Teerawat Issariyakul
Ekram Hossain

Introduction to Network Simulator NS2

Second Edition

Springer

Introduction to Network Simulator NS2

Teerawat Issariyakul • Ekram Hossain

Introduction to Network Simulator NS2

Second Edition

Springer

Teerawat Issariyakul
TOT Public Company Limited
89/2 Moo 3
Chaengwattana Rd.
Thungsonghong, Laksi
Bangkok 10210 Thailand
iteerawat@hotmail.com
ns2ultimate.com

Ekram Hossain
Department of Electrical
and Computer Engineering
University of Manitoba
75A Chancellor's Circle 15
Winnipeg, MB R3T 5V6
Canada
ekram@ee.umanitoba.ca

ISBN 978-1-4614-1405-6 e-ISBN 978-1-4614-1406-3
DOI 10.1007/978-1-4614-1406-3
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011940220

© Springer Science+Business Media, LLC 2012
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To our families

Preface

Motivation for Writing This Book

First of all, we would like to thank you for your interest in this book. This book was motivated by our frustration when we were graduate students. We were pushed to work with NS2, when we did not know what to do. We looked over the Internet, but, at the time, there were very little material on the Internet. We posted our question on every mailing list, every online forums we can think of. We did not find most of what we were looking for. We spent days and nights on many operating systems just to make NS2 run on one of the systems we have. We spent an endless amount of time, trying to understand the way NS2 moves a packet from one node to another. We did not succeed. So we were running out of time we pushed out and submitted our project/thesis as it was. Our program was as good as it was necessary to close the project. We were not satisfied with our knowledge about NS2.

After the graduation, we revisited NS2 and tried to understand it more. Bits by bits, we began to understand how NS2 does a certain thing, but perhaps more importantly why NS2 does so. From our experience, it takes months, if not years, to truly understand NS2.

We would not be surprised if our story above sounds particularly familiar to you. In fact, we expected you to experience a part, if not all, of our unpleasant experience. We wrote this book to ease your pain caused by NS2. We hope that this book can reduce your NS2 learning curve from years to months, or from months to weeks.

About NS2

NS2 is an open-source event-driven simulator designed specifically for research in computer communication networks. Since its inception in 1989, NS2 has continuously gained tremendous interest from industry, academia, and government. Having been under constant investigation and enhancement for years, NS2 now contains modules for numerous network components such as routing, transport layer protocol, and application. To investigate network performance, researchers can

simply use an easy-to-use scripting language to configure a network, and observe results generated by NS2. Undoubtedly, NS2 has become the most widely used open source network simulator, and one of the most widely used network simulators.

Unfortunately, most research needs simulation modules which are beyond the scope of the built-in NS2 modules. Incorporating these modules into NS2 requires profound understanding of NS2 architecture. Currently, most NS2 beginners rely on online tutorials. Most of the available information mainly explains how to configure a network and collect results, but does not include sufficient information for building additional modules in NS2. Despite its details about NS2 modules, the formal documentation of NS2 is mainly written as a reference book, and does not provide much information for beginners. The lack of guidelines for extending NS2 is perhaps the greatest obstacle, which discourages numerous researchers from using NS2. At the time of this writing, there is no official guide book which can help the beginners understand the architecture of NS2 in depth.

About This Book

This book is designed to be an NS2 primer. It provides information required to install NS2, run simple examples, modify the existing NS2 modules, and create as well as incorporate new modules into NS2. To this end, the details of several built-in NS2 modules are explained in a comprehensive manner.

NS2 by itself contains numerous modules. As time elapses, researchers keep developing new NS2 modules. This book does not include the details of all NS2 modules, but does so for selected modules necessary to understand the basics of NS2. We believe that once the basics of NS2 are grasped, the readers can go through other documentations, and readily understand the details of other NS2 components. The details of Network AniMator (NAM) and Xgraph are also omitted here. We understand that these two tools are nice to have and could greatly facilitate simulation and analysis of computer networks. However, we believe that they are not essential to the understanding of the NS2 concept, and their information are widely available through most of the online tutorials.

This textbook can be used by researchers who need to use NS2 for communication network performance evaluation based on simulation. Also, it can be used as a reference textbook for laboratory works for a senior undergraduate level course or a graduate level course on telecommunication networks offered in Electrical and Computer Engineering and Computer Science Programs. Potential courses include “Network Simulation and Modeling,” “Computer Networks,” “Data Communications,” “Wireless Communications and Networking,” and “Special Topics on Telecommunications.”

Book Summary

This book starts off with an introduction to network simulation in Chap. 1. We briefly discuss about computer networks and the layering concept. Then we give board statements on system analysis approaches. As one of the two main approaches, simulation can be carried out in time-driven and event-driven modes. The latter is the one NS2 was developed.

Chapter 2 provides an overview of Network Simulator 2 (NS2). Shown in this chapter are the two-language NS2 architecture, NS2 directory, and the conventions used in this book, and NS2 installation guidelines for UNIX and Windows systems. We also present a three-step simulation formulation as well as a simple example of NS2 simulation. Finally, we demonstrate how to use the `make` utility to incorporate new modules into NS2.

Chapter 3 explains the details of the NS2 two language structure, which consists of the following six main C++ classes: `Tcl`, `Instvar`, `TclObject`, `TclClass`, `TclCommand`, and `EmbeddedTcl`. Chapters 4 and 5 present the very main simulation concept of NS2. While Chap. 4 explains implementation of event-driven simulation in NS2, Chap. 5 focuses on network objects as well as packet forwarding mechanism.

Chapters 6–11 present the following six most widely used NS2 modules. First, nodes (Chap. 6) act as routers and computer hosts. Second, links, particularly `SimpleLink` objects (Chap. 7), deliver packets from one network object to another. They model packet transmission time as well as packet buffering. Third, packets (Chap. 8) contain necessary information in its header. Fourth, agents (Chaps. 9–10) are responsible for generating packets. NS2 has two main transport-layer agents: TCP and UDP agents. Finally, applications (Chap. 11) model the user demand for data transmission.

Up to Chap. 11, the book focuses mainly on wired networks. Chapter 12 demonstrates how NS2 implements Wireless Mobile Ad Hoc Networks. Discussion of packet forwarding via wireless links and node mobility is given in this chapter.

After discussing all the NS components, Chap. 13 demonstrates how a new module is developed and integrated into NS2 through two following examples: Automatic Repeat reQuest (ARQ) and packet schedulers.

Chapter 14 summarizes the postsimulation process, which consists of three main parts: debugging, variable and packet tracing, and result compilation.

Chapter 15 presents three helper modules: Timers, random number generators, and error models. It also discusses the concepts of two bit-wise operations – namely, bit masking and bit shifting, which are used throughout NS2.

Finally, Appendices A and B provide programming details which could be useful for the beginners. These details include an introduction to Tcl, OTcl, and AWK programming languages as well as a review of the polymorphism OOP concept. Last but not the least, Appendix C explains the BSD link list as well as bit level functions used throughout this book.

How to Read This Book

To Everyone

If you have not done so, we highly recommend you to visit the following online websites and learn what you can do with NS2.[1]

- Marc Greis' Tutorial: http://www.isi.edu/nsnam/ns/tutorial
- NS2 by Example: http://perform.wpi.edu/NS

To Beginners

Start simple. Learning how NS2 pass a packet from one end of a communication link to another is more challenging than what you might have imagined. In fact, if there is only one goal you should set, it is to understand this seemingly simple NS2 mechanism. Once you grasp this concept, you can quickly and effortlessly learn other NS2 mechanism.

In particular, the following steps are suggested for beginners:

Step 1 (The Basic): Learn how NS2 pass packets from one object to another from within a SimpleLink. Visit Chap. 7 for SimpleLink implementation. You may need to visit Chaps. 4–5 for details of scheduler and Network component implementation.

Step 2 (CBR and UDP): These are two high-layer components which is easy to learn. Look for the contents about UDP and CBR in Chaps. 9 and 11, respectively.

Step 3 (New Module): Next, visit Chap. 13 to learn how to build new modules into NS2. There are two examples in this chapter. Note that you may need to learn how to use `make` utility in Chap. 2.

To Instructors

This book contains 15 chapters. Different chapters may consume different amount of lecturing time. Generally, it requires more than 15 classes to cover the entire book. We suggest instructors to emphasize/omit the contents of this book at their discretion. In order to manage time efficiently, the instructors may follow the following guidelines:

- Chapters 8, 14, and 10 can be skipped without greatly impairing the content of the course.
- The instructor may pack two chapters into one lecture: Chaps. 1–2 and 4–5.

[1] Some of these contents is available in Chap. 2.

We also provide exercises at the end of each chapter. The instructors may use these exercises to set up assignments, laboratory exercises, or examination questions.

What's New in This Second Edition

In this Second edition of this book, we correct the grammatical and typological errors. We add more materials on general computer network simulation into Chap. 1. We clean up the terminologies, include more examples, and rewrite Chaps. 6 and 15. At the end of each chapter, we provide exercises which can be used as assignments, laboratory exercises, and examination questions.

In Appendix A, we include most of the information that *NS2* beginners need to know about Tcl, OTcl, and AWK. During the NS2 learning process, readers may encounter BSD-linked lists and bit level C++ functions. Readers may find the information about these two basic C++ components in Appendix C.

As of this writing, the latest NS2 version is 2.35. This version is not fully backward-compatible with version 2.29 used as a reference in the first edition of this book. Therefore, we address the compatibility issues on the link error modules in Chap. 15 and ARQ modules in Chap. 13.

We have received numerous requests to deal with wireless networking. In this regard, we develop a new chapter, namely, Wireless Mobile Ad Hoc Networks (Chap. 12). In this chapter, we explain both packet forwarding over wireless links and node mobility mechanism. The information about wireless trace formats is given in Chap. 14.

We have also developed companion online resources for this book. They contain useful NS2 information such as updates, discussion boards, detailed explanation on various topics, presentation slides, questions and answers, book corrections, and so on. The readers are highly encouraged to visit and join our online community at

- Website: http://ns2ultimate.com
- Facebook Fan Page: http://www.facebook.com/pages/Teerawat-Issariyakul/358240861417
- Twitter feed: http://twitter.com/T_Bear
- Lecture notes: http://www.ece.ubc.ca/~teerawat/NS2.htm

TOT Public Company Limited — Teerawat Issariyakul
University of Manitoba — Ekram Hossain

Acknowledgment

In this second edition, we would like to express our sincere gratitude toward our colleagues at TOT Public Company Limited, and the graduate students of the "Wireless Communications, Networks, and Services Research Group" at the University of Manitoba. We are deeply obliged to audience and followers at the NS2 website, the Facebook page, and the Twitter feed, as well as those who read the previous edition of this book and took time contacting us. Your comments and questions help us improve the content of the book in a big way. We are especially thankful to Nestor Michael C. Tiglao who went so far to help us locate the typographical errors.

We would like to thank Springer, the publisher of the book, for giving us an opportunity to share our knowledge and skill with our technical community. Special thanks go to Brett Kurzman, Springer's agent, who has keenly worked with us from the beginning to the end.

Last, but by no means the least, we would like to thank our families and friends for their understanding and continual moral support. We are forever indebted to all the members of Issariyakul and Hossain families. Thank you for giving us love, support, encouragement, and most of all for never stopping to believe in us. Thank you for not abandoning us and always being there for us, even if we have not given you time you deserved, and even have disappeared in a pile of NS2 codes.

Contents

Chapter 1
Simulation of Computer Networks

People communicate. One way or another, they exchange some information among themselves all the times. In the past several decades, many electronic technologies have been invented to aid this process of exchanging information in an efficient and creative way. Among these are the creation of fixed telephone networks, the broadcasting of television and radio, the advent of computers, the rise of the Internet, and the emergence of wireless sensation. Originally, these technologies existed and operated independently, serving their very own purposes. Not until recently that these technological wonders have started to converge, and it is a well-known fact that a computer communication network is a result of this convergence.

This chapter presents an overview of computer communication networks and the basics of simulation of such a network. Section 1.1 introduces a computer network along with the reference model which is used for describing the architecture of a computer communication network. A brief discussion on designing and modeling a complex system such as a computer network is then given in Sect. 1.2. In Sect. 1.3, the basics of computer network simulation are discussed. Section 1.4 presents one of the most common type of network simulation, namely, the time-dependent simulation. An example simulation is given in Sect. 1.5. Finally, Sect. 1.6 summarizes the chapter.

1.1 Computer Networks and the Layering Concept

A computer network is usually defined as a collection of computers interconnected for gathering, processing, and distributing information. *Computer* is used as a broad term here to include devices such as workstations, servers, routers, modems, base stations, and wireless extension points. These computers are connected by communication links such as copper cables, fiber optic cables, and microwave/satellite/radio links. A computer network can be built as a nesting and/or interconnection of several networks. The Internet is a good example of computer

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_1, © Springer Science+Business Media, LLC 2012

networks. In fact, it is a network of networks, within which tens of thousands of networks interconnect millions of computers worldwide.

1.1.1 Layering Concept

A computer network is a complex system. To facilitate design and flexible implementation of such a system, the concept of *layering* is introduced. Using a layered structure, the functionalities of a computer network can be organized as a stack of layers.

Logically, each layer communicates to its peer (a logical entity on the same layer) on the other communication node. However, the actual data transmission occurs through the lowest layer, namely, the physical layer. Therefore, data at the source node always move down the layers until reaching the physical layer. Then, it is transmitted via a physical link to a neighboring node or the destination node. At the destination node, the data are passed to the layers until reaching the corresponding peer.

Representing a well-defined and specific part of the system, each layer provides certain *services* to the above layer. When performing a task (e.g., transmit a packet), an upper layer asks its lower layer to do more specific job. Accessible (by the upper layers) through so-called interfaces, these services usually define *what* should be done in terms of network operations or primitives, but do not specifically define *how* such things are implemented. The details of how a service is implemented are defined in a so-called protocol.

A protocol is a set of rules that *multiple* peers comply with when communicating to each other.[1] As long as the peers abide to a protocol, the communication performance would be consistent and predictable. As an example, consider an error detection protocol. When a transmitter sends out a data packet, it may wait for an acknowledgment from the receiver. The receiver, on the other hand, may be responsible for acknowledging to the transmitter that the transmitted packets are received successfully.

The beauty of this layering concept is the layer independency. That is, a change in a protocol of a certain layer does not affect the rest of the system as long as the interfaces remain unchanged. Here, we highlight the words *services*, *protocol*, and *interface* to emphasize that it is the interaction among these components that makes up the layering concept.

Figure 1.1 graphically shows an overall view of the layering concept used for communication between two computer hosts: a source host and a destination host. In this figure, the functionality of each computer host is divided into four

[1]Unlike a protocol, an algorithm is a set of steps to get things done (either with or without communications).

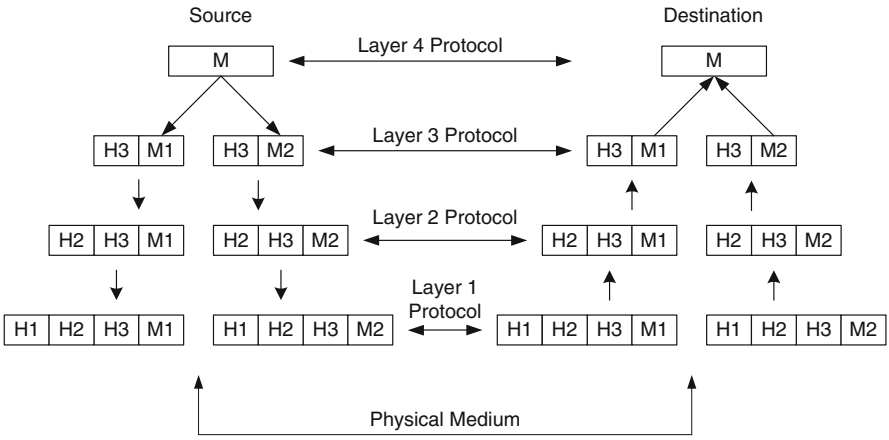

Fig. 1.1 Data flow in a layered network architecture

layers.[2] When logically linked with the same layer on another host, these layers are called *peers*.[3] Although not directly connected to each other, these peers logically communicate with one another using a protocol represented by an arrow. As mentioned earlier, the actual communication needs to propagate down the stack and use the above layering concept.

Suppose an application process running on Layer 4 of the source generates data or messages destined for the destination. The communication starts by passing a generated message `M` down to Layer 3, where the data are segmented into two chunks (`M1` and `M2`), and control information called *header* (`H3`) specific to Layer 3 is appended to `M1` and `M2`. The control information are, for example, sequence numbers, packet sizes, and error checking information. These information are understandable and used only by the peering layer on the destination to recover the data (`M`). The resulting data (e.g., `H3+M1`) are called a "protocol data unit (PDU)" and are handed to the next lower layer, where some protocol-specific control information is again added to the message. This process continues until the message reaches the lowest layer, where transmission of information is actually performed over a physical medium. Note that, along the line of these processes, it might be necessary to further segment the data from upper layers into smaller segments for various purposes. When the message reaches the destination, the reverse process takes place. That is, as the message is moving up the stack, its headers are ripped off layer by layer. If necessary, several messages are put together before being passed to the upper layer. The process continues until the original message (`M`) is recovered at Layer 4.

[2]For the sake of illustration only four layers are shown. In the real-world systems, the number of layers may vary, depending on the functionality and objectives of the networks.

[3]A peering host of a source and a destination are the destination and the source, respectively.

1.1.2 OSI and TCP/IP Reference Models

The Open Systems Interconnection (OSI) model was the first reference model developed by International Standards Organization (ISO) to provide a standard framework to describe the protocol stacks in a computer network. Its consists of seven layers, where each layer is intended to perform a well-defined function [1]. These are physical layer, data link layer, network layer, transport layer, session layer, presentation layer, and application layer. The OSI model only specifies what each layer should do; it does not specify the exact services and protocols to be used in each layer.

The Transmission Control Protocol (TCP)/Internet Protocol (IP) reference model [1], which is based on the two primary protocols, namely, TCP and IP, is used in the current Internet. These protocols have proven very powerful, and as a result have experienced widespread use and implementation in the existing computer networks. It was developed for ARPANET, a research network sponsored by the US Department of Defense, which is considered as the grandparent of all computer networks. In the TCP/IP model, the protocol stack consists of five layers – physical, data link, network, transport, and application – each of which is responsible for certain services as will be discussed shortly. Note that the application layer in the TCP/IP model can be considered as the combination of session, presentation, and application layers of the OSI model.

1.1.2.1 Application Layer

The application layer sits on top of the stack and uses services from the transport layer (discussed below). This layer supports several higher-level protocols such as Hypertext Transfer Protocol (HTTP) for World Wide Web applications, Simple Mail Transfer Protocol (SMTP) for electronic mail, TELNET for remote virtual terminal, Domain Name Service (DNS) for mapping comprehensible host names to their network addresses, and File Transfer Protocol (FTP) for file transfer.

1.1.2.2 Transport Layer

The objective of a transport layer is to perform flow control and error control for message transportation. Flow control ensures that the end-to-end transmission speed is neither too fast to make the network congested nor too slow to underutilize the network. Error control ensures that the packets are delivered to the destination properly.

Generally, when a transport protocol receives a message from the higher layer, it breaks down the message into smaller pieces. Then it generates a PDU – called a *segment* – by attaching necessary error and flow control information, and passes the segment to the lower layer.

Two well-known transport protocols, namely, TCP and User Datagram Protocol (UDP), are defined in this layer. While TCP is responsible for a reliable and connection-oriented communication between two hosts, UDP supports an unreliable

connectionless transport. TCP is ideal for applications that prefer accuracy over prompt delivery and the reverse is true for UDP.

1.1.2.3 Network Layer

This layer determines the route through which a packet is delivered from a source node to a destination node. A PDU for the network layer is called a *packet*.

1.1.2.4 Link Layer

A link layer protocol has three main responsibilities. First, flow control regulates the transmission speed in a communication link. Second, error control ensures the integrity of data transmission. Third, flow multiplexing/demultiplexing combines multiple data flows into and extracts data flows from a communication link. Choices of link layer protocols may vary from host to host and network to network. Examples of widely used link layer protocols/technologies include Ethernet, Point-to-Point Protocol (PPP), IEEE 802.11 (i.e., Wi-Fi), and Asynchronous Transfer Mode (ATM).

Link layer protocols are different from transport layer protocol as follows. The former deals with a single communication link. On the other hand, the latter does the same job for an end-to-end flow which may traverse multiple links.

1.1.2.5 Physical Layer

The physical layer deals with the transmission of data bits across a communication link. Its primary goal is to ensure that the transmission parameters (e.g., transmission power) are set appropriately to achieve the required transmission performance (e.g., to achieve the target bit error rate performance).

Finally, we point out that the five layers discussed above are common to the OSI layer. As has been mentioned already, the OSI model contains two other layers sitting on top of the transport layer, namely, session and presentation layers. The session layer simply allows users on different computers to create communication sessions among themselves. The presentation layer basically takes care of different data presentations existing across the network. For example, a unified network management system gathers data with different format from different computers and converts their format into a uniform format.

1.2 System Modeling

System modeling is an act of formulating a simple representation for an actual system. It allows investigators to look closely into the system without having to actually implement it. During the investigation, various parameters can be applied

to study system behavior. After the system is well understood, investigators can decide whether the actual system should be implemented.

System modeling often requires simplification assumptions. These assumptions exclude irrelevant details of the actual system, hence making the model cleaner and easier to implement. However, excessive assumptions may lead to inaccurate representation of the system. Design engineers need to use their discretion to achieve the best modeling trade-off.

Traditionally, there are two modeling approaches: Analytical approach and simulation approach.

1.2.1 Analytical Approach

The general concept of analytical modeling approach is to come up with a way to describe the system mathematically, and apply numerical methods to gain insight from the developed mathematical model. Examples of widely used mathematical tools are queuing and probability theories. Since analytical results derive mainly from mathematical proofs, they are true as long as the underlying conditions hold. If properly used, analytical modeling can be a cost-effective way to provide a general view of the system.

1.2.2 Simulation Approach

Simulation recreates real-world scenarios using computer programs. It is used in various applications ranging from operations research, business analysis, manufacturing planning, and biological experimentation, just to name a few. Compared to analytical modeling, simulation usually requires fewer simplification assumptions, since almost every possible detail of system specifications can be incorporated in a simulation model. When the system is rather large and complex, a straightforward mathematical formulation may not be feasible. In this case, the simulation approach is usually preferred to the analytical approach. The essence of simulation is to perform extensive experiment and make convincing argument for generalization. Due to the generalization, simulation results are usually considered not as strong as the analytical results.

1.3 Basics of Computer Network Simulation

A simulation is, more or less, a combination of art and science. That is, while the expertise in computer programming and the applied mathematics accounts for the science part, the very skills in analysis and conceptual model formulation usually represent the art portion. A long list of steps in executing a simulation process, as given in [2], seems to reflect this popular claim.

A simulation of general computer networks consists of three main parts:

- *Part 1 – Planning*: This part includes defining the problem, designing the corresponding model, and devising a set of experiments for the formulated simulation model. It is recommended that 40% of time and effort be spent on planning.
- *Part 2 – Implementing*: Implementation of simulation programs consists of three steps:
  - *Step 1 – Initialization*: This step establishes initial conditions (e.g., resetting simulation clocks and variables) so that the simulation always starts from a known state.
  - *Step 2 – Result generation*: The simulation creates and executes events, and collects necessary data generated by the created events.
  - *Step 3 – Postsimulation processing*: The raw data collected from simulation are processed and translated into performance measures of interest.

  It is recommended that 20% of time should be used for implementation.
- *Part 3 – Testing*: This part includes verifying/validating the simulation model, experimenting on the scenarios defined in Part 1 and possibly fine-tuning the experiments themselves, and analyzing the results. The remaining 40% of time should be used in this part.

This formula is in no way a strict one. Any actual simulation may require more or less time and effort, depending on the context of interest, and definitely on the modeler himself/herself.

1.3.1 Simulation Components

A computer network simulation can be thought of as a flow of interaction among network entities (e.g., nodes, packets). These entities move through the system, interact with other entities, join activities, trigger events, cause some changes to state of the system, and terminate themselves. From time to time, they contend or wait for some type of resources. This implies that there must be a logical execution sequence to cause all these actions to happen in a comprehensible and manageable way.

According to Ingalls [4], the key components of a simulation include the following:

1.3.1.1 Entities

Entities are objects that interact with one another in a simulation program. They cause some changes to the states of the system. In the context of a computer network, entities may include computer nodes, packets, flows of packets, or nonphysical objects such as simulation clocks.

1.3.1.2 Resources

Resources are limited virtual assets shared by entities such as bandwidth or power budget.

1.3.1.3 Activities and Events

From time to time, entities engage in some activities. The engagement creates events and triggers changes in the system state. Common examples of events are packet reception and route update events.

1.3.1.4 Scheduler

A scheduler maintains a list of events and their execution time. During a simulation, it moves along a simulation clock and executes events in the list chronologically.

1.3.1.5 State and Global Variables

State variables keep track of the system state. They can be classified as local variables and global variables based on their scope of operation. Local variables are valid under a limited range, while global variables are understandable globally by all program entities.

In general, global state variables hold general information shared by several entities such as the total number of nodes, the geographical area information, the reference to the scheduler, and so on.

1.3.1.6 Random Number Generator

A Random Number Generator (RNG) introduces randomness in a simulation model. It generates random numbers by sequentially picking numbers from a deterministic sequence of pseudo-random numbers [5], yet the numbers picked from this sequence appear to be random.

Without randomness, the results for every run would be exactly the same. To generate a set of different results, we may initialize the RNG for different runs with different seeds. A *seed* identifies the first location where the RNG starts picking random numbers. Two simulations whose RNG picks different initial positions would generate different simulation results.

In a computer network simulation, for example, a packet arrival process and a service process are usually modeled as random processes. These random processes are usually implemented with the aid of an RNG. The readers are referred to [6, 7] for a comprehensive treatment on random process implementation.

1.3.1.7 Statistics Gatherer

Statistics gatherers use variables to collect relevant data (e.g., packet arrival and departure time). These data can later be used to compute the performance measures such as average and standard deviation of the queuing delay for data packets traversing a network.

1.3.2 Simulation Performance

Performance of a simulation is measured by the following metrics [3]:

- *Execution speed*: How fast a simulation can be completed
- *Cost*: Expense paid to procure software/hardware, develop simulation programs, and obtain simulation results. Generally, commercial tools have more features and easier to work with at the expense of increasing cost.
- *Fidelity*: How reliable the simulation results are. Fidelity can be increased by incorporating more details (i.e., making less assumptions) into the simulation.
- *Repeatability*: An assurance that if the experiment was to be repeated, the results would be the same. Repeatability can be quantified using *confidence interval* (see the details in Sect. 1.3.3).
- *Scalability*: The impact of the size of the problem (e.g., the number of node, the input traffic) on other simulation performance measures.

1.3.3 Confidence Interval

Confidence interval [6] is a useful mathematical tool which helps quantify the level of repeatability. A confidence interval is a range between which one has confidence in finding data points. It is characterized by the width and the confidence coefficient (i.e., probability) to find data points. Figure 1.2 shows an example of confidence interval of $[-1,1]$ with the confidence coefficient is 80%. The interpretation of this example is that the probability of finding a data point within an interval $[-1,1]$ is 0.8.

It is fairly impossible/impractical to have perfectly repeatable results. Confidence interval measures such imperfection. As long as the imperfection is well defined (e.g., by confidence interval) and reasonable, the simulation results are usually deemed sufficiently reliable.

1.3.4 Choices for Network Simulation Tools

There is a wide variety of network simulation tools in the market. Each has its own strengths and weaknesses. To choose the most appropriate one, the following factors might be considered [3]:

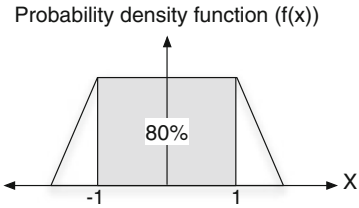

Fig. 1.2 A confidence interval of [−1,1] with confidence coefficient of 80%

1.3.4.1 Simulation Platform

The simulation platform can be software, hardware, or hybrid. Software simulation platforms can be very flexible and economical. In most cases, it can be installed in personal computers or servers. Therefore, it can be upgraded very easily. The hardware simulation platform (e.g., those using very high-level design language (VHDL)) can be very fast to run and is more suitable for computationally intensive simulation. It is also essential when input parameters need to be collected from surrounding environment. The major drawback of hardware simulation platform is that they can be prohibitively expensive to be implemented and modified. Hybrid simulation platforms combine the benefits of both the software and the hardware platforms. An example of hybrid simulation platforms is Hardware In the Loop (HIL) simulation, which is usually used to test complex real-time embedded systems [3].

1.3.4.2 Types of Simulation Tools

Simulation tools can also be classified based on how they are developed:

- *Open-source or closed-source*: By revealing its source code, open-source software opens itself for investigation. Users/programmers can find and report problems, modify the source codes, incorporate new features, and redistribute the software. The drawback of the open-source software is the lack of accountability. A lot of open-source software projects is run by volunteers. Since they can be modified by anyone at any time, they might behave differently from users' expectation. Closed-source software, on the other hand, can be modified only by the software developers. Therefore, these developers are fully accountable for the software quality.
- *Free or commercial*: Although free of charge, free software may lack the support and accountability. Commercial software, on the other hand, is usually well documented and has better technical support.
- *Publicly available or in-house*: Publicly available software can be open-source or close-source, and can be free or commercial. It can help save substantial effort

required to develop simulation software. When appropriately chosen, it can be fairly trustworthy, since well-developed software would have been extensively examined by the public. Developed internally, in-house software has greater flexibility. When the software needs to be changed or updated, the developers know what, where, and how to make the changes rapidly.

1.3.4.3 User Interface

The user interfaces of a simulation program can be Command Line Interface (CLI) or Graphic User Interface (GUI). Aiming at obtaining statistical results, a large number of academic works use CLI-based simulation tools, since these tools use most computational power for simulations. GUI-based simulation tools, on the other hand, allocate a part of computational power to improve user interface. They usually provide user-friendly network configuration interfaces, and have graphical and animation-based simulation result presentation.

1.3.4.4 Examples of Simulation Tools

The following are some of the widely used network simulation tools:

- `NS2`: An open-source software written in C++ and OTcl programming languages
- `GloMoSim`: An open-source software developed at University of California, Los Angeles (UCLA)
- `QualNet`: A commercialized version of `GloMoSim`. It supports a wider variety of protocols, has better documentation, and provides customizable simulation modules.
- `Opnet`: A commercial network simulation tool which offers several features – including HIL, parallel computing, detailed documentation, and technical support.
- `MATLAB`: A commercial multi-purpose software that can be used for network simulation and complex numerical evaluation.

1.4 Time-Dependent Simulation

As its name suggested, time-dependent simulation proceeds chronologically. It maintains a simulation clock to keep track of simulation time. Based on how events are handled, time-dependent simulation can be classified into two categories: time-driven simulation and event-driven simulation.

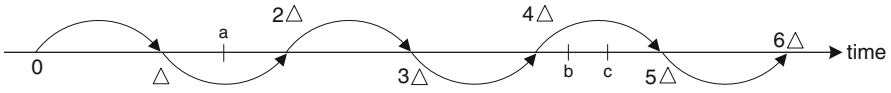

Fig. 1.3 Clock advancement in a time-driven simulation

1.4.1 Time-Driven Simulation

Time-driven simulation induces and executes events for every fixed time interval of Δ time units. In particular, it looks for events that may have occurred during this fixed interval. If found, such events would be executed as if they occurred at the end of this interval. After the execution, it advances the simulation clock by Δ time units and repeats the process. The simulation proceeds until the simulation time reaches a predefined termination time.

Figure 1.3 shows the basic idea behind time advancement in a time-driven simulation. The curved arrows represent such advances, and `a`, `b`, and `c` mark the occurrences of events. During the first interval, no event occurs. The second interval contains event `a`, which is not handled until the end of the interval.

Time interval (Δ) is an important parameter of time-driven simulation. While a large interval can lead to loss of information, a small interval can cause unnecessary waste of computational effort. Suppose, in Fig. 1.3, events `b` and `c` in the fifth interval are packet arrival and departure events, respectively. Since these two events are considered to occur exactly at the end of the interval (i.e., at 5Δ), the system state would be as if there is no packet arrival or departure events during $[4\Delta, 5\Delta]$. This is considered a loss of information. An example of waste of computational effort occurs between 2Δ and 4Δ. Although no event occurs in this interval, the simulation wastes the computational resource to stop and process events at 3Δ and 4Δ. In time-driven simulation, programmers need to use their discretion to optimize the time interval value Δ.

Example 1.1. Program 1.1 shows the pseudo codes for a time-driven simulation. Lines 1 and 2 initialize the system state variables and the simulation clock, respectively. Line 3 specifies the stopping criterion. Lines 4–6 are run as long as the simulation clock (i.e., `sim_clock`) is less than the predefined threshold `stopTime`. These lines execute events, collect statistics, and advance the simulation. □

Program 1.1 Skeleton of a time-driven simulation

```
1 initialize {system states}
2 sim_clock := startTime;
3 while {sim_clock < stopTime}
4     collect statistics from current state;
5     execute all events that occurred during
                          [sim_clock, sim_clock + step];
6     sim_clock := sim_clock + step;
7 end while
```

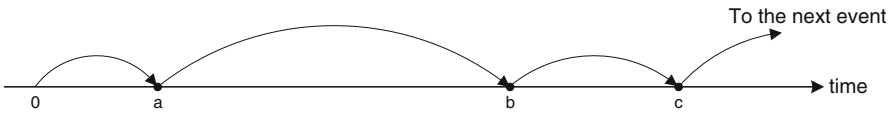

Fig. 1.4 Clock advancement in an event-driven simulation

Program 1.2 Skeleton of an event-driven simulation

```
1 initialize {system states}
2 initialize {event list}
3 while {Event list != NULL}
4     retrieve and remove an event
          whose timestamp is smallest from the event list
5     execute the retrieved event
6     set sim_clock := time corresponding to the retrieved
          event
7 end while
```

1.4.2 Event-Driven Simulation

An event-driven simulation does not proceed according to fixed time interval. Rather, it induces and executes events at any arbitrary time. Event-driven simulation has four important characteristics:

- Every event is stamped with its occurrence time and is stored in a so-called event list.
- Simulation proceeds by retrieving and removing an event with the smallest timestamp from the event list, executing it, and advancing the simulation clock to the timestamp associated with the retrieved event.
- At the execution, an event may induce one or more events. The induced events are stamped with the time when the event occurs and again are stored in the event list. The timestamp of the induced events must not be less than the simulation clock. This is to ensure that the simulation would never go backward in time.
- An event-driven simulation starts with a set of initial events in the event list. It runs until the list is empty or another stopping criterion is satisfied.

Figure 1.4 shows a graphical representation of event-driven simulation. Here, events `a, b,` and `c` are executed in order. The time gap between any pair of events is not fixed. The simulation advances from one event to another, as opposed to one interval to another in time-driven simulation. In event-driven simulation, programmers do not have to worry about optimizing time interval.

Example 1.2. Program 1.2 shows the skeleton of a typical event-driven simulation program. Lines 1 and 2 initialize the system state variables and the event list, respectively. Line 3 specifies a stopping criterion. Lines 4–6 are executed as along

as Line 3 returns `true`. Within this loop, the event whose timestamp is smallest is retrieved, executed, and removed from the list. Then, the simulation clock is set to the time associated with the retrieved event. □

1.5 A Simulation Example: A Single-Channel Queuing System

As an example, this section demonstrates a simulation of a single-channel queuing system shown in Fig. 1.5. Here, we have one communication link connecting Node A to Node B. Applications at Node A create packets according to underlying distributions for inter-arrival time and service time. After the creation, the packets are placed into a transmission buffer. When the communication link is free, the head of the line packet is transmitted to Node B, and the head of the line server fetch another packet from the buffer in a First-In-First-Out (FIFO) manner.

We now define the components of the event-driven simulation based on the framework discussed in Sect. 1.3.

1.5.1 Entities

The primary entities in this simulation include the following:

- *Applications* which generate traffic whose inter-arrival time and service time follow certain distributions,
- *A server* which stores the packet being transmitted (its state can be either `idle` or `busy`),

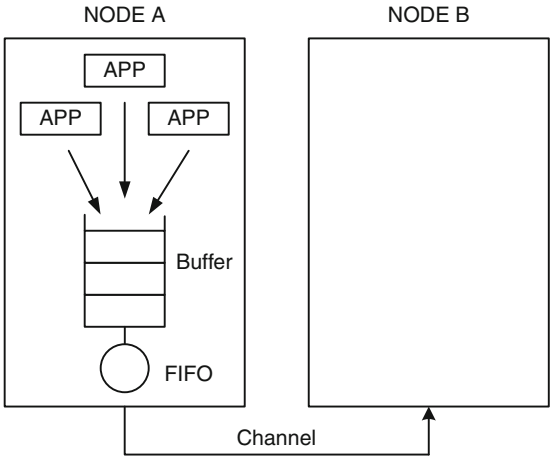

Fig. 1.5 Illustration of a single-channel queuing system

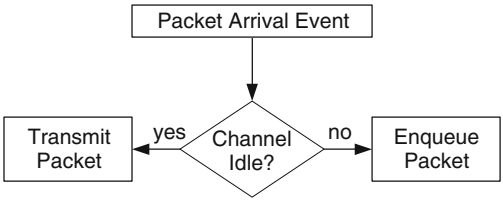

Fig. 1.6 Packet arrival event

- *A queue* which stores packets waiting to be transmitted (its state consists of the size and the current occupancy), and
- *Communication link* which carries packets from Node A to Node B.

1.5.2 State Global Variables

For simplicity, we make all variables global so that they can be accessed from anywhere in the simulation program:

- `num_pkts`: The number of packets in the systems – one in the head-of-the-line server plus all packets in the buffer.
- `link_status`: The current status of the communication link (its state can be either `idle` or `busy`).

1.5.3 Resource

Obviously, the only resource in this example is the transmission time in the channel.

1.5.4 Events

Main events in this simulation include the following:

1. `pkt_arrival` corresponds to a *packet arrival* event. This event occurs when an application generate a packet. As shown in Fig. 1.6, the packet may either be immediately transmitted or stored in the queue, depending on whether the channel is busy or idle.
2. `srv_complete` corresponds to a *successful packet transmission* event. This event indicates that a packet has been received successfully by node B. At the completion, node A begins to transmit (serve) another packet waiting in the queue. If there is no more packet to be sent, the channel becomes idle. The flow diagram of the process is shown in Fig. 1.7.

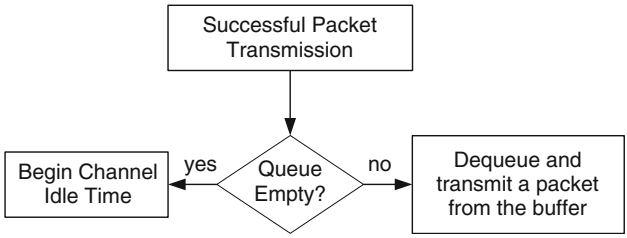

Fig. 1.7 Successful packet transmission (service completion) event

1.5.5 Simulation Performance Measures and Statistics Gatherers

Here, we consider the two following performance measures:

- *Average packet transmission latency* is the average time that a packet spends (from its arrival to its departure) in the system.
- *Average server utilization* is the percentage time where the server is busy.

It is important to note that all the above measures are the average values taken over time. The simulation time should be sufficiently long to ensure statistical accuracy of the simulation result.

In order to compute the above two performance measures, arrival time and service times of all the packets must be gathered. The computation of other performance measures from these two data will be shown later in this section.

1.5.6 Simulation Program

Program 1.3 shows the skeleton of a program which simulates the single-channel queuing system described above. It proceeds according to the three-step simulation implementation defined in Sect. 1.3:

Step 1 – Initialization (Lines 1–4): Lines 1 and 2 initialize the status of the communication link (`link_status`) to `idle` and number of packets in the systems (`num_pkts`) to zero. Line 3 sets the simulation clock to start at zero. Line 4 creates an event list (`event_list`) by invoking the procedure `create_list()`. The event list contains all events in the simulation. Again, the scheduler moves along this list and executes the events chronologically. From within the procedure `create_list()`, the initial packet arrival events created by applications are placed on the event list.

Step 2 – Result generation (Lines 5–10): This is the main part of the program where the loop runs as long as the two following conditions satisfied: (1) the event list is nonempty and (2) the simulation clock has not reached a predefined threshold.

Program 1.3 Simulation skeleton of a single-channel queuing system

```
% Initialize system states
1  link_status = idle; %The initial link status is idle
2  num_pkts = 0;       %Number of packets in system
3  sim_clock = 0;      %Current time of simulation

   %Generate packets and schedule their arrivals
4  event_list = create_list();

   % Main loop
5  while {event_list != empty} & {sim_clock < stop_time}
6    if the application creates events, insert them to the list
7    expunge the previous event from event list;
8    set sim_clock = time of current event;
9    execute the current event;
10 end while

   %Define events
11 pkt_arrival(){
12     if(link_status)
13      link_status = busy;
14      num_pkts = num_pkts + 1;
        % Update "event_list": Put "successful packet tx event"
        % into "event_list," T is random service time.
15      schedule event "srv_complete" at sim_clock + T;
16     else
17         num_queue = num_queue + 1; %Place packet in queue
18     num_pkts = num_queue + 1;
19 }

20 srv_complete(){
21     num_pkts = num_pkts - 1;
22     if(num_pkts > 0)
23         schedule event "srv_complete" at sim_clock + T;
24     else
25         link_status = idle;
26         num_pkts = 0;
27 }
```

Within the loop, Line 6 takes arrival/departure events (if any) created by applications and placed them in the event list. Lines 7–10 execute the next event on the event list by invoking either the procedure `pkt_arrival()` in Lines 11–19 or the procedure `srv_complete()` in Lines 20–27.

The procedure `pkt_arrival()` (Lines 11–19) checks whether the communication link is idle when a packet arrives. If so, the link is set to *busy*, and a *service completion* event is inserted into the `event_list` for future execution. The timestamp associated with the event is equal to the current clock time (`sim_clock`) plus the packet's randomly generated service time (`T`). If the link is

Table 1.1 Probability mass functions of inter-arrival time and service time

Time unit	Inter-arrival (probability mass)	Service (probability mass)
1	0.2	0.5
2	0.2	0.3
3	0.2	0.1
4	0.2	0.05
5	0.1	0.05
6	0.05	
7	0.05	

Table 1.2 Simulation result of a single-channel queuing system

Packet	Interarr. time	Service time	Arrival time	Service starts	Packet waiting time	Packet transmission latency
1	–	5	0	0	0	5
2	2	4	2	5	3	7
3	4	1	6	9	3	4
4	1	1	7	10	3	4
5	6	3	13	13	0	3
6	7	1	20	20	0	1
7	2	1	22	22	0	1
8	1	4	23	23	0	4
9	3	3	26	27	1	4
10	5	2	31	31	0	2
					1.0	3.5

busy, on the other hand, the packet will be enqueued into the buffer, and the packet counter (`num_pkts`) is incremented by one unit.

The procedure `srv_complete()` (Lines 20–27) first updates the number of packets in the system (`num_pkts`). Then, it checks whether the system contains any packet. If so, the head-of-the-line packet will be served. This is done by inserting another *service completion* event at time `sim_clock + T`. However, if the queue is empty, the channel is set to `idle` and the number of packets in the system is set to zero.

Step 3 – Postsimulation processing: This step collects and computes the performance measures based on the simulation results. Suppose that the inter-arrival time and the service time comply with the probability mass functions specified in Table 1.1. Table 1.2 shows the simulation results for ten packets.

In Table 1.2, the second and third columns represent the inter-arrival time and service time, respectively, of each packet. These two columns contain raw information. Data shown in other columns derive from these two columns.

The fourth and fifth columns specify the time where the packets arrive and start to be served in the head-of-the-line server, respectively. The sixth column represents the packet waiting time – the time that a packet spends in the queue. It is computed as the time difference between when the service starts and when the packet arrives. Finally, the seventh column represents the packet transmission latency – the time

Table 1.3 Evolution of number of packets in the system over time

Event	Packet no.	Simulation clock
Arrival	1	0
Arrival	2	2
Completion	1	5
Arrival	3	6
Arrival	4	7
Completion	2	9
Completion	3	10
Completion	4	11
Arrival	5	13
Completion	5	16

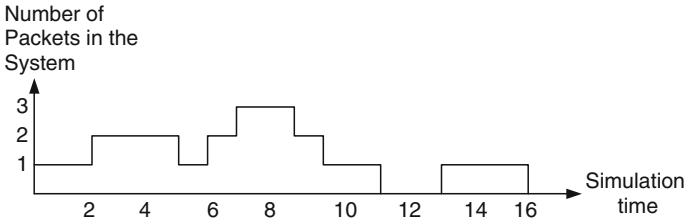

Fig. 1.8 Number of packets in the system at various instances

that a packet spends in both the queue and the channel. It is computed as the summation of the waiting time and the service time.

Based on the results in Table 1.2, we compute the average waiting time and the average packet transmission latency by averaging the sixth and seventh columns (i.e., adding all the values and dividing the result by 10). The results are therefore 1.0 and 3.5 time units, respectively.

Based on the information in Table 1.2, we also show a series of events and the dynamics of buffer occupancy with respect to the *Simulation Clock* (`sim_clock`) in Table 1.3 and Fig. 1.8, respectively. Based on Fig 1.8, the mean server utilization can be computed from the ratio of the time when the server is in use to the simulation time, which is $14/16 = 0.875$ in this case.

1.6 Chapter Summary

A computer network is a complex system. Design analysis, and optimization of a computer network can be a comprehensive task. Simulation, regarded as one of the most powerful performance analysis tools, is usually used in carrying out this task to complement the analytical tools.

This chapter focuses mainly on time-dependent simulation, which advances in a time domain. The time-dependent simulation can be classified into two categories. Time-driven simulation advances the simulation by fixed time intervals, while event-driven simulation proceeds from one event to another. NS2 is an event-driven simulation tool. Designing event-driven simulation models using NS2 is the theme of the rest of the book.

1.7 Exercises

1. What are the differences between OSI model and TCP/IP model. Draw a diagram to emphasize the differences.
2. What are the key steps in simulating a computer communication network?
3. Draw a probability density function with confidence interval $[-7, +7]$ and confidence coefficient is 95%.
4. You are given a text file. Each line of this text file contains a number representing a data point. Write a program which computes the average value and the standard deviation for the data points along with the confidence level when a confidence interval is given as an input parameter.
5. What are the two types of time-dependent simulations? Write down their main features, strengths, and weaknesses.
6. Write a sub-routine which prints out the current time slot. In a time-slotted system, write a program which executed the sub-routine at time slot 1, 7, 13, 24, and 47 by

   a. Using time-driven simulation
   b. Using event-driven simulation

Chapter 2
Introduction to Network Simulator 2 (NS2)

2.1 Introduction

Network Simulator (Version 2), widely known as NS2, is simply an event-driven simulation tool that has proved useful in studying the dynamic nature of communication networks. Simulation of wired as well as wireless network functions and protocols (e.g., routing algorithms, TCP, UDP) can be done using NS2. In general, NS2 provides users with a way of specifying such network protocols and simulating their corresponding behaviors.

Due to its flexibility and modular nature, NS2 has gained constant popularity in the networking research community since its birth in 1989. Ever since, several revolutions and revisions have marked the growing maturity of the tool, thanks to substantial contributions from the players in the field. Among these are the University of California and Cornell University who developed the REAL network simulator,[1] the foundation on which NS is invented. Since 1995 the Defense Advanced Research Projects Agency (DARPA) supported the development of NS through the Virtual InterNetwork Testbed (VINT) project [10].[2] Currently the National Science Foundation (NSF) has joined the ride in development. Last but not the least, the group of researchers and developers in the community are constantly working to keep NS2 strong and versatile.

Again, the main objective of this book is to provide the readers with insights into the NS2 architecture. This chapter gives a brief introduction to NS2. NS2 Beginners are recommended to go thorough the detailed introductory online resources. For example, NS2 official website [12] provides NS2 source code as well as detailed installation instruction. The web pages in [13] and [14] are among highly recommended ones which provide tutorial and examples for setting up basic

[1] REAL was originally implemented as a tool for studying the dynamic behavior of flow and congestion control schemes in packet-switched data networks.

[2] Funded by DARPA, the VINT project aimed at creating a network simulator that will initiate the study of different protocols for communication networking.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_2, © Springer Science+Business Media, LLC 2012

NS2 simulation. A comprehensive list of NS2 codes contributed by researchers can be found in [15]. These introductory online resources would be helpful in understanding the material presented in this book.

In this chapter an introduction to NS2 is provided. In particular, Sect. 2.2 presents the basic architecture of NS2. The information on NS2 installation is given in Sect. 2.3. Section 2.4 shows NS2 directories and conventions. Section 2.5 shows the main steps in NS2 simulation. A simple simulation example is given in Sect. 2.6. Section 2.7 describes how to include C++ modules in NS2. Finally, Sect. 2.8 concludes the chapter.

2.2 Basic Architecture

Figure 2.1 shows the basic architecture of NS2. NS2 provides users with an executable command "`ns`" which takes one input argument, the name of a Tcl simulation scripting file. In most cases, a simulation trace file is created and is used to plot graph and/or to create animation.

NS2 consists of two key languages: C++ and Object-oriented Tool Command Language (OTcl). While the C++ defines the internal mechanism (i.e., a backend) of the simulation, the OTcl sets up simulation by assembling and configuring the objects as well as scheduling discrete events (i.e., a frontend). The C++ and the OTcl are linked together using TclCL. Mapped to a C++ object, variables in the OTcl domains are sometimes referred to as *handle*s. Conceptually, a handle is just a string (e.g., "`_o10`") in the OTcl domain and does not contain any functionality. Instead, the functionality (e.g., receiving a packet) is defined in the mapped C++ object (e.g., of class `Connector`). In the OTcl domain, a handle acts as a frontend which interacts with users and other OTcl objects. It may define its own procedures and variables to facilitate the interaction. Note that the member procedures and variables in the OTcl domain are called instance procedures (instprocs) and instance variables (instvars), respectively. Before proceeding further,

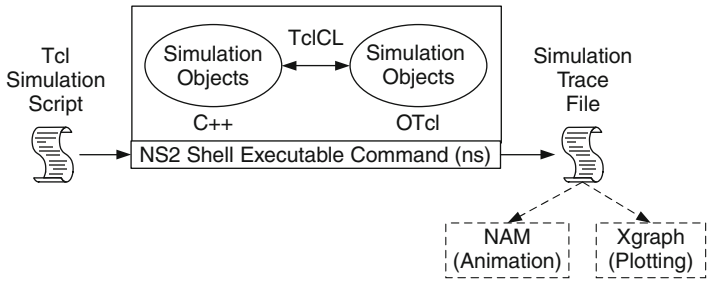

Fig. 2.1 Basic architecture of NS

the readers are encouraged to learn C++ and OTcl languages. We refer the readers to [16] for the detail of C++, while a brief tutorial of Tcl and OTcl tutorial are given in Appendices A.1 and A.2, respectively.

NS2 provides a large number of built-in C++ classes. It is advisable to use these C++ classes to set up a simulation via a Tcl simulation script. However, advance users may find these objects insufficient. They need to develop their own C++ classes and use a OTcl configuration interface to put together objects instantiated from these class.

After simulation, NS2 outputs either text-based simulation results. To interpret these results graphically and interactively, tools such as NAM (Network AniMator) and XGraph are used. To analyze a particular behavior of the network, users can extract a relevant subset of text-based data and transform it to a more conceivable presentation.

2.3 Installation

NS2 is a free simulation tool, which can be obtained from [10]. It runs on various platforms including UNIX (or Linux), Windows, and Mac systems. Being developed in the Unix environment, with no surprise, NS2 has the smoothest ride there, and so does its installation. However, due to the popularity of windows systems, the discussion in this book is based on a Cygwin (UNIX emulator) activated Windows system.

NS2 source codes are distributed in two forms: the all-in-one suite and the component-wise. With the all-in-one package, users get all the required components along with some optional components. This is basically a recommended choice for the beginners. This package provides an "`install`" script which configures the NS2 environment and creates NS2 executable file using the "`make`" utility.

The current all-in-one suite consists of the following main components:

- NS release 2.35,
- Tcl/Tk release 8.5.8,
- OTcl release 1.14, and
- TclCL release 1.20.

and the following are the optional components:

- NAM release 1.15: NAM is an animation tool for viewing network simulation traces and packet traces.
- Zlib version 1.2.3: This is the required library for NAM.
- Xgraph version 12.2: This is a data plotter with interactive buttons for panning, zooming, printing, and selecting display options.

Table 2.1 Additional Cygwin packages required to run NS2

Category	Packages
Mandatory	`gcc4`, `gcc-g++`, `gawk`, `gzip`, `tar`, `make`, `patch`, `perl`, `w32api`
Optinal (graphic-related)	`xorg-xserver`, `xinit`, `libX11-devel`, `libXmu-devel`

The idea of the component-wise approach is to obtain the above pieces and install them individually. This option save considerable amount of downloading time and memory space. However, it could be troublesome for the beginners and is therefore recommended only for experienced users.

2.3.1 Installing an All-In-One NS2 Suite on Unix-Based Systems

The all-in-one suite can be installed in the Unix-based machines by simply running the "`install`" script and following the instructions therein. The only requirement is a computer with a C++ compiler installed. The following two commands show how the all-in-one NS2 suite can be installed and validated, respectively:

```
>>./install
>>./validate
```

Validating NS2 involves simply running a number of working scripts that verify the essential functionalities of the installed components.

2.3.2 Installing an All-In-One NS2 Suite on Windows-Based Systems

To run NS2 on Windows-based operating systems, a bit of tweaking is required. Basically, the idea is to make Windows-based machines emulate the functionality of the Unix-like environment. A popular program that performs this job is Cygwin.[3] After getting Cygwin to work, the same procedure as that of Unix-based installation can be followed. For ease of installation, it is recommended that the all-in-one package be used. The detailed description of Windows-based installation can be found online at NS2's Wiki site [10], where the information on post-installation troubles can also be found.

Note that by default Cygwin does not install all packages necessary to run NS2. A user needs to manually install the addition packages shown in Table 2.1.[4]

[3]Cygwin is available online and comes free. Information such as how to obtain and install Cygwin is available online at the Cygwin website www.cygwin.com.

[4]Different versions may install different default packages. Users may need to install more or less packages depending on the version of Cygwin.

2.4 Directories and Convention

2.4.1 Directories

Suppose that NS2 is installed in Directory `nsallinone-2.35`. Figure 2.2 shows the directory structure under directory `nsallinone-2.35`. Here, the directory `nsallinone-2.35` is on the Level 1. On the Level 2, the directory `tclcl-1.20` contains classes in TclCL (e.g., `Tcl, TclObject, TclClass`). All NS2 simulation modules are in the directory `ns-2.35` on the Level 2. Hereafter, we will refer to directories `ns-2.35` and `tclcl-1.20` as *~ns/* and *~tclcl/*, respectively.

On Level 3, the modules in the interpreted hierarchy are under the directory `tcl`. Among these modules, the frequently used ones (e.g., `ns-lib.tcl, ns-node.tcl, ns-link.tcl`) are stored under the directory `lib` on Level 4. Simulation modules in the compiled hierarchy are classified in directories on Level 3. For example, directory `tools` contain various helper classes such as random variable generators. Directory `common` contains basic modules related to packet forwarding such as the simulator, the scheduler, connector, packet. Directories `queue`, `tcp`, and `trace` contain modules for queue, TCP (Transmission Control Protocol), and tracing, respectively.

2.4.2 Convention

The terminologies and formats that are used in NS2 and in this book hereafter are shown below:

2.4.2.1 Terminology

- An NS2 simulation script (e.g., `myfirst_ns.tcl`) is referred to as a *Tcl simulation script*.

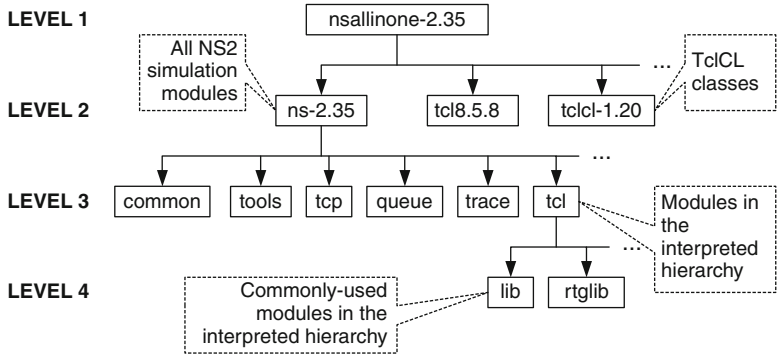

Fig. 2.2 Directory structure of NS2 [14]

- C++ and OTcl class hierarchies, which have one-to-one correspondence, are referred to as *the compiled hierarchy* and *the interpreted hierarchy*, respectively. Class (or member) variables and class (or member) functions are the variables and functions which belong to a class. In the compiled hierarchy, they are referred to simply as variables and functions, respectively. Those in the interpreted hierarchy are referred to as *instance variable*s (*instvar*s) and *instance procedure*s (*instproc*s), respectively. As we will see in Sect. 3.4, *OTcl command*, is a special instance procedure, whose implementation is in the compiled hierarchy (i.e., written in C++). An OTcl object is, therefore, associated with instance variables, instance procedures, and OTcl commands, while a C++ object is associated with variables and functions.
- A "`MyClass`" object is a shorthand for an object of class `MyClass`. A "`MyClass`" pointer is a shorthand for a pointer which points to an object of class `MyClass`. For example, based on the statements "`Queue q`" and "`Packet* p`," "`q`" and "`p`" are said to be a "`Queue`" object and a "`Packet` pointer," respectively. Also, suppose further that class `DerivedClass` and `AnotherClass` derive from class `MyClass`. Then, the term a `MyClass` object refers to any object which is instantiated from class `MyClass` or its derived classes (i.e., `DerivedClass` or `AnotherClass`).
- *Objects* and *instances* are instantiated from a C++ class and an OTcl class, respectively. However, the book uses these two terms interchangeably.
- NS2 consists of two languages. Suppose that objects "`A`" and "`B`" are written in each language and correspond to one another. Then, "`A`" is said to be *the shadow object* of "`B`." Similarly "`B`" is said to be *the shadow object* of "`A`."
- Consider two consecutive nodes in Fig. 3.2. In this configuration, an object (i.e., node) on the left always sends packets to the object on the right. The object on the right is referred to as a *downstream object* or a *target*, while the object on the left is referred to as an *upstream object*. In a general case, an object can have more than one target. A packet must be forwarded to one of those targets. From the perspective of an upstream object, a downstream object that receives the packet is also referred to as a *forwarding object*.

2.4.2.2 Notations

- As in C++, we use "`::`" to indicate the scope of functions and instprocs (e.g., `TcpAgent::send(...)`).
- Most of the texts in this book are written in regular letters. NS2 codes are written in "`this font type`." The quotation marks are omitted if it is clear from the context. For example, the Simulator is a general term for the simulating module in NS2, while a `Simulator` object is an object of class `Simulator`.
- A value contained in a variable is embraced with `<>`. For example, if a variable `var` stores an integer `7`, `<var>` will be `7`.
- A command prompt or an NS2 prompt is denoted by "`>>`" at the beginning of a line.

Table 2.2 Examples of NS2 naming convention

	The interpreted hierarchy	The compiled hierarchy
Base class	`Agent`	`Agent`
Derived class	`Agent/TCP`	`TcpAgent`
Derived class (2nd level)	`Agent/Tcp/Reno`	`RenoTcpAgent`
Class functions	`installNext`	`install_next`
Class variables	`windowOption_`	`wnd_option_`

- In this book, codes shown in figures are *partially* excerpted from NS2 file. The file name from which the codes is excerpted is shown in the first line of the figure. For example, the codes in Program 2.1 are excerpted from file "`myfirst_ns.tcl`."
- A class name may consist of several words. All the words in a class name are capitalized. In the interpreted hierarchy, a derived class is named by having the name of its parent class following by a slash character ("/") as a prefix, while that compiled classes are named by having the name of its base class as a suffix. Examples of NS2 naming convention are given in Table 2.2.
- In the interpreted hierarchy, an instproc name is written in lower case. If the instproc name consists of more than one word, each word except for the first one will be capitalized. In the compiled hierarchy, all the words are written in lower case and separated by an underscore "_" (see Table 2.2).
- The naming convention for variables is similar to that for functions and instprocs. However, the last character of the names of class variables in both the hierarchies is always an underscore ("_"; see Table 2.2). Note that this convention is only a guideline that a programmer should (but does not *have to*) follow.

2.5 Running NS2 Simulation

2.5.1 NS2 Program Invocation

After the installation and/or recompilation (see Sect. 2.7), an executable file "`ns`" is created in the NS2 home directory. NS2 can be invoked by executing the following statement from the shell environment:

```
>>ns [<file>] [<args>]
```

where `<file>` and `<args>` are optional input argument. If no argument is given, the command will bring up an NS2 environment, where NS2 waits to interpret commands from the standard input (i.e., keyboard) line-by-line. If the first input argument `<file>` is given, NS2 will interpret the input scripting `<file>` (i.e., a so-called Tcl simulation script) according to the Tcl syntax. The detail for writing a Tcl scripting file is given in Appendix A.1. Finally, the input arguments `<args>`,

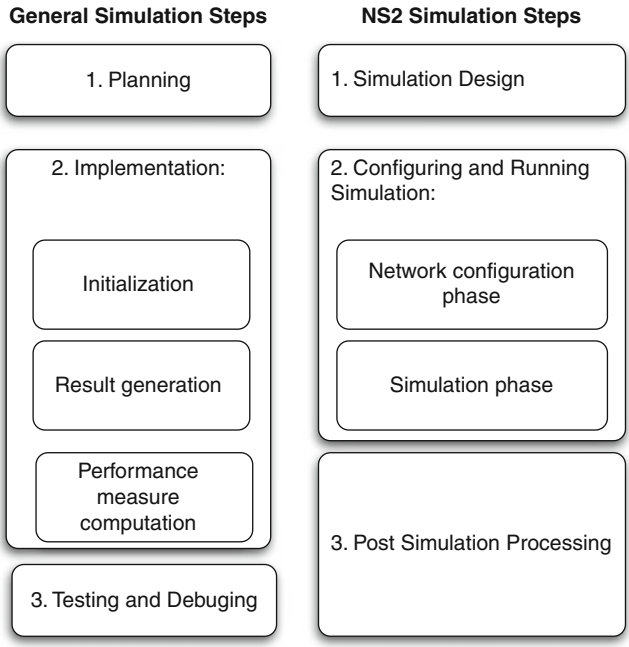

Fig. 2.3 A comparison of general simulation steps and NS2 simulation steps

each separated by a white space, are fed to the Tcl file `<file>`. From within the file `<file>`, the input argument is stored in the built-in variable "`argv`" (see Appendix A.1.1).

2.5.2 Main NS2 Simulation Steps

Section 1.3 presents the key steps for general simulation. As shown in Fig. 2.3, the general simulation steps can be tailored to fit with the NS2 framework. The key NS2 simulation steps include the following:

Step 1: Simulation Design

The first step in simulating a network is to design the simulation. In this step, the users should determine the simulation purposes, network configuration, assumptions, the performance measures, and the type of expected results.

Step 2: Configuring and Running Simulation

This step implements the design in the first step. It consists of two phases:

- *Network configuration phase*: In this phase, network components (e.g., node, TCP and UDP) are created and configured according to the simulation design. Also, the events such as data transfer are scheduled to start at a certain time.
- *Simulation Phase*: This phase starts the simulation which was configured in the Network Configuration Phase. It maintains the simulation clock and executes events chronologically. This phase usually runs until the simulation clock reaches a threshold value specified in the Network Configuration Phase.

In most cases, it is convenient to define a simulation scenario in a Tcl scripting file (e.g., `<file>`) and feed the file as an input argument of an NS2 invocation (e.g., executing "`ns <file>`").

Step 3: Postsimulation Processing

The main tasks in this steps include verifying the integrity of the program and evaluating the performance of the simulated network. While the first task is referred to as *debugging*, the second one is achieved by properly collecting and compiling simulation results (see Chap. 14).

2.6 A Simulation Example

We demonstrate a network simulation through a simple example. Again, a simulation process consists of three steps.

Step 1: Simulation Design

Figure 2.4 shows the configuration of a network under consideration. The network consists of five nodes "`n0`" to "`n4`." In this scenario, node "`n0`" sends constant-bit-rate (CBR) traffic to node "`n3`," and node "`n1`" transfers data to node "`n4`" using a file transfer protocol (FTP). These two carried traffic sources are carried by transport layer protocols User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), respectively. In NS2, the transmitting object of these two protocols are a UDP agent and a TCP agent, while the receivers are a Null agent and a TCP sink agent, respectively.

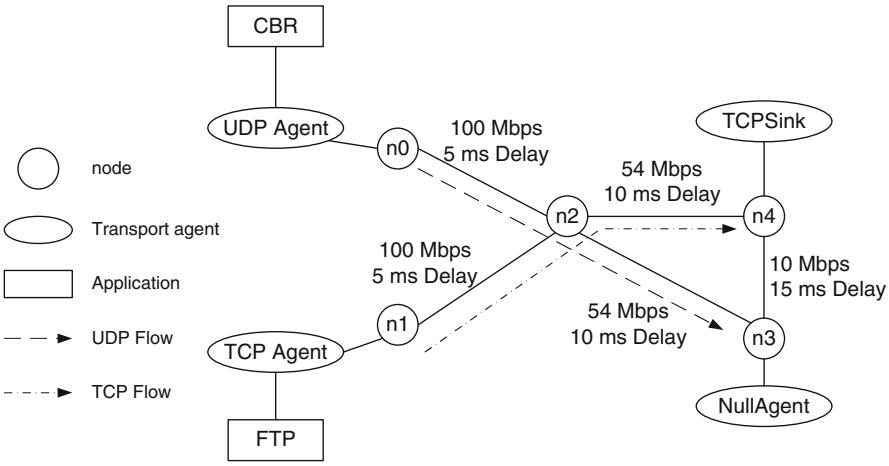

Fig. 2.4 A sample network topology

Step 2: Configuring and Running Simulation

Programs 2.1 and 2.2 show two portions of a Tcl simulation script which implements the scenario in Fig. 2.4.

Consider Program 2.1. This program creates a simulator instance in Line 1. It creates a trace file and a NAM trace file in Lines 2–3 and 4–5, respectively. It defines procedure `finish{}` in Lines 6–13. Finally, it creates nodes and links them together in Lines 14–18 and 19–24, respectively.

The Simulator is created in Line 1 by executing "`new Simulator`." The returned `Simulator` handle is stored in a variable "`ns`." Lines 2 and 4 open files "`out.tr`" and "`out.nam`," respectively, for writing. The variables "`myTrace`" and "`myNAM`" are the file handles for these two files, respectively. Lines 3 and 5 inform NS2 to collect all trace information for a regular trace and a NAM trace, respectively.

The procedure `finish{}` will be invoked immediately before the simulation terminates. The keyword `global` informs the Tcl interpreter that the variables "`ns`," "`myTrace`," "`myNAM`" are those defined in the global scope (i.e., defined outside the procedure). Line 8 flushes the buffer of the packet tracing variables. Lines 9 and 10 close the files associated with handles "`myTrace`" and "`myNAM`." Line 11 executes the statement "`nam out.nam &`" from the shell environment, where "`nam`" is an executable file which invoke the Network AniMator. Finally, Line 12 tells NS2 to exit with code 0.

Lines 14–18 create Nodes using the instproc `node{...}` of the Simulator whose handle is "`ns`." Lines 19–23 connect each pair of nodes with a bi-directional link using an instproc `duplex-link {src dst bw delay qtype}` of class `Simulator`, where "`src`" is a beginning node, "`dst`" is a terminating node, "`bw`" is the link bandwidth, "`delay`" is the link propagation delay, and "`qtype`" is the type of the queues between the node "`src`" and the node "`dst`." Similar to

Program 2.1 First NS2 Program

```
# myfirst_ns.tcl
# Create a Simulator
1  set ns [new Simulator]

# Create a trace file
2  set mytrace [open out.tr w]
3  $ns trace-all $mytrace

# Create a NAM trace file
4  set myNAM [open out.nam w]
5  $ns namtrace-all $myNAM

# Define a procedure finish
6  proc finish { } {
7        global ns mytrace myNAM
8        $ns flush-trace
9        close $mytrace
10       close $myNAM
11       exec nam out.nam &
12       exit 0
13 }

# Create Nodes
14 set n0 [$ns node]
15 set n1 [$ns node]
16 set n2 [$ns node]
17 set n3 [$ns node]
18 set n4 [$ns node]

# Connect Nodes with Links
19 $ns duplex-link  $n0 $n2 100Mb 5ms DropTail
20 $ns duplex-link  $n1 $n2 100Mb 5ms DropTail
21 $ns duplex-link  $n2 $n4 54Mb 10ms DropTail
22 $ns duplex-link  $n2 $n3 54Mb 10ms DropTail
23 $ns simplex-link $n3 $n4 10Mb 15ms DropTail
24 $ns queue-limit $n2 $n3 40
```

the instproc `duplex-link{...}`, Line 23 create a uni-directional link using an instproc `simplex-link{...}` of class `Simulator`. Finally, Line 24 sets size of the queue between node "`n2`" and node "`n3`" to be 40 packets.

Next, consider the second portion of the Tcl simulation script in Program 2.2. A UDP connection, a CBR traffic source, a TCP connection, and an FTP session are created and configured in Lines 25–30, 31–34, 35–40, and 41–42, respectively. Lines 43–47 schedule discrete events. Finally, the simulator is started in Line 48 using the instproc `run{}` associated with the simulator handle "`ns`."

To create a UDP connection, a sender "`udp`" and a receiver "`null`" are created in Lines 25 and 27, respectively. Taking a node and an agent as input argument, an instproc `attach-agent{...}` of class `Simulator` in Line 26 attaches a UDP

Program 2.2 First NS2 Program (Continued)

```
#  Create a UDP agent
25 set udp [new Agent/UDP]
26 $ns attach-agent $n0 $udp
27 set null [new Agent/Null]
28 $ns attach-agent $n3 $null
29 $ns connect $udp $null
30 $udp set fid_ 1

#  Create a CBR traffic source
31 set cbr [new Application/Traffic/CBR]
32 $cbr attach-agent $udp
33 $cbr set packetSize_ 1000
34 $cbr set rate_ 2Mb

#  Create a TCP agent
35 set tcp [new Agent/TCP]
36 $ns attach-agent $n1 $tcp
37 set sink [new Agent/TCPSink]
38 $ns attach-agent $n4 $sink
39 $ns connect $tcp $sink
40 $tcp set fid_ 2

#  Create an FTP session
41 set ftp [new Application/FTP]
42 $ftp attach-agent $tcp

#  Schedule events
43 $ns at 0.05 "$ftp start"
44 $ns at 0.1  "$cbr start"
45 $ns at 60.0 "$ftp stop"
46 $ns at 60.5 "$cbr stop"
47 $ns at 61 "finish"

#  Start the simulation
48 $ns run
```

agent "udp" and a node "n0" together. Similarly, Line 28 attaches a Null agent "null" to a node "n3." The instproc connect{<from_agt> <to_agt>} in Line 29 informs an agent <from_agt> to send the generated traffic to an agent <to_agt>. Finally, Line 30 sets the UDP flow ID to be 1. The construction of a TCP connection in Lines 35–40 is similar to that of a UDP connection in Lines 25–30.

A CBR traffic source is created in Line 31. It is attached to a UDP agent "udp" in Line 32. The packet size and generation rate of the CBR connection are set to 1,000 bytes and 2 Mbps, respectively. Similarly, an FTP session handle is created in Line 41 and is attached to a TCP agent "tcp" in Line 42.

In NS2, discrete events can be scheduled using an instproc at{...} of class Simulator, which takes two input arguments: <time> and <str>.

This instproc schedules an execution of <str> when the simulation time is <time>. Lines 43 and 44 start the FTP and CBR traffic at 0.05th second and 1st second, respectively. Lines 45 and 46 stop the FTP and CBR traffic at 60.0th second and 60.5th second, respectively. Line 47 terminates the simulation by invoking the procedure finish{} at 61st second. Note that the FTP and CBR traffic source can be started and stopped by invoking their OTcl commands start{} and stop{}, respectively.

We run the above simulation script by executing

```
>>ns myfirst_ns.tcl
```

from the shell environment. At the end of simulation, the trace files should be created and NAM should be running (since it is invoked from within the procedure finish{}).

Step 3: Post simulation Processing: Packet Tracing

Packet tracing records the detail of packet flow during a simulation. It can be classified into a text-based packet tracing and a NAM packet tracing.

Text-Based Packet Tracing

Text-based packet tracing records the detail of packets passing through network checkpoints (e.g., nodes and queues). A part of the text-based trace obtained by running the above simulation (myfirst_ns.tcl) is shown below.

```
...
+ 0.110419 1 2 tcp 1040 ------- 2 1.0 4.0 5 12
+ 0.110419 1 2 tcp 1040 ------- 2 1.0 4.0 6 13
- 0.110431 1 2 tcp 1040 ------- 2 1.0 4.0 5 12
- 0.110514 1 2 tcp 1040 ------- 2 1.0 4.0 6 13
r 0.11308 0 2 cbr 1000 ------- 1 0.0 3.0 2 8
+ 0.11308 2 3 cbr 1000 ------- 1 0.0 3.0 2 8
- 0.11308 2 3 cbr 1000 ------- 1 0.0 3.0 2 8
r 0.11316 0 2 cbr 1000 ------- 1 0.0 3.0 3 9
+ 0.11316 2 3 cbr 1000 ------- 1 0.0 3.0 3 9
- 0.113228 2 3 cbr 1000 ------- 1 0.0 3.0 3 9
r 0.115228 2 3 cbr 1000 ------- 1 0.0 3.0 0 6
r 0.115348 1 2 tcp 1040 ------- 2 1.0 4.0 3 10
+ 0.115348 2 4 tcp 1040 ------- 2 1.0 4.0 3 10
- 0.115348 2 4 tcp 1040 ------- 2 1.0 4.0 3 10
r 0.115376 2 3 cbr 1000 ------- 1 0.0 3.0 1 7
r 0.115431 1 2 tcp 1040 ------- 2 1.0 4.0 4 11
...
```

Type Identifier	Time	Source Node	Destination Node	Packet Name	Packet Size	Flags	Flow ID	Source Address	Destination Address	Sequence Number	Packet Unique ID

Fig. 2.5 Format of each line in a normal trace file

Figure 2.5 an example of a trace file. Each line in the trace file consists of 12 columns.

The general format of each trace line is shown in Fig. 2.5, where 12 columns make up a complete trace line. The *type identifier* field corresponds to four possible event types that a packet has experienced: "`r`" (received), "+" (enqueued), "−" (dequeued), and "`d`" (dropped). The *time* field denotes the time at which such event occurs. Fields 3 and 4 are the starting and the terminating nodes, respectively, of the link at which a certain event takes place. Fields 5 and 6 are packet type and packet size, respectively. The next field is a series of flags, indicating any abnormal behavior. Note the output "`-------`" denotes no flag. Following the flags is a packet flow ID. Fields 9 and 10 mark the source and the destination addresses, respectively, in the form of "`node.port`." For correct packet assembly at the destination node, NS also specifies a packet sequence number in the second last field. Finally, to keep track of all packets, a packet unique ID is recorded in the last field.

Now, having this trace at hand would not be useful unless meaningful analysis is performed on the data. In post simulation analysis, one usually extracts a subset of the data of interest and further analyzes it. For example, the average throughput associated with a specific link can be computed by extracting only the columns and fields associated with that link from the trace file. Two of the most popular languages that facilitate this process are AWK and Perl. The basic structures and usage of AWK is described in Appendix A.3.

Text-based packet tracing is activated by executing "`$ns trace-all $file`," where "`ns`" stores the Simulator handle and "`file`" stores a handle associated with the file which records the tracing strings. This statement simply informs NS2 of the need to trace packets. When an object is created, a tracing object is also created to collect the detail of traversing packets. Hence, the "`trace-all`" statement must be executed before object creation. We shall discuss the detail of text-based packet tracing later in Chap. 14.

Network AniMation (NAM) Trace

NAM trace records simulation detail in a text file and uses the text file to play back the simulation using animation. NAM trace is activated by the command "`$ns namtrace-all $file`," where "`ns`" is the Simulator handle and "`file`" is a handle associated with the file (e.g., "`out.nam`" in the above example) which stores the NAM trace information. After obtaining a NAM trace file, the animation can be initiated directly at the command prompt through the following command (See Line 11 in Program 2.1):

```
>>nam filename.nam
```

Many visualization features are available in NAM. These features are for example animating colored packet flows, dragging and dropping nodes (positioning), labeling nodes at a specified instant, shaping the nodes, coloring a specific link, and monitoring a queue.

2.7 Including C++ Modules into NS2 and the make Utility

In developing an NS2 simulation, very often it is necessary to create customized C++ modules to complement the existing libraries. As such, the developer is faced with the task of keeping track of all the created files as a part of NS2. When a change is made to one file, usually it requires recompilation of some other files that depend on it. Manual recompilation of each of such files may not be practical. In Unix, a utility tool called make is available to overcome such difficulties. In this section we introduce this tool and discuss how to use it in the context of NS2 simulation development.

As a Unix utility tool make is very useful for managing the development of software written in any compilable programming language including C++. Generally, the make program automatically keeps track of all the files created throughout the development process. By *keeping track*, we mean recompiling or relinking wherever interdependencies exist among these files, which may have been modified as a part of the development process.

2.7.1 An Invocation of a make Utility

A "make" utility can be invoked form a UNIX shell with the following command:

```
>>make [-f mydescriptor]
```

where "make" is mandatory, while the text inside the bracket is optional. By default (i.e., without optional input arguments), the "make" utility recompiles and *relinks* the source codes according to the default descriptor file "Makefile." If the descriptor file "mydescriptor" is specified, the utility uses this file in place of the default file "Makefile."

2.7.2 A make Descriptor File

A descriptor file contains instructions of how the source codes should be recompiled and relinked. Again, the default descriptor file is the file named "Makefile." A descriptor file contains the names of the source code files that make up the executable,

their interdependencies, and how each file should be rebuilt or recompiled. Such descriptions are specified through a series of so-called dependency rules. Each rule takes three components, i.e., targets, dependencies, and commands. The following is the format of the dependency rule:

```
<target1> [<target2> ...] : <dep1> [<dep2> ...]
   <command1> [<command2> ...]
```

where everything inside the brackets are optional. A target is usually the name of the file which needs to be *remade* if any modification is done to dependency files specified after the mandatory colon (`:`). If any change is noticed, the second and subsequent lines will be *executed* to regenerate the target file.

Example 2.1 (Example of a Descriptor File). Assume that we have a main executable file `channel` consisting of three separate source files named `main.c`, `fade.c`, and `model.c`. Also assume that `model.c` depends on `model.h`. The `Makefile` corresponding to this example is shown below.

```
# makefile of channel
channel : main.o fade.o model.o
    cc -o channel main.o fade.o model.o

main.o : main.c
    cc -c main.c

fade.o : fade.c
    cc -c fade.c

model.o : model.c model.h
    cc -c model.c

clean :
    rm main.o fade.o model.o
```

The first line is a comment beginning with a pound ("#") sign. When `make` is invoked, it starts checking the targets one by one. The target `channel` is examined first, and `make` finds that `channel` depends on the object files `main.o`, `fade.o`, and `model.o`. The `make` utility next checks to see if any of these object files is designated as a target file. If this is the case, `make` further checks the `main.o` object file's dependency and finds that it depends on `main.c`. Again, `make` proceeds to check whether `main.c` is listed as a target. If not, the command under the `main.o` target is executed if any change is made to `main.c`. In the command line "`cc -c main.c`,"[5] `main.c` is simply compiled

[5]The UNIX command "`cc -c file.c`" compiles the file "`file.c`" and creates an object file "`file.o`," while the command "`cc -o file.o`" links the object file "`file.o`" and create an executable file "`file`."

to obtain the main.o object. Next, make proceeds in a similar manner with the fade.o and model.o targets. Once any of these object files is updated, make returns to the channel target and executes its command, which merely compiles all of its dependent objects. Finally, we note a special target known as a *phony target* which is not really the name of any file in the dependency hierarchy. This target is "clean" and usually performs a housekeeping function such as cleaning up all the object files no longer needed after the compilation and linking. □

In Example 2.1, we notice several occurrences of certain sequences such as main.o fade.o model.o. To avoid a repetitive typing, which may introduce typos or omissions, a macro can be defined to represent such a long sequence. For example, we may define a macro to represent main.o fade.o model.o as follows:

```
OBJS = main.o fade.o model.o
```

After defining the macro, we refer to "main.o fade.o model.o" by either parentheses or curly brackets and precede that with a dollar sign (e.g., $(OBJS) or $OBJS). With this macro, Example 2.1 becomes a bit more handy as shown in Example 2.2.

Example 2.2 (Example of Makefile with Macros.). The Example 2.1 can be modified by defining macros as follows:

```
# makefile of channel
OBJS = main.o fade.o model.o
COM = cc
channel : ${OBJS}
    ${COM} -o channel ${OBJS}

main.o : main.c
    ${COM} -c main.c

fade.o : fade.c
    ${COM} -c fade.c

model.o : model.c model.h
    ${COM} -c model.c

clean :
    rm ${OBJS}
```

where ${COM} and ${OBJS} are used in the place of "cc" and "main.o fade.o model.o," respectively. □

2.7.3 NS2 Descriptor File

The NS2 descriptor file is defined in the file "Makefile" located in the home directory of NS2. It contains details needed to recompile and relink NS2. The key relevant details are those beginning with the following keywords.

- INCLUDES = : The items behind this keyword are the directory which should be *included* into the NS2 environment.
- OBJ_CC = and OBJ_STL = : The items behind these two keywords constitute the entire NS2 object files. When a new C++ module is developed, its corresponding object file name with ".o" extension should be added here.
- NS_TCL_LIB = : The items that bind these keywords are the Tcl file of NS2. Again, when a new OTcl module is developed, its corresponding Tcl file name should be added here.

Suppose a module consisting of C++ files "myc.cc" and "myc.h" and a Tcl file "mytcl.tcl." Suppose further that these files are created in a directory myfiles under the NS2 home directory. Then this module can be incorporated into NS2 using the following steps:

1. Include a string "-I./myfiles" into the Line beginning with INCLUDES = in the "Makefile."
2. Include a string "myfile/myc.o" into the Line beginning with OBJ_CC = or OBJ_STL = in the "Makefile."
3. Include a string "myfile/mytcl.tcl" into the Line beginning with NS_TCL_LIB = in the "Makefile."
4. Run "make" from the shell.

After running "make," an executable file "ns" is created. We can now use this file "ns" to run simulation.

2.8 Chapter Summary

This chapter introduces Network Simulator (Version 2), NS2. In particular, information on the installation of NS2 in both Unix and Windows-based systems is provided. The basic architecture of NS2 is described. These materials are essential for understanding NS2 as a whole and would help to get one started working with NS2.

NS2 consists of OTcl and C++. The C++ objects are mapped to OTcl handles using TclCl. To run a simulation, a user needs to define a network scenario in a Tcl Simulation script and feeds this script as an input to an executable file "ns." During the simulation, the packet flow information can be collected through text-based tracing or NAM tracing. After the simulation, an AWK program or a perl program can be used to analyze a text-based trace file. The NAM program, on the other hand, uses a NAM trace file to replay the network simulation using animation.

Simulation using NS2 consists of three main steps. First, the simulation design is probably the most important step. Here, we need to clearly specify the objectives and assumptions of the simulation. Second, configuring and running simulation implement the concept designed in the first step. This step also includes configuring the simulation scenario and running simulation. The final step in a simulation is to collect the simulation result and trace the simulation if necessary.

Written mainly in C++, NS2 uses a "`make`" utility to compile the source code, to link the created object files, and create an executable file "`ns`." It follows the instruction specified in the default descriptor file "`Makefile`." The "`make`" utility provides a simple way to incorporate a newly developed modules into NS2. After developing a C++ source code, we simply add an object file name into the dependency and rerun "`make`."

2.9 Exercises

1. Download and install NS2. Hint: When using Windows, install Cygwin first.
2. What are upstream objects, downstream objects, and targets? Draw a diagram to support your explanation.
3. Write a Tcl simulation script which prints the input arguments on screen. Format the output such that each line contains only one input argument. Run NS2 to test your program. Hint: See Tcl Tutorial in Appendix A.
4. What are the key steps in NS2 simulation? Compare your answer with general simulation steps discussed in Chap. 1.
5. Write a Tcl simulation script for the following network diagram in Fig. 2.6. Run NS2 to test your program.

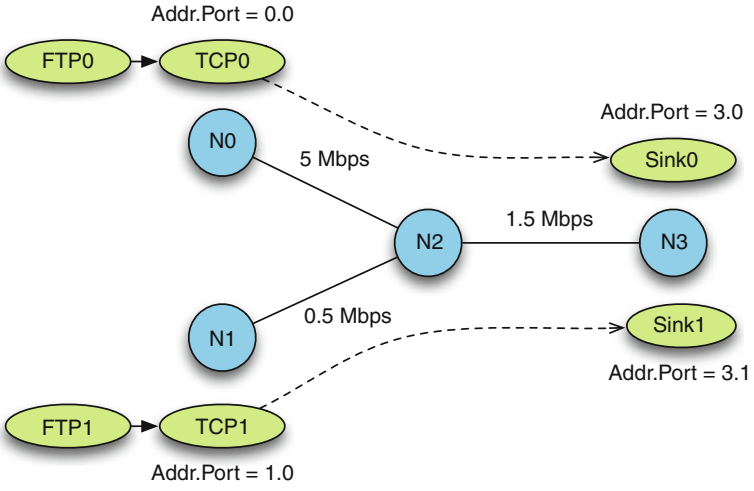

Fig. 2.6 Example network diagram

6. Design C++ and OTcl classes (e.g., Class `MyTCP`). Derive this class from the TCP Reno classes shown in Table 2.2. Use the convention defined above to name your class names, variables/instvars, and functions/instprocs in both the domains.
7. Write a program "`hello`," which prints "`Hello NS2 Users!`" on the screen

   a. Using C language,
   b. Using C++. Define at least one class,
   c. Using a `make` utility to create an executable file. Make changes in your C++ and/or header files. Run the `make` utility. Does the utility re-compile and recreate an executable file for your program?

8. Suppose you have developed a new NS2 module. Your module include C++ files, header files, and Tcl files. Where would you store these file? How would you include this new NS2 module into NS2? Demonstrate your answer with an example.

Chapter 3
Linkage Between OTcl and C++ in NS2

NS2 is an object-oriented simulator written in OTcl and C++ languages.[1] While OTcl acts as the frontend (i.e., user interface), C++ acts as the backend running the actual simulation (Fig. 2.1). From Fig. 3.1 class hierarchies of both languages can be either standalone or linked together using an OTcl/C++ interface called TclCL [17]. The OTcl and C++ classes which are linked together are referred to as *the interpreted hierarchy* and *the compiled hierarchy*, respectively.

Object construction in NS2 proceeds as follows. A programmer creates an object from an OTcl class in the interpreted hierarchy. Then, NS2 (or more precisely TclCL) automatically creates a so-called shadow object from a C++ class in the compiled hierarchy. It is important to note that no shadow object would be created when a programmer creates an object from a class in both compiled and standalone OTcl hierarchies.

Written in C++, TclCL consists of the following six main classes. First, class `TclClass` maps class names in the compiled hierarchy to class names in the interpreted hierarchy. Second, class `InstVar` binds member variables in both the hierarchies together. Third, class `TclCommand` allows the Tcl interpreter to execute non-OOP C++ statements. Fourth, class `TclObject` is the base class for all C++ simulation objects in the compiled hierarchy. Fifth, class `Tcl` provides methods to access the interpreted hierarchy from the compiled hierarchy. Finally, class `EmbeddedTcl` translates OTcl scripts into C++ codes. The details of the above classes are located in files *~tclcl/*tclcl.h, *~tclcl/*Tcl.cc, and *~tclcl/*tclAppInit.cc.

This chapter focuses on using TclCL in the following meaningful ways:

- Section 3.1 presents the motivation of having two languages in NS2.
- Section 3.2 explains class binding which maps C++ class names to OTcl class names.

[1] Refer to [16] for the C++ programming language.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_3, © Springer Science+Business Media, LLC 2012

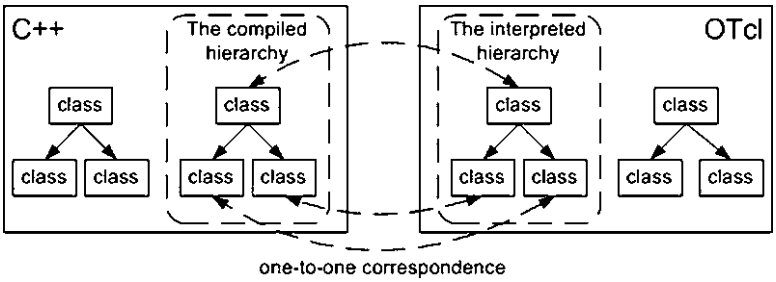

Fig. 3.1 Two language structure of NS2 [14]. Class hierarchies in both the languages may be standalone or linked together. OTcl and C++ class hierarchies which are linked together are called *the interpreted hierarchy* and *the compiled hierarchy*, respectively

- Section 3.3 discusses how NS2 binds a pair of member variables of two bound classes so that a change in one variable will be automatically reflected in the other.
- Section 3.4 shows a method to execute C++ statements from the OTcl domain.
- Section 3.5 walks through the shadow object construction process.
- Section 3.6 discusses various functionalities to access the Tcl interpreter from the C++ domain: Tcl statement execution, result passing between both the domains, and the TclObject reference retrieval.
- Section 3.7 briefly outlines how the OTcl codes are translated into the C++ code.

3.1 The Two-Language Concept in NS2

3.1.1 The Natures of OTcl and C++ Programming Languages

Why two languages? Loosely speaking, NS2 uses OTcl to create and configure a network (i.e., user frontend), and C++ to run simulation (i.e., internal mechanism). All C++ codes need to be compiled and linked to create an executable file. Since the body of NS2 is fairly large, the compilation time is not negligible. A typical Macbook Pro computer requires few seconds (long enough to annoy most programmers) to compile and link the codes with a small change such as including a C++ statement "`int i=0;`" into the program. OTcl, on the other hand, is an interpreted programming language, not a compiled one. Any change in a OTcl file can be executed without compilation. Since OTcl does not convert the codes into machine language, each line needs more execution time.[2]

[2]Although OTcl is an interpreted programming language, NS2 translates most of its OTcl codes into C++ using class `EmbeddedTcl` (see Sect. 3.7) to speed up the simulation. As a result, most change in OTcl also requires compilation.

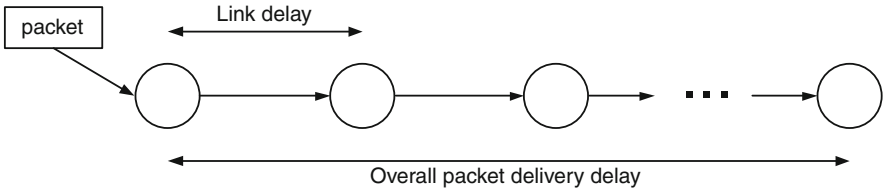

Fig. 3.2 A chain topology for network simulation

In summary, C++ is fast to run but slow to change. It is suitable for running a large simulation. OTcl, on the other hand, is slow to run but fast to change. It is suitable as a parameter configurator. NS2 is constructed by combining the advantages of these two languages.

3.1.2 C++ Programming Styles and Its Application in NS2

The motivation can be better understood by considering three following C++ programming styles.

3.1.2.1 Basic C++ Programming

This is the simplest form and involves basic C++ instructions only. This style has a flexibility problem, since any change in system parameters requires a compilation (which takes non-negligible time) of the entire program.

Example 3.1. Consider the network topology in Fig. 3.2. Define overall packet delivery delay as the time needed to carry a packet from the leftmost node to the rightmost node, where delay in link "`i`" is "`d_i`" and total number of nodes is "`num_nodes`." We would like to measure the overall packet delivery delay and show the result on the screen.

Suppose that every link has the same delay of 1 s (i.e., "`d_i = 1`" second for all "`i`"), and the number of nodes is 11 (`num_nodes = 11`). Program 3.1 shows a C++ program written in this style (the filename is "`sim.cc`"). Since the link delay is fixed, we simply increase "`delay`" for `num_nodes-1` times (Lines 4 and 5). After compiling and linking the file `sim.cc`, we obtain an executable file "`sim`." By executing "`./sim`" at the command prompt, we will see the following result on the screen:

```
>>./sim
Overall Packet Delay is 10.0 seconds.
```

Despite its simplicity, this programming style has a flexibility problem. Suppose link delay and the number of nodes are changed to 2 s and 5 nodes, respectively.

Program 3.1 A basic C++ program that simulates Example 3.1, where the delay for each of the links is 1 unit and the number of nodes is 11

```
    //sim.cc
1   main(){
2       float delay = 0, d_i = 1;
3       int i, num_nodes = 11;
4       for(i = 1; i < num_nodes; i++)
5           delay += d_i;
6       printf("Overall Packet Delay is %2.1f seconds.\n",
    delay);
7   }
```

Then, we need to modify, compile, and link the file `sim.cc` to create a new executable file "`sim`." After that, we can run "`./sim`" to generate another result (for `d_i` = 2 and `num_nodes` = 5). □

3.1.2.2 C++ Programming with Input Arguments

Addressing the flexibility problem, this programming style takes the system parameters (i.e., `argv, argc`) as input arguments [16]. As the system parameters change, we can simply change the input arguments, and do not need to recompile the entire program.

Example 3.2. Consider Example 3.1. We can avoid the above need for recompilation and relinking by feeding system parameters as input arguments of the program. Program 3.2 shows a program that feeds link delay and the number of nodes as the first and the second arguments, respectively. Line 1 specifies that the program takes input arguments. The variable `argc` is the number of input arguments. The variable `argv` is an argument vector that contains all input arguments provided by the caller (See the details on C++ programming with input arguments in [16]).

With this style, we only need to compile and link the program once. After obtaining an executable file "`sim`," we can change the simulation parameters as desired. For example,

```
>> ./sim 1 11
Overall Packet Delay is 10.0 seconds.
>> ./sim 2 5
Overall Packet Delay is 8.0 seconds.
```

Although this programming style solves the flexibility problem, it becomes increasingly inconvenient when the number of input arguments increases. For example, if delays in all the links in Example 3.1 are different, we will have to type in all the values of link delay every time we run the program. □

Program 3.2 A C++ program with input arguments: A C++ program which simulate Example 3.2. The first and second arguments are link delay and the number of nodes, respectively

```
    //sim.cc
1   int main(int argc, char* argv[]) {
2       float delay = 0, d_i = atof(argv[0]);
3       int i, num_nodes = atoi(argv[1]);
4       for(i = 1; i < num_nodes; i++)
5           delay += d_i;
6       printf("Overall Packet Delay is %2.1f seconds\n",
    delay);
7   }
```

3.1.2.3 C++ Programming with Configuration Files

This last programming style puts all system parameters in a configuration file, and the C++ program reads the system parameters from the configuration files. This style does not have the flexibility problem, and it facilitates program invocation. To change system parameters, we can simply change the content of the configuration file. In fact, this is the style from which NS2 develops.

Example 3.3. Program 3.3 applies the last C++ programming style (i.e., with configuration files). The program takes only one input argument: The configuration file name (See C++ file input/output in [16]). Function `readArgFromFile(fp,d)` reads the configuration file associated with a file pointer "`fp`," and sets variables "`num_node`" and "`d`" accordingly (the details are not shown here). In this case, the configuration file (`config.txt`) is shown in Lines 10 and 11. When invoking "`./sim config.txt`," the screen will show the following result:

```
>>./sim config.txt
Overall Packet Delay is 55.0 seconds.
```

To change the system parameters, we can simply modify the file "`config.txt`" and run "`./sim`." Clearly, this programming style removes the necessity for compiling the entire program and the lengthy invocation process. □

Recall from Sect. 2.5 that we write and feed a *Tcl simulation script* as an input argument to NS2 when running a simulation (e.g., executing "ns `myfirst_ns.tcl`"). Here, "`ns`" is a C++ executable file obtained from the compilation, while "`myfirst_ns.tcl`" is an input configuration file specifying system parameters and configuration such as nodes, link, and how they are connected. Analogous to reading a configuration file through C++, NS2 reads the system configuration from the Tcl simulation script. When we would like to change the parameters in the simulation, all we have to do is to modify the Tcl simulation script and rerun the simulation.

Program 3.3 C++ programming style with configuration files: A C++ program in file `sim.cc` (Lines 1–9) and a configuration file `config.txt` (Lines 10 and 11)

```
    //sim.cc
1   int main(int argc, char* argv[]) {
2       float delay = 0, d[10];
3       FILE* fp = fopen(argv[1],"w");
4       int i, num_nodes = readArgFromFile(fp,d);
5       for(i = 1; i < num_nodes; i++)
6           delay += d[i-1];
7       printf("Overall Packet Delay is %2.1f seconds\n",
    delay);
8       fclose(fp);
9   }

    //config.txt
10  Number of node = 11
11  Link delay = 1 2 3 4 5 6 7 8 9 10
```

3.2 Class Binding

Class binding maps C++ classes to OTcl classes. When a programmer creates an object from the interpreted hierarchy, NS2 determines the compiled class from which a shadow object should be instantiated by looking up the class binding map. As an example, an OTcl class `Agent/Tcp` is bound to the C++ class `TcpAgent`. A programmer can create an `Agent/TCP` object using the following OTcl statement

```
new Agent/TCP
```

In response, NS2 automatically creates a compiled shadow `TcpAgent` object.

3.2.1 Class Binding Process

A class binding process involves four following components:

- A C++ class (e.g., class `TcpAgent`)
- An OTcl class (e.g., class `Agent/TCP`)
- A mapping class (e.g., class `TcpClass`): A C++ class which maps a C++ class to an OTcl class
- A mapping variable (e.g., `class_tcp`): A static variable instantiated from the above mapping class to perform class binding functionalities.

Class binding is carried out by defining the mapping class as a part of the C++ file. Program 3.4 shows a mapping class `TcpClass` that binds the OTcl class `Agent/TCP` to the C++ class `TcpAgent`.

Program 3.4 Class `TcpClass` which binds the OTcl `Agent/TCP` to the C++ class `TcpAgent`

```
    //~ns/tcp/tcp.cc
1   static class TcpClass : public TclClass {
2   public:
3       TcpClass() : TclClass("Agent/TCP") {}
4       TclObject* create(int , const char*const*) {
5           return (new TcpAgent());
6       }
7   } class_tcp;
```

Every mapping class derives from the C++ class `TclClass` where all the binding functionalities are defined. Each mapping class contains only two functions: The constructor and function `create(...)`. In Line 3, the constructor feeds the OTcl class name `Agent/TCP` as an input argument to the constructor of its base class (i.e., `TclClass`).[3] In Lines 4–6, the function `create(...)` creates a shadow compiled object whose class is `TcpAgent`. This function is automatically executed when a programmer creates an `Agent/TCP` object from the interpreted hierarchy. We shall discuss the details of the shadow object construction process later in Sect. 3.5.

Note that a C++ class by itself cannot perform any operation. To bind classes, we need to instantiate a mapping object from the mapping class. In Line 7, such an object is the variable `class_tcp`. Since every class binding is unique, the mapping variable `class_tcp` is declared as `static` to avoid class binding duplication.

3.2.2 Defining Your Own Class Binding

The following steps bind an OTcl class to a C++ class:

1. Specify an OTcl class (e.g., `Agent/TCP`) and a C++ class (e.g., `TcpAgent`) which shall be bound together.
2. Derive a mapping class (e.g., `TcpClass`) from class `TclClass`.
3. Define the constructor of the mapping class (e.g., Line 3 in Program 3.4). Feed the OTcl class name (e.g., `Agent/TCP`) as an input argument to the constructor of the class `TclClass` (i.e., the base class).
4. Define function `create(...)` to construct a shadow compiled object. Invoke "`new`" to create a shadow compiled object and return the created object to the caller (e.g., `return (new TcpAgent())` in Line 5 of Program 3.4).
5. Declare a static mapping variable (e.g., `class_tcp`).

[3] A C++ operator "`:`" indicates what to be done before the execution of what is enclosed within the following curly braces [16].

Table 3.1 Examples of naming convention for mapping classes and variables

C++ class	OTcl class	Mapping class	Mapping variable
`TcpAgent`	`Agent/TCP`	`TcpClass`	`class_tcp`
`RenoTcpAgent`	`Agent/TCP/Reno`	`RenoTcpClass`	`class_reno`
`DropTail`	`Queue/DropTail`	`DropTailClass`	`class_drop_tail`

3.2.3 *Naming Convention for Class `TclClass`*

The convention to name mapping classes and mapping variables are as follows. First, every class derives directly from class `TclClass`, irrespective of its class hierarchy. For example, class `RenoTcpAgent` derives from class `TcpAgent`. However, their mapping classes `RenoTcpClass` and `TcpClass` derive from class `TclClass`.

Second, the naming convention is very similar to the C++ variable naming convention. In most cases, we simply name the mapping class by attaching the word "`Class`" to the C++ class name. Mapping variables are named with the prefix "`class_`" attached to the front. Table 3.1 shows few examples of the naming convention.

3.3 Variable Binding

Class binding, discussed in the previous section, creates connections between OTcl and C++ class names. By default, the bound classes have their own variables which are not related in any way. Variable binding is a tool that allows programmers to bind one variable in the C++ class to another variable in the bound OTcl class such that a change in one variable will be automatically reflected in the other.

3.3.1 *Variable Binding Methodology*

An OTcl variable, `iname`, can be bound to a C++ variable, `cname`, by including one of the following statements in the C++ class constructor:

- `bind("iname",&cname)` binds integer and real variables.[4]
- `bind_bw("iname",&cname)` binds a bandwidth variable.
- `bind_time("iname",&cname)` bind a time variable.
- `bind_bool("iname",&cname)` bind a boolean variable.

[4] In Sect. 3.3.3, we shall discuss the five following data types in NS2: Integer, real, bandwidth, time, and boolean.

Example 3.4. Let a C++ class `MyObject` be bound to an OTcl class `MyOTclObject`. Let `icount_`, `idelay_`, `ispeed_`, `idown_time_`, `iis_active_` be OTcl class variables whose types are integer, real, bandwidth, time, and boolean, respectively. The following C++ program binds the above variables:

```
class MyObject {
    public:
    int count_;
    double delay_,down_time_,speed_;
    bool is_active_;
    MyObject() {
        bind("icount_",&count_);
        bind("idelay_",&delay_);
        bind_bw("ispeed_",&speed_);
        bind_time("idown_time_",&down_time_);
        bind_bool("iis_active_",&is_active_);
    };
};
```

□

3.3.2 *Setting the Default Values*

NS2 sets the default values of bound OTcl class variables in the file *~ns*/tcl/lib/ns-default.tcl. The syntax for setting a default value is similar to the value assignment syntax. That is,

```
<className> set <instvar> <def_value>
```

which sets the default value of the instvar `<instvar>` of class `<className>` to be `<def_value>`. As an example, a part of file *~ns*/tcl/lib/ns-default.tcl is shown in Program 3.5.

In regards to default values, there are two important notes here. First, if no default value is provided for a bound variable, the instproc `warn-instvar {...}` of class `SplitObject` will show the following a warning message on the screen:

```
warning: no class variable <C++ class name>::<OTcl
variable name> see tcl-object.tcl in tclcl for
info about this warning.
```

The second note is that the default value setting in the OTcl domain takes precedence over that in the C++ domain. In C++, the default values are usually set in the constructor. But the values would be overwritten by those specified in the file *~ns*/tcl/lib/ns-default.tcl.

Program 3.5 Examples for default value assignment

```
    //~ns/tcl/lib/ns-default.tcl
1   ErrorModel set enable_ 1
2   ErrorModel set markecn_ false
3   ErrorModel set delay_pkt_ false
4   ErrorModel set delay_ 0
5   ErrorModel set rate_ 0
6   ErrorModel set bandwidth_ 2Mb
7   ErrorModel set debug_ false
    ...
8   Classifier set offset_ 0
9   Classifier set shift_ 0
10  Classifier set mask_ 0xffffffff
11  Classifier set debug_ false
```

3.3.3 NS2 Data Types

NS2 defines the following five data types in the OTcl domain: real, integer, bandwidth, time, and boolean.

3.3.3.1 Real and Integer Variables

These two NS2 data types are specified as `double`-valued and `int`-valued, respectively, in the C++ domain. In the OTcl domain, we can use "`e<x>`" as "$\times 10^{\texttt{<x>}}$", where `<x>` denotes the value stored in the variable `x`.

Example 3.5. Let `realvar` and `intvar` be a real instvar and an integer instvar, respectively, of an OTcl object "`obj`". The following shows various ways to set[5] `realvar` and `intvar` to be 1200:

```
$obj set realvar 1.2e3
$obj set realvar 1200
$obj set intvar 1200
```

□

3.3.3.2 Bandwidth

Bandwidth is specified as `double`-valued in the C++ domain. By default, the unit of bandwidth is bits per second (bps). In the OTcl domain we can add the following suffixes to facilitate bandwidth setting.

[5] See the OTcl value assignment in Appendix A.2.4.

- "k" or "K" means kilo or $\times 10^3$,
- "m" or "M" means mega or $\times 10^6$, and
- "B" changes the unit from bits to bytes.

NS2 only considers leading character of valid suffixes. Therefore, the suffixes "M" and "Mbps" are the same to NS2.

Example 3.6. Let bwvar be a bandwidth instvar of an OTcl object "obj." The different ways to set bwvar to be 8 Mbps (megabits per second) are as follows:

```
$obj set bwvar 8000000
$obj set bwvar 8m
$obj set bwvar 8Mbps
$obj set bwvar 8000k
$obj set bwvar 1MB
```
□

3.3.3.3 Time

Time is specified as double-valued in the C++ domain. By default, the unit of time is second. Optionally, we can add the following suffixes to change the unit.

- "m" means milli or $\times 10^{-3}$,
- "n" means nano or $\times 10^{-9}$, and
- "p" means pico or $\times 10^{-12}$.

NS2 only reads the leading character of valid suffixes. Therefore, the suffixes "p" and "ps" are the same to NS2.

Example 3.7. Let timevar also be a time instvar of an OTcl object "obj." The different ways to set timevar to 2 ms are as follows:

```
$obj set timevar 2m
$obj set timevar 2e-3
$obj set timevar 2e6n
$obj set timevar 2e9ps
```
□

3.3.3.4 Boolean

Boolean is specified as either true (or a positive number) or false (or a zero) in the C++ domain. A boolean variable will be true if the first letter of the value is greater than 0, or "t," or "T." Otherwise, the variable will be false.

Example 3.8. Let boolvar be a boolean instvar of an OTcl object "obj." The different ways to set boolvar to be true and false are as follows:

```
# set boolvar to be TRUE
$obj set boolvar 1
```

```
$obj set boolvar T
$obj set boolvar true
$obj set boolvar tasty
$obj set boolvar 20
$obj set boolvar 3.37
$obj set boolvar 4xxx

# set boolvar to be FALSE
$obj set boolvar 0
$obj set boolvar f
$obj set boolvar false
$obj set boolvar something
$obj set boolvar 0.9
$obj set boolvar -5.29
```

□

Again, NS2 ignores all letters except for the first one. As can be seen from Example 3.8, there are several strange ways for setting a boolean variable (e.g., `tasty`, `something`, `-5.29`).

Example 3.9. The following program segment shows how NS2 performs value assignment to instvars `debug_` and `rate_` of class `ErrorModel`.

```
# Create a Simulator instance
set ns [new Simulator]

# Create an error model object
set err [new ErrorModel]

# Set values for class variables
$err set debug_ something
$err set rate_ 12e3

# Show the results
puts "debug_(bool) is [$err set debug_]"
puts "rate_(double) is [$err set rate_]"
```

The results of execution of the above program are as follows:

```
>>debug_(bool) is 0
>>rate_(double) is 12000
```

□

During the conversion, the parameter suffixes are converted (e.g., "`M`" is converted by multiplying 10^6 to the value). For boolean data type, NS2 retrieves the first character in the string and throws away all other characters. If the retrieved character is a positive integer, "`t`," or "`T`," NS2 will assign a `true` value to the bound C++ variable. Otherwise, the variable will be set to `false`.

Table 3.2 NS2 data types, C++ mapping classes, and the corresponding C++ data types

NS2 data type	C++ mapping class	C++ data type
Integer	`InstVarInt`	`int`
Real	`InstVarReal`	`double`
Bandwidth	`InstVarBandwidth`	`double`
Time	`InstVarTime`	`double`
Boolean	`InstVarBool`	`bool`

3.3.4 Class Instvar

Class `Instvar` is a C++ class which binds member variables of OTcl and C++ classes together. It has five derived classes, each for one of the NS2 data types defined in Sect. 3.3.3. These five classes and their mapping class are shown in Table 3.2.

3.4 Execution of C++ Statements from the OTcl Domain

This section focuses on "method binding," which makes an *OTcl commands* available in the C++ domain. There are two types of method binding: OOP binding and non-OOP binding. The OOP binding, binds the method using the C++ function `command(...)` associated with a C++ class. In this case, the command string is called an "OTcl command." For non-OOP binding, the string – called "Tcl command" – is not bound to any class, and can be executed globally. It is not advisable to extensively use Tcl commands, since they violate the OOP principle.

3.4.1 OTcl Commands in a Nutshell

3.4.1.1 OTcl Command Invocation

The invocation of an OTcl command is similar to that of an instproc:

```
$obj <cmd_name> [<args>]
```

where `$obj` is a TclObject, `<cmd_name>` is the OTcl command string associated with `$obj`, and `<args>` is an optional list of input arguments.

Program 3.6 Function `command` of class `TcpAgent`

```
    //~ns/tcp/tcp.cc
1   int TcpAgent::command(int argc, const char*const* argv)
2   {
3       ...
4       if (argc == 3) {
5           if (strcmp(argv[1], "eventtrace") == 0) {
6               et_ = (EventTrace *)TclObject::lookup(argv[2]);
7               return (TCL_OK);
8           }
9           ...
10      }
11      ...
12      return (Agent::command(argc, argv));
13  }
```

3.4.1.2 C++ Definition of OTcl Commands

OTcl commands are defined in the function `command(argc,argv)` of C++ classes in the compiled hierarchy. This function takes two input parameters: "`argc`" and "`argv`," which are the number of input parameters and an array containing the input parameters, respectively.

Example 3.10. Consider an OTcl command `eventtrace{et}` associated with an OTcl class `Agent/TCP`, where "`et`" is an event tracing object. Program 3.6 shows the details of this OTcl command. Suppose `$tcp` and `$obj` are an `Agent/TCP` object and an event tracing object, respectively. The execution of OTcl command `eventtrace` is as follows:

```
$tcp eventtrace $obj
```

The OTcl command `eventtrace` and the event tracing object `$obj` are stored in `argv[1]` and `argv[2]`, respectively.[6] Line 5 returns `true`, and Line 6 stores the input parameter `argv[2]` in the class variable `et_`. □

3.4.1.3 Creating Your Own OTcl Commands

Here are the main steps in defining an OTcl command:

1. Pick a name, the number of input arguments, and the OTcl class for an OTcl command.

[6] As we shall see, `argv[0]` always contains the string "`cmd`."

2. Define a C++ function `command(argc,argv)` for the shadow C++ class in the compiled hierarchy.
3. Within the function `command(...)`, create two conditions which compare the number of input of arguments with `argc` and the name of the OTcl command with `argv[1]`. Specify the desired C++ statements, if both the conditions match. Return `TCL_OK` after the C++ statement execution.
4. Specify the default return statement in the case that no criterion matches with the input OTcl command. For example, Line 12 in Program 3.6 executes the function `command(...)` attributed to the base class `Agent`, and returns the execution result to the caller.

3.4.2 The Internal Mechanism of OTcl Commands

3.4.2.1 OTcl Command Invocation Mechanism

As discussed in Sect. 3.4.1, the syntax for the OTcl command invocation is similar to that of instproc invocation. Therefore, the internal process is similar to the instproc invocation mechanism discussed in Appendix A.2.4.

The process begins by executing the following OTcl statement:

```
$obj <cmd_name> [<args>]
```

If the class corresponding to `$obj` contains, either the instproc `<cmd_name>` or the instproc "`unknown`", it will execute the instproc. Otherwise, the process would move to the base class and repeat itself. In case of OTcl commands, the top-level class is class `SplitObject`. The function `unknown` of class `SplitObject` is specified in Program 3.7. The key statement in this instproc is "`$self cmd $args`" in Line 2, which according to the above OTcl command syntax can be written as follows:

```
$obj cmd <cmd_name> <args>
```

Program 3.7 Instproc `unknown` of class `SplitObject`

```
   //~tcl/tcl-object.tcl
1  SplitObject instproc unknown args {
2      if [catch "$self cmd $args" ret] {
3          set cls [$self info class]
4          global errorInfo
5          set savedInfo $errorInfo
6          error "error when calling class $cls: $args" $
           savedInfo
7      }
8      return $ret
9  }
```

Table 3.3 Description of elements of array "argv" of function command

Index (i)	Element (argv[i])
0	cmd
1	The command name (<cmd_name>)
2	The first input argument in <args>
3	The second input argument in <args>
⋮	⋮

The string "cmd" is the gateway to the C++ domain. Here, the string "cmd $args" (i.e., "cmd <cmd_name> [<args>]") is passed as an input argument vector (argv) to the function "command(argc,argv)" of the shadow class (eg., TcpAgent) as specified in Table 3.3.[7]

Next, the function command(argc,argv) compares the number of arguments and the OTcl command name with argc and argv[1], respectively. If both match, it takes the desired actions and returns TCL_OK (e.g., see Program 3.6).

3.4.2.2 OTcl Default Returning Structure

Owing to its OOP nature, NS2 allows OTcl commands to propagate up the hierarchy. That is, an OTcl command of a certain OTcl class can be specified in the function command(...) of the shadow class or in the function command() of any of its parent classes. If the input OTcl command (i.e., argv[1]) does not match with the string specified in the shadow class, function command(...) will skip to execute the default returning statement in the last line (e.g., Line 12 in Program 3.6).

The default returning statement first passes the same set of input arguments (i.e., (argc,argv)) to the function command(...) of the base class. Therefore, the same process of comparing the OTcl command and C++ statement execution will repeat in the base class. If the OTcl command does not match, the default returning process will be carried out recursively, until the top-level compiled class (i.e., class TclObject) is reached. Here, the function command(...) of class TclObject will report an error (e.g., no such method, requires additional args) and return TCL_ERROR (see file *~tclcl*/Tcl.cc).

3.4.2.3 Interpretation of the Returned Values

In file nsallinone-2.35/tcl8.5.8/generic/tcl.h, NS2 defines five following return values (as 0–5 in Program 3.8), which inform the interpreter of the OTcl command invocation result.

[7] Here, the argc is the number of nonempty element in argv.

Program 3.8 Return values in NS2

```
    //nsallinone-2.35/tcl8.5.8/generic/tcl.h
1   #define TCL_OK          0
2   #define TCL_ERROR       1
3   #define TCL_RETURN      2
4   #define TCL_BREAK       3
5   #define TCL_CONTINUE    4
```

- `TCL_OK`: The command completes successfully.
- `TCL_ERROR`: The command does not complete successfully. The interpreter will explain the reason for the error.
- `TCL_RETURN`: After returning from C++, the interpreter will exit (or return from) the current instproc without performing the rest of instproc.
- `TCL_BREAK`: After returning from C++, the interpreter will break the current loop. This is similar to executing the C++ keyword `break`, but the results prevail to the OTcl domain.
- `TCL_CONTINUE`: After returning from C++, the interpreter will immediately restart the loop. This is similar to executing the C++ keyword `continue`, but the results prevail to the OTcl domain.

Among these five types, `TCL_OK` and `TCL_ERROR` are the most common ones. If C++ returns `TCL_OK`, the interpreter may read the value passed from the C++ domain (see Sect. 3.6.3).

If an OTcl command returns `TCL_ERROR`, on the other hand, the interpreter will invoke procedure `tkerror` (defined in file `~tclcl/tcl-object.tcl`), which shows an error on the screen, and exits the program.

Example 3.11. Consider invocation of an OTcl command associated with an `Agent/TCP` object, `$tcp`. The process proceeds as follows (see also Fig. 3.3):

1. Execute an OTcl statement "`$tcp <cmd_name> <args>`" (position `(1)` in Fig. 3.3).
2. Look for an instproc `<cmd_name>` in the OTcl class `Agent/TCP`. If found, execute the instproc and complete the process. Otherwise, proceed to the next step.
3. Look for an instproc `unknown{...}` in the OTcl class `Agent/TCP`. If found, execute the instproc `unknown{...}` and complete the process. Otherwise, proceed to the next step.
4. Repeat steps (2) and (3) up the hierarchy until reaching the top level class in the interpreted hierarchy (i.e., `SplitObject`).
5. The main statement of the instproc `unknown{...}` of class `SpiltObject` in Program 3.7 is "`$self cmd $args`" in Line 2, which interpolates to

```
$tcp cmd <cmd_name> [<cmd_args>]
```

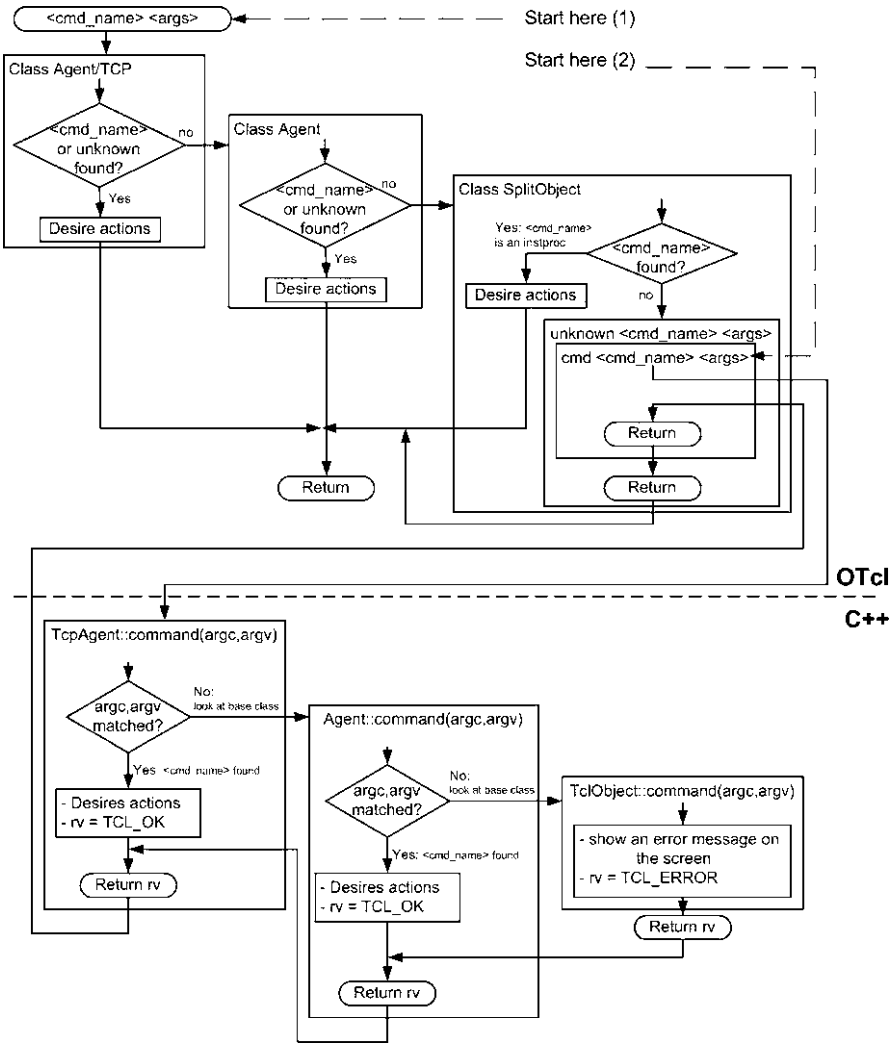

Fig. 3.3 The internal process of OTcl command invocation

6. Execute the function `command(argc,argv)` of class `TcpAgent`, where `argv[0]` and `argv[1]` are `cmd` and `<cmd_name>`, respectively.
7. Look for the matching number of arguments and OTcl command name. If found, execute the desired actions (e.g., Lines 6 and 7 in Program 3.6) and return `TCL_OK`.

8. If no criterion matches with (`argc`, `argv`), skip to the default returning statement (e.g., Line 12 in Program 3.6), moving up the hierarchy and executing the function `command(...)`.
9. Repeat steps (6)–(8) up the compiled hierarchy until the criterion is matched. Regardless of (`argc`,`argv`), the top-level class `TclObject` in the compiled hierarchy set the return value to be `Tcl-Error`, indicating that no criteria along the entire hierarchical tree matches with (`argc`,`argv`).
10. Return down the compiled hierarchy. When reaching C++ class `TcpAgent`, return to the OTcl domain with a return value (e.g., `TCL_OK` or `TCL_ERROR`). Move down the interpreted hierarchy and carry the returned value to the caller. □

3.4.3 An Alternative for OTcl Command Invocation

In general, we invoke an OTcl command by executing

```
$obj <cmd_name> <args>
```

which starts from position `(1)` in Fig. 3.3. Alternatively, we can also invoke a the command using the following syntax:

```
$obj cmd <cmd_name> <args>
```

which starts from position `(2)` in Fig. 3.3. This method explicitly tells NS2 that `<cmd_name>` is an OTcl command, not an instproc. This helps avoid the ambiguity when OTcl defines an instproc whose name is the same as an OTcl command name.

3.4.4 Non-OOP Tcl Command

Non-OOP Tcl commands are very similar to OOP Tcl command (i.e.) *OTcl commands*, discussed in the previous section. However, non-OOP Tcl command, also known as *Tcl commands*, are not bound to any class.

3.4.5 Invoking a TclCommand

A TclCommand can be invoked as if it is a global Tcl procedure. Consider the TclCommands `ns-version` and `ns-random`, specified in file *~ns*/common/misc.cc.

- TclCommand `ns-version` takes no argument and returns the NS2 version.
- TclCommand `ns-random` returns a random number uniformly distributed in $[0, 2^{31} - 1]$ when no argument is specified. If an input argument is given, it will be used to set the seed of the random number generator.

Program 3.9 Declaration and function `command(...)` of class `Random-Command`

```
     //~ns/common/misc.cc
  1  class RandomCommand : public TclCommand {
  2  public:
  3      RandomCommand() : TclCommand("ns-random") { }
  4      virtual int command(int argc, const char*const* argv);
  5  };

  6  int RandomCommand::command(int argc, const char*const* argv)
  7  {
  8      Tcl& tcl = Tcl::instance();
  9      if (argc == 1) {
  10         sprintf(tcl.buffer(), "%u", Random::random());
  11         tcl.result(tcl.buffer());
  12     } else if (argc == 2) {
  13         int seed = atoi(argv[1]);
  14         if (seed == 0)
  15             seed = Random::seed_heuristically();
  16         else
  17             Random::seed(seed);
  18         tcl.resultf("%d", seed);
  19     }
  20     return (TCL_OK);
  21 }
```

These two TclCommands can be invoked globally. For example,

```
>>ns-version
2.34
>>ns-random
729236
>>ns-random
1193744747
```

By executing `ns-version`, the version (2.34) of NS2 is shown on the screen. TclCommand `ns-random` with no argument returns a random number.

3.4.5.1 Creating a TclCommand

A TclCommand creation process consists of declaration and implementation. The declaration is similar to that of a TclClass. A TclCommand is declared as a derived class of class `TclCommand`. The name of a TclCommand is provided as an input argument of class `TclCommand` (see Line 3 in Program 3.9).

Similar to that of OTcl commands, the implementation of Tcl commands is defined in function `command(argc,argv)` as shown in Lines 6–21 of Program 3.9. Here, we only need to compare the number of input arguments (e.g., Line 9), since the name of the Tcl command was declared earlier.

Program 3.10 Function `misc_init`, which instantiates of TclCommands

```
    //~ns/common/misc.cc
1   void init_misc(void)
2   {
3       (void)new VersionCommand;
4       (void)new RandomCommand;
5       ...
6   }
```

In addition to the above declaration and implementation, we need to specify active TclCommands in the function `init_misc()` as shown in Program 3.10. Here, each active TclCommands is instantiated by the C++ statement "`(void) new <TclCommand>`." At the startup time, NS2 invokes the function `init_misc(...)` from within the file *~tclcl*/tclAppInit.cc to instantiate all active TclCommands.

3.4.5.2 Defining Your Own TclCommand

To create a TclCommand, you need to

1. Pick a name, the number of input arguments, and the class name for your TclCommand.
2. Derive a TclCommand class directly from class `TclCommand`,
3. Feed the Tcl command name to the constructor of class `TclCommand`,
4. Provide implementation (i.e., desired actions) in the function `command(...)`, and
5. Add an object instantiation statement in the function `init_misc(...)`.

Example 3.12. Let the TclCommand `print-all-args` show all input arguments on the screen. We can implement this TclCommand by including the following codes to the file *~ns*/common/misc.cc:

```
class PrintAllArgsCommand : public TclCommand {
    public:
    PrintAllArgsCommand():TclCommand("print-all-args")
    {};
    int command(int argc, const char*const* argv);
}

int PrintAllArgsCommand::command(int argc,
                        const char*const* argv) {
    cout << "Input arguments: "
    for (int i = 1; i < argc; i++) {
        count << argv[i];
    }
```

```
    return (TCL_OK);
}

void init_misc(void)
{   ...
    (void)new PrintAllArgsCommand;
    ...
}
```

□

3.5 Shadow Object Construction Process

NS2 automatically constructs a shadow compiled object when an OTcl object is created from the interpreted hierarchy. This section demonstrates the shadow object construction process. The process is defined in the top level classes in both the hierarchies – namely classes `TclObject` and `SplitObject`. However, throughout this book, we shall refer to the objects instantiated from these two hierarchies as TclObjects. The term SplitObject shall be used when a clear differentiation for both the hierarchies is needed.

3.5.1 A Handle of a TclObject

A handle is a reference to an object. As a compiler, C++ directly accesses the memory space allocated for a certain object (e.g., `0xd6f9c0`). A handler in C++ is a pointer or a reference variable to the object. OTcl, on the other hand, uses a string (e.g., `_o10`) as a reference to the object. By convention, the name string of a SplitObject is of format `_<NNN>`, where `<NNN>` is a number uniquely generated for each SplitObject.

Example 3.13. Let variables `c_obj` and `otcl_obj` contains C++ and OTcl objects, respectively. Table 3.4 shows examples of the reference value of C++ and OTcl objects.

Table 3.4 Examples of reference to (or handle of) TclObjects

Domain	Variable name	Handle
C++	`c_object`	`0xd6f9c0`
OTcl	`otcl_object`	`_o10`

We can see the value of an OTcl object stored in an OTcl variable by running the following codes:

```
//test.tcl
set ns [new Simulator]
set tcp [new Agent/TCP]
puts "The value of tcp is $tcp"
```

which show the following line on the screen:

```
>>ns test.tcl
The value of tcp is _o10
```

□

3.5.2 *TclObjects Construction Process*

In general, an OTcl object can be created and stored in a variable `$var` using the following syntax:

```
<classname> create $var [<args>]
```

where `<classname>` (mandatory) and `<args>` (optional) are the class name and the list of input arguments for the class constructor, respectively.

This general OTcl object construction approach is not widely used in NS2, since it does not create shadow objects. NS2 uses the following statement to create an object from an interpreted hierarchy:

```
new <classname> <args>
```

This section focuses on how the global procedure "`new`" automatically creates a shadow object. We shall use the OTcl class `Agent/TCP`, bound to the C++ class `TcpAgent` in the C++ compiled hierarchy, as an example to facilitate the explanation.

The TclObject construction process consists of two main parts:

Part I [OTcl Domain]: SplitObject Construction

The main steps in this part are to execute the following OTcl statements and instprocs in sequence (see also Fig. 3.4):

I.1. The OTcl statement `new <classname> [<args>]`
I.2. The OTcl statement `$classname create $o $args`
I.3. The instproc `alloc` of the OTcl class `<classname>`
I.4. The instproc `init` of the OTcl class `<classname>`
I.5. The instproc `init` of the OTcl class `SplitObject`
I.6. The OTcl statement `$self create-shadow $args`

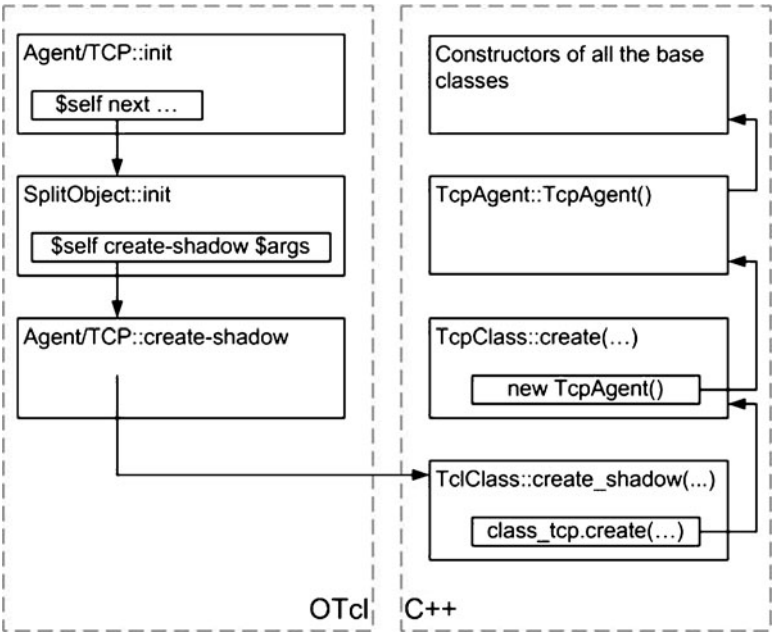

Fig. 3.4 An example of the shadow object construction process: Main steps in the constructor of class `Agent/TCP`

After these steps are complete, the constructed TclObject are returned to the caller. In most cases, the returned object is stored in a local variable (e.g., `$var`).

The details of the above six main steps are as follows. Consider Program 3.11 for the global procedure "`new{className args}`" (Step I.1). Line 2 retrieves the reference string for a SplitObject using the instproc `getid{}` of class `SplitObject`. The string is then stored in the variable "`$o`." Line 3 creates an object whose OTcl class is `$className` and associates the created object with the string stored in "`$o`" (Step I.2). Finally, if the object is successfully created, Line 11 returns the reference string "`$o`" to the caller.[8] Otherwise, an error message (Line 9) will be shown on the screen.

As discussed in Sect. A.2.4, the instproc `create` in Line 3 invokes the instproc `alloc{...}` (Step I.3) to allocate a memory space for an object of class `className`, and the instproc `init{...}` (Step I.4) to initialize the object.

The final two steps are explained through class `Agent/TCP`. Program 3.12 shows the details of constructors of OTcl classes `Agent/TCP` and `SplitObject`. The instproc `next{...}` in Line 2 invokes the instproc with the same name (i.e., `init` in this case) of the parent class. The invocation of instproc `init` therefore keeps moving up the hierarchy until it reaches the top-level class `SplitObject`

[8]Note that Line 11 returns a reference string stored in `$o`, not the variable `$o`.

Program 3.11 Global instance procedures `new` and `delete`

```
    //~tclcl/tcl-object.tcl
 1  proc new { className args } {
 2      set o [SplitObject getid]
 3      if [catch "$className create $o $args" msg] {
 4          if [string match "__FAILED_SHADOW_OBJECT_" $msg] {
 5              delete $o
 6              return ""
 7          }
 8          global errorInfo
 9          error "class $className: constructor failed:
                                                  $msg" $errorInfo
10      }
11      return $o
12  }

13  proc delete o {
14      $o delete_tkvar
15      $o destroy
16  }
```

(see Lines 6–11 in Program 3.12). Here, the instproc `create-shadow` in Line 8 marks the beginning of the C++ shadow object construction process, which will be discussed in Part II. After constructing the shadow object, the process returns down the hierarchy tree, performs the rest of the initialization in instprocs `init` (e.g., of class `Agent/TCP`), and returns the constructed object to the caller.

Program 3.12 The constructor of OTcl classes `Agent/TCP` and `SplitObject`

```
    //~ns/tcl/lib/ns-agent.tcl
 1  Agent/TCP instproc init {} {
 2      eval $self next
 3      set ns [Simulator instance]
 4      $ns create-eventtrace Event $self
 5  }

    //~tclcl/tcl-object.tcl
 6  SplitObject instproc init args {
 7      $self next
 8      if [catch "$self create-shadow $args"] {
 9          error "__FAILED_SHADOW_OBJECT_" ""
10      }
11  }
```

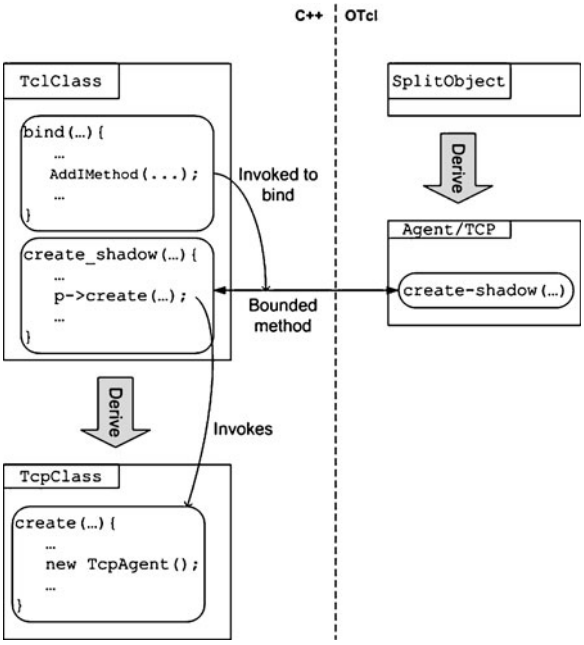

Fig. 3.5 The shadow object creation process: Moving from the OTcl domain to the C++ domain through the OTcl command `create-shadow` of class `Agent/TCP`

Part II [C++ Domain]: Shadow Object Construction

Continuing from Step 6 in Part I, the main steps in this part are to execute the following C++ statements and functions in sequence (see also Fig. 3.4):

II.1. Step 6 in Part I
II.2. The C++ function `create-shadow(...)` of class `TclClass`
II.3. The C++ function `create(...)` of class `TcpClass`
II.4. The C++ statement `new TcpAgent()`
II.5. All related C++ constructors of the class `TcpAgent`

After all the above steps, the constructed shadow object is returned to the caller in the OTcl domain.

The details of Part II are as follows. The first step in Part II is to execute the following OTcl statement from within the instproc `init` of the OTcl class `SplitObject`:

```
$self create-shadow $args
```

where `$self` is an OTcl object whose class is `Agent/TCP`.

It is here where NS2 moves from the OTcl domain to the C++ domain. From Fig. 3.5, the OTcl command `create-shadow` (Step II.1) of class `Agent/TCP` is bound to the C++ function `create_shadow()` of class `TclClass`

(Step II.2). From within the function `create_shadow()`, the statement "`p->create(...)`" is executed, where "`p`" is a pointer to a `TcpClass` object (Step II.3). From Lines 4–6 of Program 3.4, the function `create(...)` executes the C++ statement `new TcpAgent()` to create a shadow `TcpAgent` object (Step II.4). Here, all the constructors along the class hierarchy are invoked (Step II.5). Finally, the created object is returned down the hierarchy back to the caller.

3.5.3 TclObjects Destruction Process

TclObject destruction is the reverse of the TclObject construction. It destroys objects in both the C++ and OTcl domains, and returns the memory allocated to the objects to the system, using a global procedure `delete{...}`. From Program 3.11, Lines 14 and 15 destroy the objects in the OTcl and C++ domains, respectively.

In most cases, we do not need to explicitly destroy objects, since they are automatically destroyed when the simulation terminates. However, object destruction is a good practice to prevent memory leak. We destroy object when it is no longer in need. For example, the instproc `use-scheduler` of the OTcl class `Simulator` executes "`delete $schduler_` before creating a new one (see file ~*ns*/tcl/lib/ns-lib.tcl).

3.6 Access the OTcl Domain from the C++ Domain

This section discusses the following main operations for accessing the OTcl domain from the C++ domain[9]:

1. Obtain the reference to the Tcl interpreter (using the C++ function `instance()`),
2. Execute the OTcl statements from within the C++ domain (using the C++ functions `eval(...)`, `evalc(...)`, and `evalf(...)`),
3. Pass or receive results to/from the OTcl domain (using the C++ functions `result(...)` and `resultf(...)`), and
4. Retrieve the reference to TclObjects (using the C++ functions `enter(...)`, `delete(...)`, and `lookup(...)`).

[9]See the details in the file ~`tclcl/Tcl.cc`.

3.6.1 Obtain a Reference to the Tcl Interpreter

In OOP, a programmer would ask an object to take actions. Without an object, no action shall be taken. In this section, we shall demonstrate how NS2 asks the *Tcl interpreter* (i.e., object) to take actions.

NS2 obtains a C++ object which represents the Tcl interpreter using the following C++ statement:

```
Tcl& tcl = Tcl::instance();
```

where the function `instance()` of class `Tcl` returns the variable `instance_` of class `Tcl` which is a reference to the Tcl interpreter. After executing the above statement, we perform the above operations through the obtained reference (e.g., `tcl.eval(...)`, `tcl.result(...)`).

3.6.2 Execution of Tcl Statements

Class `Tcl` provides the following four functions to execute Tcl statements:

- `Tcl::eval(char* str)`: Executes the string stored in a variable "`str`" in the OTcl domain.
- `Tcl::evalc(const char* str)`: Executes the string "`str`" in the OTcl domain.
- `Tcl::eval()`: Executes the string which has already been stored in the internal variable `bp_`.
- Instproc `Tcl::evalf(const char* fmt,...)`: Uses the format "`fmt`" of `printf(...)` in C++ to formulate a string, and executes the formulated string.

Example 3.14. The followings show various ways in C++, which tell the Tcl interpreter to print out "`Overall Packet Delay is 10.0 seconds`" on the screen.

```
Tcl& tcl = Tcl::instance();

// Using eval(...)
tcl.eval("puts [Overall Packet Delay is 10.0
seconds]");

// Using evalc(...)
char s[128];
strcpy(s,"puts [Overall Packet Delay is 10.0
seconds]");
tcl.evalc(s);
// Using eval()
```

```
char s[128];
sprintf(tcl.buffer(),"puts [Overall
                  Packet Delay is 10.0 seconds]");
tcl.eval();

// Using evalf(...)
float delay = 10.0;
tcl.evalf("puts [Overall
               Packet Delay is %2.1f seconds]",
               delay);
```

Note that `tcl::buffer()` returns the internal variable `bp_`. □

3.6.3 Pass or Receive Results to/from the Interpreter

3.6.3.1 Passing Results to the OTcl Domain

Class `Tcl` provides two functions to *pass* results *to* the OTcl domain:

- `Tcl::result(const char* str)`: Passes the string `<str>` as the result to the interpreter.
- `Tcl::resultf(const char* fmt,...)`: Uses the format "`fmt`"of `printf (...)` in C++ to formulate a string, and passes the formulated string to the interpreter.

Example 3.15. Let an OTcl command `return10` of class `MyObject` returns the value "10" to the interpreter. The implementation of the OTcl command `return10` is given below:

```
int MyObject::command(int argc, const char*const*
argv)
{
    Tcl& tcl = Tcl::instance();
    if (argc == 2) {
        if (strcmp(argv[1], "return10") == 0) {
            Tcl& tcl=Tcl::instance();
            tcl.result("10");
            return TCL_OK;
        }
    }
    return (NsObject::command(argc, argv));
}
```

From OTcl, the following statement stores the value returned from the OTcl command "return10 of the C++ object whose class is MyObject in the OTcl variable "val".

```
set obj [new MyObject]
set val [$obj return10]
```

□

Example 3.16. Let an OTcl command returnVal of class MyObject return the value stored in the C++ variable "value" to the interpreter. The implementation of the OTcl command returnVal is given below:

```
int MyObject::command(int argc, const char*const*
argv)
{
    Tcl& tcl = Tcl::instance();
    if (argc == 2) {
        if (strcmp(argv[1], "returnVal") == 0) {
            tcl.resultf("%1.1f",value);
            Tcl& tcl=Tcl::instance();
            return TCL_OK;
        }
    }
    return (NsObject::command(argc, argv));
}
```

The OTcl command returnVal can be invoked as follows:

```
set obj [new MyObject]
set val [$obj returnVal]
```

□

3.6.3.2 Retrieving Results from the OTcl Domain

Class Tcl provides one function to *receive* results *from* the OTcl domain:

- Tcl::result(void): Retrieves the result from the interpreter as a string.

Example 3.17. The following statements stores the value of the OTcl variable "val" in the C++ variable "value":

```
Tcl& tcl=Tcl::instance();
tcl.evalc("set val");
char* value = tcl.result();
```

□

Class Tcl uses a private member variable "tcl_->result" to pass results between the two hierarchies, where "tcl_" is a pointer to a Tcl_Interp object. When passing a result value to the OTcl domain, the function result(...) stores

the result in this variable. The Tcl interpreter is responsible for reading the result value from the variable "`tcl_->result`." On the other hand, when passing a result value to the C++ domain, the interpreter stores the value in the same C++ variable, and the function `result()` reads the value stored in this variable.

3.6.4 TclObject Reference Retrieval

Recall that an object in the interpreted hierarchy always has a shadow compiled object. NS2 records an association of a pair of objects as an entry in its hash table.[10] Class `Tcl` provides the following three functions to deal with the hash table:

- Function `enter(TclObject* o)`: Inserts a TclObject "`*o`" into the hash table, and associates "`*o`" with the OTcl name string stored in the variable "`name_`" of the TclObject "`*o`." This function is invoked by function `TclClass:create_shadow(...)` when a TclObject is created.
- Function `delete(TclObject* o)`: Deletes the entry associated with the TclObject "`*o`" from the hash table. This function is invoked by function `TclClass:delete_shadow(...)` when a TclObject is destroyed.
- Function `lookup(char* str)`: Returns a pointer to the TclObject whose name is "`str`."

Example 3.18. Consider the OTcl command `target{...}` of the C++ class `Connector` in Program 3.13. This OTcl command sets the input argument as the forwarding NsObject (see the details in Sect. 5.3).

Program 3.13 Function `command(...)` of the C++ class `Connector`

```
    //~ns/common/connector.cc
1   int Connector::command(int argc, const char*const* argv)
2   {
3       Tcl& tcl = Tcl::instance();
4       ...
5       if (argc == 3) {
6           if (strcmp(argv[1], "target") == 0) {
7               ...
8               target_ = (NsObject*)TclObject::lookup(argv[2]);
9               ...
10          }
11      ...
12      }
13      return (NsObject::command(argc, argv));
14  }
```

[10] The definition of a hash table is given in Sect. 6.2.3.

Here, "argv[2]" is the forwarding object passed from the OTcl domain. Line 8 executes TclObject::lookup(argv[2]) to retrieve the shadow compiled object pointer corresponding to the OTcl object "argv[2]." The retrieved pointer is converted to a pointer to an object whose type is NsObject and stored in the variable "target_." □

3.7 Translation of Tcl Code

In general, Tcl is an interpreted programming language. It does not require compilation before execution. However, during the compilation, NS2 translates all built-in Tcl modules (e.g., all the script files in directory *~ns*/tcl/lib) into the C++ language using class EmbeddedTcl, to speed up the simulation.

The compilation process is carried out according to the file descriptor, Make file (see Sect. 2.7). The statement related to Tcl translation is

```
$(TCLSH) bin/tcl-expand.tcl tcl/lib/ns-lib.tcl $(NS_
  TCL_LIB_STL) \
| $(TCL2C) et_ns_lib > gen/ns_tcl.cc
```

where $(TCLSH) is the executable file which invokes the Tcl interperter, $(TCL 2C) is a Tcl script which translates Tcl codes to C++ codes, and $(NS_TCL_LIB _STL) are the list of Tcl files which will be translated to C++ programs.

The above statement has two parts, each divided by a pipeline "|" operator.[11]

First Part: Expansion

The first part is shown below:

```
$(TCLSH) bin/tcl-expand.tcl tcl/lib/ns-lib.tcl
 $(NS_TCL_LIB_STL)
```

This part asks the Tcl to interpret (i.e., run) the Tcl script file bin/tcl-expand.tcl, with two input arguments: *~ns*/tcl/lib/ns-lib.tcl and $(NS_TCL_ LIB_STL). It expands the content of all the files specified in the input arguments.

A part of the expansion is to *source* the Tcl files specified in the input files. By sourcing a Tcl file, we mean to replace a source statement with the following syntax

```
source <filename>
```

[11]The pipeline operator captures the screen output (i.e., stdout) resulting from the execution of what ahead of it. The captured string is then fed as a keyboard input (i.e., stdin) for the execution of what following it.

with the content in the file `<filename>`. Example source statements are in the file ~*ns*/tcl/lib/ns-lib.tcl:

```
source ns-autoconf.tcl
source ns-address.tcl
source ns-node.tcl
source ns-rtmodule.tcl
...
```

which tell NS2 to incorporate these files into the translation process.

Second Part: Translation

The second part is located behind the pipe operator ("|"), i.e.,

```
$(TCL2C) et_ns_lib > gen/ns_tcl.cc
```

The statement "`$(TCL2C) et_ns_lib`" translates the OTcl scripts from the former part into C++ programs using an `EmbeddedTcl` object "`et_ns_lib`." The output of the second part is the C++ programs, which are redirected using a redirection operator (i.e., ">"), to the C++ file `gen/ns_tcl.cc`.

3.8 Chapter Summary

NS2 is a network simulator tool consisting of OTcl and C++ programming languages. The main operations of NS2 (e.g., packet passing) are carried out using the C++ language, while the network configuration process (e.g., creating and connecting nodes) is carried out using the OTcl language. In most cases, programmers create object from the OTcl domain, and NS2 automatically creates *shadow* object in the C++ domain. The connection between the interpreted and compiled hierarchies is established through TclCL. In this chapter, we have discussed the main functionalities of TclCL – including class binding, variable binding, method binding, shadow object construction process, Tcl access mechanism, and Tcl code translation.

3.9 Exercises

1. Rewrite the program in Sect. 1.5 using three C++ programming styles discussed in Sect. 3.1.
2. The class binding process consists of four major components.

a. What are those components? What are their definitions?
b. Look into the NS2 codes. What are the components corresponding to the following classes: `RenoTcpAgent`, `Agent`, `Connector`.
c. Bind a C++ class `MyObject` to an OTcl class `MyOTclClass`. What are the main class binding components? Compile and run the codes. Verify the binding by printing out a string from the constructor of the C++ class. Discuss the results.

3. Consider Exercise 2.

a. Create an OTcl command named "`print-count`" which print out the value of `count_` on the screen. Write a Tcl simulation script to test the OTcl command.
b. Create an instproc "`print-count`" for the class `MyOTclObject`. If you execute "`print-count`," which body of "`print-count`" will NS2 execute (i.e., the OTcl command or the instproc)? Design an experiment to test your answer. Can you make NS2 to execute the other body? If so, how?

4. Consider Exercise 2.

a. Declare a variable `my_c_var_` and an instvar `my_otcl_var_` in classes `MyObject` and `MyOTclObject`. Bind them together.
b. Design an experiment to show that a change in one variable automatically updates the value of the other variable.
c. How would you define the default value of a variable in C++ and OTcl domain? If the default values are different, which one would be taken during run time? Design an experiment to prove your answer.

5. What are the major differences among classes `TclObject`, `TclClass`, and `InstVar`? Explain their roles during an object creation process.
6. What are the differences among a C++ function, an OTcl instproc, and an OTcl command?
7. What are the differences between functions `eval(...)` and `evalc(...)`?
8. Show a C++ statement to retrieve a `Tcl` interpreter.
9. The C++ class `TcpAgent` is bound to the OTcl class `Agent/TCP`. The C++ variable "`cwnd_`" bound to the OTcl "`cwnd_`" instvar are used to store the congestion window value.

a. Demonstrate how to set the default congestion window of TCP in the C++ domain to be 20 and in the OTcl domain to be 30. If you set the default value in both the C++ and OTcl domains, what would be the actual default value at run time?
b. Show different ways in the OTcl domain to change the congestion window value of an `Agent/Tcp` object `$tcp` to 40.

10. Can you perform the following actions? If not, what are the conditions under which you can perform such actions.

a. Bind ANY C++ variable to ANY OTcl variable.
b. Call ANY C++ statement from the OTcl domain.
c. Call ANY OTcl statement from the C++ domain.

11. What are the top-level classes in the C++ domain and in the OTcl domain?

Chapter 4
Implementation of Discrete-Event Simulation in NS2

NS2 is a event-driven simulator, where actions are associated with events rather than time. An event in a event-driven simulator consists of execution time, associated actions, and a reference to the next event (Fig. 4.1). These events connect to each other and form a *chain of events* on the *simulation timeline* (e.g., that in Fig. 4.1). Unlike a time-driven simulator, in an event-driven simulator, time between a pair of events does not need to be constant. When the simulation starts, events in the chain are executed from left to right (i.e., chronologically).[1] In the next section, we will discuss the simulation concept of NS2. In Sects. 4.2–4.4, we will explain the details of classes `Event` and `Handler`, class `Scheduler`, and class `Simulator`, respectively. Finally, we summarize this chapter in Sect. 4.5.

4.1 NS2 Simulation Concept

NS2 simulation consists of two major phases.

Phase I: Network Configuration Phase

In this phase, NS2 constructs a network and sets up an initial chain of events. The initial chain of events consists of events that are scheduled to occur at certain times (e.g., start FTP (File Transfer Protocol) traffic at 1 s). These events are called *at-events* (see Sect. 4.2). This phase corresponds to every line in a Tcl simulation script before executing instproc `run{}` of the `Simulator` object.

[1] By execution, we mean taking actions associated with an event.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_4, © Springer Science+Business Media, LLC 2012

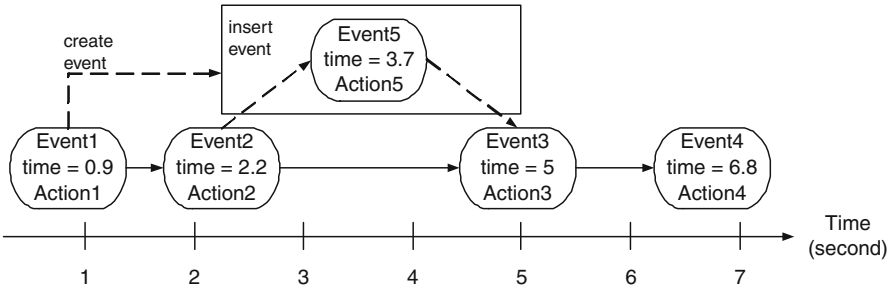

Fig. 4.1 A sample chain of events in a event-driven simulation. Each event contains execution time and a reference to the next event. In this figure, Event 1 creates and inserts Event 5 after Event 2 (the execution time of Event 5 is at 3.7 s)

Phase II: Simulation Phase

This part corresponds to a single line, which invokes the instproc `run{}` of class `Simulator`. Ironically, this single line contributes to most (e.g., 99%) of the simulation.

In this part, NS2 moves along the chain of events and executes events chronologically. Here, the instproc `run{}` starts the simulation by *dispatching* the first event in the chain of events. In NS2, "dispatching an event" or "firing an event" means "taking actions corresponding to that event." An action is, for example, starting FTP traffic or creating another event and inserting the created event into the chain of events. In Fig. 4.1, at 0.9 s, Event1 creates Event5 which will be dispatched at 3.7 s, and inserts Event5 after Event2. After dispatching an event, NS2 moves down the chain and dispatches the next event. This process repeats until the last event in the chain is dispatched, signifying the end of simulation.

4.2 Events and Handlers

4.2.1 An Overview of Events and Handlers

As shown in Fig. 4.1, an event is associated with an action to be taken at a certain time. In NS2, an event contains a *handler* that specifies the action, and the *firing time* or *dispatching time*. Program 4.1 shows declaration of classes `Event` and `Handler`. Class `Event` declares variables "`handler_`" (whose class is `Handler`; Line 5) and "`time_`" (Line 6) as its associated handler and firing time, respectively. To maintain the chain of events, each `Event` object contains pointers "`next_`" (Line 3) and "`prev_`" (Line 4) to the next and previous `Event` objects, respectively. The Variable "`uid_`" (Line 7) is an ID unique to every event.

Program 4.1 Declaration of classes `Event` and `Handler`

```
    //~/ns/common/scheduler.h
 1  class Event {
 2  public:
 3      Event* next_;        /* event list */
 4      Event* prev_;
 5      Handler* handler_;   /* handler to call when event ready */
 6      double time_;        /* time at which event is ready */
 7      scheduler_uid_t uid_;    /* unique ID */
 8      Event() : time_(0), uid_(0) {}
 9  };

10  class Handler {
11  public:
12      virtual ~Handler () {}
13      virtual void handle(Event* e) = 0;
14  };
```

Program 4.2 Function `handle` of class `NsObject`

```
    //~/ns/common/object.cc
1   void NsObject::handle(Event* e)
2   {
3       recv((Packet*)e);
4   }
```

Lines 10–14 in Program 4.1 show the declaration of an abstract class `Handler`. Class `Handler` specifies the *default action* to be taken when an associated event is dispatched in its pure virtual function `handle(e)` (Line 13).[2] This declaration forces all its instantiable derived classes to provide the action in function `handle(e)`. In the followings, we will discuss few classes which derive from classes `Event` and `Handler`. These classes are `NsObject`, `Packet`, `AtEvent`, and `AtHandler`.

4.2.2 Class `NsObject`: A Child Class of Class `Handler`

Derived from class `Handler`, class `NsObject` is one of the main classes in NS2. It is a base class for most of network components. We will discuss the details of this class in Chap. 5. Here, we only show the implementation of the function `handle(e)` of class `NsObject` in Program 4.2. The function `handle(e)` casts an `Event` object associated with the input pointer (`e`) to a `Packet` object.

[2]We call actions specified in the function `handle(e)` *default actions*, since they are taken by default when the associated event is dispatched.

Program 4.3 Declaration of classes `AtEvent` and `AtHandler`, and function `handle` of class `AtHandler`

```
     //~/ns/common/scheduler.cc
  1  class AtEvent : public Event {
  2  public:
  3      AtEvent() : proc_(0) {
  4      }
  5      ~AtEvent() {
  6          if (proc_) delete [] proc_;
  7      }
  8      char* proc_;
  9  };

 10 class AtHandler : public Handler {
 11 public:
 12     void handle(Event* event);
 13 } at_handler;

 14 void AtHandler::handle(Event* e)
 15 {
 16     AtEvent* at = (AtEvent*)e;
 17     Tcl::instance().eval(at->proc_);
 18     delete at;
 19 }
```

Then it feeds the casted object to the function `recv(p)` (Line 3). Usually, function `recv(p)`, where "p" is a pointer to a packet, indicates that the `NsObject` has received a packet "p" (see Chap. 5). Unless overridden, by derived classes, the function `handle(e)` of an NsObject simply indicates packet reception.

4.2.3 *Classes `Packet` and `AtEvent`: Child Classes of Class `Event`*

Classes `Packet` and `AtEvent` are among key NS2 classes which derive from class `Event`. These two classes can be placed on the chain of events so that their associated handler will take actions at the firing time. While the details of class `AtEvent` are discussed in this section, that of class `Packet` will be discussed later in Chap. 8.

Declared in Program 4.3, class `AtEvent` represents events whose action is the execution of an OTcl statement. It contains one string variable "`proc_`" (Line 8) which holds an OTcl statement string. At the firing time, the associated handler, whose class is `AtHandler`, will retrieve and execute the OTcl string from this variable.

Program 4.4 Instance procedure `at` of class `Simulator` and command `at` of class `Scheduler`

```
    //~/ns/tcl/lib/ns-lib.tcl
 1  Simulator instproc at args {
 2      $self instvar scheduler_
 3      return [eval $scheduler_ at $args]
 4  }

    //~/ns/common/scheduler.cc
 5  if (strcmp(argv[1], "at") == 0) {
 6      /* t < 0 means relative time: delay = -t */
 7      double delay, t = atof(argv[2]);
 8      const char* proc = argv[3];
 9      AtEvent* e = new AtEvent;
10      int n = strlen(proc);
11      e->proc_ = new char[n + 1];
12      strcpy(e->proc_, proc);
13      delay = (t < 0) ? -t : t - clock();
14      if (delay < 0) {
15          tcl.result("can't schedule command in past");
16          return (TCL_ERROR);
17      }
18      schedule(&at_handler, e, delay);
19      sprintf(tcl.buffer(), UID_PRINTF_FORMAT, e->uid_);
20      tcl.result(tcl.buffer());
21      return (TCL_OK);
22  }
```

Derived from class `Handler`, class `AtHandler` specifies the actions to be taken at firing time in its function `handle(e)` (Lines 14–19). Here, Line 16 casts the input event into an `AtEvent` object. Then Line 17 extracts and executes the OTcl statement from variable "`proc_`" of the cast event.

In the OTcl domain, an `AtEvent` object is placed in a chain of events at a certain firing time by instproc "`at`" of class `Simulator`. Whose syntax is:

```
$ns at <time> <statement>
```

where "`$ns`" is the `Simulator` object (see Sect. 4.4), `<time>` is the firing time, and `<statement>` is an OTcl statement string which will be executed when the simulation time is `<time>` second.

From Lines 1–4 of Program 4.4, the instproc `at{...}` of an OTcl class `Simulator` invokes an OTcl command "`at`" of the `Scheduler` object (Lines 5–22). The OTcl command `at{...}` of class `Scheduler` stores the firing time in a variable "`t`" (Line 7). Line 9 creates an `AtEvent` object. Lines 8 and 10–12 store the input OTcl command in the variable "`proc_`" of the created `AtEvent` object.

Line 13 converts the firing time to the "`delay`" time from the current time. Finally, Line 18 schedules the created `event *e` at "`delay`" seconds in future, feeding address of the variable "`at_handler`" (see Program 4.3) as an input argument to function `schedule(...)`.

4.3 The Scheduler

The scheduler maintains the chain of events and simulation (virtual) time. At runtime, it moves along the chain and dispatches one event after another. Since there is only one chain of events in a simulation, there is exactly one `Scheduler` object in a simulation. Hereafter, we will refer to the `Scheduler` object simply as the Scheduler. Also, NS2 supports the four following types of schedulers: List Scheduler, Heap Scheduler, Calendar Scheduler (default), and Real-time Scheduler. For brevity, we do not discuss the differences among all these schedulers here. The details of these schedulers can be found in [17].

4.3.1 Main Components of the Scheduler

Declared in Program 4.5, class `Scheduler` consists of a few main variables and functions. Variable "`clock_`" (Line 19) contains the current simulation time, and function `clock()` (Line 11) returns the value of the variable "`clock_`". Variable "`halted_`" (Line 22) is initialized to 0, and is set to 1 when the simulation is stopped or paused. Variable "`instance_`" (Line 20) is the reference to the Scheduler, and function `instance()` (Line 3) returns the variable "`instance_`". Variable `uid_` (Line 21) is the event unique ID. In NS2, the Scheduler acts as a single point of unique ID management. When an event is inserted into the simulation timeline, the Scheduler creates a new unique ID and assigns the ID to the event. Both the variables "`instance_`" and "`uid_`" are static, since there is only one Scheduler and unique ID in a simulation.

4.3.2 Data Encapsulation and Polymorphism Concepts

Program 4.5 implements the concepts of *data encapsulation* and *polymorphism* (see Appendix B). It hides the chain of events from the outside world and declares pure virtual functions `cancel(e)`, `insert(e)`, `lookup(uid)`, `deque()`, and `head()` in Lines 6–10 to manage the chain. Classes derived from class `Scheduler` provide implementation of all of the above functions. The beauty of this mechanism is the ease of modifying the scheduler type at runtime.

Program 4.5 Declaration of class `e.g., Scheduler`

```
    //~ns/common/scheduler.h
 1  class Scheduler : public TclObject {
 2  public:
 3      static Scheduler& instance() { return (*instance_); }
 4      void schedule(Handler*, Event*, double delay);
 5      virtual void run();
 6      virtual void cancel(Event*) = 0;
 7      virtual void insert(Event*) = 0;
 8      virtual Event* lookup(scheduler_uid_t uid) = 0;
 9      virtual Event* deque() = 0;
10      virtual const Event* head() = 0;
11      double clock() const { return clock_j}
12      virtual void reset();
13  protected:
14      void dispatch(Event*);
15      void dispatch(Event*, double);
16      Scheduler();
17      virtual ~Scheduler();
18      int command(int argc, const char*const* argv);
19      double clock_;
20      static Scheduler* instance_;
21      static scheduler_uid_t uid_;
22      int halted_;
22  };
```

NS2 implements most of the codes in relation to class `Scheduler`, not its derived classes (e.g., `CalendarScheduler`). At runtime (e.g., in a Tcl simulation script), we can select a scheduler to be of any derived class (e.g., `CalendarScheduler`) of class `Scheduler` without having to modify the codes for the base class (e.g., `Scheduler`).

4.3.3 Main Functions of the Scheduler

Three main functions of class `Scheduler` are `run()` (Program 4.6), `schedule(h,e,delay)` (Program 4.7) and `dispatch(p,t)` (Program 4.8). In Program 4.6, function `run()` first sets variable "`instance_`" to the address of the scheduler (`this`) in Line 3. Then, it keeps dispatching events (Line 6) in the chain until "`halted_`"$\neq 0$ or until all the events are executed (Line 5).

Function `schedule(h,e,delay)` in Program 4.7 takes three input arguments: A `Handler` pointer `(h)`, an `Event` pointer `(e)`, and the delay `(delay)`, respectively. It inserts the input `Event` object `(*e)` into the chain of events. Lines 3–12 check for possible errors. Line 13 increments the unique ID of the Scheduler and assigns it to the input `Event` object. Line 14 associates the input `Handler`

Program 4.6 Function `run` of class `Scheduler`

```
    //~ns/common/scheduler.cc
1   void scheduler::run()
2   {
3       instance_ = this;
4       Event *p;
5       while (!halted_ && (p = deque())) {
6           dispatch(p, p->time_);
7       }
8   }
```

Program 4.7 Function `schedule` of class `Scheduler`

```
    //~ns/common/scheduler.cc
1   void Scheduler::schedule(Handler* h, Event* e, double delay)
2   {
3       if (!h) { /* error: Do not feed in NULL handler */ };
4       if (e->uid_ > 0) {
5           printf("Scheduler: Event UID not valid!\n\n");
6           abort();
7       }
8       if (delay < 0) { /* error: negative delay */ };
9       if (uid_ < 0) {
10          fprintf(stderr, "Scheduler: UID space exhausted!
11          \n"); abort();
12      }
13      e->uid_ = uid_++;
14      e->handler_ = h;
15      double t = clock_ + delay;
16      e->time_ = t;
17      insert(e);
18  }
```

Program 4.8 Function `dispatch` of class `Scheduler`

```
    //~ns/common/scheduler.cc
1   void Scheduler::dispatch(Event* p, double t)
2   {
3       if (t < clock_) { /* error */ };
4       clock_ = t;
5       p->uid_ = -p->uid_;      // being dispatched
6       p->handler_->handle(p); // dispatch
7   }
```

object (*h) with the input Event object (*e). Line 15 converts input delay time (delay) to the firing time (time_) of the Event object "e." Line 17 inserts the configured Event object *e in the chain of events via function insert(e). Since the scheduler increments its unique ID when invoking function schedule(...), every scheduled event will have different unique ID.

Finally, the errors in Lines 3–12 include

1. Null handler (Line 3)
2. Positive Event unique ID (Lines 4–7; See Sect. 4.3.5)
3. Negative delay (Line 8)
4. Negative Scheduler unique ID[3] (Lines 19-12)

Function dispatch(p,t) in Program 4.8 is invoked by function run() at the firing time (Line 6 of Program 4.6). It takes a dispatching event (*p) and firing time (t) as input arguments. Since the scheduler moves forward in simulation time, the firing time (t) cannot be less than the current simulation time (clock_). From Program 4.8, Line 3 will show an error, if t < clock_. Line 4 sets the current simulation virtual time to be the firing time of the event. Line 5 inverts the sign of the "uid_" of the event, indicating that the event is being dispatched. Line 6 invokes function handle(p) of the associated handler "handler_," feeding the event (p) as an input argument.

4.3.4 Two Auxiliary Functions

Apart from the above three main function, class Scheduler provides two very useful functions: instance() (Line 3 in Program 4.5) and clock (Line 11 in Program 4.5).

- Function instance() returns *instance_ which is the address of the Scheduler.
- Function clock() returns the current simulation virtual time.

We shall see the use of these two functions throughout NS2 programming.

Example 4.1. In order to obtain the current virtual simulation time in C++, we can resort to the following statements

```
Scheduler& s = Scheduler::instance();
cout << "The current time is " << s.clock() << endl;
```

Here the upper line retrieves the reference to the Scheduler, while the lower line invokes the function clock() of the current simulation time. □

[3]The unique ID of the Scheduler is always positive. Its negative value indicates possible abnormality such as memory overflow or inadvertent memory access violation.

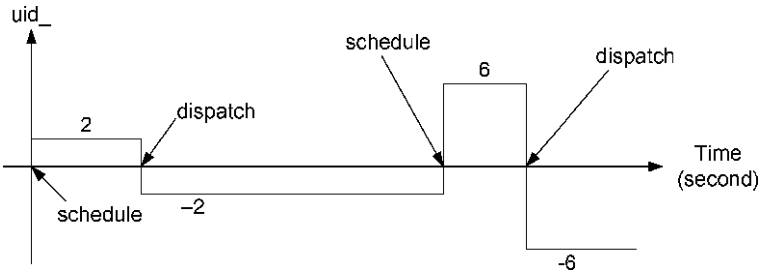

Fig. 4.2 Dynamics of `Event` unique ID `(uid)_`: Take a positive value from `Scheduler ::uid_` when being scheduled, and invert the sign when being dispatched. Increment upon `schedule` and inversion of sign upon `dispatch`

4.3.5 Dynamics of the Unique ID of an Event

The dynamics of the event's unique ID (`uid_`) is fairly subtle. In general, the Scheduler maintains the unique ID and assigns the unique ID to the event being scheduled. To make "`uid_`" unique, the Scheduler increments its "`uid_`" and assigns the incremented "`uid_`" to the scheduling event in its function `schedule(...)` (Line 13 in Program 4.7). When dispatching an event, the scheduler inverts the sign of "`uid_`" of the dispatching event (Line 5 in Program 4.8). Figure 4.2 shows the dynamics of the unique ID caused by the above `schedule(...)` and `dispatch(...)` functions. Unless the associated event is being dispatched, "`uid_`" of an event is always increasing and non-negative. The sign toggling mechanism of unique ID ensures that events will be scheduled and dispatched properly. If a scheduled event is not dispatched, or is dispatched twice, its unique ID will be positive, and an attempt to schedule this undispatched event will cause an error (Lines 5 and 6 in Program 4.7).

4.3.6 Scheduling–Dispatching Mechanism

We conclude this section through an example explaining the scheduling–dispatching mechanism. Consider the following script

```
set ns [new Simulator]
$ns at 10 [puts "An event is dispatched"]
$ns run
```

which prints out the message "`An event is dispatched`" at 10 s after the simulation has started. Figure 4.3 shows the functions (shown in rectangles) and objects (shown in rounded rectangles) related to the scheduling–dispatching mechanism, whose names are shown in boldface font. Again, an `AtEvent` object is scheduled by the OTcl command "`at`" (in the upper-left rectangle), of class `Scheduler`. The Scheduler creates an `AtEvent` object "`e`" and stores

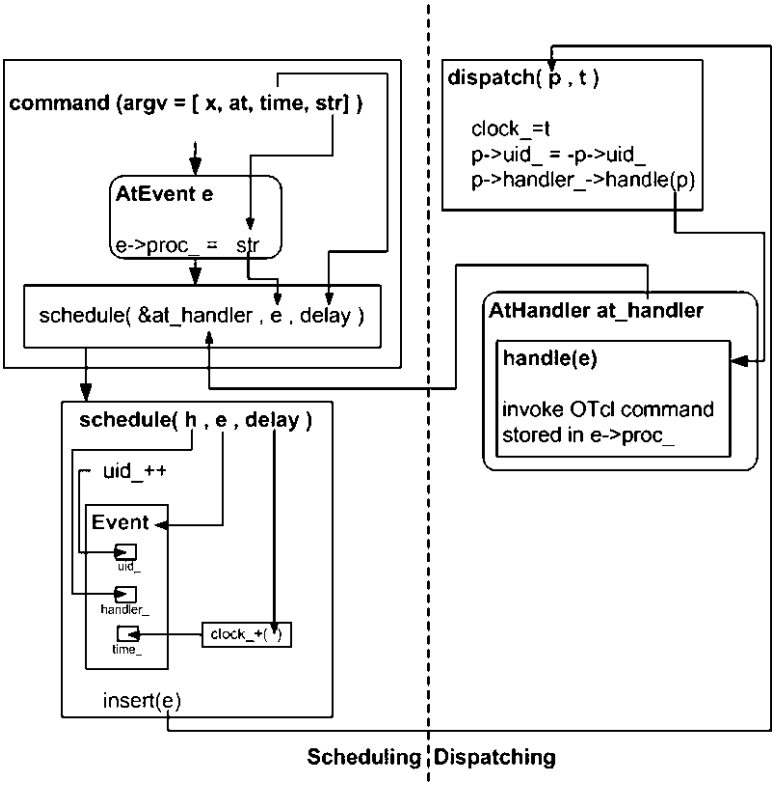

Fig. 4.3 Scheduling and dispatching mechanism of an `AtEvent`

input command (the fourth input argument "`str = puts "An event is dispatched"`)" in `e->proc_`. Then, it schedules the event "e" with `delay` converted from `time = 10` (the third input argument), feeding the address of `AtHandler` object (`at_handler` in the lower right round rectangle) as the corresponding handler.

The lower-left rectangle in Fig. 4.3 shows details of the function `schedule (h,e,delay)` of class `Scheduler`. Before inserting event "e" into the chain of events, function `schedule(...)` configures event "e" as follows: Update "`uid_`" to be the same as that of `Scheduler`, store "`at_handler`" in the handler of event "e," and set firing time to be "`clock_`" (current time) + "`delay`."

At the firing time, the scheduled `AtEvent` object is dispatched through the function `dispatch(p,t)` (the upper-right rectangle in Fig. 4.3). When the scheduled `Event` object "e"[4] is dispatched, function `dispatch(...)` inverts the sign of its variable "`uid_`," and invokes function `handle(e)` of the

[4]In Program 4.8, the first argument of function `dispatch(...)` is "p." Here, we use "e" as the first argument for the sake of explanation.

corresponding handler feeding `Event` object "`e`" as an input argument. Since the handler is "`at_handler`" (see the upper-left rectangle), the OTcl command "`puts "An event is dispatched"`" stored in "`e`" is executed.

4.3.7 Null Event and Dummy Event Scheduling

When being dispatched, an event "`p`" is fed to function `handle(p)` of the associated handler for a certain purpose. For example, the function `handle(p)` of class `NsObject` executes "`recv(p)`", where "`p`" is a packet reception event. Here, the event `*p` must have been created and fed to the function `schedule(...)` before the ongoing dispatching process.

In some cases, an event only indicates the time where the default action is taken but takes no part in such the action. For example, a queue unblocking event informs the associated `Queue` object of the completion of the ongoing transmission (see Sect. 7.3). The function `handle(p)` of the associated handler in this case simply invokes function `resume()` which take no input argument (i.e., action taking). Clearly the queue unblocking event takes no role in the dispatching process. In this case, we do not need to explicitly create an event. Instead, we can use a null event or a dummy event as an input argument to the function `schedule(...)`.

4.3.7.1 Scheduling of a Null Event

Function `schedule(h,e,delay)` takes a pointer to an event as its second input argument. A null event refers to a null pointer which is fed as the second input argument to the function `schedule(...)` (e.g., `schedule(handler, 0,delay)`).

Although simple to use, a null event could lead to runtime error which is difficult to be located. A null event is not an *actual* event. Its unique ID does not follow semantic in Fig. 4.2. The Scheduler ignores the unique ID when scheduling and dispatching a null event, and allows an undispatched event to be rescheduled. This breaks the scheduling–dispatching protection mechanism. Using null events, the users are responsible for ensuring the proper sequence of scheduling–dispatching by themselves.

4.3.7.2 Scheduling of a Dummy Event

This is another approach to schedule and dispatch events which do not take part in default actions. A dummy event is usually declared as a member variable of a C++ class, and is used repeatedly in a scheduling–dispatching process.

Consider a packet departure event which is modeled by class `LinkDelay` (see Sect. 7.2) for example. During simulation, an NsObject informs a `LinkDelay`

object to schedule packet departure events. At the firing time, the packet completely departs the NsObject, and the NsObject is allowed to fetch another packet for transmission. The packet departure event takes no part in the default action, since a new packet is fetched or created by another object.

As we shall see, a packet departure event is represented by a dummy event variable "`intr_`" of class `LinkDelay`, and the packet departure is scheduled through the variable "`intr_`" only. Since the variable "`intr_`" is a dummy `Event`, its unique ID follows the semantic in Fig. 4.2. An attempt to schedule an undispatched event would immediately cause runtime error. Note that "`intr_`" is a member variable of class `LinkDelay`. It is used over and over again to indicate packet departure from a `LinkDelay` object.

As a final note, under a simple configuration, it is recommended to use the null event scheduling approach. For a complicated configuration, on the other hand, the dummy event scheduling is preferable, since it provides a protection against scheduling of undispatched events.

4.4 The Simulator

OTcl and C++ classes `Simulator` are the main classes which supervise the entire simulation. Like the `Scheduler` object, there can be only one `Simulator` object throughout a simulation. This object will be referred to as *the Simulator* hereafter. The Simulator contains two types of key components: simulation objects and information-storing objects. While simulation objects (e.g., the Scheduler) are the key components which drive the simulation. On the other hand, Information-storing objects (e.g., the reference to created nodes) contain information which is shared among several objects. These information-storing objects are created via various instprocs (e.g., `Simulator::node{}`) during the Network Configuration Phase. Most objects access these information-storing objects via its instvar "`ns_`" (set by executing "`set ns_ [Simulator instance]`"), which is a reference to a Simulator.

4.4.1 Main Components of a Simulation

4.4.1.1 Interperted Hierarchy

Created by various instprocs, the main OTcl simulation components are as follows:

- *The Scheduler* (`scheduler_` created by the instproc `Simulator::init`) maintains the chain of events and executes the events chronologically.

- *The null agent* (`nullAgent_` created by the instproc `Simulator::init`) provides the common packet dropping point.[5]
- *Node reference* (`Node_` created by the instproc `Simulator::node`) is an associative array whose elements are the created nodes and indices are node IDs.
- *Link reference* (`link_` created by the instprocs `simplex-link{...}` or `duplex- link{...}`) is an associative array. Associated with an index with format "`sid:did`," each element of "`link_`" is the created unidirectional link which carries packet from node "`sid`" to node "`did`."
- *Reference to routing table* (`routingTable_` created by the instproc `Simulator::get-routelogic{}`,) contains the global routing table.

4.4.1.2 Compiled Hierarchy

In the compiled hierarchy, class `Simulator` also contains variables and functions as shown in Program 4.9. Variable "`instance_`" (Line 18) is a pointer to the Simulator. It is a static variable, which means that there is only one variable "`instance_`" of class `Simulator` for the entire simulation. Variable "`node list_`" (Line 14) is the link list containing all created nodes. The link list can contain up to "`size_`" elements (Line 17), while the total number of nodes is "`nn_`" (Line 16). Variable `rtobject_` (Line 15) is a pointer to a `RouteLogic` object, which is responsible for the routing mechanism (see Chap. 6).

Function `populate_flat_classifiers(...)` (Line 7) pulls out the routing information stored in the variable `*rtobject_` and installs the routing table in the created nodes and links (see Sect. 6.5). Function `add_node(...)` (Line 8) puts the input argument `node` into the link list of nodes (`nodelist_`). Function `get_link_head(...)` returns the link head object (see Chap. 7) of the link with ID "`nh`" which connects to a `ParentNode` object `*node`. Function `node_id_by_addr(addr)` (Line 10) converts node address "`addr`" to node ID. Function `alloc(n)` (Line 11) allocates spaces in `nodelist_` which can accommodate up to "`n`" nodes, and clears all components of `nodelist_` to `NULL`. Function `check(n)` immediately returns if "`n`" is less than `size_`. Otherwise, it will create more space in `nodelist_`, which can accommodate up to "`n`" nodes. Static function `instance()` in Line 3 returns the variable "`instance_`" which is the pointer to the Simulator.

4.4.2 Retrieving the Instance of the Simulator

From the interpreted hierarchy, we can also retrieve the simulator instance by invoking the instproc `instance{}` of class `Simulator` (see program 4.10).

[5]By "dropping a packet," we mean "removing a packet" from the simulation. We will discuss the dropping mechanism in Chap. 5. For the moment, it is sufficient to know that "`nullAgent_`" drops or removes all received packets from the simulation.

Program 4.9 Declaration of class `Simulator`

```
    //~ns/common/simulator.h
 1  class Simulator : public TclObject {
 2  public:
 3      static Simulator& instance() { return (*instance_); }
 4      Simulator() : nodelist_(NULL),
                rtobject_(NULL), nn_(0), size_(0) {}
 5      ~Simulator() { delete []nodelist_;}
 6      int command(int argc, const char*const* argv);
 7      void populate_flat_classifiers();
 8      void add_node(ParentNode *node, int id);
 9      NsObject* get_link_head(ParentNode *node, int nh);
10      int node_id_by_addr(int address);
11      void alloc(int n);
12      void check(int n);
13  private:
14      ParentNode **nodelist_;
15      RouteLogic *rtobject_;
16      int nn_;
17      int size_;
18      static Simulator* instance_;
19  };
```

Program 4.10 Retrieving the instance of the Simulator using instproc `instance` of class `Simulator`

```
    //~ns/tcl/lib/ns-lib.tcl
1 Simulator proc instance {} {
2     set ns [Simulator info instances]
3     if { $ns != "" } {
4         return $ns
5     }
6     ...
7 }
```

This instproc executes the OTcl built-in command "`info`" with an option "`instances`." This execution returns all the instances of a certain class. Since there is only one `Simulator` instance, the statement "`Simulator info instances`" returns the `Simulator` object as required.

4.4.3 Simulator Initialization

Simulator initialization refers to the process in the Network Configuration Phase, which creates the Simulator as well as its components. The Simulator is created by executing "`new Simulator`". This statement invokes the constructor (i.e., the instproc `init{...}` of class `Simulator`) shown in Program 4.11.

Program 4.11 Instprocs `init` and `use-scheduler` of class `Simulator`

```
    //~ns/tcl/lib/ns-lib.tcl
1   Simulator instproc init args {
2       $self create_packetformat
3       $self use-scheduler Calendar
4       $self set nullAgent_ [new Agent/Null]
5       $self set-address-format def
6       eval $self next $args
7   }

8   Simulator instproc use-scheduler type {
9       $self instvar scheduler_
10      if [info exists scheduler_] {
11          if { [$scheduler_ info class] == "Scheduler/$
                type" } {
12              return
13          } else {
14              delete $scheduler_
15          }
16      }
17      set scheduler_ [new Scheduler/$type]
18  }
```

The constructor first initializes the packet format in Line 2, and executes the OTcl statement "`use-scheduler{type}`" in Line 3 to specify type of the Scheduler. By default, type of the Scheduler is `Calendar`. Line 4 creates a null agent (`nullAgent`). Line 5 sets the address format to the default format. The instproc `use-scheduler{type}` (Lines 8–18) deletes the existing Scheduler if it is different from that specified in the input argument "`type`." Then it will create a scheduler with type = `type`, and store the created `Scheduler` object in the instvar "`scheduler_`."

4.4.4 Running Simulation

The Simulation Phase starts at the invocation of the instproc `run{}` of class `Simulator`. As shown in Program 4.12, this instproc first invokes the instproc `configure{}` of class `RouteLogic` (Line 2), which in turn computes the optimal routes and creates the routing table (see Chap. 6). Lines 5–10 reset nodes and queues. Finally, Line 11 starts the Scheduler by invoking the OTcl command `run{}` of class `Scheduler`, which in turn invokes the C++ function `run()` of class `Scheduler` shown in Program 4.6. Again, this function executes events in the chain of events one after another until the Simulator is halted (i.e., variable "`halted_`" of class `Scheduler` is 1), or until all the events are executed.

Program 4.12 Instproc : : run of class simulator

```
    //~/ns/tcl/lib/ns-lib.tcl
1   Simulator instproc run  {
2       [$self get-routelogic] configure
3       $self instvar scheduler_ Node_ link_ started_
4       set started_ 1
5       foreach nn [array names Node_] {
6           $Node_($nn) reset
7       foreach qn [array names link_] {
8           set q [$link_($qn) queue]
9           $q reset
10      }
11      return [$scheduler_ run]
12  }
```

4.4.5 Instprocs of OTcl Class Simulator

The list of useful instprocs of class `Simulator` is shown below.

`now{}`	Retrieve the current simulation time.
`nullagent{}`	Retrieve the shared null agent.
`use-scheduler{type}`	Set the scheduler to be `<type>`.
`at{time stm}`	Execute the statement `<stm>` at `<time>` second.
`run{}`	Start the simulation.
`halt{}`	Terminate the simulation.
`cancel{e}`	Cancel the scheduled event `<e>`.

4.5 Chapter Summary

This chapter explains the details of event-driven simulation in NS2. The simulation is carried out by running a *Tcl simulation script*, which consists of two parts. First, the *Network Configuration Phase* establishes a network and configures all simulation components. This phase also creates a chain of events by connecting the created events chronologically. Second, the *Simulation Phase* chronologically executes (or dispatches) the created events until the Simulator is halted, or until all the events are executed.

There are four main classes involved in an NS2 simulation:

- Class `Simulator` supervises the simulation. It contains simulation components such as the Scheduler, the null agent. It also contains information storing objects which are shared by other (simulation) components.
- Class `Scheduler` maintains the chain of events and chronologically dispatches the events.

- Class `Event` consists of the firing time and the associated handler. Events are put together to form a chain of events, which are dispatched one by one by the Scheduler. Classes `Packet` and `AtEvent` are among the classes derived from class `Event`, which can be placed on the simulation timeline (i.e., in the chain of event). They are associated with different handlers and take different actions at the firing time.
- Class `Handler`: Associated with an event, a handler specifies default actions to be taken when the associated event is dispatched. Classes `NsObject` and `AtHandler` are among classes derived from class `Handler`. They are always associated with `Packet` and `AtEvent` events, respectively. Their actions are to receive a `Packet` object and to execute an OTcl statement specified in the `AtEvent` object, respectively.

4.6 Exercises

1. What are the definitions, similarities/differences, and relationship among the following NS2 components:

   a. Simulation timeline
   b. Scheduler
   c. Event
   d. Event handler
   e. Simulator
   f. Firing time or dispatching time

   Show an example to support your answer.
2. What are the similarities/differences/relationship between a TclObject and an NSObject?
3. What is a chain of events? Explain how NS2 creates a chain of events, and how NS2 locates a particular event on the chain.
4. What is event unique ID? Where does NS2 store this value? What is its data type? What is the implication when its value is positive, negative, or zero?
5. What are the two simulation phases? Explain the key objectives of each of the phases.
6. NS2 has two types of built-in events: Packet reception events and AT events. Design another type of events. Explain their purposes, show how these events can be integrated into NS2, and write an NS2 program to support your answer.
7. Explain the sequence of actions that occurs at the firing time. Use an OTcl statement execution event as an example.
8. How does NS2 start a simulation in the OTcl domain? What happens in the C++ domain after the simulation has started? When and under what condition will the simulation terminate?

9. What are four common errors associated with event scheduling in NS2? Explain the reasons and suggest general solutions.
10. What are Null events and dummy events? What are their purposes? Explain their similarities and differences. Show example usage of both types of events.
11. Write statements for the following purposes. Run NS2 to test your answer.

    a. Show the current virtual time on the screen in both C++ and OTcl domain.
    b. At 10 s, print out "Hello NS2 Users!!" on the screen. In the C++ domain, use `printf(...)` or `cout`. In the OTcl domain, use `puts{...}`).
    c. Store a `Simulator` object in a local variable `csim` in the C++ domain, and `osim` in the OTcl domain.
    d. Send a packet `*p` to an NsObject `*obj` at 15 s in future (C++ only).

Chapter 5
Network Objects: Creation, Configuration, and Packet Forwarding

NS2 is a simulation tool designed specifically for communication networks. The main functionalities of NS2 are to set up a network of connecting nodes and to pass packets from one node (which is a network object) to another.

A network object is one of the main NS2 components, which is responsible for packet forwarding. NS2 implements network objects using the polymorphism concept in object-oriented programming (OOP). Polymorphism allows network objects to take different actions ways under different contexts. For example, a `Connector` object immediately passes the received packet to the next network object, while a `Queue`[1] object enqueues the received packets and forwards only the head of the line packet.

This chapter first introduces the NS2 components by showing four major classes of NS2 components, namely, network objects, packet-related objects, simulation-related objects, and helper objects in Sect. 5.1. A part of the C++ class hierarchy, which is related to network objects, is also shown here. Section 5.2 presents class `NsObject` which acts as a template for all network objects. An example of network objects as well as packet forwarding mechanism are illustrated through class `Connector` in Sect. 5.3. Finally, the chapter summary is given in Sect. 5.4. Note that the readers who are not familiar with OOP are recommended to go through a review of the OOP polymorphism concept in Appendix B before proceeding further.

[1]Class `Queue` is a child class of class `Connector`.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_5, © Springer Science+Business Media, LLC 2012

5.1 Overview of NS2 Components

5.1.1 Functionality-Based Classification of NS2 Modules

Based on the functionality, NS2 modules (or objects) can be classified into four following types:

- *Network objects* are responsible for sending, receiving, creating, and destroying packet-related objects. Since these objects are those derived from class `NsObject`, they will be referred to hereafter as NsObjects.
- *Packet-related objects* are various types of packets which are passed around a network.
- *Simulation-related objects* control simulation timing and supervise the entire simulation. As discussed in Chap. 4, examples of simulation-related objects are events, handlers, the Scheduler, and the Simulator.
- *Helper objects* do not explicitly participate in packet forwarding. However, they implicitly help to complete the simulation. For example, a routing module calculates routes from a source to a destination, while network address identifies each of the network objects.

In this chapter, we focus only on network objects. Note that, the simulation-related objects were discussed in Chap. 4. The packet-related objects will be discussed in Chap. 8. The main helper objects will be discussed in Chap. 15.

5.1.2 C++ Class Hierarchy

This section gives an overview of C++ class hierarchies. The entire hierarchy consists of over 100 C++ classes and `struct` data types. Here, we only show a part of the hierarchy (in Fig. 5.1). The readers are referred to [18] for the complete class hierarchy.

As discussed in Chap. 3, all classes deriving from class `TclObject` form the compiled hierarchy. Classes in this hierarchy can be accessed from the OTcl domain. For example, they can be created by the global OTcl procedure "`new{...}`." Classes derived directly from class `TclObject` include network classes (e.g., `NsObject`), packet-related classes (e.g., `PacketQueue`), simulation- related classes (e.g., `Scheduler`), and helper classes (e.g., `Routing- Module`). Again, classes that do not need OTcl counterparts (e.g., classes derived from class `Handler`) form their own standalone hierarchies. These hierarchies are not a part of the compiled hierarchy nor the interpreted hierarchy.

As discussed in Chap. 4, class `Handler` specifies an action associated with an event. Again, class `Handler` contains a pure virtual function `handle(e)` (see Program 4.1). Therefore, its derived classes are responsible for providing

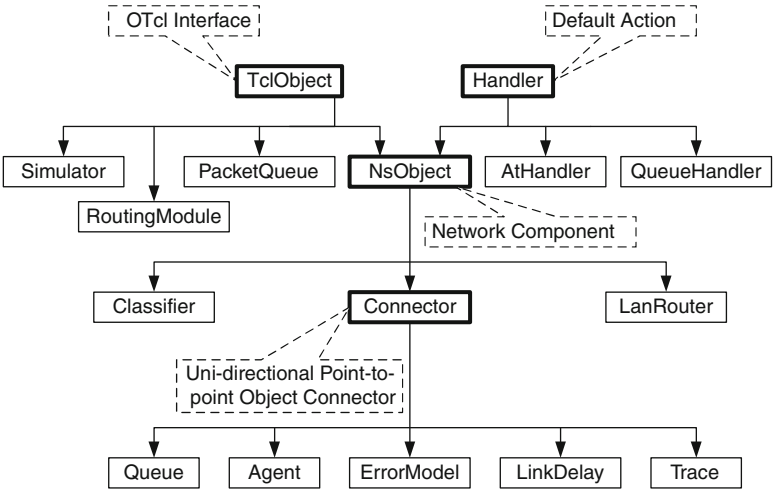

Fig. 5.1 A part of NS2 C++ class hierarchy (this chapter emphasizes on classes in *boxes* with *thick solid lines*)

implementation of the function `handle(e)`. For example, the function `handle(e)` of class `NsObject` tells the NsObject to receive an incoming packet (Program 4.2), while that of class `QueueHandler` invokes function `resume()` of the associated `Queue` object (Lines 1–4 in Program 5.1; also see Sect. 7.3.2).

Program 5.1 Function `handle(e)` of class `QueueHandler`

```
   //~/ns/queue/queue.cc
1  void QueueHandler::handle(Event*)
2  {
3      queue_.resume();
4  }
```

There are three main classes deriving from class `NsObject`: `Connector`, `Classifier`, and `LanRouter`. Connecting two NsObjects, a `Connector` object immediately forwards a received packet to the connecting NsObject (see Sect. 5.3). Connecting an NsObject to several NsObjects, a `Classifier` object classifies packets based on packet header (e.g., destination address, flow ID) and forwards the packets with the same classification to the same connecting NsObject (see Sect. 6.2). Class `LanRouter` also has multiple connecting NsObjects. However, it forwards every received packet to all connecting NsObjects.

5.2 NsObjects: A Network Object Template

5.2.1 *Class* `NsObject`

Representing NsObjects, class `NsObject` is the base class for all network objects in NS2 (see the declaration in Program 5.2). Again, the main responsibility of an NsObject is to forward packets. Therefore, class `NsObject` defines a pure virtual function `recv(p,h)` (see Line 5 in Program 5.2) as a uniform packet reception interface to force all its derived classes to implement this function.

Program 5.2 Declaration of class `NsObject`

```
    //~/ns/common/object.h
1   class NsObject : public TclObject, public Handler {
2   public:
3       NsObject();
4       virtual ~NsObject();
5       virtual void recv(Packet*, Handler* callback = 0) = 0;
6       virtual int command(int argc, const char*const* argv);
7   protected:
8       virtual void reset();
9       void handle(Event*);
10      int debug_;
11  };
```

Derived directly from class `TclObject` and `Handler` (see Program 5.2), class `NsObject` is the template class for all NS2 network objects. It inherits OTcl interfaces from class `TclObject` and the default action (i.e., function `handle(e)`) from class `Handler`. In addition, it defines a packet reception template and forces all its derived classes to provide packet reception implementation.

Function `recv(p,h)` is the very essence of packet forwarding mechanism in NS2. In NS2, an upstream object maintains a reference to the connecting downstream object. It passes a packet to the downstream object by invoking the function `recv(p,h)` of the downstream object and feeding the packet and optionally a handler as an input argument. Since NS2 focuses mainly on forwarding packets in a downstream direction, NsObjects do not need to have a reference to its upstream objects. In most cases, NsObject configuration involves downstream (not upstream) objects only.

Function `recv(p,h)` takes two input arguments: a packet "`*p`" to be received and a handler "`*h`." Most invocation of function `recv(p,h)` involves only packet "`*p`," not the handler.[2] For example, a `Queue` object (see Sect. 7.3.3) puts the received packet in the buffer and transmits the packet at the head of the buffer.

[2]We will discuss the *callback* mechanism which involves a handler in Sect. 7.3.3.

An ErrorModel object (see Sect. 15.3) imposes error probability on the received packet and forwards the packet to the connecting object if the transmission is not in error.

5.2.2 Packet Forwarding Mechanism of NsObjects

An NsObject forwards packets in two following ways:

- *Immediate packet forwarding*: To forward a packet to a downstream object, an upstream object needs to obtain a reference (e.g., a pointer) to the downstream object and invokes function `recv(p,h)` of the downstream object through the obtained reference. For example, a Connector (see Sect. 5.3) has a pointer "`target_`" to its downstream object. Therefore, it forwards a packet to its downstream object by executing `target_->recv(p,h)`.
- *Delayed packet forwarding*: To delay packet forwarding, a `Packet` object is cast to be an `Event` object, associated with a packet receiving NsObject, and placed on the simulation timeline at a given simulation time. At the firing time, the function `handle(e)` of the NsObject will be invoked, and the packet will be received through function `recv(p,h)` (see an example of delayed packet forwarding in Sect. 5.3).

5.3 Connectors

As shown in Fig. 5.2, a Connector is an NsObject which connects three NsObjects in a unidirectional manner. It receives a packet from an upstream NsObject. By default, a Connector immediately forwards the received packet to its downstream NsObject. Alternatively, it can drop the packet by forwarding the packet to a packet dropping object.[3]

From Fig. 5.2, a Connector is interested in specifying its downstream NsObject and packet dropping NsObject only. The connection from an upstream object to a Connector, on the other hand, is configured by the upstream object, not by the connector.

[3] A packet dropping network object (e.g., a null agent) is responsible for destroying packets.

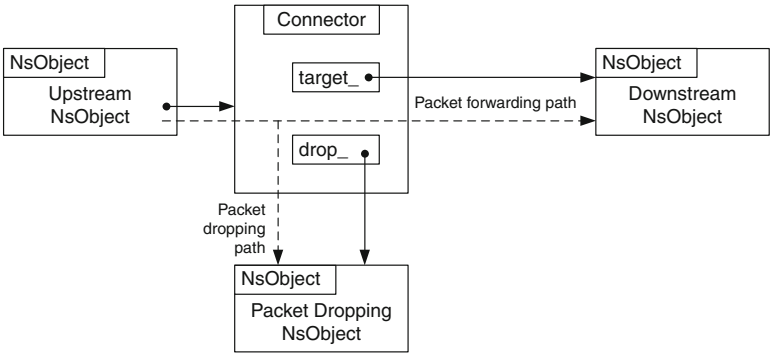

Fig. 5.2 Diagram of a connector: The *solid arrows* represent pointers, while the *dotted arrows* show packet forwarding and dropping paths

Program 5.3 Declaration and function `recv(p,h)` of class `Connector`

```
    //~/ns/common/connector.h
 1  class Connector : public NsObject {
 2  public:
 3      Connector();
 4      inline NsObject* target() { return target_; }
 5      void target (NsObject *target) { target_ = target; }
 6      virtual void drop(Packet* p);
 7      void setDropTarget(NsObject *dt) {drop_ = dt; }
 8  protected:
 9      virtual void drop(Packet* p, const char *s);
10      int command(int argc, const char*const* argv);
11      void recv(Packet*, Handler* callback = 0);
12      inline void send(Packet* p, Handler* h){target_->recv
        (p, h);}
13
14      NsObject* target_;
15      NsObject* drop_;    // drop target for this connector
16  };

    //~/ns/common/connector.cc
17  void Connector::recv(Packet* p, Handler* h){send(p, h);}
```

5.3.1 Class Declaration

Program 5.3 shows the declaration of class `Connector`. Class `Connector` contains two pointers (Lines 14 and 15 in Program 5.3) to `NsObjects`[4]: "`target_`"

[4]Since class `Connector` contains two pointers to abstract object (i.e., class `NsObject`), it can be regarded as an abstract user class for class composition discussed in Sect. B.8. We will discuss the details of how the class composition concept applies to a Connector in the next section.

and "drop_." From Fig. 5.2, "target_" is the pointer to the connecting downstream NsObject, while "drop_" is the pointer to the packet dropping object.

Class Connector derives from the abstract class NsObject. It overrides the pure virtual function recv(p,h), by simply invoking function send(p,h) (see Line 12 in program 5.3). Function send(p,h) simply forwards the received packet to its downstream object by invoking function recv(p,h) of the downstream object (i.e., target_->recv(p,h) in Line 12).

Program 5.4 Function drop of class connector

```
    //~/ns/common/connector.cc
1   void Connector::drop(Packet* p)
2   {
3       if (drop_ != 0)
4           drop_->recv(p);
5       else
6           Packet::free(p);
7   }
```

Program 5.4 shows the implementation of function drop(p), which drops or destroys a packet. Function drop(p) takes one input argument, which is a packet to be dropped. If the dropping NsObject exists (i.e., "drop_"$\neq$ 0), this function will forward the packet to the dropping NsObject by invoking drop_->recv(p,h). Otherwise, it will destroy the packet by executing "Packet::free(p)" (see Chap. 8). Note that function drop(p) is declared as virtual (Line 9). Hence, classes derived from class Connector may override this function without any function ambiguity.[5]

5.3.2 OTcl Configuration Commands

As discussed in Sect. 4.1, NS2 simulation consists of two phases: Network Configuration Phase and Simulation Phase. In the Network Configuration Phase, a Connector is set up as shown in Fig. 5.2. Again, a Connector configures its downstream and packet dropping NsObjects only.

Suppose OTcl has instantiated three following objects: a Connector object (conn_obj), a downstream object (down_obj), and a dropping object (drop_obj). Then, the Connector is configured using the following two OTcl commands (see Program 5.5):

[5]Function ambiguity is discussed in Appendix B.2.

Program 5.5 OTcl commands `target` and `drop-target` of class `Connector`

```
    //~/ns/common/connector.cc
 1  int Connector::command(int argc, const char*const* argv)
 2  {
 3      Tcl& tcl = Tcl::instance();
 4      if (argc == 2) {
 5          if (strcmp(argv[1], "target") == 0) {
 6              if (target_ != 0)
 7                  tcl.result(target_->name());
 8              return (TCL_OK);
 9          }
10          if (strcmp(argv[1], "drop-target") == 0) {
11              if (drop_ != 0)
12                  tcl.resultf("%s", drop_->name());
13              return (TCL_OK);
14          }
15      }
16      else if (argc == 3) {
17          if (strcmp(argv[1], "target") == 0) {
18              if (*argv[2] == '0') {
19                  target_ = 0;
20                  return (TCL_OK);
21              }
22              target_ = (NsObject*)TclObject::lookup(argv[2]);
23              if (target_ == 0) {
24                  tcl.resultf("no such object %s", argv[2]);
25                  return (TCL_ERROR);
26              }
27              return (TCL_OK);
28          }
29          if (strcmp(argv[1], "drop-target") == 0) {
30              drop_ = (NsObject*)TclObject::lookup(argv[2]);
31              if (drop_ == 0) {
32                  tcl.resultf("no object %s", argv[2]);
33                  return (TCL_ERROR);
34              }
35              return (TCL_OK);
36          }
37      }
38      return (NsObject::command(argc, argv));
39  }
```

- OTcl command `target` with one input argument conforms to the following syntax:

  ```
  $conn_obj target $down_obj
  ```

  This command casts the input argument `down_obj` to be of type `NsObject*` and stores it in the variable "`target_`" (Line 22).

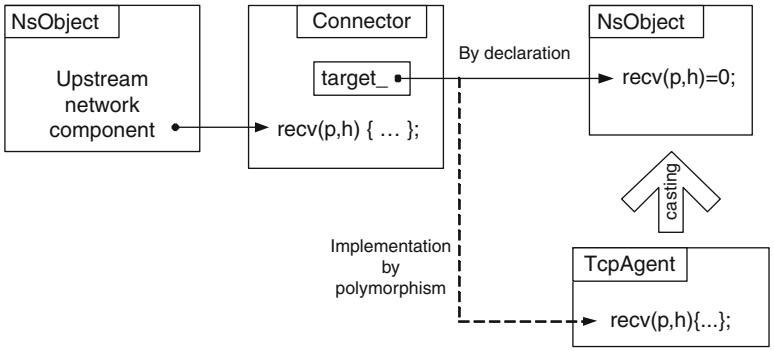

Fig. 5.3 A polymorphism implementation of a connector: A connector declares `target_` as an `NsObject` pointer. In the network configuration phase, the OTcl command `target` is invoked to setup a downstream object of the Connector, and the NsObject `*target_` is cast to a `TcpAgent` object

- OTcl command `target` with no input argument (e.g., `$conn_obj target`) returns OTcl instance corresponding to the C++ variable "`target_`" (Line 5–9). Note that function `name()` of class `TclObject` returns the OTcl reference string associated with the input argument.
- OTcl command `drop-target` with one input argument is very similar to that of the OTcl command `target` but the input argument is cast and stored in the variable "`drop_`" instead of the variable "`target_`."
- OTcl command `drop-target` with no input argument is very similar to that of the OTcl command `target` but it returns the OTcl instance corresponding to the variable "`drop_`" instead of the variable "`target_`."

Example 5.1. Consider the connector configuration in Figs. 5.2–5.3. Let the downstream object be of class `TcpAgent`, which corresponds to class `Agent/Tcp` in the OTcl domain. Also, let a `Agent/Null` object be a packet dropping NsObject. The following program shows how the network is set up from the OTcl domain:

```
set conn_obj [new Connector]
set tcp [new Agent/TCP]
set null [new Agent/Null]

$conn_obj target $tcp
$conn_obj drop-target $null
```

The first three lines create a Connector (`conn`), a TCP object (`tcp`), and a packet dropping object (`null`). The last two lines use the OTcl commands `target` and `drop-target` to set "`tcp`" and "`null`" as the downstream object and the dropping object of the Connector, respectively. □

Connector configuration complies with the class composition programming concept discussed in Appendix B.8. Table 5.1 shows the components in Example 5.1

Table 5.1 Class composition of network components in Example 5.1

Abstract class	`NsObject`
Derived class	`Agent/Tcp` and `Agent/Null`
Abstract user class	`Connector`
User class	A Tcl simulation script

and the corresponding class composition. Classes `Agent/TCP` and `Agent/Null` are OTcl classes whose corresponding C++ classes derive from class `NsObject`. Class `Connector` stores pointers (i.e., "`target_`" and "`drop_`") to NsObjects, and is therefore considered to be an abstract user class. Finally, as a user class, the Tcl Simulation Script instantiates NsObjects `tcp`, and `null` from classes `Agent/Tcp`, and `Agent/Null`, respectively, and binds `tcp` and `null` to variables `target_` and `drop_`, respectively.

When invoking `target` and `drop-target`, `tcp` and `null` are first typecast to `NsObject` pointers. Then they are assigned to pointers `target_` and to `drop_`, respectively. Functions `recv(p,h)` of both `tcp` and `null` are associated with class `Agent/TCP` and `Agent/Null`, respectively, since they both are virtual functions.

5.3.3 Packet Forwarding Mechanism of Connectors

From Sect. 5.2.2, an NsObject forwards a packet in two ways: immediate and delayed packet forwarding. This section demonstrates both the packet forwarding mechanisms through a Connector.

5.3.3.1 Immediate Packet Forwarding

Immediate packet forwarding is carried out by invoking function `recv(p,h)` of a downstream object. In Example 5.1, the Connector forwards a packet to the TCP object by invoking function `recv(p,h)` of the TCP object (i.e., `target_->recv(p,h)`, where `target_` is configured to point to a TCP object). C++ polymorphism is responsible for associating the function `recv(p,h)` to class `Agent/TCP` (i.e., the construction type), not class `NsObject` (i.e., the declaration type).

5.3.3.2 Delayed Packet Forwarding

Delayed packet forwarding is implemented with the aid of the Scheduler. Here, a packet is cast to an event, associated with a receiving NsObject, and placed on the simulation timeline. For example, to delay packet forwarding in Example 5.1 by "`d`" seconds, we may invoke the following statement instead of `target_->recv(p,h)`.

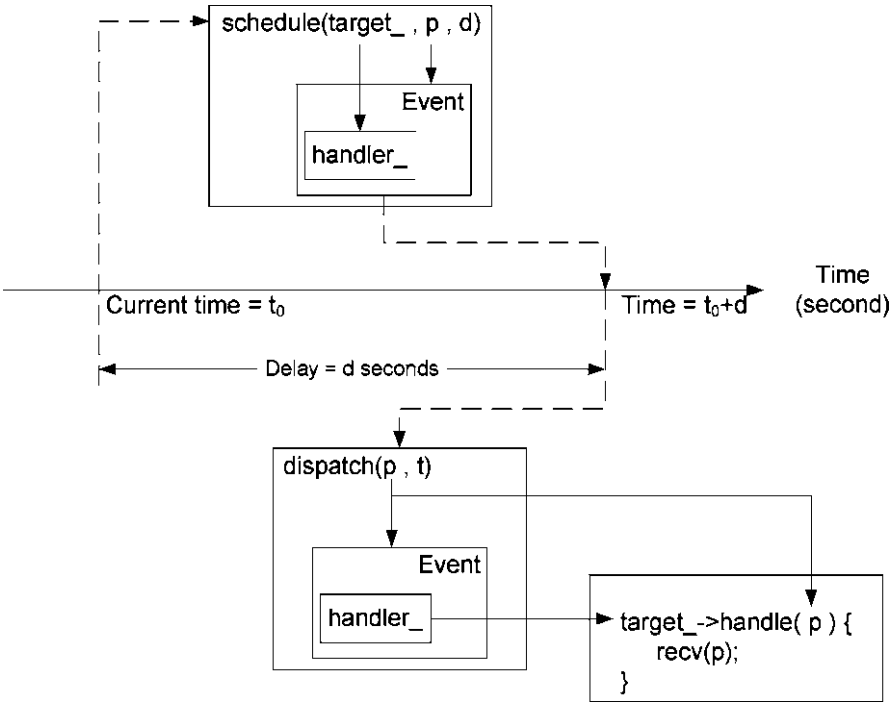

Fig. 5.4 Delayed packet forwarding mechanism

```
Scheduler& s = Scheduler::instance();
s.schedule(target_, p, d);
```

Consider Fig. 5.4 and Program 5.6 altogether. Figure 5.4 shows the diagram of delayed packet forwarding, while Program 5.6 shows the details of functions `schedule(h,e,delay)` as well as `dispatch(p,t)` of class `Scheduler`. The statement "`schedule(target_, p, d)`" casts packet `*p` and the NsObject `*target_` into `Event` and `Handler objects`, respectively (Line 1 of Program 5.6). Line 5 of Program 5.6 associates the packet `*p` with the NsObject `*target_`. Lines 6 and 7 insert the packet `*p` into the simulation timeline at the appropriate time. At the firing time, the event (`*p`) is dispatched (Lines 9–14). The Scheduler invokes function `handle(p)` of the handler associated with event `*p`. In this case, the associated handler is the NsObject `*target_`. Therefore, in Line 13, the default action `handle(p)` of "`target_`", invokes function `recv(p,h)` to receive the scheduled packet (see Program 4.2).

Program 5.6 Functions `schedule` and `dispatch` of class `Scheduler`

```
    //~/ns/common/scheduler.cc
1   void Scheduler::schedule(Handler* h, Event* e, double delay)
2   {
3       ...
4       e->uid_ = uid_++;
5       e->handler_ = h;
6       e->time_ = clock_ + delay;
7       insert(e);
8   }

9   void Scheduler::dispatch(Event* p, double t)
10      ...
11      clock_ = t;
12      p->uid_ = -p->uid_; // being dispatched
13      p->handler_->handle(p); // dispatch
14  }
```

5.4 Chapter Summary

Referred to as an NsObject, a network object is responsible for sending, receiving, creating, and destroying packets. As an object of class `NsObject`, it derives OTcl interfaces from class `TclObject` and the default action (i.e., function `handle(e)`) from class `Handler`. It defines a pure virtual function `recv(p,h)` as a uniform packet reception interface for the derived classes. Based on the polymorphism concept, the derived classes must provide their own implementation of how to receive a packet.

In NS2, an NsObject needs to create a connection to its downstream object only. Normally, an NsObject forwards a packet to a downstream object by invoking function `recv(p,h)` of its downstream object. In addition, an NsObject can defer packet forwarding by associating a packet to the downstream object and inserting the packet on the simulation timeline. At the firing time, the scheduler dispatches the packet, and the default action of the downstream object is invoked to receive the packet.

As an example, we show the details of class `Connector`, one of the main NsObject classes in NS2. Class `Connector` contains two pointers to NsObjects: "`target_`" pointing to a downstream object and "`drop_`" pointing to a packet dropping object. To configure a Connector, an object whose class derives from class `NsObject` can be set as downstream and dropping objects via OTcl command `target{...}` and `drop-target{...}`, respectively. These two OTcl commands cast the downstream and dropping objects to NsObjects, and assign them to C++ variables `*target_` and `*drop_`, respectively.

5.5 Exercises

1. What are the four types of NS2 objects? Explain their roles and differences among them.
2. Class `NsObject` contains a pure virtual function. What is the name of the function? Give a general description of the function. Why does it have to be declared as pure virtual?
3. What is the function which is central to packet reception mechanism?
4. What are the two packet reception methods? Explain their purposes and how they are implemented in NS2. Formulate an example from class `Connector` to show the process in time sequence.
5. Demonstrate how a packet is dropped in the C++ domain. Can you drop a packet from within any C++ class? Explain your answer via an example C++ class.

Chapter 6
Nodes as Routers or Computer Hosts

This chapter focuses on a basic network component, *Node*. In NS2, a Node acts as a computer host (e.g., a source or a destination) and a router (e.g., an intermediate node). It receives packets from an attached application or an upstream object, and forwards them to the attached links specified in the routing table (as a router) or delivers them from/to transport layer agents (as a host).

In the following, we first give an overview of routing mechanism and Nodes in Sect. 6.1. Sections 6.2–6.4 discuss three main routing components: classifiers, routing modules, and route logic, respectively. In Sect. 6.5, we show how the aforementioned Node components are assembled to compose a Node. Finally, the chapter summary is provided in Sect. 6.6.

6.1 An Overview of Nodes in NS2

6.1.1 Routing Concept and Terminology

In NS2, routing has a broader definition than that usually used in practice. Routing usually refers to a network layer operation which determines the route along which a packet should be forwarded to its destination. In NS2, routing is an act of forwarding a packet from one NsObject to another. It can occur within a Node (i.e., no communication), from a Node to a link (i.e., network layer), between a Node and an agent (i.e., transport layer), and so on. In order to avoid confusion, let us define the following terminologies:

- *Routing mechanism*: An act of determining and passing packets according to predefined routing rules
- *Routing rule or route entry*: A rule which determines where a packet should be forwarded to; it is usually expressed in the form of `(dst,target)` – meaning that packets destined for "`dst`" should be forwarded to "`target`."
- *Routing table*: A collection of routing rules

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_6, © Springer Science+Business Media, LLC 2012

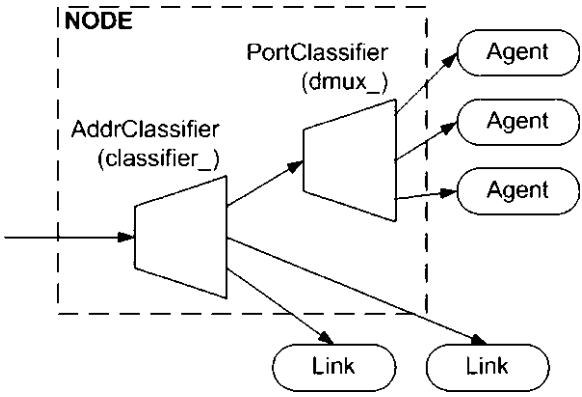

Fig. 6.1 Node architecture

- *Routing algorithm*: An algorithm which computes routing rules (e.g., Dijkstra algorithm [19])
- *Routing protocol*: A communication protocol designed to update the routing rules according to dynamic environment (e.g., Ad hoc On-demand Distance Vector (AODV) [24])
- *Routing agent*: An entity which gathers parameters (e.g., network topology) necessary to compute routing rules.
- *Route logic*: An NS2 component which runs the routing algorithm (i.e., computing routing rules)
- *Router*: An entity which run routing mechanism; in NS2, this entity is an *address classifier*.
- *Routing module*: A single point of management, which manages a group of classifiers

This chapter focuses on static routing, which involves the following main NS2 components: Nodes, classifiers, routing modules, route logic.

6.1.2 Architecture of a Node

A Node is an OTcl composite object whose architecture is shown in Fig. 6.1. Nodes are defined in an OTcl class `Node`, which is bound to C++ class with the same name. A Node consists of two main components: an address classifier (instvar `classifier_`) and a port classifier (instvar `dmux_`). These two components have one entry point and multiple forwarding targets. An address classifier acts as a router which receives a packet from an upstream object and forwards the packet to one of its connecting links based on the address embedded in packet header. A port classifier acts as a transport layer bridge – taking a packet from the address classifier (in case that the packet is destined to this particular node), and forwarding the packet to one of the attached transport layer agents.

6.1.3 Default Nodes and Node Configuration Interface

A default NS2 Node is based on flat-addressing and static routing. With flat-addressing, an address of every new node is incremented by one from that of the previously created node. Static routing assumes no change in topology. The routing table is computed once at the beginning of the Simulation phase and does not change thereafter. By default, NS2 uses the Dijkstra's shortest path algorithm [19] to compute optimal routes for all pairs of Nodes. Again, this chapter focuses on Nodes with flat-addressing and static routing only. The details about other routing protocols as well as hierarchical addressing can be found in the NS manual [17].

To provide a default Node with more functionalities such as link layer or Medium Access Control (MAC) protocol functionalities, we may use the instproc `node-config` of class `Simulator` whose syntax is as follows:

```
$ns node-config -<option> [<value>]
```

where `$ns` is the `Simulator` object.

An example use of the instproc `node-config{args}` for the default setting is shown below:

```
$ns_ node-config -addressType   flat
                 -adhocRouting
                 -llType
                 -macType
                 -propType
                 -ifqType
                 -ifqLen
                 -phyType
                 -antType
                 -channel
                 -channelType
                 -topologyInstance
```

By default, almost every option is specified as `NULL` with the exception of `addressType`, which is set to be flat addressing. The instproc `node-config` has an option `reset`, i.e.,

```
$ns node-config -reset
```

which is used to restore default parameter setting. The details of instproc `node-config` (e.g., other options) can be found in the file *~ns*/tcl/lib/ns-lib.tcl and [17].

Note that this instproc does not immediately configure the Nodes as specified in the `<option>`. Instead, it stores `<value>` in the instvars of the `Simulator` corresponding to `<option>`. This stored configuration will be used during a Node construction process. As a result, the instproc `node-config` must be executed before Node construction.

6.2 Classifiers: Multi-Target Packet Forwarders

A classifier is a packet forwarding object with multiple connecting targets. It classifies incoming packets according to a predefined criterion (e.g., destination address or transport layer port). Packets with the same category are forwarded to the same NsObject.

NS2 implements classifiers using the concept of *slots*. A slot is a placeholder for a pointer to an NsObject. It is associated with a packet category. When a packet arrives, a classifier determines the packet category and forwards the packet to the NsObject whose pointer was installed in the associated slot.

In the following, we shall discuss the details of two main processes of classifiers: configuration and internal mechanism. Configuration defines what the users ask a classifier to perform. It includes the following main steps:

1. Define the categories
2. Identify a corresponding slot as well as a forwarding NsObject for each category
3. Install the NsObject pointer in the selected slot

Internal mechanism is what a classifier does to carry out the requirement provided by users. It usually begins with the C++ function `recv(p,h)`.

For example, suppose we would like to attach a node to a transport layer agent at the port number 50. In the configuration, we install the agent in slot number 50. The internal mechanism is to tell the classifier the following: send all the packets whose port number is 50 to the NsObject whose pointer is in the slot number 50.

6.2.1 Class `Classifier` and Its Main Components

NS2 implements classifiers in a C++ class `Classifier` (see the declaration in Program 6.1), which is bound to an OTcl class with the same name. The main components of a classifier include the following.

6.2.1.1 C++ Variables

The C++ class `Classifier` has two key variables: `slot_` and `default_target_` (Lines 13 and 14 in Program 6.1). The variable `slot_` is a link list whose entries are pointers to downstream NsObjects. Each of these NsObjects corresponds to a predefined criterion. Packets matching with a certain criterion are forwarded to the corresponding NsObject. The variable `default_target_` points to a downstream NsObject for packets which do not match with any predefined criterion.

Program 6.1 Declaration of class `Classifier`

```
     //~/ns/classifier/classifier.h
 1   class Classifier : public NsObject {
 2   public:
 3       Classifier();
 4       virtual ~Classifier();
 5       virtual void recv(Packet* p, Handler* h);
 6       virtual NsObject* find(Packet*);
 7       virtual int classify(Packet *);
 8       virtual void clear(int slot);
 9       virtual void install(int slot, NsObject*);
 10      inline int mshift(int val) {return((val >> shift_) &
           mask_);}
 11  protected:
 12      virtual int command(int argc, const char*const* argv);
 13      NsObject** slot_;
 14      NsObject *default_target_;
 15      int shift_;
 16      int mask_;
 17 };
```

The class `Classifier` also have two supplementary variables: `shift_` (Line 15) and `mask_` (Line 16). These two variables are used in function `mshift (val)` (Line 10) to reformat the address (see also Sect. 15.4).

6.2.1.2 C++ Functions

The main C++ functions of class `Classifier` are shown below:

Configuration Functions

`install(slot,p)`	Store the input `NsObject` pointer "`p`" in the slot number "`slot`'.
`install_next(node)`	Install the `NsObject` pointer "`node`" in the next available slot.
`do_install (dst,target)`	Similar to `install{slot,p}` but the input parameter `dst` is a string instead of an integer.
`clear(slot)`	Remove the NsObject pointer installed in the slot number "`slot`."
`mshift(val)`	Shift `val` to the left by "`shift_`" bits. Masks the shifted value using a logical AND (`&`) operation with "`mask_`."

Packet Forwarding (i.e., Internal) Functions

`recv(p,h)`	Receive a packet `*p` and handler `*h`.
`find(p)`	Return a forwarding NsObject pointer for an incoming packet `*p`.
`classify(p)`	Return a slot number whose associated criterion matches with the header of an incoming packet `*p`.

6.2.1.3 Main Configuring Interface

C++ Functions

Program 6.2 shows the details of key C++ configuration functions. Function `install(slot,p)` stores the input `NsObject` pointer "`p`" in the slot number "`slot`" of the variable "`slot_`" (Line 5). Function `install_next(node)` installs the input `NsObject` pointer "`node`" in the next available slot (Lines 10 and 11). Function `do_install(dst,target)` converts "`dst`" to be an integer variable (Line 21), and installs the NsObject pointer "`target`" in the slot corresponding to "`dst`" (Line 22). Finally, function `clear(slot)` removes the installed `NsObject` pointer from the slot number "`slot`" of the variable "`slot_`" (Line 16).

OTcl Commands

Class `Classifier` also defines the following key OTcl commands in a C++ function `command(...)` of class `Classifier` (in the file *~ns*/classifier/classifier.cc).

`slot{index}`	Return the NsObject stored in the slot number `index`
`clear{slot}`	Clear the NsObject pointer installed in the slot number `slot`.
`install{index object}`	Install `object` in the slot number `index`.
`installNext{object}`	Install `object` in the next available slot.
`defaulttarget{object}`	Store `object` in the C++ variable `default_target_`.

6.2.1.4 Main Internal Mechanism

As an NsObject, a classifier receives a packet by having its upstream object invoke its function `recv(p,h)`, passing a packet pointer "`p`" and a handler pointer "`h`" as

Program 6.2 Functions `install`, `install_next`, `clear`, and `do_install` of class `Classifier`

```
    //~ns/classifier/classifier.cc
 1  void Classifier::install(int slot, NsObject* p)
 2  {
 3      if (slot >= nslot_)
 4          alloc(slot);
 5      slot_[slot] = p;
 6      if (slot >= maxslot_)
 7          maxslot_ = slot;
 8  }

 9  int Classifier::install_next(NsObject *node) {
10      int slot = maxslot_ + 1;
11      install(slot, node);
12      return (slot);
13  }

14  void Classifier::clear(int slot)
15  {
16      slot_[slot] = 0;
17      if (slot == maxslot_)
18          while (--maxslot_ >= 0 && slot_[maxslot_] == 0);
19  }

    //~ns/classifier/classifier.h
20  virtual void do_install(char* dst, NsObject *target) {
21      int slot = atoi(dst);
22      install(slot, target);
23  }
```

input arguments. In Program 6.3, Line 3 determines a forwarding NsObject "`node`" for an incoming packet `*p`, by invoking function `find(*p)`. Then, Line 8 passes the packet pointer "`p`" and the handler pointer "`h`" to its forwarding NsObject `*node` by executing `node->recv(p,h)`.

Function `find(p)` (Lines 10–18 in Program 6.3) examines the incoming packet `*p` and retrieves the matched `NsObject` pointer installed in the variable `slot_`. Line 13 invokes function `classify(p)` to retrieve the slot number (i.e., the variable `cl`) corresponding to the packet `*p`. Then, Lines 14 and 17 return the `NsObject` pointer (i.e., `node`) stored in the slot number `cl` of the variable `slot_`.

Function `classify(p)` is perhaps the most important function of a classifier. This is the place where the classification criterion is defined. Function `classify(p)` returns the slot number which matches with the input packet `*p` under the predefined criterion. Since the classification criteria could be different for different types of classifiers, function `classify(p)` is usually

Program 6.3 Functions `recv` and `find` of class `Classifier`

```
     //~/ns/classifier/classifier.cc
 1   void Classifier::recv(Packet* p, Handler* h)
 2   {
 3      NsObject* node = find(p);
 4      if (node == NULL) {
 5          Packet::free(p);
 6          return;
 7      }
 8      node->recv(p,h);
 9   }

10   NsObject* Classifier::find(Packet* p)
11   {
12      NsObject* node = NULL;
13      int cl = classify(p);
14      if (cl < 0 || cl >= nslot_ || (node = slot_[cl]) == 0) {
15          /*There is no potential target in the slot;*/
16      }
17      return (node);
18    }
```

Program 6.4 Function `classify` of class `PortClassifier`

```
    //~ns/classifier/classifier-port.cc
1 int PortClassifier::classify(Packet *p)
2 {
3   hdr_ip* iph = hdr_ip::access(p);
4   return iph->dport();
5 }
```

overridden in the derived classes of class `Classifier`. In Sects. 6.2.2 and 6.2.3, we show two example implementations of function `classify(p)` in classes `PortClassifier` and `DestHashClassifier`, respectively.

6.2.2 Port Classifiers

Derived from class `Classifier`, class `PortClassifier` classifies packets based on the destination port. From Lines 3 and 4 in Program 6.4, function `classify(p)` returns the destination port number of the IP header of the incoming packet `*p`.

A port classifier is used as a demultiplexer which bridges a node to receiving transport layer agents. It determines the transport layer port number stored in the header of the received packet `*p`. Suppose the port number is `cl`. Then the

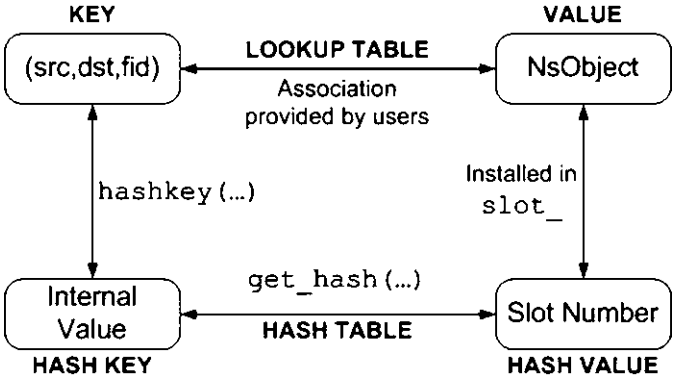

Fig. 6.2 Hash terminology and relevant functions of class `HashClassifier`

packet is forwarded to the NsObject associated with `slot_[cl]`. By installing a pointer to a receiving transport layer agent in `slot_[cl]`, the classifier forwards packets whose destination port is "`cl`" to the associated agent. The details of how a port classifier bridges a Node to a transport layer agent will be discussed later in Sect. 6.5.3.

6.2.3 Hash Classifiers

From Fig. 6.1, another important classifier in a Node is address classifier. In NS2, address classifiers are implemented in so-called hash classifiers.

6.2.3.1 An Overview of Hash Classifiers

Hash table is a data structure which facilitates a key-value lookup process.[1] The lookup process is facilitated by hashing the key into a readily manageable form. The results are stored in a so-called hash-table. The lookup is carried out over the hash table instead of the original table to expedite the lookup process.

Before proceeding further, let us introduce the following hashing terminologies. In this respect, consider, as an example, a hash classifier which classifies packets based on three input parameters: flow ID, source address, and destination address in Fig. 6.2.

[1]Suppose we have a table which associates keys and values. Given a key, the lookup process searches in the table for the matched key, and returns the corresponding value.

- *A key*: Keywords we would like to find (e.g., flow ID, source address, and destination address)
- *A value*: An entry paired with a key (e.g., a pointer to an NsObject)
- *A hash function*: A function which hashes (i.e., transforms) a key into a hash key
- *A hash key*: A transformed key; a lookup process will search over hash keys, rather than the original keys.
- *A hash value*: An entry paired with a hash key (e.g., index of the variable `slot_`)
- *A lookup table*: A table consists of (key,value) pairs.
- *A hash table*: A table consists of (hash-key, hash-value) pairs.
- *A record (or an entry)*: A pair of (key,value)
- *A hash record (or a hash entry)*: A pair of (hash-value, hash-value)[2]

Address classifiers classify packets based on the destination address. In this respect, an address and an NsObject are viewed as a key and a value, respectively. A hash classifier hashes an address into a hash key (internal to NS2), which is associated with a hash value (i.e., the slot number in which the NsObject is installed) by the underlying hash table. When receiving a packet, an address classifier looks up the slot number from the hash table, rather than the original lookup table. This eliminates the need to compare records one by one and greatly expedites the lookup process.

6.2.3.2 C++ Implementation of Class `HashClassifier`

The hash classifiers classify packets based on one or more of the following criteria: flow ID, source address, and destination address. NS2 defines a C++ class `HashClassifier` as a template. All the helper functions are defined here, but the key function `classify(p)`, which defines packet classification criteria, is defined by its derived classes.

Program 6.5 shows the details of a C++ class `HashClassifier` which is mapped to an OTcl class `Classifier/Hash`. Class `HashClassifier` has three main variables. First, variable `default_` (Line 15) contains the default slot for a packet which does not match with any entry in the table. Second, variable `ht_` (Line 16) is the hash table. Finally, variable `keylen_` (Line 17) is the number of components in a key. By default, a key consists of flow ID, source address, and destination address, and the value of `keylen_` is 3.

The key functions of class `HashClassifier` are shown below (see also Fig. 6.2):

[2]Since a record and a hash record have one-to-one relationship, we shall use these two terms interchangeably.

Program 6.5 Declaration of class `HashClassifier`

```
    //~ns/classifier/classifier-hash.h
 1  class HashClassifier : public Classifier {
 2  public:
 3      HashClassifier(int keylen): default_(-1),
          keylen_(keylen);
 4      ~HashClassifier();
 5      virtual int classify(Packet *p);
 6      virtual long lookup(Packet* p) ;
 7      void set_default(int slot) { default_ = slot; }
 8  protected:
 9      long lookup(nsaddr_t src, nsaddr_t dst, int fid);
10      void reset();
11      int set_hash(nsaddr_t src, nsaddr_t dst, int fid, long
          slot);
12      long get_hash(nsaddr_t src, nsaddr_t dst, int fid);
13      virtual int command(int argc, const char*const* argv);
14      virtual const char* hashkey(nsaddr_t, nsaddr_t, int)=0;
15      int default_;
16      Tcl_HashTable ht_;
17      int keylen_;
18  };
```

`lookup(p)`	Return the slot number which matches with the incoming packet `p`.
`lookup(src,...dst,fid)`	Return the slot number whose corresponding source address, destination address, and flow ID are `src`, `dst`, and `fid`, respectively.
`set_hash(src,...dst,fid,slot)`	Hash the key (`src,dst,fid`) into a hash key, and associates the hash key with the slot number `slot`.
`get_hash(src,...dst,fid)`	Return the slot number which matches with the key (`src,dst,fid`).
`hashkey(src,...)dst,fid)`	Return a hash key for the input key (`src,dst,fid`). This function is pure virtual and should be overridden by child classes of class `HashClassifier`.

Program 6.6 shows the details of functions `lookup(p)` and `get_hash(src, dst,fid)` of class `HashClassifier`. Function `lookup(p)` retrieves a key associated with the packet `*p`. It then asks the function `get_hash(...)` for the corresponding hash value (i.e., slot number).

In Line 6, function `get_hash(...)` invokes function `hashkey(...)` to determine the hash key corresponding to the input key (`src,dst,fid`). Then, function `Tcl_FindHashEntry` (...) locates the hash record in the hash table which matches with the hash key. If the record was found, function `Tcl_GetHash Value(ep)` will retrieve and return the corresponding hash value (i.e., slot

Program 6.6 Functions `lookup` and `get_hash` of class `HashClassifier`

```
    //~ns/classifier/classifier-hash.cc
1   long HashClassifier::lookup(Packet* p) {
2       hdr_ip* h = hdr_ip::access(p);
3       return get_hash(mshift(h->saddr()),mshift(h->daddr()),
                                                    h->flowid());
4   }

5   long HashClassifier::get_hash(nsaddr_t src,
                                  nsaddr_t dst, int fid) {
6       Tcl_HashEntry *ep= Tcl_FindHashEntry(&ht_,
                                     hashkey(src, dst, fid));
7       if (ep)
8           return (long)Tcl_GetHashValue(ep);
9       return -1;
10  }
```

number) to the caller. Note that the function `hashkey(...)` is declared as pure virtual in class `HashClassifier` and must be overridden by the child classes of class `HashClassifier`.

6.2.3.3 Child Classes of Class `HashClassifier`

Class `HashClassifier` has four major child classes (class names on the left and right are compiled and interpreted classes, respectively):

- `DestHashClassifier` ⇔ `Classifier/Hash/Dest`: classifies packets based on the destination address.
- `SrcDestHashClassifier` ⇔ `Classifier/Hash/SrcDest`: classifies packets based on source and destination addresses.
- `FidHashClassifier` ⇔ `Classifier/Hash/Fid`: classifies packets based on a flow ID.
- `SrcDestFidHashClassifier` ⇔ `Classifier/Hash/SrcDestFid`: classifies packets based on source address, destination address, and flow ID.

6.2.3.4 C++ Class `DestHashClassifier`

As an example, consider class `DestHashClassifier` (Program 6.7), a child class of class `HashClassifier`, which classifies incoming packets by the destination address only. Class `DestHashClassifier` overrides functions `classify(p)`, `do_install(dst,target)`, and `hashkey(...)`, and uses other functions (e.g., `lookup(p)`) of class `HashClassifier` (i.e., its parent class).

Program 6.7 Declaration of class `DestHashClassifier`

```
    //~ns/classifier/classifier-hash.h
1   class DestHashClassifier : public HashClassifier {
2   public:
3       DestHashClassifier() : HashClassifier(TCL_ONE_WORD_KEYS)
          {}
4       virtual int command(int argc, const char*const* argv);
5       int classify(Packet *p);
6       virtual void do_install(char *dst, NsObject *target);
7   protected:
8       const char* hashkey(nsaddr_t, nsaddr_t dst, int) {
9           long key = mshift(dst);
10          return (const char*) key;
11      }
12  };
```

Program 6.8 Functions `classify` and `do_install` of class `DestHash Classifier`

```
    //~ns/classifier/classifier-hash.cc
1   int DestHashClassifier::classify(Packet * p) {
2       int slot = lookup(p);
3       if (slot >= 0 && slot <=maxslot_)
4           return (slot);
5       else if (default_ >= 0)
6           return (default_);
7       else return (-1);
8   }

9   void DestHashClassifier::do_install(char* dst, NsObject
      *target) {
10      nsaddr_t d = atoi(dst);
11      int slot = getnxt(target);
12      install(slot, target);
13      if (set_hash(0, d, 0, slot) < 0)
14         /* show error */
15  }
```

Program 6.8 shows the implementation of function `classify(p)` of class `DestHashClassifier`. This function obtains a matching slot number "`slot`" by invoking `lookup(p)` (Line 2; See also Fig. 6.3), and returns "`slot`" if it is valid (Line 4). Otherwise, Line 6 will return the variable "`default_`."[3] If neither `slot` nor `default_` is valid, Line 7 will return –1, indicating no matching entry in the hash table.

[3]The variable "`default_`" contains the default slot number. It is defined on Line 15 of Program 6.5.

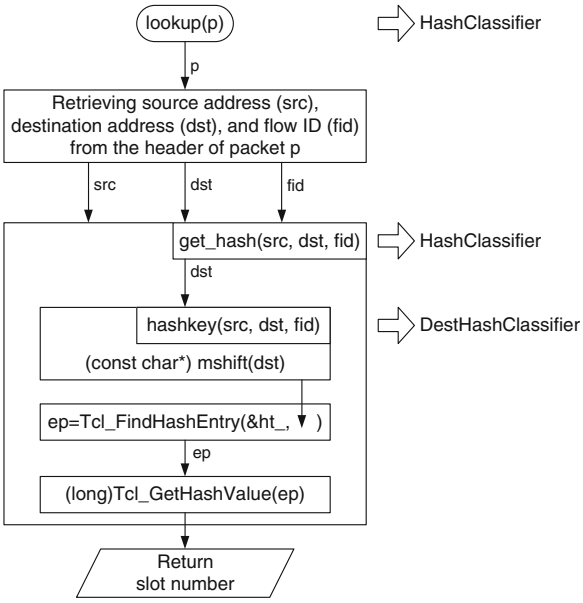

Fig. 6.3 Flowchart of function `lookup(p)` invoked from class `DestHashClassifier`

Function `do_install(dst,target)` installs (Line 12) an `NsObject` pointer `target` in the next available slot, and registers this installation in the hash table (Line 13). Defined in class `Classifier`, function `getnxt(target)` returns the available slot where `target` will be installed (see file *~ns*/classifier/classifier.cc). Again, the statement `set_hash(0,d,0,slot)` hashes the key with source address "`0`," destination address "`d`," and flow ID "`0`," and associates the result with the slot number "`slot`." Finally function `hashkey(...)` in Lines 8–11 of Program 6.7 returns the destination address, reformatted by function `mshift(...)`.

Figure 6.3 shows a process when a `DestHashClassifier` object invokes function `lookup(p)`. In this figure, the function name is indicated at the top of each box, while the corresponding class is shown in the right of a block arrow. The process follows what we have discussed earlier. The important point here is that the only function defined in class `DestHashClassifier` is the function `hashkey(...)`. Functions `lookup(p)` and `get_hash(...)` belong to class `HashClassifier`. This is a beauty of OOP, since we only need to override one function for a derived class (e.g., class `DestHashClassifier`), and are able to reuse the rest of the code from the parent class (e.g., class `HashClassifier`).

Later in Sect. 6.5.4, we shall discuss how a destination hash classifier is used to perform routing functionality.

6.2.4 Creating Your Own Classifiers

Here are the key steps for defining your own classifiers.

1. *Design*: Define criteria with tuples `(criterion,slot,NsObject)`. If a packet matches with the `criterion`, send the packet to the `NsObject` installed in `slot_[slot]`.
2. *Class construction*: Derive your C++ classifier class, for example, class `YourClassifier` from class `Classifier`. Create a shadow OTcl class.
3. *Internal mechanism*: Override function `classify(p)` according to the design in Step 1.
4. *Configuration*: In the OTcl domain, install the `NsObject` in the slot number `slot` of the `YourClassifier` object. For example, let `$clsfr` be a `YourClassifier` object and and `$obj` be an NsObject in the OTcl domain. You can install `$obj` in the slot number 10 of `$clsfr` by executing the following statement: `$clsfr install 10 $obj`.

6.3 Routing Modules

6.3.1 An Overview of Routing Modules

The main functionality of routing modules is to facilitate classifier management. For example, consider Fig. 6.4, where ten address classifiers are connected to each other. It would be rather inconvenient to configure all these ten classifiers using ten OTcl statements.

The configuration process can be facilitated by maintaining a linear topology. Even if the topology of classifiers is as complicated as a full mesh, the topology of routing modules is always linear. We can feed a configuration command to the first routing module in line, and let the routing modules propagate the configuration command toward the end of the line. Since every classifier is connected to one of these routing modules, the configuration command will eventually reach all the classifiers.

Based on the above idea, NS2 uses the following route configuration principles:

1. Assign a routing module for a classifier and connect all related routing modules in a linear topology.
2. Configure classifiers through the head routing module only.
3. Disallow direct classifier configuration.

These principles are implemented in various NS2 components such as routing agents, the route logic, and Nodes. As we shall see later on, class `Node` makes no

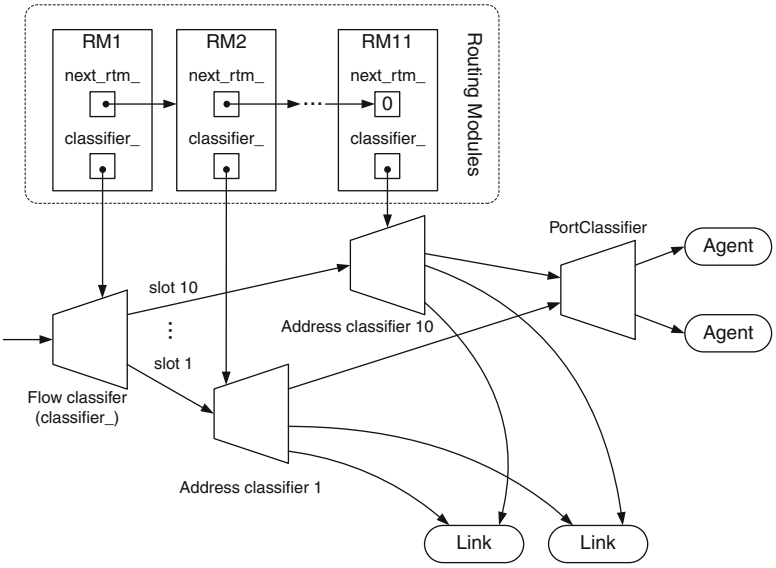

Fig. 6.4 The relationship among routing modules and classifiers in a node

attempt to directly modify its classifiers (e.g., instvars `classifier_` and `dmux_` in Fig. 6.1). Instead, it provides instprocs `add-route{...}` and `attach{...}`, which ask the related routing modules to propagate the configuration commands on its behalf.

6.3.2 C++ Class `RoutingModule`

Program 6.9 shows the declaration of class `RoutingModule`, which has three main variables. Variable `classifier_` in Line 15 is a pointer to a `Classifier` object. This variable is bound to an OTcl instvar with the same name (Line 26).

A linear topology of routing modules is created using of a pointer `next_rtm_` (Line 12), which points to another `RoutingModule` object. Finally, variable "`n_`" in Line 14 is a pointer to the associated `Node` object. These three variables are initialized to `NULL` in the constructor (Line 15).

The key functions of class `RoutingModule` include the followings (see Program 6.10):

Program 6.9 Declaration and the constructor of a C++ class `RoutingModule` which is bound to an OTcl class `RtModule`

```
    //~ns/routing/rtmodule.h
 1  class RoutingModule : public TclObject {
 2  public:
 3    RoutingModule();
 4    inline Node* node() { return n_; }
 5    virtual int attach(Node *n) { n_ = n; return TCL_OK; }
 6    virtual int command(int argc, const char*const* argv);
 7    virtual const char* module_name() const { return NULL; }
 8    void route_notify(RoutingModule *rtm);
 9    void unreg_route_notify(RoutingModule *rtm);
10    virtual void add_route(char *dst, NsObject *target);
11    virtual void delete_route(char *dst, NsObject *nullagent);
12    RoutingModule *next_rtm_;
13  protected:
14    Node *n_;
15    Classifier *classifier_;
16  };

17  static class RoutingModuleClass : public TclClass {
18  public:
19    RoutingModuleClass() : TclClass("RtModule") {}
20    TclObject* create(int, const char*const*) {
21      return (new RoutingModule);
22    }
23  } class_routing_module;

24  RoutingModule::RoutingModule() :
25          next_rtm_(NULL), n_(NULL), classifier_(NULL) {
26    bind("classifier_", (TclObject**)&classifier_);
27  }
```

`node()`	Return the attached `Node` object `n_`.
`attach(n)`	Store an input `Node` object "n" in the variable `n_`.
`module_name()`	Return the name of the routing module.
`route_notify(rtm)`	Add an input `RoutingModule *rtm` to the end of the link list.
`unreg_route_notify(rtm)`	Remove an input `RoutingModule` pointer `*rtm` from the link list.
`add_route(dst,target)`	Inform every classifier associated with the link list to add a routing rule `(dst,target)`.
`delete_route(... dst,nullagent)`	Inform every classifier in the link list to delete a routing rule with destination `dst`.

Program 6.10 Functions `route_notify`, `unreg_route_notify`, `add_route`, and `delete_route` of class `RoutingModule`

```
    //~ns/routing/rtmodule.cc
 1  void RoutingModule::route_notify(RoutingModule *rtm) {
 2      if (next_rtm_ != NULL)
 3          next_rtm_->route_notify(rtm);
 4      else
 5          next_rtm_ = rtm;
 6  }

 7  void RoutingModule::unreg_route_notify(RoutingModule *rtm) {
 8      if (next_rtm_) {
 9          if (next_rtm_ == rtm) {
10              next_rtm_ = next_rtm_->next_rtm_;
11          }
12          else {
13              next_rtm_->unreg_route_notify(rtm);
14          }
15      }
16  }

17  void RoutingModule::add_route(char *dst, NsObject *target)
18  {
19      if (classifier_)
20          classifier_->do_install(dst,target);
21      if (next_rtm_ != NULL)
22          next_rtm_->add_route(dst,target);
23  }

24  void RoutingModule::delete_route(char *dst, NsObject
      *nullagent)
25  {
26      if (classifier_)
27          classifier_->do_install(dst,nullagent);
28      if (next_rtm_)
29          next_rtm_->add_route(dst,nullagent);
30  }
```

Consider Program 6.10. Lines 1–16 show the details of functions `route_notify(rtm)` and `unreg_route_notify(rtm)`. Function `route_notify(rtm)` recursively invokes itself (Line 3) until it reaches the last routing module in the link list, where `next_rtm_` is `NULL`. Then, it attaches the input routing module `*rtm` as the last component of the link list (Line 5). Function `unreg_route_notify(rtm)` recursively searches down the link list (Line 13) until it finds and removes the input routing module pointer "`rtm`" (Lines 9 and 10).

Lines 17–30 show the details of functions `add_route(dst, target)` and `delete_route(dst,nullagent)`. Function `add_route(dst,target)` takes a destination node "`dst`" and a forwarding `NsObject` pointer "`target`" as input arguments. It installs the pointer "`target`" in all the associated classifiers

(Line 20). Again, this routing rule is propagated down the link list (Line 22), until reaching the last element of the link list. Function `delete_route(dst, nullagent)` does the opposite. It recursively installs a null agent "`nullagent`" (i.e., a packet dropping point) as the target for packets destined for a destination node "`dst`" in all the classifiers, essentially removing the routing rule with the destination "`dst`" from all the classifiers.

6.3.3 OTcl Class `RtModule`

In the OTcl domain, the routing module is defined in class `RtModule` bound to the C++ class `RoutingModule`. Class `RtModule` has two instvars: `classifier_` and `next_rtm_`. The instvar `classifier_` is bound to the class variable in the C++ domain with the same name, while the instvar `next_rtm_` is not.[4]

The OTcl class `RtModule` also defines the following instprocs and OTcl commands. For brevity, we show the details of some instprocs in Program 6.11. The details of other instprocs and OTcl command can be found in file *~ns*/tcl/lib/ns-rtmodule.tcl and *~ns*/routing/rtmodule.cc, respectively.

6.3.3.1 Initialization Instprocs

`register{node}`	Create two-way connection to the input `node` (Lines 1–13).
`unregister{}`	Remove itself from the associated node and the chain of routing modules (see the file).
`attach-node{node}`	Set the C++ variable `n_` to point to the input `node` (OTcl command; see the file).
`route-notify{module}`	Store the incoming `module` as the last element in the OTcl chain of routing modules (Lines 14–21).
`unreg-route-notify{module}`	Remove the incoming `module` from the OTcl chain of routing modules (see the file).

[4]Caution: When creating a chain of routing modules, use instproc `route_notify{...}`. If you directly configure the instvar `next_rtm_`, the C++ variable `next_rtm_` will not be automatically configured.

Program 6.11 Related Instprocs of OTcl classes `RtModule` and `RtModule/Base`

```
    //~/ns/tcl/lib/ns-rtmodule.tcl
 1  RtModule instproc register { node } {
 2      $self attach-node $node
 3      $node route-notify $self
 4      $node port-notify $self
 5  }

 6  RtModule/Base instproc register { node } {
 7      $self next $node
 8      $self instvar classifier_
 9      set classifier_ [new Classifier/Hash/Dest 32]
10      $classifier_ set mask_ [AddrParams NodeMask 1]
11      $classifier_ set shift_ [AddrParams NodeShift 1]
12      $node install-entry $self $classifier_
13  }

14  RtModule instproc route-notify { module } {
15      $self instvar next_rtm_
16      if {$next_rtm_ == ""} {
17          set next_rtm_ $module
18      } else {
19          $next_rtm_ route-notify $module
20      }
21  }

22  RtModule instproc add-route { dst target } {
23      $self instvar next_rtm_
24      [$self set classifier_] install $dst $target
25      if {$next_rtm_ != ""} {
26          $next_rtm_ add-route $dst $target
27      }
28  }

29  RtModule instproc attach { agent port } {
30      $agent target [[$self node] entry]
31      [[$self node] demux] install $port $agent
32  }
```

6.3.3.2 Instprocs for Configuring Classifiers

`add-route{dst target}`	Propagate a routing rule (`dst`, `target`) to all the attached classifiers (Lines 22–28).
`delete-route{dst nullagent}`	Remove a routing rule whose destination is "`dst`" (see the file).

`attach{agent port}`	Install the "`agent`" in the slot number "`port`" of the demultiplexer "`dmux_`" of the associated Node (Lines 29–32). We shall discuss the details of transport layer agent attachment in Sect. 6.5.3.

6.3.4 Built-in Routing Modules

6.3.4.1 The List of Built-in Routing Modules

The C++ class `RoutingModule` and the OTcl class `RtModule` are not actually in use. They are just the base classes from which the following routing module classes derive.

Routing module	C++ class	OTcl class
Routing module	`RoutingModule`	`RtModule`
Base routing module (default)	`BaseRoutingModule`	`RtModule/Base`
Multicast routing module	`McastRoutingModule`	`RtModule/Mcast`
Hierarchical routing module	`HierRoutingModule`	`RtModule/Hier`
Manual routing module	`ManualRoutingModule`	`RtModule/Manual`
Source routing module	`SourceRoutingModule`	`RtModule/Source`
Quick start for TCP/IP routing module (determine initial congestion window)	`QSRoutingModule`	`RtModule/QS`
Virtual classifier routing module	`VCRoutingModule`	`RtModule/VC`
Pragmatic general multicast routing module (reliable multicast)	`PgmRoutingModule`	`RtModule/PGM`
Light-weight multicast services routing module (reliable multicast)	`LmsRoutingModule`	`RtModule/LMS`

Among these classes, the base routing module are the most widely used. As an example, we shall discuss the details of the base routing module.

6.3.4.2 C++ Class `BaseRoutingModule` and OTcl Class `RtModule/Base`

Base routing modules are the default routing modules used for static routing. Again, they are represented in the C++ class `BaseRoutingModule` bound to the OTcl class `RtModule/Base`. From Program 6.12, class `BaseRoutingModule` derives from class `RoutingModule`. It overrides function `module_name()`, by setting its name to be "`Base`" (Line 4). A base routing module classifies packets based on its destination address only. Therefore, the type of the variable `classifier_` is defined as a `DestHashClassifier` pointer (Line 7).

Program 6.12 Declaration of class `BaseRoutingModule` which is bound to the OTcl class `RtModule/Base`

```
    //~ns/routing/rtmodule.h
 1  class BaseRoutingModule : public RoutingModule {
 2  public:
 3      BaseRoutingModule() : RoutingModule() {}
 4      virtual const char* module_name() const { return "Base";
           }
 5      virtual int command(int argc, const char*const* argv);
 6  protected:
 7      DestHashClassifier *classifier_;
 8  };

    //~ns/routing/rtmodule.cc
 9  static class BaseRoutingModuleClass : public TclClass {
10  public:
11      BaseRoutingModuleClass() : TclClass("RtModule/Base") {}
12      TclObject* create(int, const char*const*) {
13          return (new BaseRoutingModule);
14      }
15  } class_base_routing_module;
```

In the OTcl domain, class `RtModule/Base` also overrides instproc `register{node}` of class `RtModule` (Lines 6–13 in Program 6.11). In addition to creating a two-way connection to the input `Node` object `node` (performed by its base class), the base routing module creates (Line 9) and installs (Line 12) a destination hash classifier inside the `node`. We shall discuss the details of the instproc `install-entry{...}` later in Sect. 6.5.2.

6.4 Route Logic

The main responsibility of a route logic object is to compute the routing table. Route logic is implemented in a C++ class `RouteLogic` which is bound to the OTcl class with the same name (see Program 6.13).

6.4.1 C++ Implementation

The C++ Class `RouteLogic` has two key variables: "`adj_`" (Line 14), which is the adjacency matrix used to compute the routing table, and "`route_`" (Line 15), which is the routing table. It has the following three main functions:

Program 6.13 Declaration of class `RouteLogic` and the corresponding OTcl mapping class

```
    //~/ns/routing/route.h
 1  class RouteLogic : public TclObject {
 2  public:
 3      RouteLogic();
 4      ~RouteLogic();
 5      int command(int argc, const char*const* argv);
 7      virtual int lookup_flat(int sid, int did);
 8  protected:
 9      void reset(int src, int dst);
 10     void reset_all();
 11     void compute_routes();
 12     void insert(int src, int dst, double cost);
 13     void insert(int src, int dst, double cost, void* entry);
 14     adj_entry *adj_;
 15     route_entry *route_;
 16 };

    //~/ns/routing/route.cc
 17 class RouteLogicClass : public TclClass {
 18 public:
 19     RouteLogicClass() : TclClass("RouteLogic") {}
 20     TclObject* create(int, const char*const*) {
 21         return (new RouteLogic());
 22     }
 23 } routelogic_class;
```

`insert(src,... dst,cost)`	Inform the route logic of the `cost` to go from the Node `src` to the Node `dst`
`compute_routes()`	Compute the optimal routes for all source-destination pairs and store the computed routes in the variable `route_`.
`lookup_flat(... sid,did)`	Search within the variable `route_` for an entry with matching source ID (`sid`) and destination ID (`did`), and returns the index of the forwarding object (e.g., connecting link).

6.4.2 OTcl Implementation

In the interpreted hierarchy, the OTcl class `RouteLogic` has one key instvar `rtprotos_`. The instvar `rtprotos_` is an associative array whose index is the name of the routing protocol and value is the routing agent object. Again, we are dealing with static routing. Therefore, the instvar `rtprotos_` does not exist.

Program 6.14 Instprocs `configure` and `lookup` of class `RouteLogic`

```
     //~/ns/tcl/lib/ns-route.tcl
 1  RouteLogic instproc configure {} {
 2      $self instvar rtprotos_
 3      if [info exists rtprotos_] {
 4          foreach proto [array names rtprotos_] {
 5              eval Agent/rtProto/$proto init-all $rtprotos_
                  ($proto)
 6          }
 7      } else {
 8          Agent/rtProto/Static init-all
 9      }
10  }

11  RouteLogic instproc lookup { nodeid destid } {
12      if { $nodeid == $destid } {
13          return $nodeid
14      }
15      set ns [Simulator instance]
16      set node [$ns get-node-by-id $nodeid]
17      $self cmd lookup $nodeid $destid
18  }
```

The OTcl class `RouteLogic` also has two major instprocs as shown in Program 6.14).

`configure{}`	Initialize all the routing protocols (Lines 1–10).
`lookup{sid,did}`	Return the forwarding object for packets going from Node `sid` to Node `did` (Lines 11–18).

6.5 Node Construction and Configuration

So far in this chapter, we have discussed major components of a Node – classifiers, routing modules, and route logic. We now present how NS2 creates and puts together these main components.

In the following, we first show the key instvars of the OTcl class `Node` and their relationships in Sect. 6.5.1. Then, we show an approach to put classifiers into a `Node` object in Sect. 6.5.2. Sections 6.5.3 and 6.5.4 show how a `Node` is bridged to the transport (i.e., upper) layer and to the routing (i.e., lower) layer, respectively. Finally, Sect. 6.5.5 discusses the key steps to create and configure a `Node` object.

6.5.1 Key Variables of the OTcl Class `Node` and Their Relationship

The list of major instvars of the OTcl class `Node` is given below.

`id_`	Node ID
`agents_`	List of attached transport layer agents
`nn_`	Total number of Nodes
`neighbor_`	List of neighboring nodes
`nodetype_`	Node type (e.g., regular node or mobile node)
`ns_`	The Simulator
`classifier_`	Address classifier, which is the default node entry
`dmux_`	The demultiplexer or port classifier
`module_list_`	List of enabled routing modules
`reg_module_`	List of registered routing modules
`rtnotif_`	The head of the chain of routing modules which will be notified of route updates
`ptnotif_`	List of routing modules which will be notified of port attachment/detachment
`hook_assoc_`	Sequence of the chain of classifiers
`mod_assoc_`	Association of classifiers and routing modules

6.5.1.1 Routing-Related Instvars

The following five instvars of an OTcl Node plays a major role in packet routing: `module_list_`, `reg_module_`, `rtnotif_`, `ptnotif_`, and `mod_assoc_`. The instvar `module_list_` is a `list` of strings, each of which represents the name of enabled routing module. The instvar `reg_module_` is an associative array whose index and value are the name of the routing module and the routing module instance.

The instvars `rtnotif_` and `ptnotif_` contain the objects which should be notified of a route change and an agent attachment/detachment, respectively. While `rtnotif_` is the head of the link list of the routing modules, `ptnotif_` is simply an OTcl `list` whose elements are the routing modules. Finally, instvar `mod_assoc_` is an associative array whose indexes and values are classifiers and the associated routing modules, respectively.

Figure 6.5 shows an example of routing-related variable setting both in C++ and OTcl domain. Here, we assume that there are two classifiers. The first, `switch_`, classifies the geometry (i.e., circle/triangle/square). The second classifies, `classifier_`, color (i.e., black/white).

The above two classifiers are controlled by routing modules `RtModule/Mcast` and `RtModule/Base`, respectively. Since there are two routing modules, the instvar `reg_modules_` has two entries.

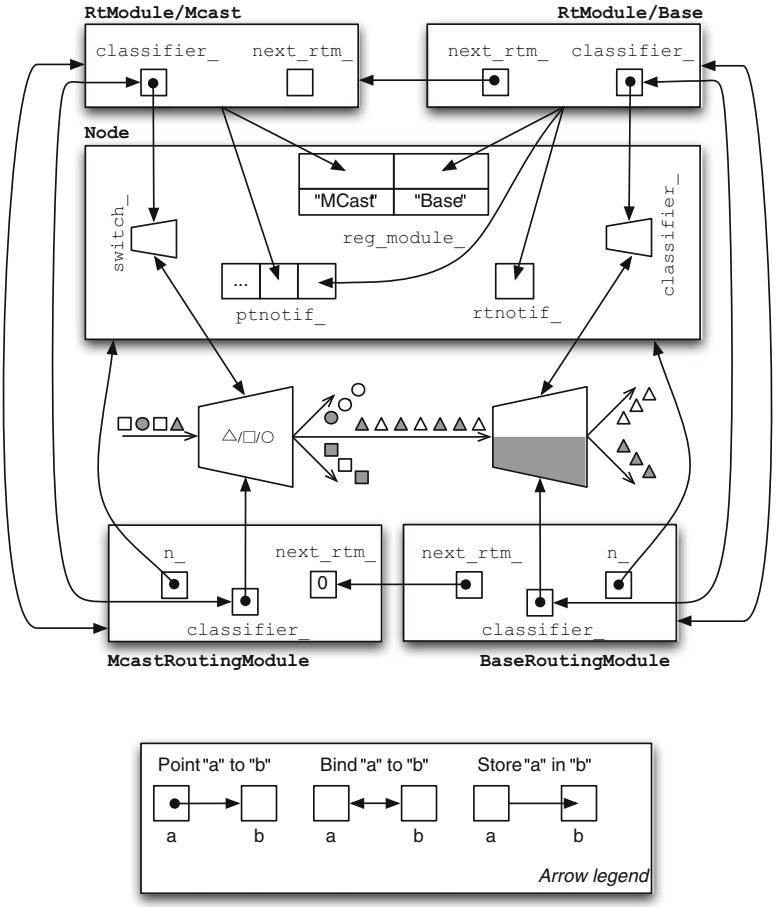

Fig. 6.5 An example of node configuration with two classifiers

Suppose further that both the classifiers need to be informed of routing change and agent attachment/detachment. We need to put both the associated routing modules in the `list` instvar `ptnotif_`. On the other hand, we only set one routing module (i.e., `RtModule/Base` associated with `classifier_` in this case) as the instvar `rtnotif_`. The route configuration command can be propagated to `RtModule/Mcast` via the variable `next_rtm_` of the head (i.e., Base) routing module.

6.5.1.2 Classifier-Related Instvars

Class `Node` has three instvars related to classifiers: `classifier_`, `hook_assoc_`, and `mod_assoc_`. Instvar `classifier_` is the default Node entry as well as the head of the chain of classifiers. Instvar `hook_assoc_` is an associative

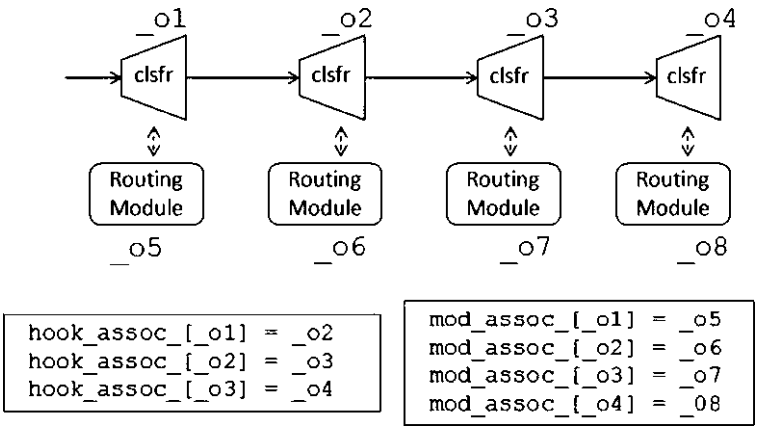

Fig. 6.6 An example of values stored in variables `hook_assoc_` and `mod_assoc_`

array whose index is a classifier and value is the downstream classifier in the chain. The index and value of the associative array `mod_assoc_` are classifiers and the associated routing modules, respectively.

Consider Fig. 6.6 for example. Here, we install classifiers `_o1`, `_o2`, `_o3`, and `_o4` into a Node, and associate them with routing modules `_o5`, `_o6`, `_o7`, and `_o8`, respectively. Then, the instvar `classifier_` would be `_o1`. The indexes and values of `hook_assoc_` and `mod_assoc_` would be as shown in the figure.

6.5.2 *Installing Classifiers in a Node*

Class `Node` provides three instprocs to configure classifiers. First, as shown in Program 6.15, the instproc `insert-entry{module clsfr hook}` takes three input arguments: a routing module "`module`", a classifier "`clsfr`", and an optional argument "`hook`." It installs the current head classifier in the slot number "`hook`" of the input classifier `clsfr` (Line 8), and replaces the head classifier with the input classifier `clsfr` (Line 12). The instvars `hook_assoc_` and `mod_assoc_` are updated in Lines 4 and 11, respectively.

The input argument "`hook`" can have one of the three following values:

- A number: The input "`clsfr`" will be configured as explained above.
- A string "`target`": The existing head classifier will be configured as a `target` of the input NsObject `clsfr`.[5]
- Null: The input "`clsfr`" will not be configured. We might have to configure it later.

[5]Note that the input `clsfr` needs not be a classifier.

Program 6.15 Instprocs `insert-entry` and `install-demux` of class `Node`

```
    //~ns/tcl/lib/ns-node.tcl
 1  Node instproc insert-entry { module clsfr {hook ""} } {
 2      $self instvar classifier_ mod_assoc_ hook_assoc_
 3      if { $hook != "" } {
 4          set hook_assoc_($clsfr) $classifier_
 5          if { $hook == "target" } {
 6              $clsfr target $classifier_
 7          } elseif { $hook != "" } {
 8              $clsfr install $hook $classifier_
 9          }
10      }
11      set mod_assoc_($clsfr) $module
12      set classifier_ $clsfr
13  }

14  Node instproc install-demux {demux {port ""} } {
15      $self instvar dmux_ address_
16      if { $dmux_ != "" } {
17           $self delete-route $dmux_
18           if { $port != "" } {
19               $demux install $port $dmux_
20           }
21       }
22       set dmux_ $demux
23      $self add-route $address_ $dmux_
24  }
```

The second classifier configuration is the instproc `install-entry{module clsfr hook}`, which is very similar to instproc `insert-entry{...}`. The only difference is that it also destroys the existing head classifier, if any. The details of the instproc `install-entry{...}` can be found in the file *~ns*/tcl/lib/ns-node.tcl.

The last classifier configuration is the instproc `install-demux{demux port}`, whose details are shown in Lines 14–24 of Program 6.15. This instproc takes two input arguments: `demux` (mandatory) and `port` (optional). It replaces the existing demultiplexer[6] "`dmux_`" with the input demultiplexer `demux` (Line 22). If "`port`" exists, the current demultiplexer "`dmux_`" will be installed in the slot number "`port`" of the input demultiplexer "`demux`" (Lines 18–20).

6.5.3 *Bridging a Node to a Transport Layer Protocol*

To attach an agent to a Node, we use instproc `attach-agent{node agent}` of class `Simulator`, where "`node`" and "`agent`" are `Node` and `Agent` objects,

[6] A demultiplexer classifies packets based on the port number specified in the packet header (see Sect. 6.2.2 for more details).

Program 6.16 Agent attachment instprocs

```
    //~ns/tcl/lib/ns-lib.tcl
 1  Simulator instproc attach-agent { node agent } {
 2      $node attach $agent
 3  }

    //~ns/tcl/lib/ns-node.tcl
 4  Node instproc attach { agent { port "" } } {
 5      $self instvar agents_ address_ dmux_
 6      lappend agents_ $agent
 7      $agent set node_ $self
 8      $agent set agent_addr_ [AddrParams addr2id $address_]
 9      if { $dmux_ == "" } {
10          set dmux_ [new Classifier/Port]
11          $self add-route $address_ $dmux_
12      }
13      if { $port == "" } {
14          set port [$dmux_ alloc-port [[Simulator
                                        instance] nullagent]]
15      }
16      $agent set agent_port_ $port
17      $self add-target $agent $port
18  }

19  Node instproc add-target { agent port } {
20      $self instvar ptnotif_
21      foreach m [$self set ptnotif_] {
22          $m attach $agent $port
23      }
24  }
```

respectively. Program 6.16 shows the instprocs related to an agent attachment process. The process proceeds as follows:

1. `Simulator::attach-agent{node agent}`: Invoke "`$node attach $agent`" (Line 2).
2. `Node::attach{agent port}`: Update instvar "`agent`" (Lines 6–8 and Line 16), create "`dmux_`" if necessary (Lines 9–15), and invoke "`$self add-target $agent $port`" (Line 17).
3. `Node::add-target{agent port}`: From Sect. 6.5.1, routing modules related to port attachment are stored in the list instvar `ptnotif_`. Therefore, Lines 22 and 23 execute instproc `attach{agent port}` of all the routing modules stored in the instvars `ptnotif_`.[7]
4. `RtModule::attach{agent port}`: Consider Lines 29–32 in Program 6.11. As a sending agent, the input "`agent`" is set as an upstream object of the node entry (Line 30). As a receiving agent, it is installed in the slot number "`port`" of demultiplexer "`dmux_`" (Line 31).

[7] Again, Nodes do not directly configure port classifiers. It asks routing modules stored in `ptnotif_` to do so on its behalf.

Program 6.17 Instprocs `add-route` of class `Node`

```
    //~ns/tcl/lib/ns-node.tcl
1   Node instproc add-route { dst target } {
2       $self instvar rtnotif_
3       if {$rtnotif_ != ""} {
4           $rtnotif_ add-route $dst $target
5       }
6       $self incr-rtgtable-size
7   }
```

Note that although an agent can be *either* a sending agent or a receiving agent, this instproc assigns both roles to every agent. This does not cause any problem at runtime due to the following reasons. A sending agent is attached to a source node, and always transmits packets destined to a destination node. It takes no action when receiving a packet from a demultiplexer. A receiving agent, on the other hand, does not generate a packet. Therefore, it can never send a packet to the node entry.

6.5.4 Adding/Deleting a Routing Rule

Class `Node` provides an instproc `add-route{dst target}` to add a routing rule (`dst,target`) to the routing table. In Program 6.17, instproc `add-route {dst target}` of class `Node` invokes the instproc `add-route{...}` of the routing module `rtnotif_` which is of class `RtModule`[8] (Line 4). From Lines 22–29 of Program 6.11, the instproc `add-route{...}` of class `RtModule` installs the routing rule (`dst,target`) in the `classifier_` of all the related routing module.

The mechanism for deleting a routing rule is similar to that for adding a routing rule, and is omitted for brevity. The readers may find the details of route entry deletion in the instproc `delete-route{dst nullagent}` of classes `Node` and `RtModule` (see file ~*ns*/tcl/lib/ns-node.tcl and file ~*ns*/tcl/lib/ns-rtmodule.tcl).

6.5.5 Node Construction and Configuration

There are two key steps to put together a Node (e.g., as shown in Figs. 6.1 and 6.5). We now discuss the details of node construction and configuration in sequence.

[8] Again, class `Node` makes no attempt to directly modify the classifiers. It asks the routing modules in the chain, whose head is `rtnotif_`, to do so on its behalf.

Program 6.18 Default value of instvar `node_factory_` and instproc `node` of class `Simulator`

```
    //~ns/tcl/lib/ns-default.tcl
1   Simulator set node_factory_ Node

    //~ns/tcl/lib/ns-lib.tcl
2   Simulator instproc node args {
3       $self instvar Node_ routingAgent_
4       set node [eval new [Simulator set node_factory_] $args]
5       set Node_([$node id]) $node
6       $self add-node $node [$node id]
7       $node nodeid [$node id]
8       $node set ns_ $self
9       return $node
10  }
```

6.5.5.1 Node Construction

A `Node` object is created using an OTcl statement "`$ns node`," where `$ns` is the `Simulator` instance. The Instproc "`node`" of class `Simulator` uses instproc "`new{...}`" to create a `Node` object (Line 4 where `node_factory_` is set to `Node` in Line 1 of Program 6.18). It also updates instvars of the Simulator so that they can later be used by other simulation objects throughout the simulation.

The construction of an OTcl `Node` object (using `new{...}`) consists of seven main steps (see also Program 6.19).

Step 1: Constructor of the OTcl Class Node

Instproc `init{...}` sets up instvars of class `Node`, and invokes instproc `mk-default-classifier{}` of the `Node` object (Line 22 in Program 6.19).

Step 2: Instproc `mk-default-classifier{}`

The instproc `mk-default-classifier{}` creates (using `new{...}`) and registers (using `register-module{mod}`) routing modules whose names are stored in the instvar `module_list_` (Lines 27–29 in Program 6.19). By default, only "`Base`" routing module is stored in the instvar `module_list_` (Line 1 in Program 6.19).

To enable/disable other routing modules, the following two instprocs of class `RtModule` must be invoked before the execution of "`$ns node`":

```
enable-module{<name>}
disable-module{<name>}
```

where `<name>` is the name of the routing module, which is to be enabled/disabled.

Program 6.19 Instprocs related to the Node Construction Process

```
    //~/ns/tcl/lib/ns-node.tcl
 1  Node set module_list_ { Base }

 2  Node instproc init args {
 3      eval $self next $args
 4      $self instvar id_ agents_ dmux_ neighbor_ rtsize_
 5         address_ \ nodetype_ multiPath_ ns_ rtnotif_ ptnotif_
 6      set ns_ [Simulator instance]
 7      set id_ [Node getid]
 8      $self nodeid $id_   ;# Propagate id_ into c++ space
 9      if {[llength $args] != 0} {
10          set address_ [lindex $args 0]
11      } else {
12          set address_ $id_
13      }
14      $self cmd addr $address_; # Propagate address_ into
           C++ space
15      set neighbor_ ""
16      set agents_ ""
17      set dmux_ ""
18      set rtsize_ 0
19      set ptnotif_ {}
20      set rtnotif_ {}
21      set nodetype_ [$ns_ get-nodetype]
22      $self mk-default-classifier
23      set multiPath_ [$class set multiPath_]
24  }

25  Node instproc mk-default-classifier {} {
26      Node instvar module_list_
27      foreach modname [Node set module_list_] {
28          $self register-module [new RtModule/$modname]
29      }
30  }

31  Node instproc register-module { mod } {
32      $self instvar reg_module_
33      $mod register $self
34      set reg_module_([$mod module-name]) $mod
35  }
```

Step 3: Instproc `register-module{mod}` of Class `Node`

This instproc invokes the instproc `register{node}` of the input routing module "`mod`" and updates the instvar "`reg_module_`" (see Lines 31–35).

Step 4: Instproc `register{node}` of Class `RtModule/Base`

This instproc first invokes instproc `register{node}` of its parent class (by the statement `$self next $node` in Line 7 of Program 6.11). Then, Lines 9–12 create (using `new{...}`) and configure (using `install-entry{...}`) the head classifier (i.e., `classifier_`) of the Node.

Step 5: Instproc `register{node}` of Class `RtModule`

From Program 6.11, this instproc attaches input `Node` object "`node`"to the routing module (Line 2). It also invokes instproc `route-notify{module}` (Line 3) and `port-notify{module}` (Line 4) of the associated Node to include the routing module into the route notification list `rtnotif_` and port notification list `ptnotif_` of the associated Node.

Step 6: Instproc `route-notify{module}` of Class `Node`

As shown in Program 6.20, the instproc `route-notify{module}` (Lines 1–9) stores the input routing module "`module`" as the last element of the link list of routing modules. It also invokes the OTcl command `route-notify` of the input routing module (Line 8). The OTcl command `route-notify` invokes the C++ function `route_notify(rtm)` associated with the attached Node (see Lines 10–16) to store the routing module as the last routing module in the link list (see Lines 17–22).

Step 7: Instproc `port-notify{module}`of Class `Node`

As shown in Lines 23–26 of Program 6.20, the instproc `port-notify{module}` appends the input argument "`module`" to the end of the list instvar `ptnotif_`.

6.5.5.2 Agent and Route Configuration

We have discussed how NS2 creates and puts main components within a `Node` object. The final step is to instruct these components what to do when receiving a packet.

From Fig. 6.1, a `Node` object contains two key components: an address classifier `classifier_`, and a port classifier/demultiplexer, `dmux_`. Class `Simulator` provides two instprocs to configure these two classifiers.[9]

[9]Since the `Simulator` object is accessible to the Tcl simulation script, users generally use these two instprocs to configure Nodes.

Program 6.20 Instprocs and functions which are related to instprocs `route-notify` and `port-notify` of the OTcl class `Node`

```
    //~/ns/tcl/lib/ns-node.tcl
 1  Node instproc route-notify { module } {
 2      $self instvar rtnotif_
 3      if {$rtnotif_ == ""} {
 4          set rtnotif_ $module
 5      } else {
 6          $rtnotif_ route-notify $module
 7      }
 8      $module cmd route-notify $self
 9  }

    //~ns/routing/rtmodule.cc
10  int BaseRoutingModule::command(int argc, const char*const*
      argv) {
11      Tcl& tcl = Tcl::instance();
12      if (argc == 3) {
13          if (strcmp(argv[1] , "route-notify") == 0) {
14              n_->route_notify(this);
15          }
16  }

    //~ns/common/node.cc
17  void Node::route_notify(RoutingModule *rtm) {
18      if (rtnotif_ == NULL)
19          rtnotif_ = rtm;
20      else
21          rtnotif_->route_notify(rtm);
22  }

    //~/ns/tcl/lib/ns-node.tcl
23  Node instproc port-notify { module } {
24      $self instvar ptnotif_
25      lappend ptnotif_ $module
26  }
```

- **Instproc** `attach-agent{...}`: Connect a Node to a transport layer agent (see the details in Sect. 6.5.3).
- **Instproc** `run{}`: Create, compute, and install routing tables in classifiers of all the Nodes. Again, by default, NS2 uses the Dijkstra's algorithm to compute routing tables for all pairs of Nodes. The key steps in the instproc `run{}` are shown below:

Step 1: Instproc `run{}` of Class `Simulator`

Shown in Line 2 of Program 4.12, instproc `run{}` of class `Simulator` executes the instproc `configure{}` of the `RouteLogic` object.

Program 6.21 Instprocs related to the route configuration process

```
    //~ns/tcl/rtglib/route-proto.tcl
1   Agent/rtProto/Static proc init-all args {
2       [Simulator instance] compute-routes
3   }

    //~ns/tcl/lib/ns-route.tcl
4   Simulator instproc compute-routes {} {
5           $self compute-flat-routes
6   }

7   Simulator instproc compute-flat-routes {} {
8       $self instvar Node_ link_
9       set r [$self get-routelogic]
10      $self cmd get-routelogic $r
11      foreach ln [array names link_] {
12          set L [split $ln :]
13          set srcID [lindex $L 0]
14          set dstID [lindex $L 1]
15          if { [$link_($ln) up?] == "up" } {
16              $r insert $srcID $dstID [$link_($ln) cost?]
17          } else {
18              $r reset $srcID $dstID
19          }
20      }
21      $r compute
22      set n [Node set nn_]
23      $self populate-flat-classifiers $n
24  }
```

Step 2: Instproc `configure{}` of Class `RouteLogic`

Defined in Lines 1–10 of Program 6.14, the instproc `configure{}` of class `Route Logic` configures the routing table for all the Nodes by invoking instproc `init-all{}` of class `Agent/rtProto/Static`.

Step 3: Instproc `init-all{}` of Class `Agent/rtProto/Static`

Defined in Lines 1–3 of Program 6.21, the instproc `init-all{}` of class `Agent /rtProto/Static` invokes the instproc `compute-routes{}` of the Simulator.

Step 4: Instproc `compute-routes{}` of Class `Simulator`

By default, this instproc invokes the instproc `compute-flat-routes{}` to compute and setup the routing table (see Lines 4–6 in Program 6.21).

Program 6.22 An OTcl command `populate-flat-classifiers`, a function `populate_flat_classifiers` of class `Simulator`, and a function `add_route` of class `Node`

```
    //~ns/common/simulator.cc
 1  int Simulator::command(int argc, const char*const* argv) {
 2        ...
 3     if (strcmp(argv[1], "populate-flat-classifiers") == 0) {
 4        nn_ = atoi(argv[2]);
 5        populate_flat_classifiers();
 6        return TCL_OK;
 7     }
 8     ...
 9  }

10  void Simulator::populate_flat_classifiers() {
11     ...
12     for (int i=0; i<nn_; i++) {
13        for (int j=0; j<nn_; j++) {
14           if (i != j) {
15              int nh = -1;
16              nh = rtobject_->lookup_flat(i, j);
17              if (nh >= 0) {
18                 NsObject *l_head=get_link_head(nodelist_[i],
                     nh);
19                 sprintf(tmp, "%d", j);
20                 nodelist_[i]->add_route(tmp, l_head);
21              }
22           }
23        }
23     }
25  }

    //~ns/common/node.cc
26  void Node::add_route(char *dst, NsObject *target) {
27      if (rtnotif_)
28         rtnotif_->add_route(dst, target);
29  }
```

Step 5: Instproc `compute-flat-routes{}` of Class `Simulator`

Defined in Lines 7–24 of Program 6.21, this instproc computes and installs the routing table in all related address classifiers.

- Retrieve and store the route logic in a local variable `$r` (Lines 9–10)
- Collect and insert topology information into the retrieved route logic `$r` (Lines 11–20)
- Compute the optimal route (Line 21)
- Add routing rules into all related classifiers (Lines 22 and 23)

Program 6.22 shows the details of how the computed routing rules are propagated to all the nodes. Lines 1–9 show the details of OTcl command `populate-flat-classifiers{n}`. This OTcl command stores the input number of nodes "n" in the variable `nn_` (Line 4), and invokes the function `populate_flat_classifiers()` (Line 5) to install the computed routing rules in all the classifiers.

Function `populate_flat_classifiers()` adds the routing rules for all pairs `(i,j)` of `nn_` nodes (Lines 10–25). For each pair, Line 16 retrieves the next hop (i.e., forwarding) referencing point "`nh`" of a forwarding object for a packet traveling from Node "`i`" to Node "`j`." Line 18 retrieves the link entry point "`l_head`" corresponding to the variable "`nh`." Lines 19 and 20 add a new routing rule for the node `i` (i.e., `nodelist_[i]`). The rule specifies the link entry "`l_head`" as a forwarding target for packet destined for a destination node `j`. The rule is added to the Node "`i`" via its function `add_route(dst,target)`.

Function `add_route(dst,target)` simply invokes function `add_route (dst,target)` of the associated `RoutingModule` object `rtnotif_` (Lines 26–29). Defined in Program 6.10, function `add_route(dst,target)` of class `RoutingModule` recursively installs the input routing rule `(dst,target)` down the link list of routing modules.

6.6 Chapter Summary

A Node is a basic component which acts as a router and a computer host. Its main responsibilities are to forward packets according to a routing table and to bridge the high-layer protocols to a low-level network. A Node consists of three key components: classifiers, routing modules, and the route logic. A classifier is a multi-target packet forwarder. It forwards packets in the same category to the same forwarding NsObject. In a Node, an address classifier and a port classifier act as a router and a bridge to the transport layer, respectively.

Routing modules are responsible for managing classifiers. By convention, all the configuration commands must go though routing modules only. Finally, the route logic collects network topology, computes the optimal routing rules, and install the resulting rules in all the Nodes.

During the Network Configuration Phase, a Node is created by executing `$ns node` where `$ns` is the `Simulator` object. Here, address classifiers and routing modules are installed in the Node. The instruction of what to do when receiving a packet is provided later when the following two OTcl statements are executed. First, the transport layer connections are created using the instproc `attach-agent{...}` of class `Simulator`. Second, the instproc `run{}` of class `Simulator` computes the optimal routes for all pairs of nodes and installs the computed routing tables in relevant classifiers.

6.7 Exercises

1. What is a Classifier? What are the similarities/differences between a Connector and a Classifier?
2. Explain and give example for the following terminologies:

   a. Routing mechanism, b. Routing rules, c. Routing table,
   d. Routing algorithm, e. Routing protocol, f. Routing agent,
   g. Route logic, h. Routers, i. Routing module.

3. What are routing modules? Explain their roles and necessities.
4. Explain how classifiers work.

   a. What are slots? What does "installing an NsObject in a slot" mean?
   b. How does a packet enter a classifier? Explain the packet flow mechanism since a packet enters a classifier until it leaves the classifier. Give an example and draw a diagram to support your answer.

5. What is a hash function? Explain your answer and show few applications which use hash functions.
6. What are the components in NS2 which

   a. Find the optimal route,
   b. Propagate topology and routing information,
   c. Forward packet based on the routing information.

7. Consider a packet size classifier which classifies packet into small (smaller than 40 bytes), medium (not smaller than 40 bytes but smaller than 1,000 bytes), and large (not smaller than 1,000 bytes) packets.

   a. Create a packet size classifier.
   b. Configure the classifier such that small, medium, and large packets are sent to the NsObject whose addresses are stored in variables "`sm`," "`md`," and "`lg`," respectively.
   c. Explain the packet flow mechanism and run an NS2 program to test your answer.

   Hint: Packet size can be obtained by C++ statements "`hdr_cmn* ch = hdr_cmn::access(p); ch->size_`"; see Chap. 8.
8. Draw a Node diagram. What OTcl commands do you use to create a Node like in the diagram. Explain, step-by-step, how NS2 creates the Node.
9. What is the default routing algorithm in NS2? How does NS2 setup the routing in a Node?
10. What is a node entry? Show an OTcl statement to retrieve a reference to the node entry.

11. How does NS2 inform a Node of

    a. New route,
    b. New agent which shall be attached to a certain port?

    Show the step-by-step processes in NS2 via examples.

Chapter 7
Link and Buffer Management

A Link is an OTcl object that connects two nodes and carries packets from the beginning node to the terminating node. This chapter focuses on a class of most widely used Link objects, namely, `SimpleLink` objects. Conveying packets from one node to another, a `SimpleLink` object models packet transmission time, link propagation delay, and packet buffering. Here, packet transmission time refers to the time required by a transmitter to send out a packet. It is determined by the link bandwidth and packet size. Link propagation delay is the time needed to convey a data bit from the beginning to the end of a link. In the presence of bursty traffic, a transmitter may receive packets while transmitting a packet. The packets entering a busy transmitter could be placed in a buffer for future transmission. `SimpleLink` also models this packet buffering mechanism.

In the followings, we first give an introduction to classes `Link` and `SimpleLink` in Sect. 7.1. Then, we show how NS2 models packet transmission time and propagation delay in Sect. 7.2. Next, the packet buffering, queue blocking, and callback mechanisms are discussed in Sect. 7.3. Section 7.4 shows a network construction and packet flow example. Finally, the chapter summary is provided in Sect. 7.5.

7.1 Introduction to SimpleLink Objects

NS2 models a link using classes derived from OTcl class `Link` object, among which OTcl class `SimpleLink` is the simplest one which can be used to connect two Nodes.

7.1.1 Main Components of a SimpleLink

Figure 7.1 shows the composition of class `SimpleLink`, which consists of the following basic objects and tracing objects in the interpreted hierarchy:

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_7, © Springer Science+Business Media, LLC 2012

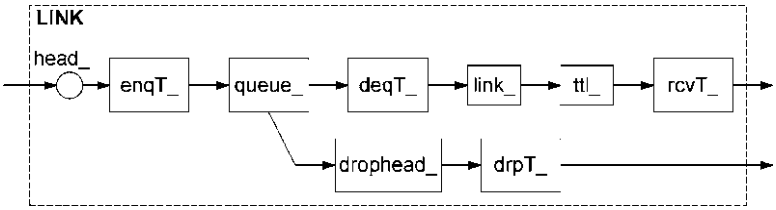

Fig. 7.1 Architecture of a `SimpleLink` object

7.1.1.1 Basic Objects

`head_`	The entry point of a `SimpleLink` object.
`queue_`	A `Queue` object which models packet buffering of a real router (see Sect. 7.3).
`link_`	A `DelayLink` object which models packet transmission time and link propagation delay (see Sect. 7.2).
`ttl_`	A *time to live* checker object whose class is `TTLChecker`. It decrements the time to live field of an incoming packet. After the decrement, if the time to live field is still positive, the packet will be forwarded to the next element in the link. Otherwise, it will be removed the packet from the simulation (see file *~ns*/common/ttl.h,cc).
`drophead_`	The common packet dropping point for the link. The dropped packets are forwarded to this object. It is usually connected to a null agent so that all `SimpleLink` objects share the same dropping point.

7.1.1.2 Tracing Objects

These objects will be inserted only if instvar `$traceAllFile_` of the `Simulator` object is defined. We will describe the details of tracing objects in detail in Chap. 13. These objects are

`enqT_`	Trace packets entering `queue_`.
`deqT_`	Trace packets leaving `queue_`.
`drpT_`	Trace packets dropped from `queue_`.
`rcvT_`	Trace packets leaving the `SimpleLink` or equivalently received by the next node.

Program 7.1 Instproc `simplex-link` of class `Simulator`

```
    //~ns/tcl/lib/ns-lib.tcl
1   Simulator instproc simplex-link { n1 n2 bw delay qtype
       args } {
2       $self instvar link_ queueMap_ nullAgent_ useasim_
3       switch -exact $qtype {
4           ...
5           default {
6               set q [new Queue/$qtype $args]
7           }
8       }
9       switch -exact $qtypeOrig {
10          ...
11          default {
12              set link_($sid:$did) [new SimpleLink    \
                                        $n1 $n2 $bw $delay $q]
13          }
14      }
15  }
```

7.1.2 *Instprocs for Configuring a `SimpleLink` Object*

In the OTcl domain, a `SimpleLink` object is created using the instprocs `simplex-link{..}` and `duplex-link{...}` of class `Simulator` whose syntax is as follows:

```
$ns simplex-link $n1 $n2 <bandwidth> <delay> <qtype>
$ns duplex-link $n1 $n2 <bandwidth> <delay> <qtype>
```

where `$ns` is the `Simulator` object, and `$n1` and `$n2` are `Node` objects.

Instproc `simplex-link{...}` above creates a unidirectional `SimpleLink` object connecting Node `$n1` to Node `$n2` (Program 7.1). The speed and the propagation delay of the link are given as `<bandwidth>` (in bps) and `<delay>` (in seconds), respectively. As opposed to a "real" router, NS2 incorporates a queue in a `SimpleLink` object, not in a `Node` object. The type of the queue in the link is specified by `<qtype>`.

Program 7.1 shows details of instproc `Simulator::simplex-link{...}`. Line 6 creates an object of class `Queue/$qtype`. Line 12 constructs a `SimpleLink` object, connecting node `$n1` to `$n2`. It specifies delay, bandwidth, and `Queue` object of the link to be `$bw`, `$delay`, and `$q`, respectively. The Simulator stores the created `SimpleLink` object in its instance associative array "`link_`" `($sid:$did)`, where `$sid` is the source node ID, and `$did` is the destination node ID, respectively, (see Chap. 4).

Program 7.2 The constructor of the OTcl class `SimpleLink`

```
     //~ns/tcl/lib/ns-link.tcl
  1  SimpleLink instproc init { src dst bw delay q {
                                          lltype "DelayLink"} } {
  2   set ns [Simulator instance]
  3   set drophead_ [new Connector]
  4   $drophead_ target [$ns set nullAgent_]
  5   set head_ [new Connector]
  6   if { [[$q info class] info heritage ErrModule] ==
                                                  "ErrorModule" } {
  7       $head_ target [$q classifier]
  8   } else {
  9       $head_ target $q
  10  }
  11  set queue_ $q
  12  set link_ [new $lltype]
  13  $link_ set bandwidth_ $bw
  14  $link_ set delay_ $delay
  15  $queue_ target $link_
  16  $link_ target [$dst entry]
  17  $queue_ drop-target $drophead_
  18  set ttl_ [new TTLChecker]
  19  $ttl_ target [$link_ target]
  20  $self ttl-drop-trace
  21  $link_ target $ttl_
  22 }
```

Instproc `duplex-link{...}` creates two `SimpleLink` objects: one connecting Node `$n1` to Node `$n2` and another connecting Node `$n2` to Node `$n1`. The readers are encouraged to find details of the instproc `duplex-link{...}` in file *~ns*/tcl/lib/ns-lib.tcl.

7.1.3 The Constructor of Class `SimpleLink`

Program 7.2 shows details of the instproc `init{...}` (i.e., the constructor) of class `SimpleLink`, which constructs and connects objects according to Fig. 7.1. Lines 3, 5, 11, 12, and 18 create instvars "`drophead_`," "`head_`," "`queue_`," "`link_`," and "`ttl_`," whose OTcl classes are `Connector`, `Connector`, `Queue`, `DelayLink`, and `TTLChecker`, respectively. Bandwidth and delay of the instvar "`link_`" are configured in Lines 13 and 14.

Apart from creating the above objects, the constructor also connects the created objects as in Fig. 7.1. Derived from class `Connector`, each of the created objects uses commands `target{...}` and `drop-target{...}` to specify the next downstream object and the dropping point, respectively (see Chap. 5). Line 9 sets the target of "`head_`" to be "`q`." Line 15 sets the target of "`queue_`" (which is set

to "q" in Line 11) to be "link_." Line 16 sets the target of "link_" to be the entry of the next node. Lines 19 and 21 insert "ttl_" between "link_" and the entry of the next node. Line 17 sets the dropping point of "queue_" to be "drophead_." Finally, Line 4 sets the target of "drophead_" to be the null agent of the Simulator.

7.2 Modeling Packet Departure

7.2.1 Packet Departure Mechanism

NS2 models packet departure using a C++ class `Linkdelay` (see Program 7.3), which is bound to an OTcl class `DelayLink`. Again, the OTcl class `DelayLink` is used to instantiate the instvar `SimpleLink::link_` which models the packet departure process.

Program 7.3 Declaration of class `LinkDelay`

```
    //~ns/link/delay.h
 1  class LinkDelay : public Connector {
 2      public:
 3          LinkDelay(): dynamic_(0), latest_time_(0), itq_(0){
 4              bind_bw("bandwidth_", &bandwidth_);
 5              bind_time("delay_", &delay_);
 6          }
 7          void recv(Packet* p, Handler*);
 8          void send(Packet* p, Handler*);
 9          void handle(Event* e);
10          inline double txtime(Packet* p) {Packet TXT Time
11              return (8. * hdr_cmn::access(p)->size() /
                    bandwidth_);
12          }
13      protected:
14          int command(int argc, const char*const* argv);
15          double bandwidth_;
16          double delay_;
17          PacketQueue* itq_;
18          Event intr_; /* In transit */
19  };

    //~ns/link/delay.cc
20  static class LinkDelayClass : public TclClass {
21  public:
22      LinkDelayClass() : TclClass("DelayLink") {}
23      TclObject* create(int argc , const char*const*  argv ) {
24          return (new LinkDelay);
25      }
26  } class_delay_link;
```

A packet departure process consists of packet transmission time and link propagation delay. While the former defines the time a packet stays in an upstream node, the summation of the former and the latter determines the time needed to deliver the entire packet to the connecting downstream node. Conceptually, when a `LinkDelay` object receives a packet, it places these two events on the simulation timeline:

1. *Packet departure* from an upstream object: Define *packet transmission time* $= \frac{\text{packet size}}{\text{bandwidth}}$ as time needed to transmit a packet over a link. After a period of packet transmission time, the packet completely leaves (or departs) the transmitter, and the transmitter is allowed to transmit another packet. Upon a packet reception, a `LinkDelay` object waits for a period of packet transmission time, and informs its upstream object that it is ready to receive another packet.
2. *Packet arrival* at a downstream node: Define *propagation delay* as the time needed to deliver a data bit from the beginning to the end of the link. Again, an entire packet needs a period of "packet transmission time + propagation delay" to reach the destination. A `LinkDelay` object, therefore, schedules a packet reception event at the downstream node after this period.

7.2.2 C++ Class `LinkDelay`

Program 7.3 shows the declaration of C++ class `LinkDelay`, which is mapped to the OTcl class `DelayLink`. Class `LinkDelay` has the following four main variables. Variables "`bandwidth_`" (Line 15) and "`delay_`" (Line 16) store the link bandwidth and propagation delay, respectively. In Lines 4 and 5, these two variables are bound to OTcl instvars with the same name. In a link with large bandwidth-delay product, a transmitter can send a new packet before the previous packet reaches the destination. Class `LinkDelay` stores all packets *in-transit* in its buffer "`itq_`" (Line 17), which is a pointer to a `PacketQueue` object (see Sect. 7.3.1). Finally, variable "`intr_`" (Line 18) is a dummy `Event` object, which represents a packet departure (from the transmitting node) event. As discussed in Sect. 4.3.7, the packet departure is scheduled using variable "`intr_`" which does not take part in event dispatching.[1]

The main functions of class `LinkDelay` are `recv(p,h)`, `send(p,h)`, `handle (e)`, and `txttime(p)`. Function `txttime(p)` calculates the packet transmission time of packet `*p` (Lines 10–12 in Program 7.3). Function `send(p,h)` sends packet `*p` to the connecting downstream object (see Line 12 in Program 5.3). Function `handle(e)` is invoked when the Scheduler dispatches an event corresponding to the `LinkDelay` object (see Chap. 4). Function `recv(p,h)`

[1] As a dummy `Event` object, variable "`intr_`" ensures that an error message will be shown on the screen, if an undispatched event is rescheduled.

Program 7.4 Function `recv(p,h)` of class `LinkDelay`

```
    //~ns/link/delay.cc
1   void LinkDelay::recv(Packet* p, Handler* h)
2   {
3       double txt = txtime(p);
4       Scheduler& s = Scheduler::instance();
5       if (dynamic_) { ... }
6       else if (avoidReordering_) { ... }
7       else {
8           s.schedule(target_, p, txt + delay_);
9       }
10      s.schedule(h, &intr_, txt);
11  }
```

(Program 7.4) takes a packet `*p` and a handler `*h` as input arguments, and schedules packet departure and packet arrival events.

1. *Packet departure event*: Since a packet spends "packet transmission time" (`txt` in Line 3) at the upstream object, function `recv(p,h)` schedules a packet departure event at "`txt`" seconds after the `LinkDelay` object receives the packet. To do so, Line 10 invokes function `schedule(h,&intr,txt)` of class `Scheduler`, where the first, second, and third input arguments are a handler pointer, a dummy event pointer, and delay, respectively (see Chap. 4). After "`txt`" seconds, the Scheduler dispatches this event by invoking function `handle(e)` associated with the handler `intr` to inform the upstream object of a packet departure. In most cases, the upstream object responds by transmitting another packet, if available (see Sect. 7.3.3 for the callback mechanism).
2. *Packet arrival*: Class `LinkDelay` also passes the packet to its downstream object (`*target_`). Line 8 schedules an event cast from the input packet `*p` with delay `txt+delay_` seconds, where "`txt`" is the packet transmission time and "`delay_`" is the link propagation delay. Here, `*target_` is passed to the function `schedule(...)` as a handler pointer. After "`txt+delay_`" seconds, `h.handle(p)` will invoke function `recv(p)` (see Program 4.2), and packet `*p` will be passed to `*target_` after `txt+delay_` seconds.

The major difference between scheduling packet departure and arrival events is as follows. While a node can hold only one (head of the line) packet, a link can contain more than one packet. Correspondingly, at an instance, a link can schedule only one packet departure event (using "`intr_`"), and more than one packet arrival event (using `*p` which represents a packet). Every time a `LinkDelay` object receives a packet, it schedules the packet departure event using the same variable "`intr_`." If variable "`intr_`" has not been dispatched, such a scheduling will cause runtime error, because it attempts to place a packet in the head of the buffer which is currently occupied by another packet. A packet arrival event, at the connecting node on the other hand, is tied to incoming packet. A `LinkDelay` object schedules a

new packet arrival event for every received packet (see Line 8 in Program 7.4). Therefore, a link can schedule another packet arrival event, even if the previous arrival event has not been dispatched. This is essentially the case for a link (with large bandwidth-delay product) which can contain several packets.

7.3 Buffer Management

Another major component of a `SimpleLink` object is a `Queue` object. Implemented with, class `Queue`, it models the buffering mechanism in a network router. It stores the received packets in the buffer and forwards them to its downstream object when the ongoing transmission is complete.

As shown in Program 7.5, class `Queue` derives from class `Connector` and can be used to connect two NsObjects. It uses a `PacketQueue` object (see Sect. 7.3.1), `*pq_` in Line 20, for packet buffering. The buffer size is specified in variable "`qlim_`" (Line 16). The variables "`blocked_`" (Line 17), "`unblock_on_resume_`" (Line 18), and "`qh_`" (Line 19) are related to the so-called callback mechanism and shall be discussed later in Sect. 7.3.3.

Program 7.5 Declaration of class `Queue`

```
    //~ns/queue/queue.h
1   class Queue : public Connector {
2    public:
3        virtual void enque(Packet*) = 0;
4        virtual Packet* deque() = 0;
5        virtual void recv(Packet*, Handler*);
6        void resume();
7        int blocked() const { return (blocked_ == 1); }
8        void unblock() { blocked_ = 0; }
9        void block() { blocked_ = 1; }
10       int limit() { return qlim_; }
11       int length() { return pq_->length(); }
12       virtual ~Queue();
13  protected:
14       Queue();
15       void reset();
16       int qlim_;
17       int blocked_;
18       int unblock_on_resume_;
19       QueueHandler qh_;
20       PacketQueue *pq_;
21  };
```

Program 7.6 Declaration of class `PacketQueue`

```
    //~ns/queue/queue.h
1   class PacketQueue : public TclObject {
2       public:
3           PacketQueue() : head_(0), tail_(0), len_(0),
            bytes_(0) {}
4           virtual int length() const { return (len_); }
5           virtual Packet* enque(Packet* p);
6           virtual Packet* deque();
7           virtual void remove(Packet*);
8           Packet* head() { return head_; }
9           Packet* tail() { return tail_; }
10      protected:
11          Packet* head_;
12          Packet* tail_;
13          int len_;
14  };
```

There are a number of important functions of class `Queue`. Function `enque(p)` and `deque()` (Lines 3 and 4) place and take, respectively, a packet from the `PacketQueue` object `*pq_`. They are declared as pure virtual and must be implemented by instantiable derived classes of class `Queue`. Inherited from class `NsObject`, the function `recv(p,h)` (Line 5) is the main packet reception function. Function `blocked()` in Line 7 indicates whether the `Queue` object is in a blocked state. Functions `resume()` (Line 6), `unblock()` (Line 8), and `block()` (Line 9) are used in the callback mechanism which will be discussed in Sect. 7.3.3. Finally, functions `limit()` and `length()` return the buffer size and current buffer occupancy, respectively.

7.3.1 *Class `PacketQueue`: A Model for Packet Buffering*

Declared in Program 7.6, class `PacketQueue` models low-level operations of the buffer including storing, enqueuing, and dequeuing packet. It contains several variables and functions which implement a link list of `Packets`. Variable "`head_`" in Line 11 is the pointer to the beginning of the link list. Variable "`tail_`" in Line 12 is the pointer to the end of the link list. The variable "`len_`" in Line 13 is the number of packets in the buffer. Function `enque(p)` in Line 5 puts the input packet `*p` to the end of the buffer. Function `deque()` in Line 6 returns the head of the line `Packet` pointer or returns `NULL` when the buffer is nonempty or empty, respectively. Function `remove(p)` in Line 7 searches for a matching packet `*p` and removes it from the buffer (if found). Note that packet admitting/dropping is the functionality of class `Queue`, not of class `PacketQueue`. We will show an example of packet admitting/dropping of class `DropTail` in Sect. 7.3.4.

Program 7.7 Declaration and function `handle` of class `QueueHandler`, and the constructor of class `Queue`

```
     //~ns/queue/queue.h
1    class QueueHandler : public Handler {
2    public:
3       inline QueueHandler(Queue& q) : queue_(q) {}
4       void handle(Event*);
5    private:
6       Queue& queue_;
7    };

     //~ns/queue/queue.cc
8    void QueueHandler::handle(Event*)
9    {
10      queue_.resume();
11   }

12   Queue::Queue() : Connector(), blocked_(0),
                      unblock_on_resume_(1), qh_(*this),pq_(0)
13   { ... }
```

7.3.2 *Queue Handler*

Derived from class `Handler` (see Line 1 in Program 7.7), class `QueueHandler` is closely related to the (event) Scheduler. Again, a `QueueHandler` object defines its default actions in its function `handle(e)`. These default actions will be taken when an associated event is dispatched. As shown in Lines 8–11 of Program 7.7, the default action of a `QueueHandler` object is to execute function `resume()` of the associated `Queue` object "`queue_`." We will discuss the details of function `resume()` in Sect. 7.3.3. In the rest of this section, we will demonstrate how a connection between `QueueHandler` and `Queue` objects is created.

To associate a `Queue` object with a `QueueHandler` object, classes `Queue` and `QueueHandler` declare their member variables "`qh_`" (Line 19 in Program 7.5) and "`queue_`" (Line 6 in Program 7.7), as a `QueueHandler` pointer and a `Queue` reference, respectively. These two variables are initialized when a `Queue` object is instantiated (Line 12 in Program 7.7). The constructor of class `Queue` invokes the constructor of class `QueueHandler`, feeding itself as an input argument (i.e., `qh_(*this)`). The constructor of "`qh_`" then sets its member variable "`queue_`" to share the same address as the input `Queue` object (i.e., `queue_(q)` in Line 3 of Program 7.7). These two constructors create a two-way connection between the `Queue` and `QueueHandler` objects. After this point, the `Queue` and the `QueueHandler` objects refer to each other by the variables `qh_` and `queue_`, respectively.

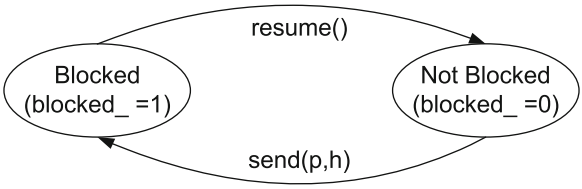

Fig. 7.2 State diagram of the queue blocking mechanism

7.3.3 Queue Blocking and Callback Mechanism

7.3.3.1 Queue Blocking

NS2 uses the concept of *queue blocking*[2] to indicate whether a queue is currently transmitting a packet. By default, a queue can transmit one packet at a time. It is not allowed (i.e., blocked) to transmit another packet until the ongoing transmission is complete. A queue is said to be blocked or unblocked (i.e., `blocked_ = 1` or `blocked_ = 0`), when it is transmitting a packet or is not transmitting a packet, respectively.

Figure 7.2 shows the state diagram of the queue blocking mechanism. When in the "Not Blocked" state, a queue is allowed to transmit a packet by executing "`target_->recv(p,&qh_)`," after which it enters the "Blocked" state. Here, a queue waits until the ongoing transmission is complete where the function `resume()` is invoked. After this point, the queue enters the "Not Blocked" state and the process repeats.

7.3.3.2 Callback Mechanism

As discussed in Chap. 5, a node in NS2 passes packets to a downstream node by executing function `recv(p,h)`, where `*p` denotes a packet and `*h` denotes a handler. A callback mechanism refers to a process where a downstream object invokes an upstream object along the downstream path for a certain purpose. In a queue blocking process, a callback mechanism occurs when a downstream object `Queue` object by invoking function `resume()` of the unblocks an upstream `Queue` object.

We now explain the callback mechanism process for queue unblocking via an example network in Fig. 7.3. Here, we assume that the following objects are sequentially connected: an upstream NsObject, a `Queue` object, a `LinkDelay` object, and a downstream NsObject. Again, an NsObject passes a packet `*p` by invoking function `recv(p,h)` of its downstream object, where `*h` is a handler. In most cases, the input handler `*h` is passed along with the packet `*p` as input argument of function `recv(p,h)`. However, this mechanism is different for `Queue` objects.

[2]Queue blocking has no relation to packet blocking when the buffer is full.

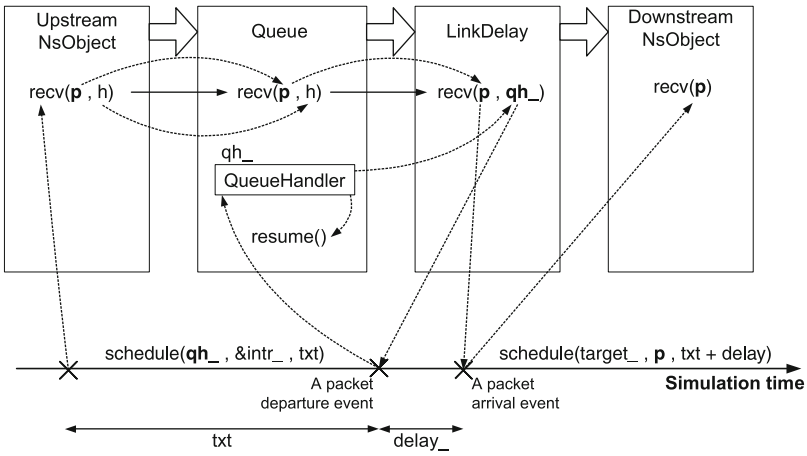

Fig. 7.3 Diagram of callback mechanism for a queue unblocking process

Program 7.8 Function `recv` of class `Queue`

```
    //~ns/queue/queue.cc
1   void Queue::recv(Packet* p, Handler*)
2   {
3       enque(p);
4       if (!blocked_) {
5           p = deque();
6           if (p != 0) {
7               blocked_ = 1;
8               target_->recv(p, &qh_);
9           }
10      }
11  }
```

Consider function `recv(p,h)` of class `Queue` in Program 7.8. Instead of immediately passing the incoming packet `*p` to its downstream object, Line 3 places the packet in the buffer. Again, a `Queue` object is allowed to transmit a packet only when it is not blocked (Line 4). In this case, Line 5 retrieves a packet from the buffer. If the packet exists (Line 6), Line 7 will set the state of the `Queue` object to be "blocked," and Line 8 will forward the packet to its downstream object (i.e., `*target_`). The `Queue` object passes its `QueueHandler` pointer "`qh_`" (instead of the incoming handler pointer) to its downstream object. This `QueueHandler` pointer acts as a reference point for a queue blocking callback mechanism.

From Fig. 7.3, the downstream object of the `Queue` object is a `LinkDelay` object. Upon receiving a packet, it schedules two events: packet departure and

Program 7.9 Function `resume` of class `Queue`

```
    //~ns/queue/queue.cc
1   void Queue::resume()
2   {
3       Packet* p = deque();
4       if (p != 0)
5           target_->recv(p, &qh_);
6       else
7           if (unblock_on_resume_)
8               blocked_ = 0;
9           else
10              blocked_ = 1;
11  }
```

arrival events (see Lines 10 and 8 in Program 7.4). A packet arrival event is associated with the downstream object (i.e., `*target_`). At the firing time, the function `handle(p)` of the downstream object will invoke function `recv(p)` to receive packet `*p` (see Program 4.2).

Function `recv(p)` of class `LinkDelay` also schedules a packet departure event. The departure event is associated with the `QueueHandler` object "`qh_`." At the firing time, the Scheduler invokes function `handle(p)` of the associated `QueueHandler` object `qh_`. In Program 7.7, this function in turn invokes function `resume()` to unblock the associated `Queue` object. Essentially, the `LinkDelay` object schedules an event which *calls back* to unblock the upstream `Queue` object.

Program 7.9 shows the details of function `resume()` invoked when the ongoing transmission is complete. Function `resume()` first retrieves the head of the line packet from the buffer (Line 3). If the buffer is nonempty (Line 4), Line 5 will send the packet to the downstream object of the queue. If the queue is idle (i.e., the buffer is empty), variable "`blocked_`" will be set to zero and one in case that the flag "`unblock_on_resume_`" is one and zero, respectively.

7.3.4 Class `DropTail`: A Child Class of Class `Queue`

Consider class `DropTail`, a child class of class `Queue`, which is bound to the OTcl class `Queue/DropTail` in Program 7.10. The constructor of class `DropTail` creates a pointer "`q_`" (Line 13) to a `PacketQueue` object and sets "`pq_`" derived from class `Queue` to be the same as "`q_`" (Line 5). Throughout the implementation, class `DropTail` refers to its buffer by "`q_`" instead of "`pq_`." Class `DropTail` overrides function `enque(p)` (Line 11 and Program 7.11) and `deque()` (Line 12) of class `Queue`. It also allows packet dropping at the front of the buffer, if the flag "`drop_front_`" (Line 14) is set to 1. Class `DropTail` does not override function `recv(p,h)`. Therefore, it receives a packet through the function `recv(p,h)` of class `Queue`.

Program 7.10 Declaration of class `DropTail`

```
    //~ns/queue/drop-tail.h
 1  class DropTail : public Queue {
 2    public:
 3      DropTail() {
 4          q_ = new PacketQueue;
 5          pq_ = q_;
 6          bind_bool("drop_front_", &drop_front_);
 7      };
 8      ~DropTail() { delete q_; };
 9    protected:
10      int command(int argc, const char*const* argv);
11      void enque(Packet*);
12      Packet* deque();
13      PacketQueue *q_;
14      int drop_front_;
15  };

    //~ns/queue/drop-tail.cc
16  static class DropTailClass : public TclClass {
17      public:
18      DropTailClass() : TclClass("Queue/DropTail") {}
19      TclObject* create(int, const char*const*) {
20          return (new DropTail);
21      }
22  } class_drop_tail;
```

Program 7.11 Function `enque` of class `DropTail`

```
    //~ns/queue/drop-tail.cc
 1  void DropTail::enque(Packet* p)
 2  {
 3      if ((q_->length() + 1) >= qlim_)
 4          if (drop_front_) {
 5              q_->enque(p);
 6              Packet *pp = q_->deque();
 7              drop(pp);
 8          } else
 9              drop(p);
10      else
11          q_->enque(p);
12  }
```

In Program 7.11, the function `enque(p)` first checks whether the incoming packet will cause buffer overflow (Line 3). If so, it will drop the packet either from the front (Lines 5–7) or from the tail (Line 9), where function `drop(p)` (Lines 7 and 9) belongs to class `Connector` (see Program 5.4). If the buffer has enough space, Line 11 will enqueue packet (`p`) to its buffer (`q_`).

7.4 A Sample Two-Node Network

We have introduced two basic NS2 components: nodes and links. Based on these two components, we now create a two-node network with a unidirectional link and show the packet flow mechanism within this network in Fig. 7.4.

Fig. 7.4 A two-node network with a unidirectional link and the instprocs of class `Simulator`

7.4.1 Network Construction

The network in Fig. 7.4 consists of a beginning node (`n1`), a termination node (`n2`), a `SimpleLink` connecting `n1` and `n2`, a source transport layer agent (`udp`), and a sink transport layer agent (`null`). This network can be created using the following Tcl simulation script:

```
set ns [new Simulator]
set n1 [$ns node]
set n2 [$ns node]
$ns simplex-link $n1 $n2 <bw> <delay> DropTail
set udp [new Agent/UDP]
set null [new Agent/Null]
$ns attach-agent $n1 $udp
$ns attach-agent $n2 $null
```

Here, a command "`$ns node`" creates a `Node` object. The internal mechanism of the node construction process was described in Sect. 6.5. The statement "`$ns simplex-link $n1 $n2 <bw> <delay> DropTail`" creates a unidirectional `SimpleLink` object, which connects node `$n1` to node `$n2`. The link bandwidth and delay are `<bw>` bps and `<delay>` seconds, respectively. The buffer in the link is of class `DropTail`. From Sect. 6.5.3, the commands "`$ns attach-agent $n1 $udp`" and "`$ns attach-agent $n2 $null`" set the target of the agent "`udp`" to be the entry of Node `$n1` and installs agent `$null` in the demultiplexer of Node `$n2`.

7.4.2 Packet Flow Mechanism

To deliver a packet "`*p`" from agent `$udp` to `$null`,

1. Agent $udp sends the packet *p to the entry of Node $n1.[3]
2. Packet *p is sent to the head classifier "classifier_" (which is of class DestHashClassifier) of Node $n1.
3. The DestHashClassifier object "classifier_" examines the header of the packet *p. In this case, the packet is destined to the Node $n2. Therefore, it forwards the packet to the link head of the connecting SimpleLink object.
4. The link head forwards the packet to the connecting Queue object.
5. The Queue object enqueues the packet. If not blocked, it will forward the head of the line packet to the connecting LinkDelay object and set its status to blocked.
6. Upon receiving a packet, the LinkDelay object schedules the two following events:
   a. Packet departure event, which indicates that packet transmission is complete. This event unblocks the associated Queue object.
   b. Packet arrival event, which indicates the packet arrival at the connecting TTLChecker object.
7. The TTLChecker object receives the packet and decrements the TTL field of the packet header. If the TTL field of the packet is nonpositive, the TTLChecker object will drop the packet. Otherwise, it will forward the packet to the entry of Node $n2 (see file *~ns*/common/ttl.cc).
8. Node $n2 forwards the packet to the head classifier (classifier_). Since the packet is destined to itself, the packet is forwarded to the demultiplexer (dmux_).
9. The demultiplexer forwards the packet to the agent $null installed in the demultiplexer.

7.5 Chapter Summary

This chapter focuses on class SimpleLink, a basic link class that can be used to connect two nodes. The connection between two nodes $n1 and $n2 can be created by the following instprocs:

```
$ns simplex-link $n1 $n2 <bw> <delay> <queue_type>
$ns duplex-link $n1 $n2 <bw> <delay> <queue_type>
```

where the bandwidth and delay of the SimpleLink object are <bw> bps and <delay> seconds, respectively. Also the type of queue implemented in the SimpleLink object is <queue_type>.

A SimpleLink object models packet transmission time, link propagation delay, and packet buffering. Here, packet transmission time is the time required

[3]Note that, each object sends a packet *p to its downstream object by invoking target_->recv(p,h), where target_ is a pointer to the downstream object.

to transmit a packet and is computed by $\frac{\text{packet size}}{\text{bandwidth}}$, while the link propagation time is the time required to deliver a data bit from the beginning to the end of the `SimpleLink` object. These two attributes are implemented in the C++ class LinkDelay. Packet buffering is implemented in the abstract class `Queue`. Finally, classes `LinkDelay` and `Queue`, together, help model packet arrival and departure event.

7.6 Exercises

1. What are features of a simple link? Draw a diagram and explain each of its object components. What are the purposes, and the OTcl and C++ classes of those components?
2. Define packet transmission time and propagation delay. What are their differences? Explain your answer using one example.
3. What are the OTcl and C++ classes responsible to model packet transmission time and propagation delay? Draw a diagram and explain how NS2 implements packet latency during packet forwarding.
4. What are the purposes of the variable "`intr_`" in class `LinkDelay`?
5. Lines 8 and 10 of Program 7.4 invoke the function `schedule(...)` of the `Scheduler` object. What do these lines do?
6. What are the differences/similarities between C++ classes `PacketQueue` and `Queue`?
7. Explain the relationship among class `QueueHandler`, `Queue`, `LinkDelay`, `Packet`, and `Scheduler`. Draw a diagram and explain the callback mechanism.
8. Explain how a `DropTail` queue receives and drops packets from its tail when its buffer is full.
9. Write an NS2 statement which creates a `SimpleLink` object whose bandwidth is 2 Mbps, propagation delay is 10 ms, and the queue type is DropTail.

Chapter 8
Packets, Packet Headers, and Header Format

Generally, a packet consists of packet header and data payload. Packet header stores packet attributes (e.g., source and destination IP addresses) necessary for packet delivery, while data payload contains user information. Although this concept is typical in practice, NS2 models packets differently.

In most cases, NS2 extracts information from data payload and stores the information into packet header. This idea removes the need to process data payload at runtime. For example, instead of counting the number of bits in a packet, NS2 stores packet size in the variable `hdr_cmn::size_` (see Sect. 8.3.5), and accesses this variable at runtime.[1]

This chapter discusses how NS2 models packets. Section 8.1 gives an overview on NS2 packet modeling. Section 8.2 discusses the packet allocation and deallocation processes. Sections 8.3 and 8.4 show the details of packet header and data payload, respectively. We give a guideline of how to customize packets (i.e., to define a new payload type and activate/deactivate new and existing protocols) in Sect. 8.5. Finally, the chapter summary is given in Sect. 8.6.

8.1 An Overview of Packet Modeling Principle

8.1.1 Packet Architecture

Figure 8.1 shows the architecture of an NS2 packet model. From Fig. 8.1, a packet model consists of four main parts: actual packet, class `Packet`, protocol-specific headers, and packet header manager.

- **Actual Packet:** An actual packet refers to the portion of memory which stores packet header and data payload. NS2 does not directly access either the

[1]For example, class `LinkDelay` determines packet size from a variable `hdr_cmn::size_` when computing packet transmission time (see Line 11 of Program 7.3).

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_8, © Springer Science+Business Media, LLC 2012

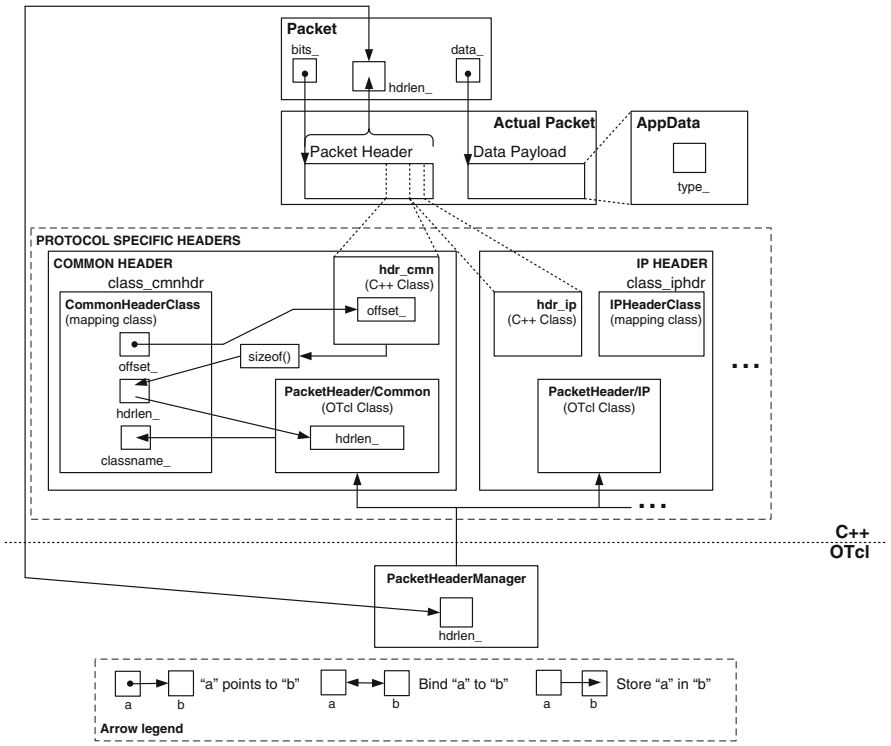

Fig. 8.1 Packet modeling in NS2

packet header or the data payload. Rather, it uses member variables "`bits_`" and "`data_`" of class `Packet` to access packet header and data payload, respectively. The details of packet header and data payload will be given in Sects. 8.3 and 8.4, respectively.

- **Class `Packet`:** Declared in Program 8.1, class `Packet` is the C++ main class which represents packets. It contains the following variables and functions:
  - **C++ Variables of Class `Packet`**

`bits_`	A string which contains packet header
`data_`	A pointer to an `AppData` object which contains data payload
`fflag_`	Set to `true` if the packet is currently referred to by other objects and `false` otherwise
`free_`	A pointer to the head of the packet free list
`ref_count_`	The number of objects which currently refer to the packet
`next_`	A pointer to the next packet in the link list of packets
`hdr_len_`	Length of packet header

Program 8.1 Declaration of class `Packet`

```
    //~/ns/common/packet.h
 1  class Packet : public Event {
 2  private:
 3      unsigned char* bits_;
 4      AppData* data_;
 5      static void init(Packet*) {bzero(p->bits_, hdrlen_);}
 6      bool fflag_;
 7  protected:
 8      static Packet* free_;
 9      int    ref_count_;
10  public:
11      Packet* next_;
12      static int hdrlen_;

        //Packet Allocation and Deallocation
13      Packet() : bits_(0), data_(0), ref_count_(0), next_(0) { }
14      inline unsigned char* const bits() { return (bits_); }
15      inline Packet* copy() const;
16      inline Packet* refcopy() { ++ref_count_; return this; }
17      inline int& ref_count() { return (ref_count_); }
18      static inline Packet* alloc();
19      static inline Packet* alloc(int);
20      inline void allocdata(int);
21      static inline void free(Packet*);

        //Packet Access
22      inline unsigned char* access(int off){return &bits_[off]);};
23  }
```

– **C++ Functions of Class `Packet`**

`init(p)`	Clear the packet header `*bits_` of the input packet `*p`.
`copy()`	Return a pointer to a duplicated packet.
`refcopy()`	Increase the number of objects, which refer to the packet, by one.
`alloc()`	Create a new packet and return the pointer to the created packet.
`alloc(n)`	Create a new packet with "`n`" bytes of data payload and return a pointer to the created packet.
`allocdata(n)`	Allocate "`n`" bytes of data payload to the variable `data_`.
`free(p)`	Deallocate the packet "`p`."
`access(off)`	Retrieve a reference to a certain point (specified by the offset "`off`") of the variable "`bits_`" (i.e., packet header).

- **Protocol Specific Header:** From Fig. 8.1, packet header consists of several protocol-specific headers. Each protocol-specific header uses a contiguous portion of packet header to store its packet attributes. In common with most TclObjects, there are three classes related to each protocol-specific header:
  - A C++ class (e.g., `hdr_cmn` or `hdr_ip`) provides a structure to store packet attributes.
  - An OTcl class (e.g., `PacketHeader/Common` or `PacketHeader/IP`) acts as an interface to the OTcl domain. NS2 uses this class to configure packet header from the OTcl domain.
  - A mapping class (e.g., `CommonHeaderClass` or `IPHeaderClass`) binds a C++ class to an OTcl class.

  We will discuss the details of protocol-specific header later in Sect. 8.3.5.
- **Packet Header Manager:** A packet header manager maintains a list of active protocols and configures all active protocol-specific headers to setup packet header. It has an instvar "`hdrlen_`" which indicates the length of packet header consisting of protocol-specific headers. The instvar "`hdrlen_`" is bound to a variable "`hdrlen_`" of class `Packet`. Any change in one of these two variables will result in an automatic change in another.
- **Data Payload:** From Line 4 in Program 8.1, the pointer "`data_`" points to data payload, which is of class `AppData`. We will discuss the details of data payload in Sect. 8.4.

8.1.2 A Packet as an Event: A Delayed Packet Reception Event

Derived from class `Event` (Line 1 in Program 8.1), class `Packet` can be placed on the simulation time line (see the details in Chap. 4). In Sect. 4.2, we mentioned two main classes derived from class `Event`: class `AtEvent` and class `Packet`. We also mentioned that an `AtEvent` object is an event created by a user from a Tcl simulation script. This section discusses details of another derived class of class `Event`: class `Packet`.

As discussed in Sect. 5.2.2, NS2 implements *delayed packet forwarding* by placing a packet reception event on the simulation timeline at a certain delayed time. Derived from class `Event`, class `Packet` can be placed on the simulation timeline to signify a delayed packet reception. For example, the following statement (see Line 8 in Program 7.4) schedules a packet reception event, where the NsObject `*target_` receives a packet `*p` at `txt+delay_` seconds in future:

```
s.schedule(target_, p, txt + delay_)
```

Note that a `Packet` pointer is cast to be an `Event` pointer before being fed as the second input argument of the function `schedule(...)`.

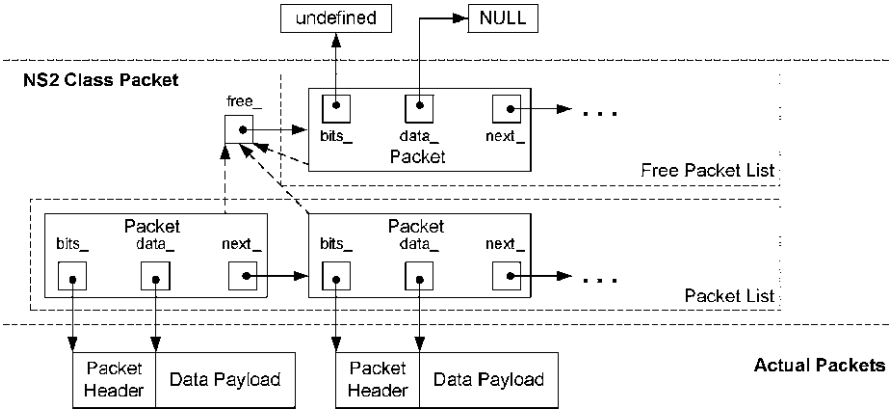

Fig. 8.2 A link list of packets and a free packet list

At the firing time, the Scheduler dispatches the scheduled event (i.e., `*p`) and invokes `target->handle(p)`, which executes "`target_->recv(p)`" to forward packet `*p` to the NsObject pointer `*target_`.

8.1.3 A Link List of Packets

Apart from the above four main packet components, a `Packet` object contains a pointer "`next_`" (Line 11 in Program 8.1), which helps formulating a link list of `Packet` objects (e.g., Packet List in Fig. 8.2). Program 8.2 shows the implementation of functions `enque(p)` and `deque()` of class `PacketQueue`. Function `enque(p)` (Lines 3–13) puts a `Packet` object `*p` to the end of the queue. If the `PacketQueue` is empty, NS2 sets "`hdrlen_`," "`tail_`," and "`p`" to point to the same place[2] (Line 5). Otherwise, Lines 7 and 8 set `*p` as the last packet in the `PacketQueue`, and shift variable "`tail_`" to the last packet pointer "`p`." Since the pointer "`tail_`" is the last pointer of `PacketQueue`, Line 10 sets the pointer `tail_->next_` to 0 (i.e, points to `NULL`).

Function `deque()` (Lines 14–21) retrieves a pointer to the packet at the head of the buffer. If there is no packet in the buffer, the function `deque()` will return a `NULL` pointer (Line 15). If the buffer is not empty, Line 17 will shift the pointer "`head_`" to the next packet, Line 19 will decrease the length of `PacketQueue` object by one, and Line 20 will return the `packet` pointer "`p`" which was set to the pointer "`head_`" in Line 16.

[2]Note that, `head_` and `tail_` are pointers to the first and the last `Packet` objects, respectively, in a `PacketQueue` object.

Program 8.2 Functions `enque` and `deque` of class `PacketQueue`

```
    //~/ns/common/queue.h
1   class PacketQueue : public TclObject {
2       ...
3       virtual Packet* enque(Packet* p) {
4           Packet* pt = tail_;
5           if (!tail_) head_= tail_= p;
6           else {
7               tail_->next_= p;
8               tail_= p;
9           }
10          tail_->next_= 0;
11          ++len_;
12          return pt;
13      }

14      virtual Packet* deque() {
15          if (!head_) return 0;
16          Packet* p = head_;
17          head_= p->next_; // 0 if p == tail_
18          if (p == tail_) head_= tail_= 0;
19          --len_;
20          return p;
21      }
22      ...
23  };
```

8.1.4 Free Packet List

Unlike most NS2 objects, a `Packet` object, once created, will not be destroyed until the simulation terminates. NS2 keeps `Packet` objects which are no longer in use in a *free packet list* (see Fig. 8.2). When NS2 needs a new packet, it first checks whether the free packet list is empty. If not, it will take a `Packet` object from the list. Otherwise, it will create another `Packet` object. We will discuss the details of how to *allocate* and *deallocate* a `Packet` object later in Sect. 8.2.

There are two variables which are closely related to the packet allocation/deallocation process: "`fflag_`" and "`free_`." Each `Packet` object uses a variable "`fflag_`" (Line 6 in Program 8.1) to indicate whether it is in use. The variable "`fflag_`" is set to `true`, when the `Packet` object is in use, and set to `false` otherwise. Shared by all the `Packet` objects, a static pointer "`free_`" (Line 8 in Program 8.1) is a pointer to the first packet on the free packet list. Each packet on the free packet list uses its variable "`next_`" to form a link list of free `Packet` objects. This link list of free packets is referred to as a *free packet list*. Although NS2 does not return memory allocated to a `Packet` object to the system, it does return the memory used by packet header (i.e., "`bits_`") and data payload (i.e., "`data_`") to the system (see Sect. 8.2.2), when the packet is deallocated. Since most

memory required to store a Packet object is consumed by packet header and data payload, maintaining a free packet list does not result in a significant waste of memory.

8.2 Packet Allocation and Deallocation

Unlike most of the NS2 objects,[3] a Packet object is allocated and deallocated using static functions alloc() and free(p) of class Packet, respectively. If possible, function alloc() takes a Packet object from the free packet list. Only when the free packet list is empty, does the function alloc() creates a new Packet object using "new". Function free(p) deallocates a Packet object, by returning the memory allocated for packet header and data payload to the system and storing the not-in-use Packet pointer "p" in the free packet list for future reuse. The details of packet allocation and deallocation will be discussed below.

8.2.1 Packet Allocation

Program 8.3 shows details of the function alloc() of class Packet, the packet allocation function. The function alloc() returns a pointer to an allocated Packet object to the caller. This function consists of two parts: packet allocation in Lines 3–15 and packet initialization in Lines 16–22.

Consider the packet allocation in Lines 3–15. Line 3 declares "p" as a pointer to a Packet object and sets the pointer "p" to point to the first packet on the free packet list.[4] If the free packet list is empty (i.e., p = 0), NS2 will create a new Packet object (in Line 11) and allocate memory space with size "hdrlen_" bytes for the packet header in Line 12. The variable "hdrlen_" is not configured during the construction of a Packet object. Rather, it is set up in the Network Configuration Phase (see Sect. 8.3.8) and is used by the function alloc() to create packet header.

Function alloc() does not allocate memory space for data payload. When necessary, NS2 creates data payload using the function allocdata(n) (see Lines 8–14 in Program 8.4), which will be discussed in detail later in this section.

If the free packet list is nonempty, the function alloc() will execute Lines 5–9 in Program 8.3 (see also the diagram in Fig. 8.3). In this case, the function alloc() first makes sure that nobody is using the Packet object "*p," by asserting that

[3]Generally, NS2 creates and destroys most objects using procedures new{...} and delete{...}, respectively.

[4]Again, "free_" is the pointer to the first packet on the free packet list.

Program 8.3 Function `alloc` of class `Packet`

```
    //~/ns/common/packet.h
1   inline Packet* Packet::alloc()
2   {
        //Packet Allocation
3       Packet* p = free_;
4       if (p != 0) {
5           assert(p->fflag_ == FALSE);
6           free_ = p->next_;
7           assert(p->data_ == 0);
8           p->uid_ = 0;
9           p->time_ = 0;
10      } else {
11          p = new Packet;
12          p->bits_ = new unsigned char[hdrlen_];
13          if (p == 0 || p->bits_ == 0)
14              abort();
15      }

        //Packet Initialization
16      init(p); // Initialize bits_[]
17      (HDR_CMN(p))->next_hop_ = -2; // -1 reserved for
           IP_BROADCAST
18      (HDR_CMN(p))->last_hop_ = -2; // -1 reserved for
           IP_BROADCAST
19      p->fflag_ = TRUE;
20      (HDR_CMN(p))->direction() = hdr_cmn::DOWN;
21      p->next_ = 0;
22      return (p);
23  }
```

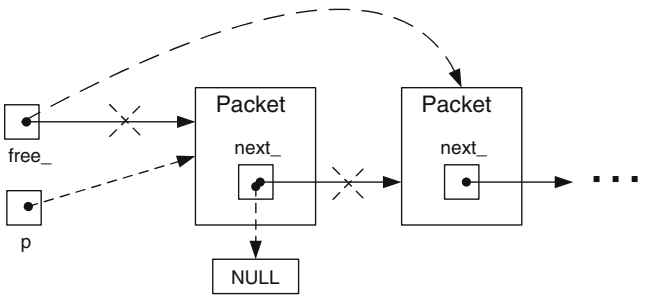

Fig. 8.3 Diagram of packet allocation when the free packet list is nonempty. The *dotted lines* show the actions caused by the function `alloc` of class `Packet`

Program 8.4 Functions `alloc`, `allocdata`, and `copy` of class `Packet`

```
    //~/ns/common/packet.h
 1  inline Packet* Packet::alloc(int n)
 2  {
 3      Packet* p = alloc();
 4      if (n > 0)
 5          p->allocdata(n);
 6      return (p);
 7  }

 8  inline void Packet::allocdata(int n)
 9  {
10      assert(data_ == 0);
11      data_ = new PacketData(n);
12      if (data_ == 0)
13          abort();
14  }

15  inline Packet* Packet::copy() const
16  {
17      Packet* p = alloc();
18      memcpy(p->bits(), bits_, hdrlen_);
19      if (data_)
20          p->data_ = data_->copy();
21      return (p);
22  }
```

"`fflag_`" is `false` (Line 5).[5] Then, Line 6 shifts the pointer "`free_`" by one position. Lines 8–9 initialize two variables ("`uid_`" and "`time_`") of class `Event` (i.e., the mother class of class `Packet`) to be zero. Line 21 removes the packet from the free list by setting `p->next_` to zero.

After the packet allocation process is complete, Lines 16–22 initialize the allocated `Packet` object. Line 16 invokes function `init(p)`, which initializes the header of packet `*p`. From Line 5 in Program 8.1, invocation of function `init(p)` executes "`bzero(p-> bits_,hdrlen_)`," which clears "`bits_`" to zero.[6] Line 19 sets `fflag_` to be `true`, indicating that the packet `*p` is now in use. Line 21 sets the pointer `p->next_` to be zero. Lines 17, 18, and 20 initialize the common header. We will discuss packet header in greater detail in Sect. 8.3.2.

[5]The C++ function `assert(cond)` can be used for an integrity check. It does nothing if the input argument "`cond`" is `true`. Otherwise, it will initiate an error handling process (e.g., showing an error on the screen).

[6]Function `bzero(...)` takes two arguments – the first is a pointer to the buffer and the second is the size of the buffer – and sets all values in a buffer to zero.

Apart from the function `alloc()`, other relevant functions include `alloc(n)`, `allocdata(n)`, and `copy()` (see Program 8.4). The function `allocdata(n)` data allocates a packet (Line 3), and invokes `allocdata(n)` (Line 5). The function `allocdata(n)` creates data payload with size "n" bytes (by invoking `new Packet Data(n)` in Line 11). We will discuss the details of data payload later in Sect. 8.4.

Function `copy()` returns a replica of the current `Packet` object. The only difference between the current and the replicated `Packet` objects is the unique ID (`uid_`) field. This function is quite useful, since we often need to create a packet which is the same as or slightly different from an original packet. This function first allocates a packet in Line 17. Then, it copies packet header and data payload to the created packet `*p` in Lines 18 and 20, respectively.

Despite its name, function `refcopy()` (Line 16 in Program 8.1) does not create a copy of a `Packet` object. Rather, it returns the pointer to the current `Packet` object and increment the variable `ref_count_` bg 1. The variable `ref_count_` keeps track of the number of objects which share the same `Packet` object. It is initialized to 0 in the constructor of class `Packet` (Line 13 in Program 8.1), and is incremented by one when the function `ref_copy()` (Line 16 in Program 8.1) is invoked, indicating that a new object starts using the current `Packet` object. Similarly, it is decremented by one when the function `free(p)` (see Sect. 8.2.2) is invoked, indicating that an object has stopped using the current `Packet` object.

8.2.2 Packet Deallocation

When a packet `*p` is no longer in use, NS2 deallocates the packet using a function `free(p)`. By deallocation, NS2 returns the memory used to store packet header and data payload to the system, sets the pointer "`data_`" to zero, and stores the `Packet` object in the free packet list. Note that although the value of "`bits_`" is not set to zero, the memory location stored in "`bits_`" is no longer accessible. It is very important not to use "`bits_`" after packet deallocation. Otherwise, NS2 will encounter a (memory share violation) runtime error.

Details of the function `free(Packet*)` are shown in Program 8.5. Before returning a `Packet` object to the free packet list, we need to make sure that

1. The packet is in use (i.e., `p->fflag_ = 1` in Line 3), since there is no point in deallocating a packet which has already been deallocated.
2. No object is using the packet. In other words, the variable `ref_count_` is zero (Line 4), where `ref_count_` stores the number of objects which are currently using this packet.
3. The packet is no longer on the simulation time line (i.e., `p->uid_<=0` in Line 5). Deallocating a packet while it is still on the simulation timeline will

Program 8.5 Function `free` of class `Packet`

```
    //~/ns/common/packet.h
1   inline void Packet::free(Packet* p)
2   {
3       if (p->fflag_) {
4           if (p->ref_count_ == 0) {
5               assert(p->uid_ <= 0);
6               if (p->data_ != 0) {
7                   delete p->data_;
8                   p->data_ = 0;
9               }
10              init(p);
11              p->next_ = free_;
12              free_ = p;
13              p->fflag_ = FALSE;
14          } else {
15              --p->ref_count_;
16          }
17      }
18  }
```

cause event mis-sequencing and runtime error. Line 5 asserts that the event unique ID corresponding to the `Packet` object "p" (i.e., `p->uid_`) is non-positive, and therefore is no longer on the simulation timeline.[7]

NS2 allows more than one simulation object to share the same `Packet` object. To deallocate a packet, NS2 must ensure that the packet is no longer used by *any* simulation object. Again, NS2 keeps the number of objects sharing a packet in the variable `ref_count_`. If `ref_count_>0`, meaning an object is invoking the function `free(p)` while other objects are still using the packet `*p`, the function `free(p)` will simply reduce `ref_count_` by one, indicating that one object stops using the packet (Line 15).[8] On the other hand, if `ref_count_` is zero, meaning that no other object is using the packet, Lines 5–13 will then clear packet header and data payload and store the `Packet` object in the free packet list.

If all the above three conditions are satisfied, function `free(p)` will execute Lines 6–13 in Program 8.5. The schematic diagram for this part is shown in Fig. 8.4. Line 7 returns the memory used by data payload to the system. Line 8 sets the pointer "`data_`" to zero. Line 10 returns the memory used by header of the packet `*p` to the system by invoking the function `init(p)` (see Line 5 of Program 8.1). Lines 11 and 12 place the packet as the first packet on the free packet list. Finally, Line 13 sets "`fflag_`" to `false`, indicating that the packet is no longer in use.

[7] From Fig. 4.2, an event with positive unique ID (e.g, "`uid_`" is 2 or 6) was scheduled but has not been dispatched.

[8] If the `Packet` object is deallocated when `ref_count_ > 0`, simulation objects may later try to access the deallocated `Packet` object and cause a runtime error.

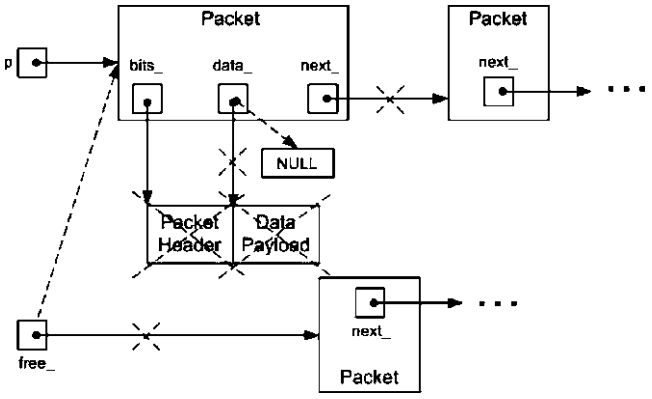

Fig. 8.4 The process of returning a packet to the packet free list. The *dotted lines* show the action caused by the function `free` of class `Packet`

8.3 Packet Header

As a part of a packet, packet header contains packet attributes such as packet unique ID and IP address. Again, packet header is stored in the variable "`bits_`" of class `Packet` (see Line 3 of Program 8.1). The variable "`bits_`" is declared as a string (i.e., a *Bag of Bits* (BOB)) and has no structure to store packet attributes. However, NS2 imposes a two-level structure on variable "`bits_`," as shown in Fig. 8.5.

The first level divides the entire packet header into protocol-specific headers. The location allocated to each protocol specific header on "`bits_`" is identified by its variable `offset_`. The second level imposes a packet attribute-storing structure on each protocol-specific header. On this level, packet attributes are stored as members of a C++ `struct` data type.

In practice, a packet contains only relevant protocol-specific headers. An NS2 packet, on the other hand, includes *all* protocol-specific headers into a packet header, regardless of packet type. Every packet uses the same amount of memory to store the packet header. The amount of memory is stored in the variable "`hdrlen_`" of class `Packet` in Line 12 of Program 8.1, and is declared as a static variable. The variable "`hdrlen_`" has no relationship to simulation packet size. For example, TCP and UPD packets may have different sizes. The values stored in the corresponding variable `hdr_cmn::size_` may be different; however, the values stored in the variable `Packet::hdrlen_` for both TCP and UDP packets are the same.

In the following, we first discuss the first level packet header composition in Sect. 8.3.1. Sections 8.3.2 and 8.3.3 show examples of protocol-specific headers: common packet header and IP packet header. Section 8.3.4 discusses one of the main packet attributes: payload type. Section 8.3.5 explains the details of protocol-specific header (i.e., the second level packet header composition). Section 8.3.6 demonstrates how packet attributes stored in packet header are

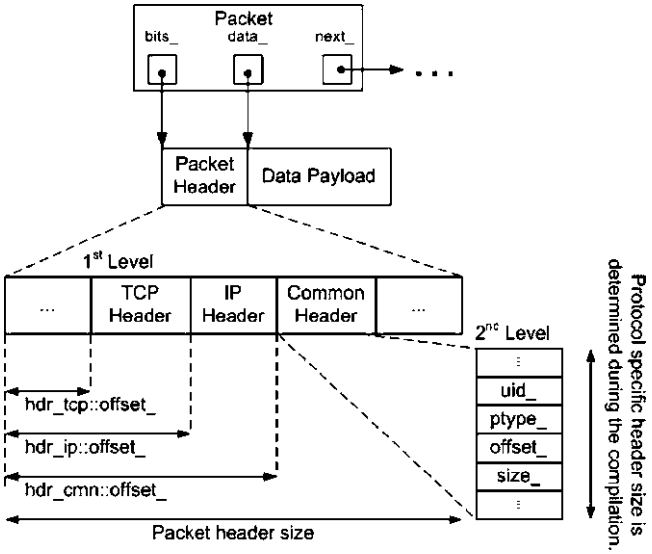

Fig. 8.5 Architecture of packet header: During the construction of the Simulator, the packet header size is determined and stored in the instvar `PacketHeaderManager::hdrlen_` which is bound to the variable `PacketHeaderManager::hdrlen_`. See the details in Step 2 in Sect. 8.3.8

accessed. Section 8.3.7 discusses one of the main packet header component, a packet header manager, which maintains the active protocol list and sets up the offset value for each protocol. Finally, Sect. 8.3.8 presents the packet header construction process.

8.3.1 An Overview of First Level Packet Composition: Offseting Protocol-Specific Header on the Packet Header

On the first level, NS2 puts together all relevant protocol-specific headers (e.g., common header, IP header, TCP header) and composes a packet header (see Fig. 8.5). Conceptually, NS2 allocates a contiguous part on the packet header for a protocol-specific header. Each protocol-specific header is offset from the beginning of packet header. The distance between the beginning of packet header and that of a protocol-specific header is stored in the member variable `offset_` of the protocol-specific header. For example, `hdr_cmn`, `hdr_ip`, and `hdr_tcp` – which represent common header, IP header, and TCP header – store their offset values in variables `hdr_cmn::offset_`, `hdr_ip::offset_`, and `hdr_tcp::offset_`, respectively.

Program 8.6 Declaration of C++ `hdr_cmn struct` data type

```
    //~/ns/common/packet.h
1   struct hdr_cmn {
2       enum dir_t { DOWN= -1, NONE= 0, UP= 1 };
3       packet_t ptype_;     // payload type
4       int size_;       // simulated packet size
5       int uid_;        // unique id
6       dir_t   direction_; // direction: 0=none, 1=up, -1=down
7       static int offset_; // offset for this header

8       inline static hdr_cmn* access(const Packet* p) {
9           return (hdr_cmn*) p->access(offset_);
10      }
11      inline static int& offset() { return offset_; }
12      inline packet_t& ptype() { return (ptype_); }
13      inline int& size() { return (size_); }
14      inline int& uid() { return (uid_); }
15      inline dir_t& direction() { return (direction_); }
16  };
```

8.3.2 Common Packet Header

Common packet header contains packet attributes which are common to all packets. It uses C++ `struct` data type `hdr_cmn` to indicate how the packet attributes are stored. Program 8.6 shows a part of `hdr_cmn` declaration. The main member variables of `hdr_cmn` are as follows:

`ptype_` The payload type (see Sect. 8.3.4).

`size_` The packet size in bytes. Unlike actual packet transmission, the number of bits requires to hold a packet has no relationship to simulation packet size. During simulation, NS2 uses the variable `hdr_cmn::size_` as the packet size.

`uid_` The ID which is unique to every packet.

`dir_t` The direction to which the packet is moving. It is used mainly in wireless networks: A packet can move "`UP`" (i.e., `dir_t = 1`) toward higher layers or move "`DOWN`" toward lower layer components (i.e., `dir_t = -1`). It can also set to `0`, when not in use, by default, "`dir_t`" is set to "`DOWN`" (see Line 20 in Program 8.3).

`offset_` The memory location relative to the beginning of packet header from which the common header is stored (see Sect. 8.3.1 and Fig. 8.5).

From Fig. 8.6, most functions of class `hdr_cmn` act as an interface to access its variables. Perhaps, the most important function of class `hdr_cmn` is the function `access(p)` in Lines 8–10. This function returns a pointer to the common protocol-specific header of the input `Packet` object `*p`. We will discuss the packet header access mechanism in greater detail in Sect. 8.3.6.

Program 8.7 Declaration of C++ `hdr_ip` `struct` data type

```
    //~/ns/common/ip.h
1   struct hdr_ip {
2       ns_addr_t   src_;
3       ns_addr_t   dst_;
4       int     ttl_;
5       int     fid_;
6       int     prio_;
7       static int offset_;
8       inline static int& offset() { return offset_; }
9       inline static hdr_ip* access(const Packet* p) {
10          return (hdr_ip*) p->access(offset_);
11      }
12      ns_addr_t& src() { return (src_); }
13      nsaddr_t& saddr() { return (src_.addr_); }
14      int32_t& sport() { return src_.port_;}
15      ns_addr_t& dst() { return (dst_); }
16      nsaddr_t& daddr() { return (dst_.addr_); }
17      int32_t& dport() { return dst_.port_;}
18      int& ttl() { return (ttl_); }
19      int& flowid() { return (fid_); }
20      int& prio() { return (prio_); }
21  };
```

8.3.3 IP Packet Header

Represented by a C++ struct data type `hdr_ip`, IP packet header contains information about source and destination of a packet. Program 8.7 shows a part of `hdr_ip` declaration. IP packet header contains the following five main variables which contain IP-related packet information (see Lines 2–6 in Program 8.7):

`src_`	Source node's address indicated in the packet
`dst_`	Destination node's address indicated in the packet
`ttl_`	Time to live for the packet
`fid_`	Flow ID of the packet
`prio_`	Priority level of the packet

NS2 uses data type `ns_addr_t` defined in the file *~ns*/config.h to store node address. From Program 8.8, `ns_addr_t` is a struct data type, which contains two members: `addr_` and `port_`. Both members are of type `int32_t`, which is simply an alias for `int` data type (see Line 5 and file *~ns*/autoconf-win32.h). While `addr_` specifies the node address, `port_` identifies the attached port (if any).

The variables `src_` and `dst_` of IP header are of class `ns_addr_t`. Hence, "`src_.addr_`" and "`src_.port_`" store the node address and the port of the sending agent, respectively. Similarly, the packet will be sent to a receiving agent attached to port "`dst_.port_`" of a node with address "`dst_.addr_`."

Program 8.8 Declaration of C++ `ns_addr_t` struct data type, and `int32_t`

```
    //~/ns/config.h
1   struct ns_addr_t {
2       int32_t addr_;
3       int32_t port_;
4   };

    //~/ns/autoconf-win32.h
5   typedef int int32_t;
```

Lines 7–11 in Program 8.7 declare the variable `offset_`, function `offset(off)` and function `access(p)`, which are essential to access IP header of a packet. Lines 12–20 in Program 8.7 are functions that return the values of the variables.

8.3.4 Payload Type

Although stored in common header, payload type is attributed to the entire packet, not to a protocol-specific header. Each packet corresponds to only one payload type but may contain several protocol-specific headers. For example, a packet can be encapsulated by both TCP and IP protocols. However, its type can be either audio *or* TCP packet, *but not both*.

NS2 stores a payload type in a member variable `ptype_` of a common packet header. The type of the variable `ptype_` is `enum packet_t` defined in Program 8.9. Again, members of `enum` are integers which are mapped to strings. From Program 8.9, `PT_TCP` (Line 2) and `PT_UDP` (Line 3) are mapped to 0 and 1, respectively. Since `packet_t` declares `PT_NTYPE` (representing undefined payload type) as the last member, the value of `PT_NTYPE` is $N_p - 1$, where N_p is the number of `packet_t` members. NS2 provides 60 built-in payload types, meaning the default value of `PT_NTYPE` is 59.

From Lines 11–30 in Program 8.9, class `p_info` maps each member of `packet_t` to a description string. It has a static associative array variable, `name_` (Line 28). The index and value of `name_` are the payload type and the corresponding description string, respectively. Class `p_info` also has one important function `name(p)` (Lines 23–26), which translates a `packet_t` variable to a description string.

At the declaration, NS2 declares a global variable `packet_info` (using `extern`), which is of class `p_info` (Line 30). Accessible at the global scope, the variable `packet_info` provides an access to the function `name(p)` of class `p_info`. To obtain a description string of a `packet_t` object "p," one may invoke

```
packet_info.name(ptype)
```

Program 8.9 Declaration of `enum packet_t` type and class `p_info`

```
    //~/ns/common/packet.h
1   enum packet_t {
2       PT_TCP,
3       PT_UDP,
4       PT_CBR,
5       PT_AUDIO,
6       PT_VIDEO,
7       PT_ACK,
8       ...
9       PT_NTYPE // This MUST be the LAST one
10  }

11  class p_info {
12  public:
13      p_info() {
14          name_[PT_TCP]= "tcp";
15          name_[PT_UDP]= "udp";
16          name_[PT_CBR]= "cbr";
17          name_[PT_AUDIO]= "audio";
18          name_[PT_VIDEO]= "video";
19          name_[PT_ACK]= "ack";
20          ...
21          name_[PT_NTYPE]= "undefined";
22      }
23      const char* name(packet_t p) const {
24          if ( p <= PT_NTYPE ) return name_[p];
25          return 0;
26      }
27  private:
28      static char* name_[PT_NTYPE+1];
29  };
30  extern p_info packet_info; /* map PT_* to string name */
```

Example 8.1. Class `Agent` is responsible for creating and destroying network layer packets (see Chap. 9). It is the base class of TCP and UDP transport layer protocol modules. Class `Agent` provides a function `allocpkt()`, which is responsible for allocating (i.e., creating) a packet.

To print out the type of every allocated packet on the screen, we modify function `allocpkt()` of class `Agent` in file *~ns*/common/agent.cc as follows:

```
    //~/ns/common/agent.h
1   Packet* Agent::allocpkt() const
2   {
3       Packet* p = Packet::alloc();
4       initpkt(p);
5       /*----- Begin Additional Codes -----*/
6       hdr_cmn* ch = hdr_cmn::access(p);
7       packet_t pt = ch->ptype();
```

```
8       printf("Example Test: Class Agent allocates a
                        packet with type %s\n", packet_
                          info.name(pt));
9       getchar();
10      /*----- End Additional Codes -----*/
11      return (p);
12  }
```

where Lines 5–10 are added to the original codes. Line 6 retrieves the reference "ch" to the common packet header (see Sect. 8.3.6). Line 7 obtains the payload type stored in the common header using the function `ptype()`, and assigns the payload type to variable "pt." Note that, the variable `packet_info` is a global variable of class `p_info`. When the variable "pt" is fed as an input argument, the statement `packet_info.name(pt)` returns the description string corresponding to the `packet_t` object "pt" (Line 8).

After recompiling the code, the simulation should show the type of every allocated packet on the screen. For example, when running the Tcl simulation script in Programs 2.1–2.2, the following result should appear on the screen:

```
>> ns myfirst_ns.tcl
Example Test: Class Agent allocates a packet with type cbr
Example Test: Class Agent allocates a packet with type cbr
Example Test: Class Agent allocates a packet with type cbr
...
```

8.3.5 Protocol-Specific Headers

A protocol-specific header stores packet attributes relevant to the underlying protocol only. For example, common packet header holds basic packet attributes such as packet unique ID, packet size, payload type, and so on. IP packet header contains IP packet attributes such as source and destination IP addresses and port numbers. There are 48 classifications of packet headers. The complete list of protocol-specific headers with their descriptions is given in [17].

Each protocol-specific header involves three classes: A C++ class, and OTcl class, and a mapping class.

8.3.5.1 Protocol-Specific Header C++ Classes

In C++, NS2 uses a `struct` data type to represent a protocol-specific header. It stores packet attributes and its offset value in members of the `struct` data type. It also provides a function `access(p)` which returns the reference to the protocol-specific header of a packet `*p`. Representing a protocol specific header,

each struct data type is named using the format hdr_<XXX>, where <XXX> is an arbitrary string representing the type of a protocol-specific header. For example, the C++ class name for common packet header is hdr_cmn.

In the C++ domain, protocol specific headers are declared but not instantiated. Therefore, NS2 uses a struct data type (rather than a class) to represent protocol-specific headers. No constructor is required for a protocol-specific header. Hereafter, we will refer to struct and class interchangeably.

8.3.5.2 A Protocol-Specific Header OTcl Class

NS2 defines a shadow OTcl class for each C++ protocol specific header class. An OTcl class acts as an interface to the OTcl domain. It is named with the format PacketHeader/<XXX>, where <XXX> is an arbitrary string representing a protocol-specific header. For example, the OTcl class name for common packet header is PacketHeader/Common.

8.3.5.3 A Protocol-Specific Header Mapping Class

A mapping class is responsible for binding OTcl and C++ class names together. All the packet header mapping classes derive from class PacketHeaderClass which is a child class of class TclClass. A mapping class is named with format <XXX>HeaderClass, where <XXX> is an arbitrary string representing a protocol-specific header. For example, the mapping class name for common packet header is CommonHeaderClass.

Program 8.10 shows the declaration of class PacketHeaderClass, which has two key variables: hrdlen_ in Line 8 and offset_ in Line 9. The variable "hdrlen_" represents the length of the protocol-specific header.[9] It is the system memory needed to store a protocol-specific header C++ class. The variable offset_ indicates the location on packet header where the protocol-specific header is used.

The constructor of class PacketHeaderClass in Lines 3 and 4 takes two input arguments. The first input argument classname is the name of the corresponding OTcl class name (e.g., PacketHeader/Common). The second one, hdrlen, is the length of the protocol-specific header C++ class. In Lines 3 and 4, the constructor feeds classname to the constructor of class TclClass, stores hdrlen in the member variable hdrlen_, and resets offset_ to zero.

Function method(argc,argv) in Line 5 is an approach to take a C++ action from the OTcl domain. Functions bind_offset(off) in Line 6 and offset(off) in Line 7 are used to configure and retrieve, respectively, value

[9]While the variable hdrlen_ in class PacketHeaderClass represents the length of a protocol specific header, the variable hdrlen_ in class Packet represents total length of packet header.

Program 8.10 Declaration of class `PacketHeaderClass`

```
    //~/ns/common/packet.h
 1  class PacketHeaderClass : public TclClass {
 2  protected:
 3      PacketHeaderClass(const char* classname, int hdrlen) :
 4              TclClass(classname), hdrlen_(hdrlen),
                offset_(0);{};
 5      virtual int method(int argc, const char*const* argv);
 6      inline void bind_offset(int* off) { offset_ = off; };
 7      inline void offset(int* off) {offset_= off;};
 8      int hdrlen_;        // # of bytes for this header
 9      int* offset_;       // offset for this header
10  public:
11      TclObject* create(int argc,const char*const* argv)
          {return 0;};
12      virtual void bind(){
13          TclClass::bind();
14          Tcl& tcl = Tcl::instance();
15          tcl.evalf("%s set hdrlen_ %d", classname_, hdrlen_);
16          add_method("offset");
17      };
18  };
```

of the variable "`offset_`". Function `create(argc,argv)` in Line 11 does nothing, since no protocol-specific header C++ object is created. It will be overridden by the derived classes of class `PacketHeaderClass`. Function `bind()` in Lines 12–17 glues the C++ class to the OTcl class. Line 13 first invokes the function `bind()` of class `TclClass`, which performs the basic binding actions. Line 15 exports variable "`hdrlen_`" to the OTcl domain. Line 16 registers the *OTcl method* `offset` using function `add_method`("`offset`").

Apart from the OTcl commands discussed in Sect. 3.4, an *OTcl method* is another way to invoke C++ functions from the OTcl domain. It is implemented in C++ via the following two steps. The first step is to define a function `method(ac,av)`. As can be seen from Program 8.11, the structure of function `method` is very similar to that of the function `command`. A *method* "`offset`" stores the input argument in the variable `*offset_` (Line 7 in Program 8.11). The second step in method implementation is to register the name of the *method* using a function "`add_method(str)`," which takes the method name as an input argument. For class `PacketHeaderClass`, the *method* `offset` is registered from within function `bind(...)` (Line 16 of Program 8.10).

A protocol-specific header is implemented using a `struct` data type, and hence does not derive function `command(...)` from class `TclObject`.[10] It resorts to OTcl *method*s defined in the mapping class to take C++ actions from the OTcl

[10]Since NS2 does not instantiate a protocol specific header object, it models a protocol specific header using `struct` data type.

Program 8.11 Function `method` of class `PacketHeaderClass`

```
    //~/ns/common/packet.cc
1   int PacketHeaderClass::method(int ac, const char*const* av)
2   {
3       Tcl& tcl = Tcl::instance();
4       ...
5       if (strcmp(argv[1], "offset") == 0) {
6           if (offset_) {
7               *offset_ = atoi(argv[2]);
8               return TCL_OK;
9           }
10          tcl.resultf("Warning: cannot set
                            offset_ for %s",classname_);
11          return TCL_OK;
12      }
13      ...
14      return TclClass::method(ac, av);
15  }
```

Table 8.1 Classes and objects related to common packet header

Class/object	Name
C++ class	`hdr_cmn`
OTcl class	`PacketHeader/Common`
Mapping class	`CommonHeaderClass`
Mapping variable	`class_cmnhdr`

domain. We will show an example use of the method `offset` later in Sect. 8.3.8, when we discuss packet construction mechanism.

Consider, for example, a common packet header. Its C++, OTcl, and mapping classes are `hdr_cmn`, `PacketHeader/Common`, and `CommonPacketHeaderClass`, respectively (see Table 8.1). Program 8.12 shows the declaration of class `CommonPacketHeaderClass`. As a child class of `TclClass`, a class mapping variable `class_cmnhdr` is instantiated at the declaration. Line 3 of the constructor invokes the constructor of its parent class `PacketHeaderClass`, which takes the OTcl class name (i.e., `PacketHeader/Common`) and the amount of memory needed to hold the C++ class (i.e., `hdr_cmn`) as input arguments. Here, "`sizeof (hdr_cmn)`" computes the required amount of memory, for `hdr_cmn`. The result of this statement is fed as the second input argument. In Line 6 of Program 8.10, the statement `bind_offset(&hdr_cmn::offset_)` sets the variable `offset_` to share the address with the input argument. Therefore, a change in `hdr_cmn::offset_` will result in an automatic change in the variable `*offset_` of class `CommonHeaderClass`, and vice versa.

Program 8.12 Declaration of class `CommonHeaderClass`

```
    //~/ns/common/packet.cc
1   class CommonHeaderClass : public PacketHeaderClass {
2   public:
3       CommonHeaderClass() : PacketHeaderClass("PacketHeader/
                              Common", sizeof(hdr_cmn)) {
4           bind_offset(&hdr_cmn::offset_);
5       }
6   } class_cmnhdr;
```

8.3.6 Packet Header Access Mechanism

This section demonstrates how packet attributes stored in packet header can be retrieved and modified. NS2 uses a two-level packet header structure to store packet attributes. On the first level, protocol-specific headers are stored within a packet header. On the second level, each protocol-specific header uses a C++ `struct` data type to store packet attributes.

Before proceeding further, let us have a look at how packet header can be modified.

Example 8.2. Given a pointer to a `Packet` object `*p`, the following statements set the packet size to be 1000 bytes.

```
hdr_cmn* ch = hdr_cmn::access(p);
ch->size_ = 1000;
```

The upper line retrieves the reference to the common header and stores the reference in the pointer "`ch`," while the lower line modifies the packet size through the field `size_` of the common packet header (through `*ch`). □

The header access mechanism consists of two major steps: (1) Retrieve a reference to a protocol-specific header, and (2) Follow the structure of the protocol-specific header to retrieve or modify packet attributes. In this section, we will explain the access mechanism through common packet header (see the corresponding class names in Table 8.1).

8.3.6.1 Retrieving a Reference to Protocol-Specific Header

NS2 obtains a reference to a protocol-specific header of a packet `*p` using a function `access(p)` of the C++ class `hdr_cmn`.

Example 8.3. Consider function `allocpkt()` of class `Agent` shown in Program 8.13, which shows the details of functions `allocpkt()` and

Program 8.13 Functions `allocpkt` and `initpkt` of class `Agent`

```
    //~/ns/common/agent.cc
1   Packet* Agent::allocpkt() const
2   {
3       Packet* p = Packet::alloc();
4       initpkt(p);
5       return (p);
6   }

7   Packet* Agent::initpkt(Packet* p) const
8   {
9       hdr_cmn* ch = hdr_cmn::access(p);
10      ch->uid() = uidcnt_++;
11      ch->ptype() = type_;
12      ch->size() = size_;
13      ...
14      hdr_ip* iph = hdr_ip::access(p);
15      iph->saddr() = here_.addr_;
16      iph->sport() = here_.port_;
17      iph->daddr() = dst_.addr_;
18      iph->dport() = dst_.port_;
19      ...
20  }
```

`initpkt(p)`. Function `allocpkt()` in Lines 1–6 creates a `Packet` object and returns a pointer to the created object. It first invokes function `alloc()` of class `Packet` in Line 3 (see the details in Sect. 8.2.1). Then, Line 4 initializes the allocated packet by invoking the function `initpkt(p)`. Finally, Line 5 returns the pointer "`p`" which points to the initialized `Packet` object.

Function `initpkt(p)` follows the structure defined in the protocol-specific header C++ classes to set packet attributes to the default values. Lines 9 and 14 in Program 8.13 execute the first step in the access mechanism: retrieve references to common packet header "`ch`" and IP header "`iph`," respectively.

After obtaining the pointers "`ch`" and "`iph`," Lines 10–12 and Lines 15–18 carry out the second step in the access mechanism: access packet attributes through the structure defined in the protocol-specific headers. In this step, the relevant packet attributes such as unique packet ID, payload type, packet size, source IP address and port, and destination IP address and port, are configured through the pointers "`ch`" and "`iph`." Note that `uidcnt` (i.e., uid count) is a static member variable of class `Agent` which represents the total number of generated packets. We will discuss the details of class `Agent` later in Chap. 9. □

Figure 8.6 shows an internal mechanism of the function `hdr_cmn::access(p)` where "`p`" is a `Packet` pointer. When `hdr_cmn::access(p)` is executed Line 9 in Program 8.6 executes `p->access(offset_)`, where `offset_` is the member variable of class `hdr_cmn`, specifying the location on

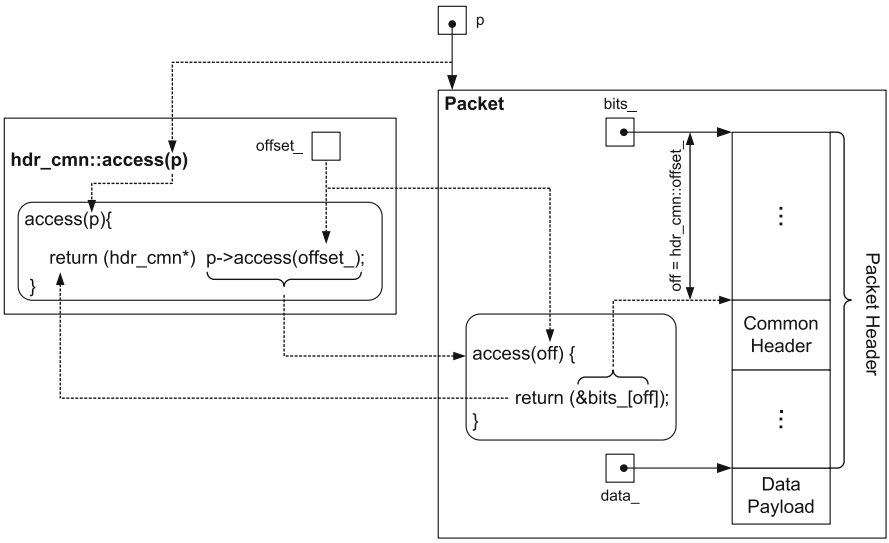

Fig. 8.6 The internal mechanism of the function `access(p)` of the `hdr_cmn struct` data type, where "`p`" is a pointer to a `Packet` object

the packet header allocated to the common header (see also Fig. 8.5). On the right-hand side of Fig. 8.6, the function `access(off)` simply returns `&bits_[off]`, where "`bits_`" is the member variable of class `Packet` storing the entire packet header. Since the input argument `offset_` belongs to `hdr_cmn`, the statement `access(offset_)` essentially returns `&bits_[hdr_cmn::offset_]`, which is the reference to the common header stored in the `Packet` object `*p`. This reference is returned as an `unsigned char*` variable. Then, class `hdr_cmn` casts the returned reference to `hdr_cmn*` data type and returns it to the caller.

Note that NS2 simplifies the retrival of protocal specific header reference, by defining pre-processing statements:

```
//~ns/common/packet.h
#define HDR_CMN(p) chdr_cmn::access(p))
#define HDR_ARP(p) chdr_arp::access(p))
...
```

8.3.6.2 Accessing Packet Attributes in a Protocol-Specific Header

After obtaining a reference to a protocol-specific header, the second step is to access the packet attributes according to the structure specified in the protocol-specific

Program 8.14 Declarations of C++ class `PacketHeaderManager` and mapping class `PacketHeaderManagerClass`

```
    //~/ns/common/packet.cc
1   class PacketHeaderManager : public TclObject {
2   public:
3       PacketHeaderManager() {bind("hdrlen_",
                                &Packet::hdrlen_);}
4   };

5   static class PacketHeaderManagerClass : public TclClass {
6   public:
7       PacketHeaderManagerClass() :
                            TclClass("PacketHeaderManager") {}
8       TclObject* create(int, const char*const*) {
9           return (new PacketHeaderManager);
10      }
11  } class_packethdr_mgr;
```

header C++ class. Since NS2 declares a protocol-specific header as a `struct` data type, it is fairly straightforward to access packet attributes once the reference to the protocol-specific header is obtained (see Example 8.3).

8.3.7 Packet Header Manager

A packet header manager is responsible for keeping the list of active protocols and setting the offset values of all the active protocols. It is implemented using a C++ class `PacketHeaderManager` which is bound to an OTcl class with the same name. Program 8.14 and Fig. 8.7 show the declaration of the C++ class `PacketHeaderManager` as well as the corresponding binding class, and the diagram of the OTcl class `PacketHeaderManager`, respectively.

The C++ class `PacketHeaderManager` has one constructor (Line 3) and has neither variables nor functions. The constructor binds the instvar "`hdrlen_`" of the OTcl class `PacketHeaderManager` to the variable "`hdrlen_`" of class `Packet` (see also Fig. 8.1). The OTcl class `PacketHeaderManager` has two main instvars: "`hdrlen_`" and "`tab_`." The instvar "`hdrlen_`" stores the length of packet header. It is initialized to zero in Line 1 of Program 8.15, and is incremented as protocol-specific headers are added to the packet header. Representing the active protocol list, the instvar "`tab_`" (Line 2 in Program 8.16) is an associative array whose indexes are protocol-specific header OTcl class names and values are 1 if the protocol-specific header is active (see Line 12 in Program 8.5). If the protocol-specific header is inactive, the corresponding value of "`tab_`" will not be available (i.e., NS2 `unsets` all entries corresponding to inactive protocol-specific headers; see Line 7 in Program 8.20).

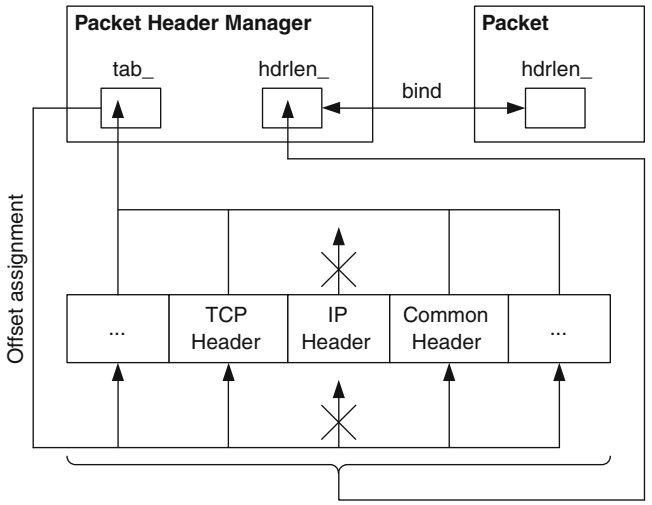

Fig. 8.7 Architecture of an OTcl `PacketHeaderManager` object

Program 8.15 Initialization of a `PacketHeaderManager` object

```
    //~/ns/tcl/lib/ns-packet.tcl
1   PacketHeaderManager set hdrlen_ 0

2   foreach prot {
3       Common
4       Flags
5       IP
6       ...
7   }  {
8       add-packet-header $prot
9   }

10  proc add-packet-header args {
11      foreach cl $args {
12          PacketHeaderManager set tab_(PacketHeader/$cl) 1
13      }
14  }
```

8.3.8 Protocol-Specific Header Composition and Packet Header Construction

Packet header is constructed through the following three-step process:

Program 8.16 Function `create_packetformat` of class `Simulator` and function `allochdr` of class `PacketHeaderManager`

```
    //~/ns/tcl/lib/ns-packet.tcl
1   Simulator instproc create_packetformat { } {
2       PacketHeaderManager instvar tab_
3       set pm [new PacketHeaderManager]
4       foreach cl [PacketHeader info subclass] {
5           if [info exists tab_($cl)] {
6               set off [$pm allochdr $cl]
7               $cl offset $off
8           }
9       }
10      $self set packetManager_ $pm
11  }

12  PacketHeaderManager instproc allochdr cl {
13      set size [$cl set hdrlen_]
14      $self instvar hdrlen_
15      set NS_ALIGN 8
16      set incr [expr ($size + ($NS_ALIGN-1)) & ~($NS_ALIGN-1)]
17      set base $hdrlen_
18      incr hdrlen_ $incr
19      return $base
20  }
```

Step 1: At the Compilation Time

During the compilation, NS2 translates all C++ codes into an executable file. It sets up all necessary variables (including the length of all protocol-specific headers) for all built-in protocol-specific headers, and includes all built-in protocol-specific headers into the active protocol list. There are three main tasks in this step.

Task 1: Construct All Mapping Variables, Configure the Variable `hdrlen_`, Register the OTcl Class Name, and Binds the Offset Value

Since all mapping variables are instantiated at the declaration, they are constructed during the compilation using their constructors. As an example, consider the common packet header[11] whose construction process shown in Program 8.10, Program 8.12, and Fig. 8.8 proceeds as follows:

1. Store the corresponding OTcl class name (e.g., `PacketHeader/Common`) in the variable `classname_` of class `TclClass`.
2. Determine the size (using function `sizeof (...)`) of the protocol-specific header, and store it in the variable "`hdrlen_`" of class `PacketHeaderClass`.
3. Bind the variable `offset_` of the `PacketHeader` to that of class `hdr_cmn`.

[11] NS2 repeats the following process for all protocol specific headers. For brevity, we show the construction process through common packet header only.

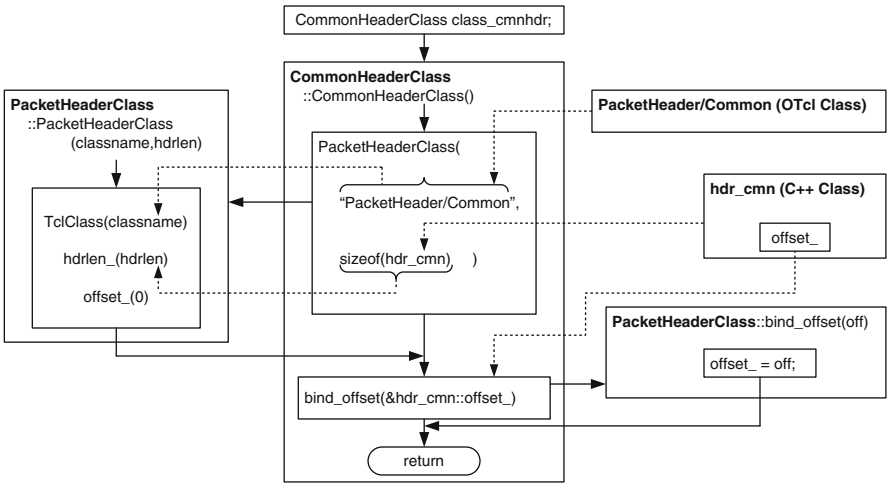

Fig. 8.8 Construction of the static mapping variable `class_cmnhdr`

Task 2: Invocation of Function `bind()` of Class `TclClass` Which Exports the Variable `hdrlen_`

The main NS2 function (i.e., `main(argc,argv)`) invokes the function `init(...)` of class `Tcl`, which in turn invokes the function `bind()` of class `TclClass` of all mapping variables. The function `bind()` registers and binds an OTcl class name to the C++ domain (see file *~tclcl/*Tcl.cc). This function is overridden by class `PacketHeaderClass`.

As shown in Lines 12–17 of Program 8.10, class `PacketHeaderClass` overrides function `bind()` of class `TclClass`. Line 13 first invokes the function `bind()` of class `TclClass`. Line 15 exports the variable "`hdrlen_`" to the OTcl domain. Finally, Line 16 registers the OTcl method `offset`.

In case of class `CommonHeaderClass`, `classname_` is `PacketHeader/Common` and "`hdrlen_`" is 104 bytes. Therefore, Line 15 of Program 8.10 executes the following OTcl statement:

```
PacketHeader/Common set hdrlen_ 104
```

which sets instvar "`hdrlen_`" of class `PacketHeader/Common` to be 104. Note that this instvar "`hdrlen_`" is not bound to the C++ domain.

After Task 1 and Task 2 are completed, the related protocol-specific classes, namely `hdr_cmn`, `PacketHeader/Common`, and `CommonHeaderClass`, would be as shown in Fig. 8.9. The mapping object `class_cmnhdr` is of class `CommonHeaderClass`, which derives from classes `PacketHeaderClass` and `TclClass`, respectively. It inherits variables `classname_`, `hdrlen_`, and `offset_` from its parent class. After object construction is complete, variable `classname_` will store the name of the OTcl common packet header

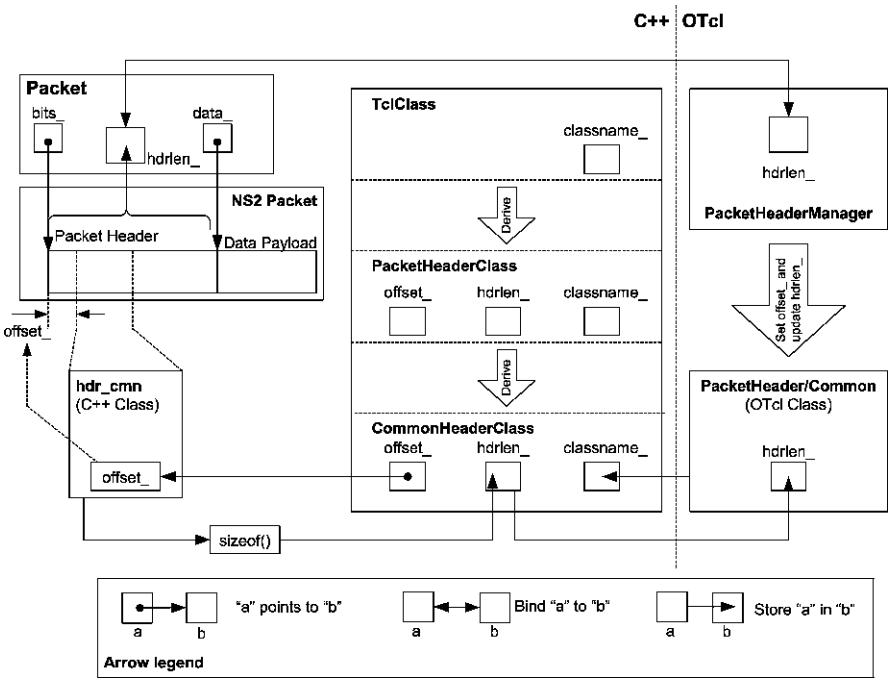

Fig. 8.9 A schematic diagram of a static mapping object `class_cmnhdr`, class `hdr_cmn`, class `PacketHeader/Common`, and class `Packet`

class (i.e., `PacketHeader/Common`), `hdr_len_` will store the amount of memory in bytes needed to store common header, and `offset_` will point to `hdr_cmn:: offset_`. However, at this moment, the offset value is set to zero. The dashed arrow in Fig. 8.9 indicates that the value of variable `hdr_cmn:: offset_` will be later set to store an offset from the beginning of a packet header to the point where the common packet header is stored. Also, after the function `Tcl::init()` invokes the function `bind()` of class `PacketHeaderClass`, the instvar "`hdrlen_`" of class `PacketHeader/Common` will store the value of the variable "`hdrlen_`" of class `CommonHeaderClass`. Note that tasks 1 and 2 only set up C++ OTcl class, and mapping class. However, the packet header manager is not configured at this phase.

*Task 3: Sourcing the File ~*ns*/tcl/lib/ns-packet.tcl to Setup an Active Protocol List*

As discussed in Sect. 3.7, NS2 sources all scripting Tcl files during the compilation process. In regards to packet header, Program 8.15 shows a part of the file *~ns*/tcl/lib/ns-packet. Here, Line 8 invokes procedure `add-packet-`

header{prot} for all built-in protocol-specific headers indicated in Lines 3–6. Line 12 sets the value of the associative array "tab_" whose index is the input protocol-specific header name to be 1.

Step 2: During the Network Configuration Phase

In regards to packet header construction, the main task in the Network Configuration Phase is to setup variables offset_ of all active protocol-specific headers and formulate a packet header format. Subsequent packet creation will follow the packet format created in this step.

The offset configuration process takes place during the simulator construction. From Line 2 of Program 4.11, the constructor of the Simulator invokes the instproc create_packetformat{} of class Simulator.

As shown in Program 8.16, the instproc create_packetformat{} creates a PacketHeaderManager object "pm" (Line 3). Here, the constructor of C++ class PacketHeaderManager is invoked. From Program 8.14, the constructor binds its OTcl instvar "hdrlen_" to the variable "hdrlen_" of the C++ class Packet.

After creating a PacketHeaderManager object "pm," the instproc create_packetformat{} computes the offset value of all active protocol-specific headers using the instproc allochdr{cl} (Line 6), and configures the offset values of all protocol specific headers (Line 7). The foreach loop in Line 4 runs for all built-in protocol-specific headers which are child classes of class PacketHeader. Line 5 filters out those which are not in the active protocol list (see Sect. 8.3.7). Lines 6 and 7 are executed for all active protocol-specific headers specified in the variable "tab_" (which was configured in Step 1 – Task 3) of the PacketHeaderManager object "pm." Line 7 configures offset values using the OTcl *method* offset (see Program 8.11) of protocol specific header mapping classes. The OTcl *method* offset stores the input argument in the variable *offset_ of the protocol-specific header mapping class (e.g., CommonHeaderClass).

Lines 12–19 in Program 8.16 and Fig. 8.10 show the OTcl source codes and the diagram, respectively, of the instproc allochdr{cl} of an OTcl class PacketHeaderManager. The instproc allochdr{cl} takes one input argument "cl" (in Line 12) which is the name of a protocol-specific header, computes the memory requirement, and returns the offset value corresponding to the input argument "cl." Line 13 stores header length of a protocol-specific header "cl" (e.g., the variable "hdrlen_" of class PacketHeader/Common) in a local variable "size."[12] Based on "size," Lines 15 and 16 compute the amount of

[12]The variable hdrlen_ of a protocol specific header OTcl class was configured in Step 1 – Task 2.

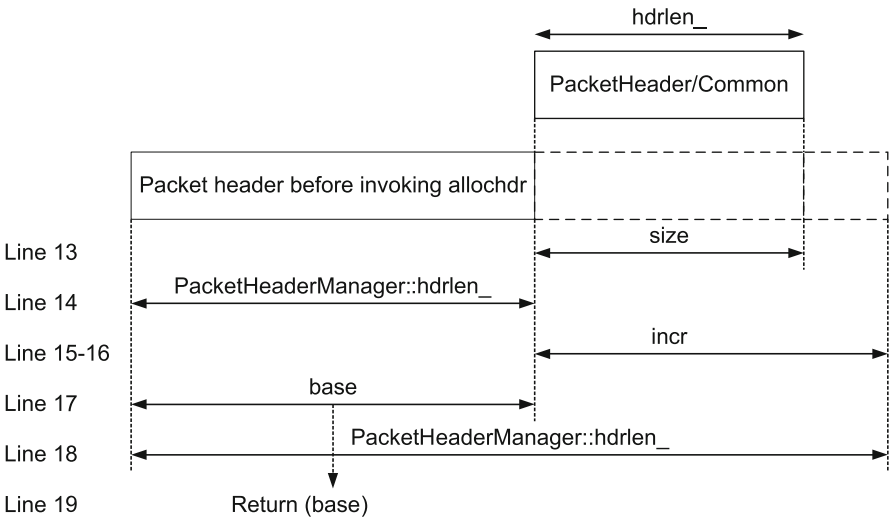

Fig. 8.10 A diagram representing the instproc `allochdr` of class `PacketHeaderManager`. Line numbers shown on the *left* correspond to the lines in Program 8.16. The action corresponding to each line is shown on the *right*

memory (`incr`) needed to store the header.[13] Line 17 stores the current packet header length (excluding the input protocol-specific header) in a local variable "`base`." Since "`base`" is an offset distance from the beginning of packet header to the input protocol-specific header, it is returned to the caller as the offset value in Line 19. After Line 18 increases the header length (i.e., the instvar "`hdrlen_`" of class `PacketHeaderManager`) by "`incr`."

During the `Simulator` construction, the packet header manager also updates its variable "`hdrlen_`" (Line 19 in Program 8.16). Note that the instvar "`hdrlen_`" of class `PacketHeaderManager` was set to zero at the compilation (Line 1 in Program 8.15). As Lines 6 and 7 in Program 8.16 repeat for every protocol-specific header, the offset value is added to the instvar "`hdrlen_`" of an OTcl class `PacketHeaderManager`. At the end, the instvar "`hdrlen_`" will represent the total header length, which embraces all protocol-specific headers.

Step 3: During the Simulation Phase

During the Simulation Phase, NS2 creates packets based on the format defined in the former two steps. For example, an `Agent` object creates and initializes a packet using its function `allocpkt()`. Here, a packet is created using the function

[13] The variable "`incr`" could be greater than "`size`," depending on the underlying hardware.

alloc() of class Packet and initialized using the function initpkt(p) of class Agent. Again, the function alloc() takes a packet from the free packet list, if it is nonempty. Otherwise, it will create a new packet using "new". After retrieving a packet, it clears the values stored in the packet header and data payload. The function initpkt(p) assigns default values to packet attributes such as packet unique ID, payload type, and packet size (see Program 8.13). The initialization is performed by retrieving a reference (e.g., "ch") to the relevant protocol-specific header and accessing packet attributes using the predefined structure.

8.4 Data Payload

Implementation of data payload in NS2 differs from actual data payload. In practice, user information is transformed into bits and are stored in data payload. Such the transformation is not necessary in simulation, since NS2 stores the user information in the packet header. NS2 rarely needs to maintain data payload. In Line 11 of Program 7.3, packet transmission time, i.e., the time required to send out a packet, is computed as $\frac{\text{packet size}}{\text{bandwidth}}$. Class LinkDelay determines the size of a packet by hdr_cmn::size_ (not by counting the number of bits stored in packet header and data payload) to compute packet transmission time. In most cases, users do not need to explicity deal with data payload.

NS2 also provides a support to hold data payload. In Line 4 of Program 8.1, class Packet provides a pointer "data_" to an AppData object.[14] Program 8.17 shows the declaration of an abstract class AppData. Class AppData has only one member variable type_ in Line 11. Among its functions, and one is a pure virtual function copy() shown in Line 18. Indicating the type of application, the variable type_ is of type enum AppDataType defined in Lines 1–8. The function copy() duplicates an AppData object to a new AppData object. It is a pure virtual function, and must be overridden by child instantiable classes of class AppData. Function size() in Line 17 returns the amount of memory required to store an AppData object.

Class AppData provides two constructors. One is in Line 13, where the caller feeds an AppData type as an input argument. Another is in Line 14, where a reference to a AppData object is fed as an input argument. In both the cases, the constructor simply sets the variable type_ to a value as specified in the input argument.

Program 8.18 shows the declaration of class PacketData, a child class of class AppData. This class has two new member variables: "data_" (a string variable which stores data payload) in Line 3 and datalen_ (the length of "data_") in Line 4. Line 25 defines a function data() which simply returns "data_." Lines 26 and 27 override the virtual functions size() and copy(), respectively, of

[14] However, no memory is allocated to the AppData object unless it is needed.

Program 8.17 Declaration of `enum AppDataType` and class `AppData`

```
    //~/ns/common/ns-process.h
 1  enum AppDataType {
 2      ...
 3      PACKET_DATA,
 4      HTTP_DATA,
 5      ...
 6      ADU_LAST
 7
 8  };

 9  class AppData {
10  private:
11      AppDataType type_;      // ADU type
12  public:
13      AppData(AppDataType type) { type_ = type; }
14      AppData(AppData& d) { type_ = d.type_; }
15      virtual ~AppData() {}
16      AppDataType type() const { return type_; }
17      virtual int size() const { return sizeof(AppData); }
18      virtual AppData* copy() = 0;
19  };
```

class `AppData`. Function `size()` simply returns `datalen_`. Function `copy()` creates a new `PacketData` object which has the same content as the current `PacketData` object, and returns the pointer to the created object to the caller.

Class `PacketData` has two constructors. One is to construct a new object with size "`sz`," using the constructor in Lines 6–12. This constructor simply sets the default application data type to be `PACKET_DATA` (Line 6), stores "`sz`" in "`datalen_`" (Line 7), and allocates memory of size "`datalen_`" to "`data_`" (Line 9). Another construction method[15] is to create a copy of an input `PacketData` object (Lines 13–20). In this case, the constructor feeds an input `PacketData` object "`d`" to the parent class (Line 13), copies the variable `datalen_` (Line 14), and duplicates its data payload (Line 17).[16]

NS2 creates a `PacketData` object through two functions of class `Packet`: `alloc(n)` and `allocdata(n)`. In Program 8.4, the function `alloc(n)` allocates a packet in Line 3 and creates data payload using the function `allocdata(n)` in Line 5. The function `allocdata(n)` creates a `PacketData` object of size "n," by executing "`new PacketData(n)`" in Line 11.

Program 8.19 shows four functions which can be used to manipulate data payload. Functions `accessdata()` (Lines 4–9) and `userdata()` (Line 10) are both data payload access functions. The difference is that the function

[15]Function `copy()` in Line 27 uses this constructor to create a copy of a `PacketData` object.

[16]Function `memcpy(dst,src,num)` copies "`num`" data bytes from the location pointed by "`src`" to the memory block pointed by "`dst`."

Program 8.18 Declaration of class `PacketData`

```
    //~/ns/common/packet.h
 1  class PacketData : public AppData {
 2  private:
 3      unsigned char* data_;
 4      int datalen_;
 5  public:
 6      PacketData(int sz) : AppData(PACKET_DATA) {
 7          datalen_ = sz;
 8          if (datalen_ > 0)
 9              data_ = new unsigned char[datalen_];
10          else
11              data_ = NULL;
12      }
13      PacketData(PacketData& d) : AppData(d) {
14          datalen_ = d.datalen_;
15          if (datalen_ > 0) {
16              data_ = new unsigned char[datalen_];
17              memcpy(data_, d.data_, datalen_);
18          } else
19              data_ = NULL;
20      }
21      virtual ~PacketData() {
22          if (data_ != NULL)
23              delete []data_;
24      }
25      unsigned char* data() { return data_; }
26      virtual int size() const { return datalen_; }
27      virtual AppData* copy() { return new PacketData(*this);}
28  };
```

`accessdata()` returns a direct pointer to *a string* "`data_`" which contains data payload while the function `userdata()` returns a pointer to *an* `AppData` *object* which contains data payload. Function `setdata(d)` (Lines 11–15) sets the pointer "`data_`" to point to the input argument "`d`." Finally, function `datalen()` in Line 16 returns the size of data payload.

8.5 Customizing Packets

8.5.1 Creating Your Own Packet

When designing a new protocol, programmers may need to change the packet format. This section gives a guideline of how packet header, data payload, or both can be modified. Note that, it is recommended *not to* use data payload in simulation. If possible, include information related to the new protocol in a protocol-specific header.

Program 8.19 Functions `accessdata`, `userdata`, `setdata` and `datalen` of class `Packet`

```
    //~/ns/common/packet.h
1   class Packet : public Event {
2       ...
3   public:
4       inline unsigned char* accessdata() const {
5           if (data_ == 0)
6               return 0;
7           assert(data_->type() == PACKET_DATA);
8           return (((PacketData*)data_)->data());
9       }
10      inline AppData* userdata() const {return data_;}
11      inline void setdata(AppData* d) {
12          if (data_ != NULL)
13              delete data_;
14          data_ = d;
15      }
16      inline int datalen() const { return data_ ? data_
                                ->size() : 0; }
17      ...
18  };
```

8.5.1.1 Defining a New Packet Header

Suppose we would like to include a new protocol-specific header, namely "My Header," into the packet header. We need to define a C++ class (e.g., `hdr_myhdr`), an OTcl class (e.g., `PacketHeader/MyHeader`), and a mapping class (e.g., `MyHeaderClass`), and include the OTcl class into the active protocol list. In particular, we need to perform the following four steps:

- **Step 1:** Define a C++ protocol-specific header structure (e.g., see Program 8.6).
  - Pick a name for the C++ `struct` data type, say `struct hdr_myhdr`.
  - Declare a variable `offset_` to identify where the protocol-specific header reside in the entire header.
  - Define a function `access(p)` which returns the reference to the protocol-specific header (see Lines 8–10 in Program 8.6).
  - Include all member variables required to hold new packet attributes.
  - [Optional] Include a new payload type into `enum packet_t` and class `p_info` (e.g., see Program 8.9). Again, a new payload type does not need to be added for every new protocol-specific header.
- **Step 2:** Pick an OTcl name for the protocol specific header, e.g., `PacketHeader/MyHeader`.

- **Step 3:** Bind the OTcl name with the C++ protocol-specific header structure. Derive a mapping class `MyHeaderClass` from class `PacketHeaderClass` (e.g., see class `CommonHeaderClass` in Program 8.12).
  - At the construction, feed the OTcl name (i.e., `PacketHeader/MyHeader`) and the size needed to hold the protocol-specific header (i.e., `sizeof(hdr_myhdr)`) to the constructor of class `PacketHeaderClass` (e.g., see Line 3 in Program 8.12).
  - From within the constructor of the mapping class, invoke function `bind_offset(...)` feeding the address of the variable `offset_` of the C++ `struct` data type as an input argument (i.e., invoke `bind_offset(&hdr_myhdr::offset_)`).
  - Instantiate a mapping variable `class_myhdr` at the declaration.
- **Step 4:** Activate the protocol-specific header from the OTcl domain. Add the OTcl name to the list defined within the packet header manage. In particular, modify Lines 2–9 of Program 8.15 as follows:

```
foreach prot {
    Common
    Flags
    ...
    MyHeader
} {
    add-packet-header $prot
}
```

  where only the suffix of the new protocol-specific header (i.e., `MyHeader`) is added to the `foreach` loop.

8.5.1.2 Defining a New Data Payload

Data payload can be created in four levels:

1. None: NS2 rarely uses data payload in simulation. To avoid any complication it is suggested not to use data payload in simulation.
2. Use class `PacketData`: The simplest form of storing data payload is to use class `PacketData` (see Program 8.18). Class `Packet` has a variable "`data_`" whose class is `PacketData`, and provides functions (in Program 8.19) to manipulate the variable "`data_`."
3. Derive a class (e.g., class `MyPacketData`) from class `PacketData`: This option is suitable when new functionalities (i.e., functions and variables) in addition to those provided by class `PacketData` are needed. After deriving a new `PacketData` class, programmers may also derive a new class (e.g., class `MyPacket`) from class `Packet`, and override the variable "`data_`" of class `Packet` to be a pointer to a `MyPacketData` object.

4. Define a new data payload class: A user can also define a new payload type if needed. This option should be used when the new payload has nothing in common with class `PacketData`. The following are the main tasks needed to define and use a new payload type `MY_DATA`.
   - Include the new payload type (e.g., `MY_DATA`) into `enum AppDataType` data type (see Program 8.17).
   - Derive a new payload class `MyData` from class `AppData`.
     - Feed the payload type `MY_DATA` to the constructor of class `AppData`.
     - Include any other necessary functions and variables.
     - Override functions `size()` and `copy()`.
   - Derive a new class `MyPacket` from class `Packet`
     - Declare a variable of class `MyData` to store data payload.
     - Include functions to manipulate the above `MyData` variable.

8.5.2 Activate/Deactivate a Protocol-Specific Header

By default, NS2 includes *all* built-in protocol-specific headers into packet header (see Program 8.15). This inclusion can lead to unnecessary wastage of memory especially in a *packet-intensive* simulation, where numerous packets are created. For example, common, IP, and TCP headers together use only 0.1 kB, while the default packet header consumes as much as 1.9 kB [17]. Again, NS2 does not return the memory allocated to a `Packet` object until the simulation terminates. Selectively including protocol-specific header can lead to huge memory saving.

The packet format is defined when the Simulator is created. Therefore, a protocol-specific headers must be activated/deactivated before the creation of the Simulator. NS2 provides the following OTcl procedures to activate/ deactivate protocol-specific headers:

- To add a protocol-specific header `PacketHeader/MH1`, execute

  ```
  add-packet-header MH1
  ```

  In Lines 10–14 of Program 8.15, the above statement includes `PacketHeader/MH1` to the variable "`tab_`" of class `PacketHeaderManager`.
- To remove a protocol-specific header `PacketHeader/MH1` from the active list, execute

  ```
  remove-packet-header MH1
  ```

  The details of procedure `remove-packet-header{args}` are shown in Lines 1–9 of Program 8.20. Line 7 removes the entries with the index `PacketHeader/MH1` from the variable "`tab_`" of class `PacketHeaderManager`.

Program 8.20 Procedures `remove-packet-header`, and `remove-all-packet-header`

```
    //~ns/tcl/ns-packet.tcl
1   proc remove-packet-header args {
2       foreach cl $args {
3           if { $cl == "Common" } {
4               warn "Cannot exclude common packet header."
5               continue
6           }
7           PacketHeaderManager unset tab_(PacketHeader/$cl)
8       }
9   }

10  proc remove-all-packet-headers {} {
11      PacketHeaderManager instvar tab_
12      foreach cl [PacketHeader info subclass] {
13          if { $cl != "PacketHeader/Common" } {
14              if [info exists tab_($cl)] {
15                  PacketHeaderManager unset tab_($cl)
16              }
17          }
18      }
19  }
```

- To remove all protocol-specific headers, execute

  ```
  remove-all-packet-header
  ```

  In Lines 10–19 of Program 8.20, the procedure `remove-all-packet-header{}` uses `foreach` to remove all protocol-specific headers (except for common header) from the active protocol list.

8.6 Chapter Summary

Consisting of packet header and data payload, a packet is represented by a C++ class `Packet`. Class `Packet` consists of pointers "`bits_`" to its packet header and "`data_`" to its data payload. It uses a pointer "`next_`" to form a link list of packets. It also has a pointer "`free_`" which points to the first `Packet` object on the free packet list. When a `Packet` object is no longer in use, NS2 stores the `Packet` object in the free packet list for future reuse. Again, `Packet` objects are not destroyed until the simulation terminates. When allocating a packet, NS2 first tries to take a `Packet` object from the free packet list. Only when the free packet list is empty, will NS2 create a new `Packet` object.

During simulation, NS2 usually stores relevant user information (e.g., packet size) in packet header, and rarely uses data payload. It is recommended not to use data payload if possible, since storing all information in packet header greatly simplifies the simulation yet yields the same simulation results.

Packet header consists of several protocol-specific headers. Each protocol-specific header occupies a contiguous part in packet header and identifies the occupied location using its variable `offset_`. NS2 uses a packet header manager (represented by an OTcl class `PacketHeaderManager`) to maintain a list of active protocols, and define packet header format using the list when the Simulator is created. The packet header construction process proceeds in the three following steps:

1. *At the Compilation*: NS2 defines the following three classes for each of protocol-specific headers:
   - *A C++ class*: NS2 uses C++ `struct` data type to define how packet attributes are stored in a protocol specific header. One of the important member variables is `offset_`, which indicates the location of the protocol-specific header on the packet header.
   - *An OTcl class*: During the Network Configuration Phase, the packet header manager configures packet header from the OTcl domain.
   - *A mapping class*: A mapping class binds the OTcl and C++ class together. It declares a *method* `offset`, which is invoked by a packet header manager from the OTcl domain to configure the value of the variable `offset_` of the C++ class `PacketHeaderClass`.
2. *At the Network Configuration Phase*: A user may add/remove protocol specific headers to/from the active protocol list. When the Simulator is created, the packet header manager computes and assigns appropriate offset values to all protocol-specific headers specified in the active list.
3. *At the Simulation Phase*: NS2 follows the above packet header definitions when allocating a packet.

8.7 Exercises

1. What are actual packets, class `Packet`, data payload, packet header, protocol-specific header? Explain their similarities/differences/relationships. Draw a diagram to support your answer.
2. Consider standard IP packet header as specified in [21]. How does NS2 store this packet header information?
3. What is the free packet list? How does it relate to the variable "`free_`" of class `Packet`? Draw a diagram to support your answer.
4. What is the purpose of the variable "`fflag_`" of class `Packet`? When is it used? How is it used?
5. Explain the packet destruction process. Draw a diagram to support your answer.

6. What is the purpose of the variable "`offset_`" defined for each protocol-specific header? Explain your answer via few examples of protocol specific header.
7. Where and how does NS2 define active protocol list? Write few statements to activate/deactivate protocol-specific headers.
8. Design a new packet header which can record a collection of time values. Pass the packet with new header through a network. When the packet enters a `LinkDelay` object, add the current simulation time into the time value collection and print out all the values in the collection.
9. Design a new data payload type. For simplicity, set every bit in the payload to "1". Run a program to test your answer.
10. Write C++ statements to perform the following tasks:
    a. Show the following information for a packet `*p` on the screen: Size, source and destination IP addresses and ports, payload type, and flow ID.
    b. Create a new packet.
    c. Destroy the packet `*p`.

Chapter 9
Transport Control Protocols Part 1: An Overview and User Datagram Protocol Implementation

A typical communication system consists of applications, transport layer agents, and a low-level network. An application models user demand to transmit data. Taking user demand as an input, a sending transport layer agent creates packets and forwards them to the associated receiving transport layer agent through a low-level network services. Having discussed the details of low-level network functionalities in Chaps. 5–7, the details of transport layer agents are presented here in Chaps. 9 and 10. Also, the details of applications will be presented in Chap. 11.

This chapter provides an overview of transport layer agents and shows NS2 implementation of User Datagram Protocol (UDP) agents. In particular, Sect. 9.1 introduces two most widely used transport control protocols: Transmission Control Protocol (TCP) and UDP. Section 9.2 explains NS2 implementation of basic agents. Section 9.3 shows the implementation of UDP agents and Null agents. Finally, the chapter summary is given in Sect. 9.4.

9.1 UDP and TCP Basics

9.1.1 UDP Basics

Defined in [22], UDP is a connectionless transport-layer protocol, where no connection setup is needed before data transfer. UDP offers minimal transport layer functionalities – non-guaranteed data delivery – and gives applications a direct access to the network layer. Aside from the multiplexing/demultiplexing functions and some light error checking, it adds nothing to IP packets. In fact, if the application developer uses UDP as a transport layer protocol, then the application is communicating almost directly with the network layer.

UDP takes messages from an application process, attaches source and destination ports for the multiplexing/demultiplexing service, adds two other fields of error

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_9, © Springer Science+Business Media, LLC 2012

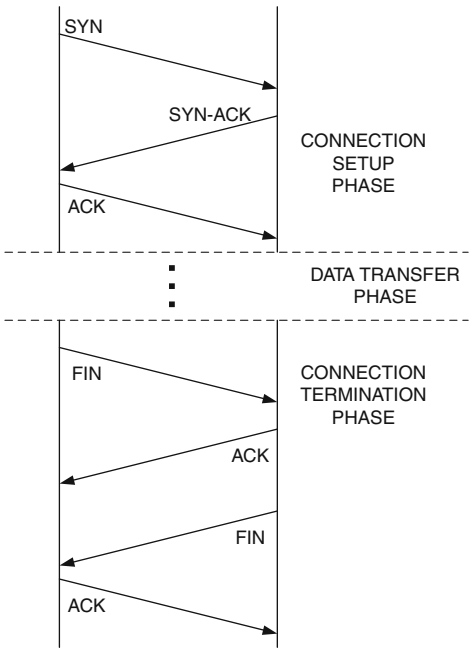

Fig. 9.1 Main phases of TCP operation: Connection setup, data transfer, and connection termination

checking and length information, and passes the resulting packet to the network layer [22]. The network layer encapsulates the UDP packet into a network layer packet and then delivers the encapsulated packet at the receiving host. When a UDP packet arrives at the receiving host, it is delivered to the receiving UDP agent identified by the destination port field in the packet header.

9.1.2 TCP Basics

As shown in Fig. 9.1, TCP [20–22] is a connection-oriented reliable transport protocol consisting of three phases of operations: connection setup, data transfer, and connection termination. In the *connection setup* phase, a TCP sender initiates a three-way handshake (i.e., sending `SYN`, `SYN-ACK`, and `ACK` messages). After a connection is established, TCP enters the *data transfer* phase where a TCP sender transfers data to a TCP receiver. Finally, after the data transfer is complete, TCP tears down the connection in the *connection termination* phase using a four-way handshake (i.e., sending two pairs of `FIN-ACK` messages.)

The main operation of TCP lies in the data transfer phase, which implements the following two mechanisms: (1) error control using basic acknowledgment and timeout, and (2) congestion control using a window-based mechanism.

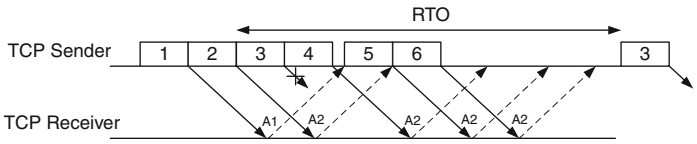

Fig. 9.2 An example of TCP error control using acknowledgment: A TCP sender realizes the loss of TCP packet number 3 after transmitting the packet number 3 for a period of RTO (i.e., timeout)

9.1.2.1 Error Control Using Basic Acknowledgment and Timeout

As a reliable transport layer protocol, TCP provides connection reliability by means of acknowledgment (ACK). For every received packet, a TCP receiver returns an ACK packet to the sender. If an ACK packet is not received within a given *timeout* value, the TCP sender will assume that the packet is lost, and will retransmit the lost packet. Note that in the literature, a timeout period is also referred to as Retransmission TimeOut (RTO). Hereafter, we will refer to these two terms interchangeably.

TCP uses a *cumulative* acknowledgment mechanism. With this mechanism, a TCP receiver always acknowledges to the sender with the highest sequence number up to which all packets have been successfully received. For example, in Fig. 9.2, packet 3 is lost. Therefore, the TCP receiver returns ACK for packet 2 (A2) even when packets 4, 5, and 6 have been received. These ACK packets (e.g., A2), which acknowledge the same TCP packet (e.g., packet 2), are referred to as the *duplicated acknowledgment packets*. From Fig. 9.2, the TCP sender does not receive an ACK packet which acknowledges packet 3. After a period of RTO, the sender will assume that packet 3 is lost and will retransmit packet 3.

The RTO value is optimized according to the following tradeoff: a small RTO value leads to unnecessary packet retransmission, while a large RTO value results in high latency of packet loss detection. In general, an RTO value should be a function of network Round-Trip Time (RTT), which is the time required for a data bit to travel from a source node to a destination node and travel back to the source node. Due to network dynamics, RTT of one packet could be different from that of another. In TCP, smoothed (i.e., average) RTT ($\bar{t}$) and RTT variation (σ_t) are computed based on the collected RTT samples, and are used to compute the RTO value.

According to [23], instantaneous smoothed RTT, RTT variation, and instantaneous RTO are computed as follows. Let $t(k)$ be the kth RTT sample collected upon ACK reception. Also, let $\bar{t}(k)$, $\sigma_t(k)$, and $RTO(k)$ be the values of $\bar{t}$, σ_t, and RTO, respectively, when the kth RTT sample is determined. Then,

$$\bar{t}(k+1) = \alpha \times \bar{t}(k) + (1-\alpha) \times t(k+1), \tag{9.1}$$

$$\sigma_t(k+1) = \beta \times \sigma_t(k) + (1-\beta) \times |t(k+1) - \bar{t}(k)|, \tag{9.2}$$

$$\text{RTO}(k+1) = \min\{ub, \max\{lb, \gamma \times [\bar{t}(k+1) + 4 \times \sigma_t(k+1)]\}\}, \tag{9.3}$$

where ub and lb are fixed upper and lower bounds on the RTO value.[1] The constants $\alpha \in (0, 1)$ and $\beta \in (0, 1)$ are usually set to 7/8 and 3/4, respectively. The variable γ is a *binary exponential backoff* (BEB) factor. It is initialized to 1 and doubled for every timeout event, and is reset to 1 when a new ACK packet arrives.

9.1.2.2 Window-Based Congestion Control

A transport layer protocol is also responsible for network congestion. It limits the transmission rate of a data flow to help control network congestion. As a window-based congestion control protocol, TCP limits the transmission rate by adjusting the *congestion window* (i.e., transmission window) which basically refers to the amount of data that a sender can transmit without waiting for acknowledgment. For example, the congestion window size of the TCP connection in Fig. 9.2 is initialized to 4. Therefore, the TCP sender pauses after sending packets 1–4. After receiving ACK corresponding to packet 1 (i.e., A1), the number of unacknowledged packets becomes 3 and TCP is able to send out packet 5.

Congestion window of transmission window refers to a range of sequence numbers of TCP packets which can be transmitted at a moment. For example, the congestion window at the beginning of Fig. 9.2 is $\{1, 2, 3, 4\}$ and the congestion window size is 4. When A1 is received, the congestion window becomes $\{2, 3, 4, 5\}$. In this case, we say that the congestion window *slides* to the right. Suppose that the congestion window changes to $\{2, 3, 4, 5, 6\}$ (the size is 5). In this case, we say that the congestion window is *opened* by one unit. On the contrary, if the window becomes $\{2, 3, 4\}$, we say the congestion window is *closed* by one unit. Again, a larger window size allows the sender to transmit more data in a given interval implying a higher transmission rate at the transport layer. TCP increases and decreases its transmission rate by opening and closing its congestion window.

Window Increasing Mechanism

One of the key features of TCP is network-based rate adaptability. TCP slowly opens its congestion window to fill up the underlying network, when the network is underutilized. When the network is overutilized, TCP rapidly closes the congestion window to help relieve the congestion. TCP window opening mechanism consists of two phases, each of which is identified by the current congestion window size (w) and a *slow-start* threshold (w_{th}):

1. *Slow-start phase:* If $w < w_{\text{th}}$, TCP increases w by one for every received ACK packet.
2. *Congestion avoidance phase:* If $w \geq w_{\text{th}}$, TCP increases w by $\frac{1}{w(t)}$ for every received ACK packet.

[1] RFC 2988 recommends ub to be at least 60 s, and lb to be 1 s [23].

Note that, TCP receiver may advertise its maximum window size ($w_{\max}$) which does not fill its buffer too rapidly. This $w_{\max}$ acts as an upper-bound for the above window increasing mechanism. In NS2, congestion window (w) evolves according to the above two phases, regardless of $w_{\max}$. However, TCP uses the minimum of ω and $w_{\max}$ to determine the amount of data it can transmit.

Packet Loss Detection Mechanism

In the literature, various TCP variants use different combinations of the following packet loss detection mechanisms:

- *Timeout*: As discussed earlier, TCP starts its retransmission timer for every transmitted packet, and assumes a packet loss upon timer expiration.
- *Fast Retransmit*: By default, an RTO has granularity of 0.5 s, which could lead to large latency in packet loss detection. Fast retransmit expedites the packet loss detection by means of duplicated acknowledgment detection. Upon detection of the kth (which is equal to 3 by default) duplicated acknowledgment (excluding the first one which is a new acknowledgment), the TCP sender stops waiting for the timeout, assumes a packet loss, and retransmits the lost packet. From Fig. 9.2, if the fast retransmit mechanism is used, the TCP sender will assume that packet 3 is lost and it retransmits packet 3 upon receiving the 4th A2 packet (i.e., the third duplicated acknowledgment). Note that based on the cumulative acknowledgment principle, upon receiving the retransmitted packet 3, TCP receiver sends A6 back to the sender, since packets 4, 5, and 6 have been successfully received earlier.

Window Decreasing Mechanism

Originally conceived to combat congestion in a wired network, TCP assumes that all packet losses occur due to congestion (i.e., buffer overflow at the routers in the network). It reacts to every packet loss by reducing its transmission rate (or window size) to lessen the congestion. The following approaches are among the most popular window decreasing mechanisms for a TCP variant used in the literature.

- *Reset to 1*: Conventionally, TCP reacts to packet loss by resetting the window size to 1, and setting the slow-start threshold to half of the current congestion window size. However, this option is usually deemed too radical and could lead to TCP throughput degradation in the presence of random packet loss.
- *Fast Recovery*: Upon detection of a packet loss, the fast recovery mechanism sets both current window size and slow-start threshold to half of the current congestion window size. Then, it increases the window size by one for each duplicated acknowledgment received during the fast retransmit phase. At this moment, the sender may transmit a new packet if the congestion window allows.

Table 9.1 Differences among basic TCP variants: different window closing mechanisms upon detection of a packet loss

TCP variant	Loss detection	
	Timeout	Fast retransmit
Old-Tahoe	Reset w to 1	N/A
Tahoe	Reset w to 1	Reset w to 1
Reno	Reset w to 1	Fast recovery (single packet)
New Reno	Reset w to 1	Fast recovery (all packets)

Upon receiving a new acknowledgment, the sender exits Fast Recovery and sets the window size to the slow-start threshold value, after which TCP operates normally in a congestion avoidance phase.

9.1.2.3 TCP Variants

There are numerous TCP variants in the literature. This section discusses only de facto TCP variants namely Old Tahoe, Tahoe, Reno, and new Reno. These TCP variants use the same window increasing mechanism (i.e., slow start and congestion avoidance). However, they differ in how they detect a packet loss and decrease the window size. Table 9.1 shows the differences in window size adjustment mechanism, when packet loss is detected through timeout and Fast Retransmit (i.e., duplicated ACKs).

The very first TCP variant, Old-Tahoe, detects packet loss through timeout only. When packet loss is detected, it always resets congestion window size to 1. Developed from Old-Tahoe, TCP Tahoe uses the Fast Retransmit mechanism to expedite packet loss detection rather than waiting for the timeout. It always sets the window size to 1 upon detection of a packet loss. Both TCP Reno and New-Reno reset the window size to 1, when a packet loss is detected through timeout. However, they will use Fast Recovery if packet loss is detected through Fast Retransmit. The difference between TCP Reno and TCP New-Reno is that TCP Reno exits the fast recovery process as soon as the lost packet which triggered Fast Retransmit is acknowledged. If there are multiple packet losses within a congestion window, Fast Recovery could be invoked for several times, and the window size will decrease significantly. To avoid the multiple window closures, TCP NewReno stays in the Fast Recovery phase until all packets in the loss window are acknowledged or until timeout occurs.

9.2 Basic Agents

An agent is an NsObject which is responsible for creating and destroying packets. There are two main types of NS2 agents: routing agents and transport-layer agents. A routing agent creates, transmits, and receives routing control packets, and commands routing protocols to act accordingly. Connecting an application to a low-

Program 9.1 Class `AgentClass` which binds OTcl and C++ class `Agent`

```
    //~/ns/common/agent.cc
1   static class AgentClass : public TclClass {
2   public:
3       AgentClass() : TclClass("Agent") {}
4       TclObject* create(int, const char*const*) {
5           return (new Agent(PT_NTYPE));
6       }
7   } class_agent;
```

level network, a transport-layer agent controls the congestion and reliability of a data flow based on an underlying transport layer protocol (e.g., UDP or TCP). This book focuses on transport layer agents only.

NS2 implements agents in a C++ class `Agent`, which is bound to an OTcl class with the same name (see Program 9.1). In the following, we first discuss the relationship among a transport-layer agent, an application, and a low-level network in Sect. 9.2.1. Agent configuration and internal mechanisms are discussed in Sects. 9.2.2 and 9.2.3, respectively. Finally, Sect. 9.2.4 provides guidelines to define a new transport-layer agent.

9.2.1 Applications, Agents, and a Low-Level Network

An agent acts as a bridge which connects an application and a low-level network. Based on the user demand provided by an application, a sending agent constructs packets and transmits them to a receiving agent through a low-level network. Figure 9.3 shows an example of such a connection.

Consider Fig. 9.3. On the top level, a constant bit rate CBR application, which models a user demand to periodically transmit data, is used as an application. The demand is passed to a UDP sending agent, which in turn creates UDP packets. Here, the UDP agent stores source and destination IP addresses and transport layer ports in the packet header, and forwards the packet to the attached node (e.g., Node 1 in Fig. 9.3). Using the precalculated routing table, the low-level network delivers the packet to the destination node (e.g., Node 3 in Fig. 9.3) specified in the packet header. The destination node uses its demultiplexer to forward the packet to the agent attached to the port specified in the packet header. Finally, a Null receiving agent simply destroys the received packet.

From the above discussion, an agent can be used as a sending agent (e.g., a UDP agent) or a receiving agent (e.g., a Null agent). A sending agent has connections to both an application and a low-level network, while a receiving agent may not have a connection to an application (because it does not need any). An application (e.g., CBR) uses its variable "`agent_`" as a reference to an agent (e.g., UDP and Null agents), while an agent uses its variable "`app_`" as a reference to an application. It is mandatory to configure the variables "`agent_`" and

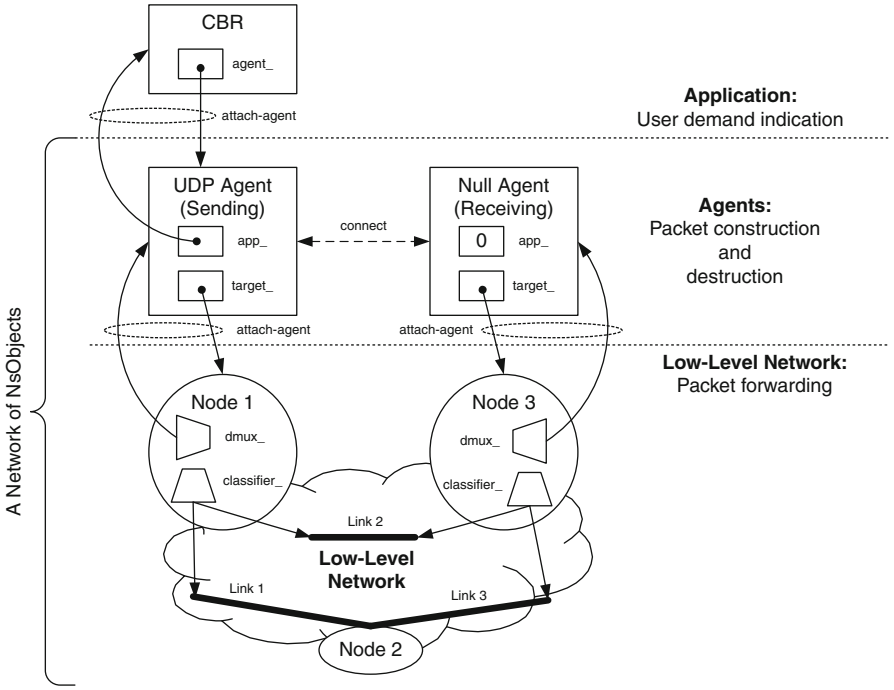

Fig. 9.3 A CBR application over UDP configuration

"`app_`" (i.e., create the connection) for a sending agent, while it is optional for a receiving agent. This is mainly because the application needs to inform the agent of user demand (i.e., by invoking function `sendmsg(...)`), and the sending agent needs to inform the application of the completion of data transmission (i.e., by invoking function `resume()`). Since a receiving agent simply destroys the received packet, it does not need a connection to an application.

Both sending and receiving agents connect to a low-level network in the same manner. They use a pointer "`target_`," to point to the attached node. The Node, on the other hand, *install*s the agent in the slot number "`port`" of its demultiplexer "`dmux_`" (which is of class `PortClassifier`), where "`port`" is the corresponding port number of the agent (see Sect. 6.5.3).

Table 9.2 shows the key differences between a sending agent and a receiving agent. The upstream object of a sending agent is an application, which informs a sending agent of incoming user demand through function `sendmsg(...)` of the sending agent. The upstream object of a receiving agent, on the other hand, is a `Node` object, which forwards packets to the receiving agent via function `recv(p,h)`. The downstream object of a sending agent is a `Node` object. The sending agent passes a packet `*p` to a `Node` object by executing `target_->recv(p,h)`. A receiving agent usually has no downstream object, since it simply destroys the received packets.

Table 9.2 Key differences between a sending and a receiving agent

	Sending agent	Receiving agent
Upstream		
– Object	`Application`	`Node`
– Packet forwarding function	`sendmsg`	`recv`
Downstream object		
– Object	`Node`	N/A
– Packet forwarding function	`recv`	N/A

9.2.2 Agent Configuration

From Fig. 9.3 and Program 9.2, agent configuration consists of four main steps:

1. Create a sending agent, a receiving agent, and an application using "`new{...}`" (Lines 8–10).
2. Attach agents to the application using OTcl Command `attach-agent-{agent}` of class `Application` (Line 11).
3. Attach agents to the a low-level network using instproc `attach-agent-{node agent}` of class `Simulator` (Lines 12 and 13).
4. Associate the sending agent with the receiving agent using instproc `connect {src dst}` of class `Simulator` (Line 14).

The details of these four steps will be discussed in greater detail in Sect. 11.1.

Example 9.1 (A Network Construction Example). The example network in Fig. 9.3 uses CBR, a UDP agent, and a Null agent as an application, a sending agent, and a receiving agent, respectively. To setup the example network, we may use the Tcl simulation script in Program 9.2.

Program 9.2 A simulation script which creates the network in Fig. 9.3

```
1  set ns [new Simulator]
2  set n1 [$ns node]
3  set n2 [$ns node]
4  set n3 [$ns node]
5  $ns duplex-link $n1 $n2 5Mb 2ms DropTail
6  $ns duplex-link $n2 $n3 5Mb 2ms DropTail
7  $ns duplex-link $n1 $n3 5Mb 2ms DropTail

   #=== UDP-Null peering starts here ===
8  set udp [new Agent/UDP]
9  set null [new Agent/Null]
10 set cbr [new Application/Traffic/CBR]
11 $cbr attach-agent $udp
12 $ns attach-agent $n1 $udp
13 $ns attach-agent $n3 $null
14 $ns connect $udp $null
```

While Lines 1–7 create a low-level network (see the details in Chaps. 6 and 7), Lines 8–14 set up a CBR application, a UDP agent, and a Null agent on top of the low-level network. □

9.2.3 Internal Mechanism for Agents

The internal mechanisms for agents are defined in the C++ domain as follows:

- *A sending agent*: Receive user demand by having the associated application invoke its function `sendmsg(...)`. From within `sendmsg(...)`, create packets using function `allocpkt()` and forward the created packets to the low-level network by executing `target_->recv(p,h)`.
- *A receiving agent*: Receive packets by having a low-level network demultiplexer invoke its function `recv(p,h)`. Destroy received packets by invoking function `free(p)` of class `Packet`.

In this section, we will discuss the detail of variables and functions required to perform the above mechanisms.

9.2.3.1 Related C++ and OTcl Variables

The main variables of C++ class `Agent` and their bound OTcl instvars are shown in Table 9.3. Of type `ns_addr_t` (see Sect. 8.3.3), variables "`here_`" and "`dst_`" contain addresses and ports of the Node attached to the sending agent and the receiving agent, respectively. An IPv6 priority level is stored in the variable "`prio_`." Variable "`app_`" acts as a reference to an `Application` object. Since class `Agent` is responsible for packet generation, it must assign a unique ID to every packet. Therefore, it maintains a static variable "`uidcnt_`," which counts the total number of generated packets. When a packet is created, an `Agent` object sets the unique ID of the packet to be "`uidcnt_`," and increases "`uidcnt_`" by one (see function `initpkt(p)` in Line 10 of Program 9.3).

9.2.3.2 Key C++ Functions

A list of key C++ functions with their descriptions is given below (see the declaration of class `Agent` in file *~ns*/common/agent.cc,h). Since class `Agent` is a template for transport layer agents, it provides no implementation for some of its

Table 9.3 The list of C++ and OTcl variables of class `Agent`

C++ type	C++ variable	OTcl instvar	Description
`ns_addr_t`	`here_`		
	`here_.addr_`	`agent_addr_`	Address of the attached node
	`here_.port_`	`agent_port_`	Port where the sending agent is attached
`ns_addr_t`	`dst_`		
	`dst_.addr_`	`dst_addr_`	Address of the attached node attaching to a peering agent
	`dst_.port_`	`dst_port_`	Port where the receiving agent is attached
`int`	`fid_`	`fid_`	Flow ID
`int`	`prio_`	`prio_`	IPv6 priority field(e.g., 0 = unspecified, 1 = background traffic)
`int`	`flags_`	`flags_`	Flags
`int`	`defttl_`	`ttl_`	Default time to live value
`int`	`size_`	N/A	Packet size
`packet_t`	`type_`	N/A	Payload type
`int`	`seqno_`	N/A	Current sequence number
`Application*`	`app_`	N/A	A pointer to an application
`int`	`uidcnt_`	N/A	Total number of packets generated by all agents

functions. The child classes of class `Agent` are responsible for implementing these functions.

`recv(p,h)`	Receive a packet `*p`
`send(p,h)`	Send a packet `*p`
`send(nbytes)`	Send a message with `nbytes` bytes
`sendmsg(nbytes)`	Send a message with `nbytes` bytes
`timeout(tno)`	Action to be performed at timeout
`connect(dst)`	Connect to a dynamic destination `dst`
`close()`	Close a connection-oriented session
`listen()`	Wait for a connection-oriented session
`attachApp(app)`	Store `app` in the variable `app_`
`allocpkt()`	Create a packet
`initpkt(p)`	Initialize the input packet `*p`
`recvBytes(bytes)`	Send data of `bytes` bytes to the attached application
`idle()`	Tell the application that the agent has nothing to transmit

Program 9.3 Constructor, function `allocpkt`, and function `initpkt` of class `Agent`

```
    //~/ns/common/agent.cc
1   Agent::Agent(packet_t pkttype):size_(0),type_(pkttype),
                                              app_(0){}

2   Packet* Agent::allocpkt() const
3   {
4       Packet* p = Packet::alloc();
5       initpkt(p);
6       return (p);
    }

7   void Agent::initpkt(Packet* p)
    {
8       hdr_cmn* ch = hdr_cmn::access(p);
9       ch->uid() = uidcnt_++;
10      ch->ptype() = type_;
11      ...
12      hdr_ip* iph = hdr_ip::access(p);
13      iph->saddr() = here_.addr_;
14      iph->sport() = here_.port_;
15      iph->daddr() = dst_.addr_;
16      iph->dport() = dst_.port_;
17      ...
18  }
```

The Constructor

Class `Agent` has no default constructor. Its only constructor takes a `packet_t` (see Sect. 8.3.4 and Program 8.9) object as an input argument (see Line 1 of Program 9.3). The constructor sets the variable "`type_`" to be as specified in the input argument and resets other variables to zero. This payload type setting implies that one agent is able to transmit packets of one type only. We need several agents to transmit packets of several types.

Functions `allocpkt()` and `initpkt(p)`

Shown in Program 9.3, function `allocpkt()` is the main packet construction function. It creates a packet by invoking function `alloc()` of class `Packet` in Line 4, and initializes the packet by invoking function `initpkt(p)` in Line 5. After initialization, function `allocpkt()` returns a pointer to the constructed packet "`p`" to the caller.

The details of function `initpkt(p)` are shown in Lines 8–18 of Program 9.3. Function `initpkt(p)` sets the initial values in the packet header of the input packet "`*p`" to the default values. The uniqueness of the unique ID field "`uid_`" in the common header is assured by setting "`uid_`" to be the total number of packets

Program 9.4 Functions `attachApp` and `recv` of class `Agent`

```
    //~/ns/common/agent.cc
1   void Agent::attachApp(Application *app)
2   {
3       app_ = app;
4   }

5   void Agent::recv(Packet* p, Handler*)
6   {
7       if (app_)
8           app_->recv(hdr_cmn::access(p)->size());
9       Packet::free(p);
10  }
```

allocated so far. Class `Agent` stores the total number of allocated packets in its static variable "`uidcnt_`." Since the variable "`uidcnt_`" is distinct and unique to all agents, assigning this variable to the field "`uid_`" of the common header (Line 9) assures the uniqueness of packet unique ID.

Other initialization includes setting up the payload type in the common header to be as specified in the variable "`type_`" (Line 10). Also, Lines 12–16 configure source and destination IP addresses and port numbers in the variables "`here_`" and "`dst_`."

Function `attachApp(app)`

Lines 1–4 in Program 9.4 show the details of function `attachApp(app)`. To bind an application to an agent, function `attachApp(app)` stores the input pointer "`app`" in its pointer to a `Application` object, "`app_`." After this point, the agent may invoke `public` functions of the attached application through the pointer "`app_`."

Functions `recv(p,h)`, `send(p,h)`, and `sendmsg(nbytes)`

These functions are used by sending and receiving agents in the packet forwarding process. On the sender side, an application informs a sending agent of user demand by invoking functions `send(nbytes)`, and `sendmsg(...)` of class `Agent`. As an NsObject, the sending agent forwards an incoming packet `*p` to a downstream NsObject by executing `target_->recv(p,h)`. Functions `send(nbytes)` and `sendmsg(...)` have no implementation in the scope of class `Agent`, and must be implemented by the child classes of class `Agent`.

On the receiver side, an NsObject forwards packets to a receiving agent by invoking its function `recv(p,h)`. Shown in Lines 5–10 of Program 9.4, function `recv(p,h)` deallocates the received packet (Line 9) and may inform the attached

application (if it exists) of packet reception by invoking function `recv(size)` of the attached `Application` object (Lines 7 and 8), where `size` is the size of packet `*p`.

9.2.4 Guidelines to Define a New Transport Layer Agent

Class `Agent` provides the basic functionalities necessary for most agents. A new agent can be created based on these functionalities, following the guidelines below:

1. Define an inheritance structure: Select a base class and derive a new agent class from the selected base class. Bind the C++ and OTcl agent class names together.
2. Define necessary C++ variables and OTcl instvars.
3. Implement the constructors of both C++ and OTcl classes. Bind the variables and the instvars if necessary. Feed payload type (i.e., `packet_t`) as an input argument of the C++ constructor.
4. Implement the necessary functions including functions `send(nbyte)`, `sendmsg(...)`, `recv(p,h)`, and `timeout(tno)`. Also define OTcl instprocs if necessary.
5. Define necessary OTcl commands as interfaces to the C++ domain from the OTcl domain.
6. [Optional] Define a timer (see Sect. 15.1).

9.3 UDP and Null Agents

UDP is a connectionless transport layer protocol, which provides neither congestion control nor error control. In NS2, a UDP agent is used as a sending agent. It is usually peered with a Null (receiving) agent, which is responsible for packet destruction. Figure 9.3 shows a network configuration example where a Constant Bit Rate (CBR) traffic source uses a UDP agent and a Null agent as its transport later agents. Here, the CBR asks the UDP agent to transmit a burst of packets for every fixed interval. The UDP agent creates and forwards packets to the low-level network, irrespective of the network condition. On the receiving end, the Null agent simply destroys the packets received from the low-level network. In the following, we will discuss the details of UDP and Null agents.

9.3.1 Null (Receiving) Agents

A Null agent is the simplest but one of the most widely used receiving agents. The main responsibility of a Null agent is to deallocate packets through function

Program 9.5 Mapping class `UdpAgentClass` which binds a C++ class `UdpAgent` to an OTcl class `Agent/UDP`

```
    //~/ns/apps/udp.cc
1   static class UdpAgentClass : public TclClass {
2   public:
3       UdpAgentClass() : TclClass("Agent/UDP") {}
4       TclObject* create(int, const char*const*) {
5           return (new UdpAgent());
6       }
7   } class_udp_agent;
```

`free(p)` of class `Packet` (see Line 9 in Program 9.4). A Null agent is represented by an OTcl class `Agent/Null` which is derived directly from an OTcl class `Agent` (see file *~ns*/tcl/lib/ns-agent.tcl). Due to its simplicity, Null agents have no implementation in the C++ domain.

9.3.2 UDP (Sending) Agent

A UDP agent is perhaps the simplest form of sending agents. It receives user demand to transmit data by having the attached application invoke its function (e.g., `sendmsg(...)`), creates packets based on the demand, and forwards the created packet to a low-level network. An application may use the three following ways to tell a UDP agent to send out packets: via a C++ function `sendmsg(...)` of class `UdpAgent`, via an OTcl command `send{...}` of OTcl class `Agent/UDP`, or via an OTcl command `sendmsg{...}` of OTcl class `Agent/UDP`.

Again, NS2 defines a UDP sending agent based on the guideline in Sect. 9.2.4. Since a UDP agent implements no acknowledgment mechanism and needs no timer, we can skip the last step in the guideline.

Step 1: Define Inheritance Structure

A UDP agent is represented by a C++ class `UdpAgent` and an OTcl class `Agent/UDP`. These two classes derive from class `Agent` in their domains, and are bound using a mapping class `UdpAgentClass` (see Program 9.5).

Step 2: Define C++ Variables and OTcl Instvars

The key variable of class `UdpAgent` is "`seqno_`" (Line 12 in Program 9.6), which counts the number of packets generated by a `UdpAgent` object. Note that every

Program 9.6 Declaration and the constructors of class `UdpAgent` as well as the default value of the instvar `packetSize_` of class `Agent/UDP`

```
    //~/ns/apps/udp.h
 1  class UdpAgent : public Agent {
 2  public:
 3      UdpAgent();
 4      UdpAgent(packet_t);
 5      virtual void sendmsg(int nbytes, const char *flags = 0){
 6          sendmsg(nbytes, NULL, flags);
 7      }
 8      virtual void sendmsg(int nbytes, AppData* data, ...
                                        const char *flags = 0);
 9      virtual void recv(Packet* pkt, Handler*);
10      virtual int command(int argc, const char*const* argv);
11  protected:
12      int seqno_;
13  };

    //~/ns/apps/udp.cc
14  UdpAgent::UdpAgent() : Agent(PT_UDP), seqno_(-1){
15      bind("packetSize_", &size_);
16  }

17  UdpAgent::UdpAgent(packet_t type) : Agent(type){
18      bind("packetSize_", &size_);
19  }

    //~/ns/tcl/lib/ns-default.tcl
20  Agent/UDP set packetSize_ 1000
```

packet has a unique ID "`uid_`." Also, every packet generated by the *same agent* has a unique sequence number "`seqno_`." However, two packets generated by different agents may have the *same* sequence number "`seqno_`" but they must have different unique ID "`uid_`."

Step 3: Implement the Constructors in the C++ and OTcl Domains

NS2 implements constructor for UDP agents in the C++ domain only. From Program 9.6, the default constructor in Lines 14–16 feeds UDP payload type (i.e., `PT_UDP`) to the constructor of class `Agent`, essentially storing `PT_UDP` in the variable `type_`. It also sets the sequence number (i.e., `seqno_`) to be −1. By specifying the payload type, the constructor in Lines 17–19 sets the payload type to be as specified in the input argument. The constructor in this case does not set the value of "`seqno_`" since the packets of specified type may not have sequence number. For both constructors, the C++ variable "`size_`," which specifies the packet size,

Program 9.7 Function `sendmsg` of class `UdpAgent` and function `idle` of class `Agent`

```
    //~/ns/apps/udp.cc
 1  void UdpAgent::sendmsg(int nbytes,AppData* data,const char*
                                                          flags)
 2  {
 3      Packet *p;
 4      int n = nbytes / size_;
 5      while (n-- > 0) {
 6          p = allocpkt();
 7          /* packet header configuration */
 8          hdr_cmn::access(p)->size() = size_;
 9          ...
10          /* --------------------------- */
11          target_->recv(p);
12      }

13      n = nbytes % size_;
14      if (n > 0) {
15          p = allocpkt();
16          /* packet header configuration */
17          hdr_cmn::access(p)->size() = n;
18          ...
19          /* --------------------------- */
20          target_->recv(p);
21      }
22      idle();
23  }

    //~/ns/common/agent.cc
24  void Agent::idle() { if (app_) app_->resume(); }
25  }
```

is bound to instvar `packetSize_` in the OTcl domain (Lines 15 and 19). By default, the packet size is set to 1,000 bytes in the file *~ns*/tcl/lib/ns-default.tcl (Line 20).

Step 4: Define the Necessary C++ Functions

As a sending agent, a UDP agent needs to define a function `sendmsg(...)` to receive a user demand from the application. Program 9.7 shows the details of function `sendmsg(nbytes,data,flags)`, which takes three input arguments: "`nbytes`," "`data`," and "`flags`." Function `sendmsg(...)` divides data payload with size "`nbytes`" bytes into "`n`" (see Line 4) or "`n+1`" parts (depending on "`nbytes`"), stores each part into a UDP packet (which contains a payload of "`size_`" bytes), and transmits all ("`n`" or "`n+1`") packets to the attached low-level network.

Program 9.8 OTcl Commands `send` and `sendmsg` of class `Agent/UDP`

```
    //~/ns/apps/udp.cc
1   int UdpAgent::command(int argc, const char*const* argv)
2   {
3       if (argc == 4) {
4           if (strcmp(argv[1], "send") == 0) {
5               PacketData* data = new PacketData(1 + strlen
                                                    (argv[3]));
6               strcpy((char*)data->data(), argv[3]);
7               sendmsg(atoi(argv[2]), data);
8               return (TCL_OK);
9           }
10      } else if (argc == 5) {
11          if (strcmp(argv[1], "sendmsg") == 0) {
12              PacketData* data = new PacketData(1 + strlen
                                                    (argv[3]));
13              strcpy((char*)data->data(), argv[3]);
14              sendmsg(atoi(argv[2]), data, argv[4]);
15              return (TCL_OK);
16          }
17      }
18      return (Agent::command(argc, argv));
19  }
```

Since NS2 rarely sends actual payload along with a packet, Line 8 sets the size of packet to be "`size_`." Line 11 sends out the created packet, by executing `target_->recv(p)`.[2] Lines 6–11 are repeated "`n`" times to transmit all "`n`" packets.

After transmitting the first "`n`" packets, the entire application payload is left with `nbytes % size_`, where `%` is the modulus operator. If the remainder is nonzero, Lines 15–20 will transmit the remaining application payload in another packet. Finally, Line 22 invokes function `idle()` to inform the attached application that the UDP agent has finished data transmission. From Line 24, function `idle()` does so by invoking function `resume()` of the attached application (if any).

There are two important notes for UDP agents. First, since a UDP agent is a sending agent its function `recv(p,h)` is generally not to be used. Second, in Program 9.7, function `sendmsg(...)` transmits packets, irrespective of network condition.

Step 5: Define OTcl Commands and Instprocs

Class `Agent/UDP` defines the two following OTcl commands defined in Program 9.8:

[2] Variable `target_` is configured to point to a node entry during the network configuration phase (see Sect. 9.2.1).

- `send{nbytes str}`: Send a payload of size "`nbytes`" containing a message "`str`."
- `sendmsg{nbytes str flags}`: Similar to the OTcl command `send` but also passes the input flag "`flags`" when sending a packet.

Lines 5–8 in Program 9.8 show the details of the OTcl command `send{...}`. Line 5 creates a `PacketData` object. Line 6 stores the input message "`str`" in the created `PacketData` object. Line 7 sends out the application payload by invoking function `sendmsg(...)`. Note that the size of application payload does not depend on the size of the message in the `PacketData` object (i.e., `argv[3]` or "`str`"). Rather, the size is specified in the first input argument (i.e., `argv[2]` or "`nbytes`"). The implementation of the OTcl command `sendmsg(...)` is similar to that of the OTcl command `send{...}`. However, it also feeds a flag "`flags`" as an input argument of function `sendmsg(...)` (see Line 14).

9.3.3 Setting Up a UDP Connection

A UDP connection can be created by the network configuration method provided in Sect. 9.2.2. An example connection where a UDP agent, a Null agent, and a CBR traffic source are used as a sending agent, a receiving agent, and an application is shown in Example 9.1.

9.4 Chapter Summary

An agent is a connector which bridges an application to a low-level network. Its main responsibilities are to create packets based on user demand received from an application, to forward packets to a low-level network, and to destroy packets received from a low-level network. From this point of view, an agent can be used to model transport layer protocols and routing protocols. This chapter focuses on transport layer (protocol) agents only.

Class `Agent` is a base class, which represents both sending and receiving agents. It connects to an application and a low-level network using pointers "`app_`" and "`target_`," respectively. An application also has a pointer "`agent_`" to the agent, while a low-level network uses a pointer "`target_`" as a reference to the agent. Class `Agent` provides basic functionalities for creating, forwarding, and destroying packets. Its functions `send(...)` and `sendmsg(...)` are invoked by an attached application to pass on user demand. An agent creates packets based on the demand, and forwards the created packet to a low-level network by executing `target_->recv(p,h)`. A low level network sends a packet to a receiving agent by invoking function `recv(p,h)` of the receiving agent.

UDP and TCP are among the most widely used transport layer protocols. UDP is a simple transport layer protocol and it can be flexibly used by other network

protocols. In NS2, UDP is implemented in the C++ class `UdpAgent` which is bound to an OTcl class `Agent/UDP`. A UDP agent is usually peered with a Null agent, which simply destroys received packets.

TCP is a reliable transport control protocol. Its main features are end-to-end error control and network congestion control. It implements timeout and acknowledgment to provide end-to-end error control and adopts a window-based rate adjustment to control network congestion. We will discuss the details of TCP implementation in NS2 in the Chap. 10.

9.5 Exercises

1. What is the default TCP version used in NS2? How does it react to duplicate acknowledgments and timeout?
2. Suppose a series of packets measures the following set of round trip time (RTT): 9.7, 10.1, 11.2, 15.7, 8.8, 7.2, 12.2, and 15.8 s. Suppose further that a new packet with round-trip time 13.1 s has arrived. Use [35] to compute smoothed RTT, RTT variation, and instantaneous retransmission timeout (RTO).
3. What is the characteristic unique to an Agent?
4. What are the NS2 objects which are responsible for putting IP addresses and ports in packets? Explain how these objects write the addresses and the port into the packet header.
5. What are C++ and OTcl classes which model senders and receivers for TCP and UDP packets?
6. Design a connection-oriented transport layer protocol in NS2. This protocol sets up the connection in the same way as TCP does, but transmit data packets in the same way as UDP does. Incorporate the protocol into NS2 and write a program to test the developed module.
7. Where does NS2 store the following information: the packet sequence number, the maximum window size, the current congestion windows, and flow ID? Give you answer for both TCP and UDP packets, wherever applicable.
8. How does NS2 ensure the uniqueness of packet unique ID?

Chapter 10
Transport Control Protocols
Part 2: Transmission Control Protocol

As a transport control protocol, Transmission Control Protocol (TCP) bridges an application to a low-level network, controls network congestion, and provides reliability to an end-to-end connection. This chapter discusses the details of TCP agents. Section 10.1 gives an overview of TCP agents. Here, we show a TCP network configuration method, a brief overview of TCP internal mechanism, TCP header format, and the main steps in defining TCP senders and TCP receivers. Sections 10.2 and 10.3 discuss the implementation of TCP receivers and senders, respectively. Sections 10.4–10.7 present the implementation of four main functionalities of a TCP sender. Finally, the chapter summary is provided in Sect. 10.8.

10.1 An Overview of TCP Agents in NS2

Based on user demand from an application, a TCP sender creates and forwards packets to a low-level network. It controls the congestion by limiting the rate (i.e., by adjusting the congestion window) at which packets are fed to the low-level network. It enforces an acknowledgment mechanism to provide connection reliability. A TCP receiver must acknowledge every received TCP packet. Based on the acknowledgment pattern, a TCP sender determines whether the transmitted packet was lost or not. If so, it will retransmit the packet. A TCP sender is responsible for sending packets as well as controlling the transmission rate, while the role of a TCP receiver is only to return acknowledgments to the associated TCP sender.

10.1.1 Setting Up a TCP Connection

As a transport layer agent, TCP can be incorporated into a network using the method discussed in Sect. 9.2.2.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_10, © Springer Science+Business Media, LLC 2012

Example 10.1. Consider Fig. 9.3. Replace the CBR application with File Transfer Protocol (FTP), the UDP agent with a TCP sender, and the Null agent with a TCP receiver. The modified network can be created using the following Tcl simulation script.

```
1  set ns [new Simulator]
2  set n1 [$ns node]
3  set n2 [$ns node]
4  set n3 [$ns node]
5  $ns duplex-link $n1 $n2 5Mb 2ms DropTail
6  $ns duplex-link $n2 $n3 5Mb 2ms DropTail
7  $ns duplex-link $n1 $n3 5Mb 2ms DropTail

   #=== TCP connection setup starts here ===
8  set tcp [new Agent/TCP]
9  set sink [new Agent/TCPSink]
10 set ftp [new Application/FTP]
11 $ns attach-agent $n1 $tcp
12 $ns attach-agent $n3 $sink
13 $ftp attach-agent $tcp
14 $ns connect $tcp $sink
15 $ns at 0.0 "$ftp start"
```

Similar to those in Example 9.1, Lines 8–14 above create a TCP connection on top of a low-level network. □

10.1.2 Packet Transmission and Acknowledgment Mechanism

TCP provides connection reliability by means of acknowledgment and packet retransmission. Figure 10.1 shows a diagram for TCP packet transmission and acknowledgment mechanisms. The process starts when an application (e.g., `FTP`) informs a TCP sender (e.g., `TcpAgent`) of user demand by invoking function `sendmsg(nbytes)` of the `TcpAgent` object through its variable "`agent_`." The TCP sender creates TCP packets, and forwards them to its downstream object by executing `target_->recv(p,h)`. The low-level network delivers the packets to the destination node attached to the TCP receiver (i.e., `TcpSink`). The destination node forwards the packet to the TCP receiver (i.e., a `TcpSink` object) by invoking function `recv(p,h)` of the TCP receiver installed in its demultiplexer (e.g., "`dmux_`"). Upon receiving a TCP packet, the TCP receiver creates an ACK packet and returns it to the TCP sender by executing `target_->recv(p,h)`, where in this case `p` is a pointer to the created ACK packet. The low-level network delivers the ACK packet to the sending node, which forwards the ACK packet to the TCP sender via its demultiplexer.

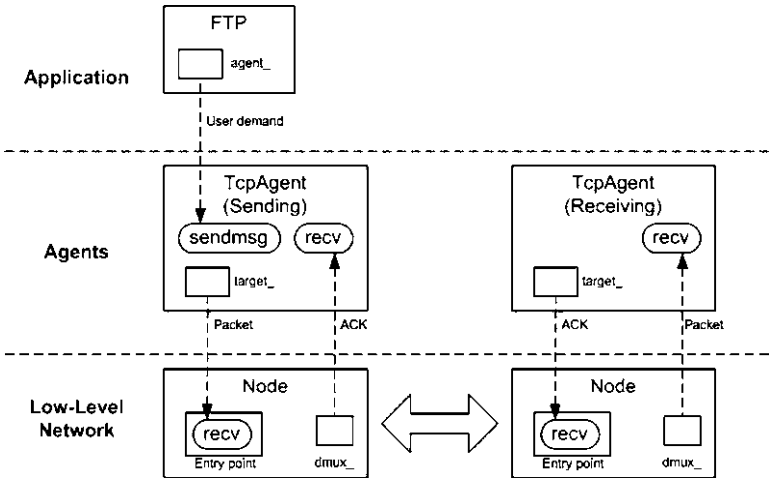

Fig. 10.1 TCP packet transmission and acknowledgment mechanisms

If a TCP packet or an ACK packet is lost (or delayed for a long period of time), the TCP sender will assume that the packet is lost. In this case, the TCP sender will retransmit the lost TCP packet based on the retransmission process explained in Sect. 9.1.2.

10.1.3 TCP Header

TCP packet header is defined in the "`hdr_tcp`" `struct` data type shown in Program 10.1. The key variables of `hdr_tcp` include

`seqno_`	TCP sequence number
`ts_`	Timestamp: The time when the packet was generated
`ts_echo_`	Timestamp echo: The time when the peering TCP received the packet
`reason_`	Reason for packet transmission (e.g., 0 = normal transmission)

In common with other packet header, `hdr_tcp` contains function `access(p)` (Lines 8–10), which can be used to obtain the reference to a TCP header stored in the input packet `*p`. This reference can then be used to access the attributes of a TCP packet header.

10.1.4 Defining TCP Sender and Receiver

We follow the guidelines provided in Sect. 9.2.4 to define a TCP sender and a TCP receiver.

Program 10.1 Declaration of `hdr_tcp struct` data type

```
    //~/ns/tcp/tcp.h
1   struct hdr_tcp {
2       double ts_;          //time packet generated (at source)
3       double ts_echo_;     //the echoed timestamp
4       int seqno_;          //sequence number
5       int reason_;         //reason for a retransmit
6       static int offset_; // offset for this header
7       inline static int& offset() { return offset_; }
8       inline static hdr_tcp* access(Packet* p) {
9           return (hdr_tcp*) p->access(offset_);
10      }
11      int& seqno() { return (seqno_); }
12      ...
13  };
```

Step 1: Define the Inheritance Structure

NS2 defines TCP sender in a C++ class `TcpAgent` which is bound to an OTcl class `Agent/TCP` through a mapping class `TcpClass`, as shown in Lines 1–7 of Program 10.2. Similarly, TCP receiver is defined in a C++ class `TcpSink` which is bound to an OTcl class `Agent/TCPSink` through a mapping class `TcpSinkClass`, as shown in Lines 8–14 of Program 10.2.

Step 2: Define the Necessary C++ and OTcl Variables

While class `TcpSink` has only one C++ key variable "`acker_`" which is of class `Acker`,[1] class `TcpAgent` has several variables. We classify the key C++ variables of class `TcpAgent` into four categories. First, C++ variables, whose values change dynamically during a simulation, are shown in Table 10.1. Second, C++ variables, which are usually configured once, are listed in Table 10.2. Third, Table 10.3 shows the variables that are related to TCP timer mechanism. Finally, Table 10.4 shows the Miscellaneous variables of class `TcpAgent`.

10.1.4.1 Step 3: Implement the Constructor

The constructors of both TCP senders and TCP receivers set their variables to the default values, and bind C++ variables to OTcl instvars as specified in Tables 10.1–10.3. In addition, the constructor of the TCP sender invokes the

[1] We will be discuss the details of class `Acker` later in Sect. 10.2.1.

Program 10.2 Class `TcpClass` which binds a C++ class `TcpAgent` and an OTcl class `Agent/TCP` together, class `TcpSinkClass`, which binds a C++ class `TcpSink` and an OTcl class `Agent/TCPSink` together, and the constructor of class `TcpSink`

```
    //~/ns/tcp/tcp.cc
1   static class TcpClass : public TclClass {
2   public:
3       TcpClass() : TclClass("Agent/TCP") {}
4       TclObject* create(int , const char*const*) {
5           return (new TcpAgent());
6       }
7   } class_tcp;

    //~/ns/tcp/tcp-sink.cc
8   static class TcpSinkClass : public TclClass {
9   public:
10      TcpSinkClass() : TclClass("Agent/TCPSink") {}
11      TclObject* create(int, const char*const*) {
12          return (new TcpSink(new Acker));
13      }
14  } class_tcpsink;

15  TcpSink::TcpSink(Acker* acker) : Agent(PT_ACK),
              acker_(acker) {...}
```

Program 10.3 Components of `TcpAgent` related to TCP retransmission timer

```
    //~/ns/tcp/tcp.h
1   class TcpAgent : public Agent {
2       ...
3       protected:
4       RtxTimer rtx_timer_;
5       ...
6   }

    //~/ns/tcp/tcp.cc
7   TcpAgent::TcpAgent() : ... Agent (PT_TCP), rtx_timer_(this),
      ...
8   {     ...     }
```

constructor of its parent class (i.e., `Agent`) with an input argument `PT_TCP`, setting the instantiated `TcpAgent` object to transmit TCP packet only. It also initializes the retransmission timer "`rtx_timer_`" with the pointer "`this`" to itself. The details of `TcpAgent` construction and timers are given in file *~ns*/tcp/tcp.cc and Sect. 15.1.

A TCP receiver is somewhat different from a TCP sender, since it does not have a default constructor. From Line 15 of Program 10.2, the constructor takes a pointer to an `Acker` object as an input argument (see Sect. 10.2.1), and initializes

Table 10.1 Key operating variables of class `TcpAgent`

C++ variable	OTcl variable	Default value	Description
`t_seqno_`	`t_seqno_`	0	Current TCP sequence number
`curseq_`	`seqno_`	0	Total number of packets need to be transmitted specified by the application. A TCP sender transmits packets as long as its sequence number is less than `curseq_`.
`highest_ack_`	`ack_`	0	Highest ACK number (not frozen during fast recovery)
`lastack_`	N/A	0	Highest ACK number (frozen during fast recovery)
`cwnd_`	`cwnd_`	0	Congestion window size in packets
`ssthresh_`	`ssthresh_`	0	Slow-start threshold
`dupacks_`	`dupacks_`	0	Duplicated ACK counter
`maxseq_`	`maxseq_`	0	Highest transmitted sequence number
`t_rtt_`	`rtt_`	0	RTT sample
`t_srtt_`	`srtt_`	0	Smoothed RTT
`t_rttvar_`	`rttvar_`	0	RTT deviation
`t_backoff_`	`backoff_`	0	Current RTO backoff multiplicative factor
`rtt_active_`	N/A	0	Status of the RTT collection process
`rtt_ts_`	N/A	–1	Time at which the packet is transmitted
`rtt_seq_`	N/A	0	Sequence number of the tagged packet
`t_rtxcur_`	N/A	0	Current value of unbounded retransmission timeout
`ts_peer_`	N/A	0	Latest timestamp provided by the peering TCP receiver
`rtx_timer_`	N/A	N/A	Retransmission timer object

its variable "`acker_`" with this input pointer. It also initializes its parent constructor with `PT_ACK`, an ACK packet type. Finally, it binds few C++ variables to OTcl instvars (see the detailed construction of class `TcpSink` in file *~ns*/tcp/tcp-sink.cc).

Steps 3, 4, and 5: Implement the Necessary Functions, OTcl Commands, and Instprocs, and Define Timers if Necessary

The detailed implementation of C++ functions of TCP receivers are shown in the next section, while those of TCP senders are given in Sects. 10.3–10.7. For brevity, we will not discuss the details of implementation of OTcl command and instproc. The readers are encouraged to study the details of TCP senders and TCP receivers in files *~ns*/tcp/tcp.cc,h, *~ns*/tcp/tcp-sink.cc,h, and *~ns*/tcl/lib/ns-agent.tcl.

Table 10.2 Key variables of class `TcpAgent`

C++ variable	OTcl variable	Default value	Description
`wnd_`	`window_`	20	Upper bound on window size
`numdupacks_`	`numdupacks_`	3	Number of duplicated ACKs which triggers fast retransmit
`wnd_init_`	`windowInit_`	2	Initial value of window size
`size_`	`packetSize_`	1,000	TCP packet size in bytes
`tcpip_base_`	`tcpip_base_hdr_size_`	40	TCP basic header size in bytes
`useHeaders_`	`useHeaders_`	`true`	If `true`, TCP and IP header size will be added to packet size
`maxburst_`	`maxburst_`	0	Maximum number of bytes that a TCP sender can transmit in one transmission
`maxcwnd_`	`maxcwnd_`	0	Upper bound on `cwnd_`
`control_`	`control_increase_`	0	If set to 1, do not open the congestion window when the network is limited (See Sect. 10.5).
`increase_`			

Table 10.3 Timer-related variables of class `TcpAgent`

C++ variable	OTcl variable	Default value	Description
`srtt_init_`	`srtt_init_`	0	Initial value of `t_srtt_`
`rttvar_init_`	`rttvar_init_`	12	Initial value of `t_rttvar_`
`rtxcur_init_`	`rtxcur_init_`	3.0	Initial value of `t_rtxcur_`
`T_SRTT_BITS`	`T_SRTT_BITS`	3	Multiplicative factor for smoothed RTT
`T_RTTVAR_BITS`	`T_RTTVAR_BITS`	2	Multiplicative factor for RTT deviation
`rttvar_exp_`	`rttvar_exp_`	2	Multiplicative factor for RTO computation
`decrease_num_`	`decrease_num_`	0.5	Window decreasing factor
`increase_num_`	`increase_num_`	1.0	Window increasing factor
`tcp_tick_`	`tcp_tick_`	0.01	Timer granularity in seconds
`maxrto_`	`maxrto_`	100,000	Upper bound on RTO in seconds
`minrto_`	`minrto_`	0.2	Lower bound on RTO in seconds

10.2 TCP Receiver

A TCP receiver is responsible for deallocating received TCP packets and returning cumulative ACK packets to the TCP sender. As discussed in Sect. 9.1.2, a cumulative ACK packet acknowledges a TCP packet with the highest contiguous sequence

Table 10.4 Miscellaneous variables of class `TcpAgent`

C++ variable	Default value	Description
`cong_action_`	0	`true` when the congestion has occurred.
`singledup_`	1	If set to 1, the TCP sender will transmit new packets upon receiving first few duplicated ACK packets.
`prev_highest_ack_`	N/A	Sequence number of an ACK packet received before the current ACK packet.
`last_cwnd_action_`	N/A	The latest action on congestion window
`recover_`	N/A	The highest transmitted sequence number during the previous packet loss event

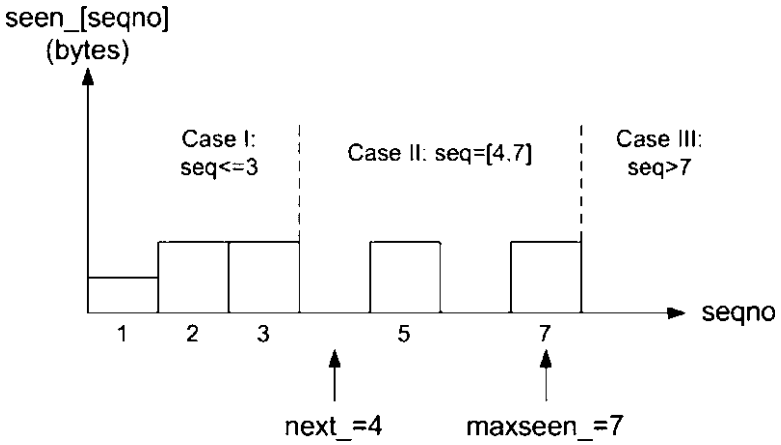

Fig. 10.2 Information necessary to generate a cumulative acknowledgement

number. Upon receiving a cumulative ACK packet, the TCP sender assumes that all packets whose sequence numbers are lower than or equal to that of the ACK packet have been successfully received. A cumulative ACK packet has the capability of acknowledging multiple packets. For example, suppose Packet 3 in Fig. 9.2 is not lost but is delayed and that it arrives at the receiver right after Packet 6 is received. Upon receiving Packet 3, the receiver acknowledges with A6, since it has received Packets 4–6 earlier.

In NS2, C++ implementation of TCP receivers involves two main classes: `Acker` and `TcpSink`. Class `Acker` is a helper class responsible for generating ACK packets. Class `TcpSink` contains an `Acker` object and acts as interfaces to a peering TCP sender.

Program 10.4 Declaration of class `Acker`

```
    //~/ns/tcp/tcp-sink.h
 1  class Acker {
 2  public:
 3      Acker();
 4      virtual ~Acker() { delete[] seen_; }
 5      inline int Seqno() const { return (next_ - 1); }
 6      inline int Maxseen() const { return (maxseen_); }
 7      int update(int seqno, int numBytes);
 8  protected:
 9      int next_;
10      int maxseen_;
11      int wndmask_;
12      int *seen_;
13      int is_dup_;
14  };
```

10.2.1 Class `Acker`

Program 10.4 shows the declaration of a C++ class `Acker`.[2] Class `Acker` stores necessary information required to generate cumulative ACK packets in the following variables:

`seen_`	An array whose index and value are the sequence number and the corresponding packet size, respectively
`next_`	Expected sequence number
`maxseen_`	Highest sequence number ever received
`wndmask_`	Modulus mask, initialized to maximum window size-1 (set to 63 by default; see Sect. 15.4.1)
`is_dup_`	True if the latest received TCP packet was received earlier

Figure 10.2 shows an example of information stored in an `Acker` object. In this case, Packets 1, 2, 3, 5, and 7 are received, but Packets 4 and 6 are missing. Therefore, "`next_`" and "`maxseen_`" are set to 4 and 7, respectively. Also, variable "`seen_`" stores the size in bytes of Packets 1–7 in its respective entries. To determine whether packet n is missing, class `Acker` checks the value of `seen_[n]`. The packet is missing if and only if `seen_[n]` is zero. Suppose a TCP receiver receives a TCP packet number 4 when the status of the `Acker` object is as in Fig. 10.2. The `Acker` object will generate an ACK packet with sequence number 5. However, if the sequence number of the received packet is not 4 (e.g., 7, 8, 9), the `Acker` object will create an ACK packet with sequence number 3.

[2]Class `Acker` is not implemented in the OTcl domain.

Program 10.5 The constructor of class `Acker`

```
    //~/ns/tcp/tcp-sink.cc
1   Acker::Acker() : next_(0), maxseen_(0), wndmask_(MWM)
2   {
3       seen_ = new int[MWS];
4       memset(seen_, 0, (sizeof(int) * (MWS)));
5   }

    //~/ns/tcp/tcp-sink.cc
6   #define MWS 64
7   #define MWM (MWS-1)
```

As discussed in Sect. 9.1.2, a TCP sender can transmit at most w unacknowledged packets in a network, where w is the current congestion window size. Let `MWS` be the Maximum Window Size in a simulation (see Line 6 in Program 10.5). Then, $w \in \{0, \cdots, \texttt{MWS-1}\}$ and there can be at most `MWS` unacknowledged packets during the entire simulation. An `Acker` object needs only `MWS` entries in the array "`seen_`" to store information about unacknowledged packets.

Program 10.5 shows the constructor of the C++ class `Acker`. The constructor resets "`next_`" and "`maxseen_`" to zero in Line 1. Line 3 allocates memory space for array variable "`seen_`" with `MWS` entries. Line 4 clears the allocated memory to zero. Also, "`wndmask_`" is set to `MWM` (Maximum Window Mask which is set to 63 in Line 7).

The above `MWS` (set by default to 64 in Line 6 of Program 10.5) entries of "`seen_`" are reused to store the packet size corresponding to all incoming TCP sequence numbers. Class `Acker` uses a modulus operation to map a sequence number to an array index. Upon receiving a TCP packet with sequence number "`seqno`," an `Acker` object stores the packet size in the entry `seqno % MWS` (which is the remainder of `seqno/MWS`), of the array "`seen_`," where "`%`" is a modulus operator. When "`seqno`" exceeds `MWS`, `seqno % MWS` will be restarted from the first entry (i.e, the index number "o") to reuse the memory allocated to "`seen_`."

As discussed in Sect. 15.4.1, a modulus operation can also be implemented by bit masking. In particular, `seqno % MWS` is in fact equivalent to `seqno & wndmask_`, where "`wndmask_`" is set initially to `MWM` in the constructor (Line 1 in Program 10.5), and `MWM` (Maximum Window Mask) is defined as 63 (Lines 6 and 7 in Program 10.5).[3] To facilitate the understanding, readers may consider the case where seqno is less than 64, where `seqno & wndmask_` is simply `seqno`.

Class `Acker` has two key functions: `Seqno()` and `update(seq, numBytes)`. Function `Seqno()` (Line 5 in Program 10.4) returns the highest sequence number of a burst of contiguously received packets. As shown in Program 10.6, function `update(seq,numBytes)` updates its internal variables according to the input arguments.

[3]While the possible value of `seqno` is in {0,..., 65535}, the possible value of `seqno % 64` (which is equal to `seqno & 63` is in {0,...,63}.

Program 10.6 Function `update` of class `Acker`

```
    //~/ns/tcp/tcp-sink.cc
 1  int Acker::update(int seq, int numBytes)
 2  {
 3      bool just_marked_as_seen = FALSE;
 4      is_dup_ = FALSE;
 5      int numToDeliver = 0;
 6      if (seq > maxseen_) {
 7          int i;
 8          for (i = maxseen_ + 1; i < seq; ++i)
 9              seen_[i & wndmask_] = 0;
10          maxseen_ = seq;
11          seen_[maxseen_ & wndmask_] = numBytes;
12          seen_[(maxseen_ + 1) & wndmask_] = 0;
13          just_marked_as_seen = TRUE;
14      }
15      int next = next_;
16      if (seq < next)
17          is_dup_ = TRUE;
18      if (seq >= next && seq <= maxseen_) {
19          if (seen_[seq & wndmask_] && !just_marked_as_seen)
20              is_dup_ = TRUE;
21          seen_[seq & wndmask_] = numBytes;
22          while (seen_[next & wndmask_]) {
23              numToDeliver += seen_[next & wndmask_];
24              ++next;
25          }
26          next_ = next;
27      }
28      return numToDeliver;
29  }
```

Function `update(seq,numBytes)` takes two input arguments: "`seq`" and "`numBytes`" which are the sequence number and the size of an incoming TCP packet, respectively. It updates variables "`next_`," "`maxseen_`," "`seen_`," and "`is_dup_`", and returns the number of in-sequence bytes which is ready to be delivered to the application. From Fig. 10.2, "`seq`" can be (I) less than "`next_`," (II) between "`next_`" and "`maxseen_`," and (III) greater than "`maxseen_`." Function `update(seq,numBytes)` reacts to these three cases as follows:

(i) If "`seq`" < "`next_`," function `update(seq,numBytes)` will set "`is_dup_`" to be `true` (Line 17). This case implies that this packet was received earlier, and therefore, this packet is a duplicate packet.

(ii) If "`seq`" lies in between "`next_`" and "`maxseen_`," function `update(seq, numBytes)` will execute Lines 19–26. Line 19 determines whether "`seq`" was received earlier. This happens to be true under the two following conditions: (1) the corresponding entry of "`seen_`" is nonzero and (2) `just_marked_as_seen` is `false`. The latter condition is added since if `seq` is a new sequence number, Line 11 would also have store the packet size is `seen_`. However, the variable `Just_marked_as_seen` would have been set as `true` by Line 13. For Case (ii), "`is_dup_`" is set to `true`.

Line 21 stores the packet size in `seen_[seq & wndmask_]`.[4] Lines 22–26 update "`next_`" by advancing "`next_`" until `seen_[next_ & wndmask_]` is empty. Also, Line 23 keeps adding the packet size to `numToDeliver`, which are returned in Line 28. Essentially, the returned value is the number of bytes which corresponds to "`next_`" advancement.

(iii) If `seq > maxseen_`, implying a new TCP packet, function `update(...)` will execute Lines 7–13. Lines 8–9 and 12 clear the `seen_[maxseen_+1]` through `seen_[seq-1]`. It updates "`maxseen_`" in Line 10 and stores the packet size in `seen_[seq & wndmask_]` in Line 11. Since Line 10 stores "`seq`" in "`maxseen_`," the condition in Line 18 is satisfied and Lines 19–26 are to be executed. If Case (iii) is executed, Case (ii) will also be executed. Therefore, Line 13 sets "`just_marked_as_seen`" to be `true`, which simply indicates that the current packet is not a duplicated packet, and Line 20 will be skipped.

10.2.2 Class `TcpSink`

Representing TCP receivers, class `TcpSink` reacts to received TCP packets as follows:

1. Extract the sequence number `(seq)` from the received TCP packet,
2. Inform the `Acker` object of the sequence number (`seq`) and the size of the TCP packet (`numBytes`) through function `update(seq,numBytes)` of class `Acker`,
3. Create and send an ACK packet to the TCP sender by invoking function `ack(p)` of class `TcpSink`. The sequence number in the ACK packet is obtained from function `Seqno()` of the `Acker` object (invoked from within function `ack(p)`).

Program 10.7 shows the declaration of a C++ class `Tcpsink`, which is bound to an OTcl class `Agent/TCPSink`. The only key variable of class `TcpSink` is a pointer to an `Acker` object, "`acker_`" in Line 8. Two main functions of class `TcpSink` include `recv(p,h)` and `ack(p)`.

Shown in Program 10.8, function `recv(p,h)` is invoked by an upstream object to hand a TCP packet over to a `TcpSink` object. Lines 4–6 inform the `Acker` object, "`acker_`," of an incoming TCP packet "`pkt`." Here, the sequence number (i.e., `th->seqno()`) and packet size (i.e., `numBytes`) are passed to "`acker_`" through this function. Again, function `update(seq,numBytes)` returns the number of in-order bytes which can be delivered to the application. If this number is nonzero, it will be delivered to the application through function `recvBytes(bytes)` in Line 8. Line 9 invokes function `ack(pkt)` to generate an ACK packet and send it to the TCP sender. Finally, Line 10 deallocates the received TCP packet.

[4] Bit masking with "`wndmask_`" has the same impact as a modulus with "`wndmask_+1`" does.

Program 10.7 Declaration of class `TcpSink`

```
    //~/ns/tcp/tcp-sink.h
1   class TcpSink : public Agent {
2   public:
3       TcpSink(Acker*);
4       void recv(Packet* pkt, Handler*);
5       int command(int argc, const char*const* argv);
6   protected:
7       void ack(Packet*);
8       Acker* acker_;
9   };
```

Program 10.8 Function `recv` of class `TcpSink`

```
    //~/ns/tcp/tcp-sink.cc
1   void TcpSink::recv(Packet* pkt, Handler*)
2   {
3       int numToDeliver;
4       int numBytes = hdr_cmn::access(pkt)->size();
5       hdr_tcp *th = hdr_tcp::access(pkt);
6       numToDeliver = acker_->update(th->seqno(), numBytes);
7       if (numToDeliver)
8           recvBytes(numToDeliver);
9       ack(pkt);
10      Packet::free(pkt);
11  }
```

Program 10.9 Function `ack` of class `TcpSink`

```
    //~/ns/tcp/tcp-sink.cc
1   void TcpSink::ack(Packet* opkt)
2   {
3       Packet* npkt = allocpkt();
4       hdr_tcp *otcp = hdr_tcp::access(opkt);
5       hdr_tcp *ntcp = hdr_tcp::access(npkt);
6       ntcp->seqno() = acker_->Seqno();
7       double now = Scheduler::instance().clock();
8       ntcp->ts() = now;
9       hdr_ip* oip = hdr_ip::access(opkt);
10      hdr_ip* nip = hdr_ip::access(npkt);
11      nip->flowid() = oip->flowid();
12      send(npkt, 0);
13  }
```

Program 10.9 shows the details of function `ack(p)`. In this function, variables whose name begins with "o" and "n" are used for an old packet and a new packet, respectively. Line 6 puts an ACK number in the ACK packet. Lines 7–8 and 9–11 configure timestamp and flow ID of the ACK packet, respectively.

Finally, the configured packet is sent out using function `send(npkt,0)` of class `Agent` in Line 12, where a new packet "`npkt`" is transmitted along with a Null handler.

10.3 TCP Sender

A TCP sender has the following four main responsibilities:

- *Packet transmission*: Based on user demand from an application, a TCP sender creates and forwards TCP packets to a TCP receiver.
- *ACK processing*: A TCP sender observes a received ACK pattern and determines whether transmitted packets were lost. If so, it will retransmit the lost packets. From the ACK pattern, it can also estimate the network condition (e.g., end-to-end bandwidth) and adjust the congestion window accordingly.
- *Timer-related mechanism*: A retransmission timer is used to provide connection reliability. Unless reset by an ACK packet arrival, the retransmission timer informs the TCP sender of packet loss after the packet has been transmitted for a period of *Retransmission TimeOut (RTO)*.
- *Window adjustment*: Based on the ACK pattern and timeout event, a TCP sender adjusts its congestion window to fully use the network resource and prevent network congestion.

The details of these four responsibilities will be discussed in the next four sections.

10.4 TCP Packet Transmission Functions

Class `TcpAgent` provides the following four main packet transmission functions:

- `sendmsg(nbytes)`: Send "`nbytes`" of application payload. When `nbytes=-1`, the payload is assumed to be infinite.
- `sendmuch(force,reason,maxburst)`: Send out a packet whose sequence number is "`t_seqno_`." Keep sending out packets as long as the congestion window allows and the total number of transmitted packets during a function invocation does not exceed "`maxburst`."
- `output(seqno,reason)`: Create and send a packet with a sequence number and a transmission reason as specified by "`seqno`" and "`reason`," respectively.
- `send_one()`: Send a TCP packet with a sequence number "`t_seqno_`."

Among the above functions, function `sendmsg(nbytes)` is the only `public` function derived from class `Agent`, while the other three functions are internal to

class TcpAgent. Again, function sendmsg(nbytes) is invoked by an application to inform a TcpAgent object of user demand. Function sendmsg(nbytes) does not directly send out packets. Rather, it computes the number of TCP packets required to hold "nbytes" of data payload, and increases variable "curseq_" by the computed value. In NS2, a TcpAgent object keeps transmitting TCP packets as long as the sequence number does not exceed "curseq_." Increasing "curseq_" is therefore equivalent to feeding data payload to a TcpAgent object.

Another important variable is "t_seqno_," which contains the default TCP sequence number. Unless otherwise specified, a TCP sender always transmits a TCP packet with the sequence number stored in "t_seqno_." Both the functions sendmuch(force,reason,maxburst) and send_one() use function output(t_seqno_,reason) to send out a TCP packet whose sequence number is "t_seqno_."

Function send_much(...) acts as a foundation for TCP packet transmission. In most cases, TCP agent first stores the sequence number of the packet to be transmitted in "t_seqno_." Then, it invokes the function send_much(...) to send TCP packets-starting with that with the sequence number "t_seqno_" as long as the transmission window permits. As we shall see in Program 10.11, each packet transmission is carried out using function output(t_seqno_, reason).

10.4.1 Function sendmsg(nbytes)

Function send_msg(nbytes) is the main data transmission interface function derived from class Agent. A user (e.g., application) informs a TCP sender of transmission demand through this function. Function sendmsg(nbytes) usually takes one input argument, "nbytes," which is the amount of application payload in bytes that a user needs to send. When the user has infinite demand, "nbytes" is specified as −1.

Program 10.10 shows the details of function sendmsg(nbytes). Lines 4–7 transform the input user demand to the number of TCP packets to be transmitted (i.e., "curseq_"). Line 8 starts data transmission by invoking function send_much(0,0,maxburst_). Note that Line 1 specifies the limit (i.e., TCP_MAXSEQ) on the number of TCP sequence numbers which can be created by a certain TCP sender. Again, if nbytes = -1, the TCP sender will be backlogged until "TCP_MAXSEQ" TCP packets are transmitted. If "nbytes" is greater than −1, Line 7 will compute the number of TCP packets (each with size "size_" bytes) which can accommodate "nbytes" of application payload.

Program 10.10 Function `sendmsg` of class `TcpAgent`

```
    //~/ns/tcp/tcp.h
1 #define TCP_MAXSEQ 1073741824

   //~/ns/tcp/tcp.cc
2 void TcpAgent::sendmsg(int nbytes, const char* /*flags*/)
3 {
4     if (nbytes == -1 && curseq_ <= TCP_MAXSEQ)
5         curseq_ = TCP_MAXSEQ;
6     else
7         curseq_ += (nbytes/size_ + (nbytes%size_ ? 1 : 0));
8     send_much(0, 0, maxburst_);
9 }
```

10.4.2 Function `send_much(force,reason,maxburst)`

There are three important points in regards to function `send_much(force, reason,maxburst)`. First, it creates and sends out as many packets as the current transmission window allows, but not greater than "`maxburst`" packets. Second, every packet is transmitted by executing `output(t_seqno_,reason)`. Finally, function `send_much(...)` always sends out a TCP packets with sequence number "`t_seqno_`."

Function `send_much(force,reason,maxburst)` takes three following input arguments, where a typical invocation of this function is `send_much(0,0, maxburst_)`:

- "`force`": This value is usually set to zero. When "`force`" = `1`, TCP sender will try to transmit data packets even if some conditions are not met.[5]
- "`reason`": This value specifies the reason for data transmission. For a normal transmission, "`reason`" is set to 0. Other possible values of "`reason`" are shown in Lines 1–4 in Program 10.11. This input argument is later placed in the field "`reason_`" of TCP packet header (i.e., `hdr_tcp::reason_`) and will be used for various purposes in simulation.
- "`maxburst`": The maximum number of packets that can be transmitted for each invocation of function `send_much(force,reason,maxburst)`.

Program 10.11 shows the details of function `send_much(force, reason, maxburst)`. Function `send_much(force,reason,maxburst)` first stores

[5]For example, a variable "`overhead_`" adds a certain delay time specified by a `DelSndTimer` object before data transmission. By default, TCP sender does not transmit when "`overhead_`" is nonzero. However, it can transmit packets immediately when `force = 1`. Note that we do not discuss the details of class `DelSndTimer` here. The readers may find the details of class `DelSndTimer` in files *~ns*/tcp/tcp.cc,h.

Program 10.11 Functions `send_much` and window() of class `TcpAgent`

```
    //~/ns/tcp/tcp.h
 1  #define TCP_REASON_TIMEOUT     0x01 //Timeout
 2  #define TCP_REASON_DUPACK      0x02 //Duplicated ACK
 3  #define TCP_REASON_RBP         0x03 //Rate Based Pacing
 4  #define TCP_REASON_PARTIALACK  0x04 //Partial ACK

    //~/ns/tcp/tcp.cc
 5  void TcpAgent::send_much(int force, int reason, int
     maxburst)
 6  {
 7      int win = window(), npackets = 0;
 8      while (t_seqno_ <= highest_ack_ + win && t_seqno_ <
         curseq_) {
 9          if (overhead_ == 0 || force ) {
10              output(t_seqno_, reason);
11              npackets++;
12              t_seqno_++;
13          }
14          win = window();
15          if (maxburst && npackets == maxburst)
16              break;
17      }
18  }

19  int TcpAgent::window()
20  {
21      return (cwnd_ < wnd_ ? (int)cwnd_ : (int)wnd_);
22  }
```

the current congestion window[6] in a variable "`win`" and sets the variable "`npackets`" to zero in Line 7. Then, Line 8 checks whether the TCP sender is allowed to send a TCP packet with sequence number "`t_seqno_`." If so, Line 10 will invoke function `output(t_seqno_,reason)` to send out a TCP packet. Again, a TCP sender is allowed to transmit a packet if the following two conditions are satisfied:

1. Congestion window allows packet transmission: Function `window()` in Line 7 returns the minimum of the current congestion window and the maximum window size. This minimum value is stored in the variable "`win`" in Line 7. Since the latest received ACK number is "`highest_ack_`," the TCP sender can transmit TCP packets with sequence numbers "`t_seqno_`" through "`highest_ack_+win`".

[6]From Lines 19–22 of Program 10.11, function `window()` returns the minimum of "`window_`" (the maximum window size) and "`cwnd_`" (the current congestion window size) as the current bounded congestion window.

2. TCP sender still has data to transmit: The sender will send TCP packets unit the sequence number reaches "curseq_." Specified in the user demand, "curseq_" is the highest TCP sequence number that the sender needs to transmit.

After a packet transmission, the default sequence number "t_seqno_" (Line 12) and the congestion window size "win" (Line 14) are updated. Lines 15–17 stop the transmission, if TCP sender has sent out "maxburst" packets. The above process repeats until the condition in Line 8 becomes false.

10.4.3 Function output(seqno,reason)

Taking two input arguments, function output(seqno,reason) creates a packet, sets the sequence number and the reason field of TCP header to the input arguments "seqno" and "reason," respectively, and forwards the packet to the low-level network using function send(p,h) of class Agent.

Programs 10.12 and 10.13 show the details of function output(seqno, reason), which consists of five main parts. First, Line 5 creates a packet "p" using function allocpkt() of class Agent. Second, Lines 6–26 configure common, TCP, and flag headers of the created packets. For the common packet header, function output(...) configures packet size in Lines 18–26. If "useHeaders_" is true, "tcpip_base_hdr_size_" (40 bytes by default) will be added to the packet size. Since an SYN packet (with seqno_=0 and syn_=1) contains no pay-load, its size is set to be "tcpip_base_hdr_size_" bytes (Line 22). The following TCP header fields are configured in Lines 6–12: sequence number, timestamp, timestamp echo, transmitting reason, and latest observed round trip time (RTT). Finally, function output(...) configures the congestion flag[7] in the flag header (Lines 13–16). This congestion flag is set to be true when TCP is trying to transmit a new packet under congestion, i.e., both of the following conditions in Line 13 are true:

1. Congestion has occurred: During network congestion, TCP sender closes the congestion window by invoking function slowdown(how), within which the variable "cong_action_" is set to true.
2. TCP sender is transmitting a new packet (is_retx = false): This flag set to true, when a *new* packet (not a *retransmitted* packets) is experiencing congestion.

[7]For example, a router in the network may drop packets marked with a *congestion action* flag to help relieve network congestion. However, dropping a retransmitted packet may lead to TCP connection reset. Therefore, a TCP sender does not mark retransmitted packets with a congestion action flag.

Program 10.12 Function `output` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
 1  void TcpAgent::output(int seqno, int reason)
 2  {
 3      int force_set_rtx_timer = 0;
 4      int is_retransmit = (seqno < maxseq_);
 5      Packet* p = allocpkt();
 6      hdr_tcp *tcph = hdr_tcp::access(p);
 7      tcph->seqno() = seqno;
 8      tcph->ts() = Scheduler::instance().clock();
 9      tcph->ts_echo() = ts_peer_;
10      tcph->reason() = reason;
11      tcph->last_rtt() = int(int(t_rtt_)*tcp_tick_*1000);
12      int databytes = hdr_cmn::access(p)->size();
13      if (cong_action_ && !is_retransmit) {
14          hdr_flags* hf = hdr_flags::access(p);
15          hf->cong_action() = TRUE;
16          cong_action_ = FALSE;
17      }
18      if (seqno == 0) {
19          if (syn_) {
20              databytes = 0;
21              curseq_ += 1;
22              hdr_cmn::access(p)->size() = tcpip_base_hdr_
                  size_;
23          }
24      } else if (useHeaders_ == true) {
25          hdr_cmn::access(p)->size() += headersize();
26      }
```

The third part of function `output(seqno,reason)` is used to send out the configured packet using function `send(p,h)` of class `Agent` in Line 27. The fourth part updates the relevant variables of the `TcpAgent` object in Lines 28–48. If the condition in Line 30 is `true`, TCP sender will no longer have data to transmit. In this case, Line 31 informs the application so by invoking function `idle()` of class `Agent`. Relevant variables to be updated are `ndatapack_`, `ndatabytes_`, `nrexmitpack_`, `nremitbytes_`, in Lines 28, 29, 42, and 43, respectively. The former two variables denote the data transmitted by the `TcpAgent` object in packets and bytes, while the latter two are those corresponding to the retransmitted packets only. Lines 33–39 update the related variables when "`seqno`" > "`maxseq_`." These variables include "`maxseq_`" and other RTT estimation variables. We will discuss about the RTT estimation later in Sect. 10.6.

The final part is to start the retransmission timer by invoking function `set_rtx_timer()` in Line 48. Note that each TCP sender has only one retransmission timer. Under a normal situation, the timer is started only when it is idle (i.e., its status is not `TIMER_PENDING`). However, it is also started when `highest_ack_ == maxseq_`, regardless of the timer's status (see Line S45–48).

Program 10.13 Function `output` of class `TcpAgent` (cont.)

```
27      send(p, 0);
28      ++ndatapack_;
29      ndatabytes_ += databytes;
30      if (seqno == curseq_ && seqno > maxseq_)
31          idle();
32      if (seqno > maxseq_) {
33          maxseq_ = seqno;
34          if (!rtt_active_) {
35              rtt_active_ = 1;
36              if (seqno > rtt_seq_) {
37                  rtt_seq_ = seqno;
38                  rtt_ts_ = Scheduler::instance().clock();
39              }
40          }
41      } else {
42          ++nrexmitpack_;
43          nrexmitbytes_ += databytes;
44      }
45      if (highest_ack_ == maxseq_)
46          force_set_rtx_timer = 1;
47      if (!(rtx_timer_.status() == TIMER_PENDING)
                                  || force_set_rtx_timer)
48          set_rtx_timer();
49  }
```

Program 10.14 Function `send_one` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
1  void TcpAgent::send_one()
2  {
3      if (t_seqno_ <= highest_ack_ + wnd_ && t_seqno_ < curseq_
           && t_seqno_ <= highest_ack_ + cwnd_ +
              dupacks_ ) {
4          output(t_seqno_, 0);
5          t_seqno_ ++ ;
6      }
7  }
```

10.4.4 Function `send_one()`

Figure 10.4 shows the details of function `send_one()`. Function `send_one()` is very similar to function `send_much(...)`. It prepares sequence numbers starting at "`t_seqno_`" and passes them to function `output(t_seqno_, 0)` for packet creation and transmission. The main difference is that while function `send_much(...)` may send out several packets, function `send_one(...)` sends out only one packet. Function `send_one(...)` is designed to send a new packet during a fast retransmit phase for every received duplicated ACK packet

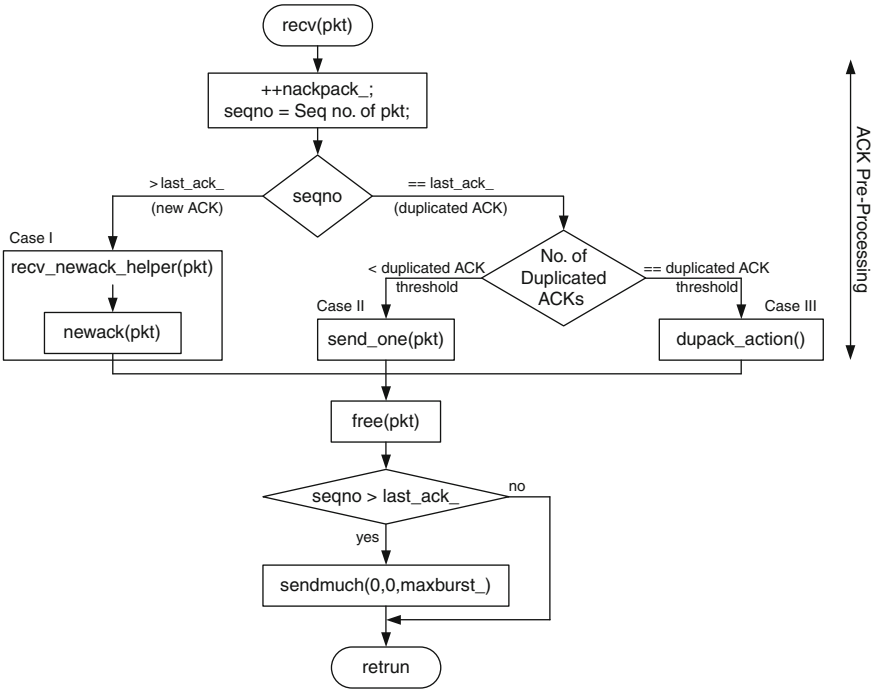

Fig. 10.3 Function `recv(p,h)` of class `TcpAgent`

(see Sect. 10.5). In this case, Line 3 inflates the congestion windows by the number of duplicated ACK (`dupacks_`). As will be discussed in Sect. 10.5, this function is invoked if the option "`singledup_`" is set to 1 during the reception of the first duplicated ACK packets.

10.5 ACK Processing Functions

The second responsibility of a TCP sender is to process ACK packets. An ACK packet could be a new ACK packet or a duplicated ACK packet. A new ACK packet slides the congestion window to the right and opens the congestion window to allow the TCP sender to transmit more packets. A duplicated ACK packet, on the other hand, indicates out-of-order packet delivery or packet loss (see Fig. 9.2, for example). Again, TCP Tahoe assumes that packet loss upon detecting the "`numdupacks_`"th (3rd by default) duplicated ACK packet. It sets the slow-start threshold to half of the current congestion window, sets the congestion window size to "`wnd_init_`" (which is usually set to 1), and retransmits the lost packet. During a Fast Retransmit phase, the TCP sender transmits a new packet for every

received duplicated ACK packet (due to inflated congestion window). When a new ACK packet is received, the TCP sender sets its congestion window to the same as slow-start threshold, and returns to its normal operation.

Class `TcpAgent` provides the four following key ACK Processing functions:

- `recv(p,h)`: This is the main ACK reception function. It determines whether the received packet (`*p`) is a new ACK packet or a duplicated ACK packet, and acts accordingly.
- `recv_newack_helper(p)`: This function is invoked from within function `recv(p,h)` when a new ACK packet is received. It invokes function `newack(p)` to update relevant variables, and opens the congestion window if necessary.
- `newack(p)`: Invoked from within function `recv_newack_helper(p)`, this function updates variables related to sequence number, ACK number, and RTT estimation process, and restarts the retransmission timer.
- `dupack_action()`: This function is invoked from within function `recv(p,h)` when a duplicated ACK packet is received and Fast Retransmit process is launched. It cuts down the congestion window, prepares the sequence number of the lost packet for retransmission, and resets the retransmission timer.

10.5.1 Function `recv(p,h)`

Figure 10.3 and Program 10.15 show the diagram and implementation, respectively, for function `recv(p,h)`. Function `recv(p,h)` pre-processes the received ACK packets in Lines 6–14, where "`t_seqno_`" and "`cwnd_`" are adjusted. Depending of the received ACK type (i.e., new or duplicated), Lines 6–14 (ACK pre-processing) process an ACK packet according to the following three cases:

- **Case I (New ACK)**: If a new ACK packet is received (i.e., Line 6 returns `true`), Line 7 will invoke function `recv_newack_helper(p)` to adjust congestion window (`cwnd_`) and prepare a new sequence number (`t_seqno_`) for packet transmission.
- **Case II (Duplicated ACK)**: In this case, a duplicated ACK packet is received (i.e., Line 6 returns `false`) but the number of duplicated ACK packets received so far has not reached "`numdupacks_`" (i.e., Line 9 returns `false`). Line 12 will invoke function `send_one()` to transmit new TCP packets under the congestion window inflated by the number of received duplicated ACK packets (see the definition of Fast Recovery in sect. 9.1.2.2). Note that variable "`singledup_`" is an NS2 option for congestion window inflation. The above actions are executed when "`singledup_`" is `true` only. If "`singledup_`" is `false`, the TCP sender will not send a new packet for every received ACK packet.

Program 10.15 Function `recv` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
 1  void TcpAgent::recv(Packet *pkt, Handler*)
 2  {
 3      hdr_tcp *tcph = hdr_tcp::access(pkt);
 4      int valid_ack = 0;
 5      ++nackpack_;
 6      if (tcph->seqno() > last_ack_) {
 7          recv_newack_helper(pkt);
 8      } else if (tcph->seqno() == last_ack_) {
 9          if (++dupacks_ == numdupacks_ && !noFastRetrans_) {
10              dupack_action();
11          } else if (dupacks_ < numdupacks_ && singledup_ ) {
12              send_one();
13          }
14      }
15      if (tcph->seqno() >= last_ack_)
16          valid_ack = 1;
17      Packet::free(pkt);
18      if (valid_ack)
19          send_much(0, 0, maxburst_);
20  }
```

- **Case III (Fast retransmit)**: If the received ACK is the last (i.e., "`numdupacks_`"th) duplicated ACK packet, the TCP sender will enter the Fast Retransmit phase, by invoking function `dupack_action()` (Line 10). Note that an option Flag "`noFastRetrans_`" is an NS option for a Fast Retransmit phase. The TCP sender will not enter a Fast Retransmit phase, if "`noFastRetrans_`" is `true`.

After executing one of the above three cases, Line 17 deallocates the ACK packet `*pkt` by executing `free(pkt)`. If the received ACK is valid (i.e., `valid_ack_=1`), Line 19 will create and transmit TCP packets using function `send_much(0,0,maxburst_)`. Here a received ACK packet is said to be valid if it is a new ACK packet (i.e., `tcph->seqno() > last_ack_`) or a duplicated ACK (i.e., `tcph->seqno() = last_ack_`). If an ACK packet is invalid, a TCP sender will only destroy the ACK packet, but will not create and forward new packets.

10.5.2 Function `recv_newack_helper(pkt)`

Function `recv_newack_helper(pkt)` is a helper function invoked when a new ACK packet is received. As shown in Program 10.16, the function `recv_newack_helper(pkt)` first invokes function `newack(pkt)` in Line 2 to update relevant variables and to process the retransmission timer. When Explicit Congestion Notification (ECN) is not enabled (i.e., by default ECT (ECN Capable

Program 10.16 Function `recv_newack_helper` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
1   void TcpAgent::recv_newack_helper(Packet *pkt) {
2       newack(pkt);
3       if (!ect_) {
4           if (!control_increase_ ||
            (control_increase_ && (network_limited() == 1)))
5                   opencwnd();
6       }
7       if ((highest_ack_ >= curseq_-1) && !closed_) {
8           closed_ = 1;
9           finish();
10      }
11  }
```

Program 10.17 Function `network_limited` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
1  int TcpAgent::network_limited() {
2      int win = window () ;
3      if (t_seqno_ > (prev_highest_ack_ + win))
4          return 1;
5      else
6          return 0;
7  }
```

Transport System) is set to zero), Line 5 will open the congestion window (by invoking function `opencwnd()`) when at least one of the following conditions is true (Line 4):

- "`control_increase_`" = 0: Variable "`control_increase_`," when set to 1, suppresses the congestion window opening. When "`control_increase_`" is zero, a TCP sender can freely increase the congestion window.
- "`control_increase_`" ≠ 0 but the network is limited: When "`control_increase_`" is 1, the TCP sender is allowed to open the congestion window only when the previous congestion window is not sufficient to transmit the current packet (i.e., the network is limited). In NS2, a network is said to be limited when "`t_seqno_`" is less than `prev_highest_ack_ + win`, where "`prev_highest_ack_`" is the ACK number before the reception of the current ACK packet and "`win`" is the current congestion window (see Program 10.17). In this case, it is necessary to open the congestion window, in order to transmit a new packet. Note that if the TCP sender stops transmission due to any reason other than the reason that the network is limited, function `recv_newack_helper(pkt)` will not open the congestion window.

Finally, if the TCP sender no longer has data to transmit, Line 8 in Program 10.16 will close the connection by setting "`closed_`" to 1, and Line 9 will invoke function `finish()`.

Program 10.18 Function `newack` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
 1  void TcpAgent::newack(Packet* pkt)
 2  {
 3      double now = Scheduler::instance().clock();
 4      hdr_tcp *tcph = hdr_tcp::access(pkt);
 5      dupacks_ = 0;
 6      last_ack_ = tcph->seqno();
 7      prev_highest_ack_ = highest_ack_ ;
 8      highest_ack_ = last_ack_;
 9      if (t_seqno_ < last_ack_ + 1)
10          t_seqno_ = last_ack_ + 1;
11      hdr_flags *fh = hdr_flags::access(pkt);
12      if (rtt_active_ && tcph->seqno() >= rtt_seq_) {
13          if (!ect_) {
14              t_backoff_ = 1;
15              ecn_backoff_ = 0;
16          }
17          rtt_active_ = 0;
18          rtt_update(now - rtt_ts_);
19      }
20      newtimer(pkt);
21  }
```

10.5.3 Function `newack(pkt)`

Program 10.18 shows the details of function `newack(pkt)`. Lines 5–10 update variables `dupack_`, `last_ack_`, `prev_highest_ack_`, `highest_ack_`, and `t_seqno_`. Lines 12–19 update RTT estimation variables and timeout backoff value. Finally, Line 20 starts a retransmission timer for the transmitting packet. We will discuss the details of RTT estimation and retransmission timer later in Sect. 10.6.

10.5.4 Function `dupack_action()`

The main responsibilities of function `dupack_action()` are to: (1) decrease congestion window size, (2) set "`t_seqno_`" to the sequence number of the lost TCP packet, and (3) restart retransmission timer. Program 10.19 shows the details of function `dupack_action()`. Line 5 registers fast retransmission event (i.e., `FAST_RETX`) for tracing. Line 6 records `CWND_ACTION_DUPACK` as the latest window adjustment action (i.e., "`last_cwnd_action_`"). Line 7 closes the congestion window by invoking function `slowdown(CLOSE_SSTHRESH_HALF | CLOSE_CWND_ONE)`, feeding how the slow-start threshold and congestion window are to be configured as an input argument. Finally, Line 8 invokes function `reset_rtx_timer(0,0)` to set "`t_seqno_`" to `highest_ack_+1`,

Program 10.19 Function `dupack_action` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
1   void TcpAgent::dupack_action()
2   {
3       if (highest_ack_ > recover_) {
4       recover_ = maxseq_;
5       trace_event("FAST_RETX");
6       last_cwnd_action_ = CWND_ACTION_DUPACK;
7       slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_ONE);
8       reset_rtx_timer(0,0);
9       return;
10  }
```

and restarts the retransmission timer. The details of functions `reset_rtx_timer(...)` and `slowdown(...)` will be discussed in Sects. 10.6 and 10.7, respectively.

TCP Tahoe reacts to a duplicated ACK packet differently. Lines 4–9 in Program 10.19 are executed only when all the packets transmitted during this packet loss have been acknowledged. Here, variable "`recover_`" records the highest TCP sequence number (i.e., "`maxseq_`") transmitted during this packet loss event. Line 4 sets "`recover_`" to be "`maxseq_`" so that it can be used in the next packet loss event. The condition in Line 3, `highest_ack_ > recover_`, implies that the TCP packet with highest sequence number transmitted during this previous loss must be acknowledged. If this condition is not satisfied, the TCP sender will wait for timeout and retransmit the lost packet.

10.6 Timer-Related Functions

Another responsibility of a TCP sender is to use a retransmission timer to provide connection reliability. The main components of this part include estimation of smoothed RTT (round trip time) and RTT variation, computation of RTO (retransmission timeout), implementation of BEB (binary exponential backoff), utilization of a retransmission timer, and defining actions to be performed at timeout.

10.6.1 RTT Sample Collection

A TCP sender needs to collect RTT samples to estimate smoothed RTT and RTT variation, and to compute retransmission timeout (RTO) value. An RTT sample is measured as the time difference between the point where a packet is transmitted and the point where the associated ACK packet arrives at the sender.

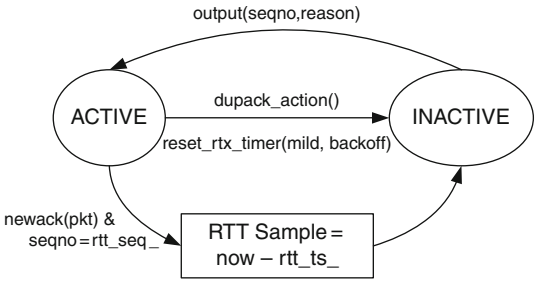

Fig. 10.4 The RTT sampling process

In NS2, each TCP sender has only one set of variables including variables `rtt_active_`, `rtt_ts_`, and `rtt_seq_` (see Table 10.1) to track RTT samples. It can collect only one RTT sample at a time – meaning not all the packets are used to collect RTT samples.

Figure 10.4 shows the diagram of the RTT collection process. The process starts in the inactive state where `rtt_active_=0`. The collection is activated (i.e., the process enters the active state) when a TCP sender sends out a new packet using function `output(seqno,reason)`. From Program 10.13, Line 35 sets "`rtt_active_`" to be 1.[8] Lines 37 and 38 record the TCP sequence number and the current time in the variables "`rtt_seq_`" and `rtt_ts_`, respectively.

An RTT sample is collected when the associated ACK packet returns (see Lines 12–19 of function `newack(pkt)` in Program 10.18). Given that the collection process is active (i.e., `rtt_active_=1`), Line 12 determines whether the incoming ACK packet belongs to the same collecting sample. It is so if the sequence number in the received ACK packet is the same as that stored in "`rtt_seq_`" (set at the beginning of the collecting process). Note that the logical relation here is "`>=`" rather than "`==`," since some TCP variants may not generate an ACK packet for every received TCP packet. At the end of the collection process, Line 17 sets "`rtt_active_`" to zero indicating that the collecting process has completed (i.e., the process moves back to inactive state), and Line 18 takes an RTT sample by invoking `rtt_update(now-rtt_ts_)` (defined in Program 10.22).

The above RTT collection process operates fairly well under normal situations. However, a packet loss may inflate an RTT sample and affect the accuracy of RTO collection process. In this case, the measured RTT would be the RTT value plus the time used to retransmit the lost packets. To keep it simple, NS2 simply cancels the RTT collection process when a packet loss occurs. In particular, functions `dupack_action()` (Line 8 in Program 10.19) and `timeout(tno)` (Lines 14 and 16 in Program 10.26) invoke function `reset_rtx_timer(...)` to set "`rtt_active_`" to zero, essentially cancelling the RTT collection process.

[8] If the "`rtt_active_`" is nonzero, TCP sender will skip the collection process.

10.6.2 RTT Estimation

After collecting an RTT sample, a TCP sender feeds the sample "`tao`" to function `rtt_update(tao)` to estimate smoothed RTT (`t_srtt_`), RTT variation (`t_rttvar_`), and unbounded RTO (`t_rtxcur_`)[9] based on (9.1)–(9.3), where $\alpha = 7/8$, $\beta = 3/4$, and $\gamma = 1$. Instead of directly computing these three variables, NS2 manipulates (9.1)–(9.3) such that each term in these equations is multiplied with 2^n, where n is an integer. As discussed in Sect. 15.4.2, multiplication and division by 2^n can be implemented in C++ by shifting a binary value to the left and right, respectively, by n bits. This bit shifting technique is used in function `rtt_update(tao)` to compute "`t_srtt_`," "`t_rttvar_`," and "`t_rtxcur_`."

At time k, let $t(k)$ be the RTT sample, $\bar{t}(k)$ be the smoothed RTT value, $\sigma_t(k)$ be the RTT variation, and Δ refer to $t(k+1) - \bar{t}(k)$. From (9.1) to (9.3),

$$\begin{aligned}
\bar{t}(k+1) &= \frac{1}{8}\left(7\bar{t}(k) + t(k+1)\right) \\
&= \frac{1}{8}\left(7\bar{t}(k) + \bar{t}(k) + t(k+1) - \bar{t}(k)\right) \\
&= \frac{1}{8}\left(8\bar{t}(k) + \Delta\right) \qquad (10.1) \\
\sigma_t(k+1) &= \frac{1}{4}\left(3\sigma_t(k) + |\Delta|\right) \\
&= \frac{1}{4}\left(3\sigma_t(k) - 4\sigma_t(k) + 4\sigma_t(k) + |\Delta|\right) \\
&= \frac{1}{4}\left(-\sigma_t(k) + 4\sigma_t(k) + |\Delta|\right) \qquad (10.2) \\
\mathrm{RTO}_u(k+1) &= \gamma \times [t(k+1) + 4\sigma_t(k+1)] \qquad (10.3)
\end{aligned}$$

where $\mathrm{RTO}_u(k+1)$ is an unbounded RTO. Equations (10.1)–(10.3) are now rearranged so that all the multiplicative fact are $2^n, n = \{0, 2, 3\}$ (i.e., the multiple of 1, 4, and 8). NS2 uses bit shifting operation in place of multiplication to implement (10.1)–(10.3).

[9] An actual value of RTO must be bounded by a minimum value and a maximum value.

Program 10.20 Function `rtt_init()` of class `TcpAgent`, and default values for the timer-related variables

```
    //~/ns/tcp/tcp.cc
1   void TcpAgent::rtt_init()
2   {
3       t_rtt_ = 0;
4       t_srtt_ = int(srtt_init_ / tcp_tick_) << T_SRTT_BITS;
5       t_rttvar_ = int(rttvar_init_ / tcp_tick_) << T_RTTVAR_
          BITS;
6       t_rtxcur_ = rtxcur_init_;
7       t_backoff_ = 1;
8   }

    //~/ns/tcl/lib/ns-default.tcl
9   Agent/TCP set T_SRTT_BITS 3     #in bits
10  Agent/TCP set T_RTTVAR_BITS 2   #in bits
11  Agent/TCP set srtt_init_ 0      #in seconds
12  Agent/TCP set rttvar_init_ 12   #in seconds
13  Agent/TCP set rtxcur_init_ 3.0  #in seconds
14  Agent/TCP set T_SRTT_BITS 3     #in bits
15  Agent/TCP set T_RTTVAR_BITS 2   #in bits
16  Agent/TCP set rttvar_exp_ 2     #in bits
17  Agent/TCP set tcp_tick_ 0.1     #in seconds
18  Agent/TCP set maxrto_ 100000    #in seconds
19  Agent/TCP set minrto_ 0.2       #in seconds
```

10.6.3 Overview of State Variables

State of variables contain the current status of a TCP agent. Related timer state variables are shown in Tables 10.1 and 10.3. Most of the variables are well explained by their descriptions. We now discuss a few points related to these variables.

First, C++ timer variables are initialized in function `rtt_init()` (Lines 1–8 in Program 10.20). OTcl timer instvars, on the other hand, are initialized in file *~ns*/tcl/lib/ns-default.tcl shown in Lines 9–19 of Program 10.20.

Second, "`tcp_tick_`" is a simulation time unit (i.e., granularity) in seconds. Hereafter, we will refer to a simulation time unit as a "tick." The default value of "`tcp_tick_`" is 100 ms. In other words, one "tick" is set by default to 0.1 (see Line 17 in Program 10.20).

Third, "`t_backoff_`" is used as a binary exponential backoff factor (i.e., γ in (10.3)). A TCP sender doubles its retransmission timer for every timeout event. In NS2, a TCP sender doubles "`t_backoff_`" for every timeout event and computes the unbounded RTO as "`t_rtxcur_ * t_backoff_`" (see Line 7 in Program 10.23).

Finally, there are two main points related to variables "`t_srtt_`" and "`t_rttvar_`." One is that these variables are stored in "ticks," rather than seconds. However, their initial values are in seconds. Lines 4 and 5 in Program 10.20 divide

Program 10.21 Class `RtxTimer` and related components

```
    //~/ns/tcp/tcp.h
1   class RtxTimer : public TimerHandler {
2   public:
3       RtxTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
4   protected:
5       virtual void expire(Event *e);
6       TcpAgent *a_;
7   };

    //~/ns/tcp/tcp.cc
8   void RtxTimer::expire(Event*)
9   {
10      a_->timeout(TCP_TIMER_RTX);
11  }

12  void TcpAgent::set_rtx_timer()
13  {
14      rtx_timer_.resched(rtt_timeout());
15  }
```

the initial values of smoothed RTT and RTT variation by "`tcp_tick_`" to obtain the time in "ticks" (rather than in seconds). Another point is about the division operation (by 8 and 4, respectively). To avoid round-off error during a division, these two variables are multiplied by 8 and 4, respectively, at the initialization. Again, Lines 4 and 5 in Program 10.20 shift "`t_srtt_`" and "`t_rttvar_`" to the left by `T_SRTT_BITS=3` bits and `T_RTTVAR_BITS = 2` bits, respectively. This bit shifting is equivalent to multiplying 8 and 4 to "`t_srtt_`" and "`t_rttvar_`," respectively.

10.6.4 Retransmission Timer

A TCP sender uses a retransmission timer to provide end-to-end reliability. When transmitting a packet, it starts a retransmission timer. Upon the timer expiration, the timer informs the TCP sender of a packet timeout. Here the TCP sender assumes that the packet is lost and retransmits the lost packet. If an ACK packet is received before the timeout, the timer will be stopped (i.e., cancelled). The details of NS2 timer implementation is given in Sect. 15.1.

NS2 models retransmission timers using a C++ class `RtxTimer` shown in Program 10.21. Derived from class `TimerHandler`, class `RtxTimer` has one variable "`a_`" which is a pointer to a `TcpAgent` object. It derives three main functions: `sched(delay)`, `resched(delay)`, and `cancel()`. It overrides one function `expire(e)` of class `TimerHandler`. Function `sched(delay)` starts the timer and sets the timer to expire at "`delay`" seconds in future. Function

`cancel()` stops the pending timer. Function `resched(delay)` restarts the timer and again sets the timer to expire at "`delay`" seconds in future. Finally, function `expire(e)` defines a set of actions which are taken at the timer expiration.

NS2 creates a two-way connection between `TcpAgent` and `RtxTimer` objects using the following mechanism. First, class `TcpAgent` declares an `RtxTimer` object (`rtx_timer_` in Line 4 in Program 10.3) as its member variable. Every `TcpAgent` object therefore has a direct access to an `RtxTimer` object. Second, on the reverse direction, class `RtxTimer` declares a pointer "`a_`" to a `TcpAgent` object in Line 6 of Program 10.21 as its member variable. Finally, a `TcpAgent` object is specified as a target of the pointer "`a_`" in the constructor of the `RtxTimer` object. From Line 7 in Program 10.3, the constructor of class `TcpAgent` creates a "`rtx_timer_`" by feeding "`this`" (i.e., a pointer to itself) as an input argument. From Line 3 in Program 10.21, the constructor of "`rtx_timer_`" stores "`this`" in its variable "`a_`," creating a connection from the "`rtx_timer_`" back to the `TcpAgent` object.

Note that in Line 4 in Program 10.3, a TCP sender has only one retransmission timer. Therefore, the TCP timeout mechanism applies to only one packet at a time. The retransmission timer is started when a new packet is transmitted (by function `output(...)`; see Line 48 in Program 10.13). Here, the timer is not allowed to start if it is in use (i.e., its status is `TIMER_PENDING`). This is in contrast to the actual TCP implementation where retransmission timers are set for all transmitted packets.

10.6.5 Function Overview

Class `TcpAgent` provides the following seven key timer-related functions:

- `rtt_update(tao)`: Takes an RTT sample "`tao`" as an input argument, updates smoothed RTT (`t_srtt_`), RTT variation (`t_rttvar_`), and unbounded RTO (`t_rtxcur_`) according to (10.1), (10.2), and (10.3), respectively.
- `rtt_timeout()`: Computes the bounded RTO value based on `t_rtxcur_`, `minrto_`, and `maxrto_`, as well as TCP binary exponential backoff (BEB) mechanism which make use of the current value of `t_backoff_`.
- `rtt_backoff()`: Double the binary exponential backoff multiplicative factor `t_backoff_`.
- `set_rtx_timer()`: Restart the retransmission timer.
- `reset_rtx_timer(mild,backoff)`: Restart the retransmission timer and cancel the RTT sample collecting process. If "`mild`" is zero, set `t_seqno_` to `highest_ack_+1`. Also, invoke function `rtt_backoff()` if "`backoff`" is nonzero.

- `newtimer(pkt)`: Take an ACK packet "`pkt`" as an input argument. Start the retransmission timer if TCP connection is active.[10] Cancel the timer, otherwise.
- `timeout(tno)`: If the connection is active, close the congestion window, adjust "`t_backoff_`," retransmit the lost packet, and restart the retransmission timer. Otherwise, restart the retransmission timer (but does not perform other action).[11]

10.6.6 Function `rtt_update(tao)`

Function `rtt_update(tao)` updates three main timer variables: smoothed RTT (`t_srtt_`), RTT variation (`t_rttvar_`), and Retransmission TimeOut (RTO; `t_rtxcur_`). Shown in Program 10.22, function `rtt_update(tao)` takes an RTT sample as an input argument. It is invoked from within function `newack(pkt)`, when a new ACK packet is received and a new RTT sample is `now - rtt_ts_` (see Line 18 in Program 10.18).

Function `rtt_update(tao)` aligns the input argument "`tao`" with "`tcp_tick_`" and stores the aligned valued in variable "`t_rtt_`" as the latest RTT sample (Lines 4–6). Before proceeding further, let us define the following variables

$$\bar{t} = \frac{\texttt{t\_srtt\_}}{8} = \texttt{t\_srtt\_>>T\_SRTT\_BITS} \tag{10.4}$$

$$\sigma_t = \frac{\texttt{t\_rttvar\_}}{4} = \texttt{t\_rttvar\_>>T\_RTTVAR\_BITS} \tag{10.5}$$

$$\Delta = \texttt{t\_rtt\_} - \bar{t} = \texttt{t\_rtt\_} - (\texttt{t\_srtt\_>>T\_SRTT\_BITS}) \tag{10.6}$$

where `T_SRTT_BITS` and `T_RTTVAR_BITS`, are defined in Program 10.20 as 3 and 2, respectively. Again, variables "`t_srtt_`" and "`t_rttvar_`" are stored in multiples of 8 and 4 (see Lines 4 and 5 in Program 10.20). Therefore, their relationship to actual smoothed RTT ($\bar{t}$) and RTT variation (σ_t) is given by (10.4) and (10.5), respectively.

Based on the above variables, Lines 8–15 compute the smoothed RTT value. In (10.1) and (10.2), we rearrange the variables "`t_srtt_`" and "`t_rttvar_`" as follows:

[10] A TCP connection is said to be active and idle when it has data to transmit and does not have data to transmit, respectively.

[11] As we will see, a retransmission timer does not stop when a TCP connection becomes idle. At the expiration, a TCP sender does nothing but restarts the timer. By keeping the timer running, the timer will be available as soon as the TCP sender becomes active.

Program 10.22 Function `rtt_update` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
 1  void TcpAgent::rtt_update(double tao)
 2  {
 3      double now = Scheduler::instance().clock();
 4      double tickoff = fmod(now-tao+boot_time_, tcp_tick_);
 5      if ((t_rtt_ = int((tao + tickoff) / tcp_tick_)<1);
 6          t_rtt_ = 1;
 7      if (t_srtt_ != 0) {
 8          register short delta = t_rtt_ - (t_srtt_ >>
              T_SRTT_BITS);
 9          if ((t_srtt_ += delta) <= 0)
10              t_srtt_ = 1;
11          if (delta < 0)
12              delta = -delta;
13          delta -= (t_rttvar_ >> T_RTTVAR_BITS);
14          if ((t_rttvar_ += delta) <= 0)
15              t_rttvar_ = 1;
16      } else {
17          t_srtt_ = t_rtt_ << T_SRTT_BITS;
18          t_rttvar_ = t_rtt_ << (T_RTTVAR_BITS-1);
19      }
20      t_rtxcur_ = (((t_rttvar_ << (rttvar_exp_ + (T_SRTT_
            BITS - T_RTTVAR_BITS))) + t_srtt_)  >> T_SRTT_BITS )
              * tcp_tick_;
21      return;
22  }
```

$$\texttt{t\_srtt\_}(k+1) = 8\bar{t}(k+1) = 8\bar{t}(k) + \Delta(k) = \texttt{t\_srtt\_}(k) + \Delta(k) \tag{10.7}$$

$$\begin{aligned}\texttt{t\_rttvar\_}(k+1) &= 4\sigma_t(k+1) = |\Delta| - \sigma_t(k) + 4\sigma_t(k) \\ &= |\Delta| - [\texttt{t\_rttvar\_(k)} >> \texttt{T\_SRTT\_BITS}] \\ &\quad + \texttt{t\_rttvar\_}(k).\end{aligned} \tag{10.8}$$

In Program 10.22, Line 8 computes "`delta`" (i.e., Δ) as indicated in (10.6). Line 9 updates "`t_srtt_`" according to (10.7) and Lines 11 and 12 compute $|\Delta|$. Lines 13 and 14 update "`t_rttvar_`" according to (10.8). From Lines 9–10 and 14–15, both "`t_srtt_`" and "`t_rttvar_`" will be set to 1, if their updated values are less than zero. Also, Lines 8–15 are invoked when "`t_srtt_`" is nonzero only. When "`t_srtt_`" is zero, "`t_srtt_`" and "`t_rttvar_`" are simply set to 8 times (Line 17) and twice of (Line 18) the RTT sample (i.e., "`t_rtt_`"), respectively.

NS2 computes (using (10.3)) and stores the unbounded value of RTO in variable "`t_rtxcur_`" (Line 20). It is computed as $\bar{t} + 4\sigma_t$ shown in (9.3). The upper-bound and the lower-bound in (9.3) will be implemented when an unbounded RTO

is assigned to the retransmission timer (e.g., in function `rtt_timeout()`). The computation of "`t_rtxcur_`" in Line 20 consists of four steps:

(i) Scale "`t_rttvar_`": Variables "`t_srtt_`" and "`t_rttvar_`" are stored as multiples of $2^{\text{T\_SRTT\_BITS}} = 8$ and $2^{\text{T\_RTTVAR\_BITS}} = 4$, respectively. Line 20 converts the scale of "`t_rtt_var_`" into the same scale of "`t_srtt_`" as follows:

$$\begin{aligned} \texttt{t\_rttvar\_} &\rightarrow \texttt{t\_rttvar\_} \times \frac{8}{4} \\ &= \texttt{t\_rttvar\_>>T\_RTTVAR\_BITS<<T\_SRTT\_BITS} \\ &= \texttt{t\_rttvar\_<<(T\_SRTT\_BITS} - \texttt{T\_RTTVAR\_BITS)} \end{aligned}$$

(ii) Multiply $2^{\texttt{rttvar\_exp\_}} = 2^2 = 4$ to the value obtained from Step (i). Denote the result from Step (i) as "`t_rtt_var_`"$^{(1)}$. See the default value of `rttvar_exp_` in Line 16 of program 10.20.

$$\begin{aligned} \texttt{t\_rttvar\_}^{(1)} &\rightarrow 4 \times \texttt{t\_rttvar\_}^{(1)} \\ &= \texttt{t\_rttvar\_}^{(1)}\texttt{<<rttvar\_exp\_} \end{aligned}$$

(iii) Denote the value computed in Step (ii) be `t_rtt_var_`$^{(2)}$. Add `t_srtt_` to `t_rtt_var_`$^{(2)}$.

$$\texttt{t\_rttvar\_}^{(2)} \rightarrow \texttt{t\_rttvar\_}^{(2)} + \texttt{t\_srtt\_}$$

(iv) Convert the computed value to seconds: Let `t_rtt_var_`$^{(3)}$ be the value computed in Step (iii). This value is stored in ticks and is in the scale of `t_srtt_` (i.e., multiple of 8). To change the unit of `t_rtt_var_`$^{(3)}$ to seconds,

$$\texttt{t\_rttvar\_}^{(3)} \rightarrow \texttt{t\_rttvar\_}^{(3)}\texttt{>>T\_SRTT\_BITS*tcp\_tick\_}$$

which is equivalent to Line 20 in Program 10.22.

10.6.7 Function `rtt_timeout()`

Shown in Program 10.23, function `rtt_timeout()` computes the bounded RTO, based on unbounded RTO (`t_rtxcur_`), RTO lower bound (`minrto_`), RTO upper bound (`maxrto_`), and TCP binary exponential backoff (BEB) mechanism. NS2 implements the BEB mechanism using a multiplicative factor "`t_backoff_`." The timeout value, which is used by the retransmission timer, is a product of "`t_rtxcur_`" and "`t_backoff_`" (see Line 7). The lower bound and the upper-bound are implemented in Lines 4–5 and Lines 8–9, respectively. Note that while

Program 10.23 Functions `rtt_timeout` and `rtt_backoff` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
1   double TcpAgent::rtt_timeout()
2   {
3       double timeout;
4       if (t_rtxcur_ < minrto_)
5           timeout = minrto_ * t_backoff_;
6       else
7           timeout = t_rtxcur_ * t_backoff_;
8       if (timeout > maxrto_)
9           timeout = maxrto_;
10      if (timeout < 2.0 * tcp_tick_)
11          timeout = 2.0 * tcp_tick_;
12      return (timeout);
13  }

14  void TcpAgent::rtt_backoff()
15  {
16      if (t_backoff_ < 64)
17          t_backoff_ <<= 1;
18      if (t_backoff_ > 8) {
19          t_rttvar_ += (t_srtt_ >> T_SRTT_BITS);
20          t_srtt_ = 0;
21      }
22  }
```

the lower-bound applies to `t_rtxcur_` before applying the BEB mechanism, the upper bound does so after the BEB. Hence, Lines 10–11 place another lower-bound constraint (i.e., `2.0*tcp_tick_`) for the value after the BEB.

10.6.8 Function `rtt_backoff()`

Function `rtt_backoff()` applies TCP binary exponential backoff (BEB) mechanism to a multiplicative factor "`t_backoff_`." As discussed in Sect. 9.1.2, "`t_backoff_`" is doubled for every timeout and is reset to its initial value when a new ACK packet is received. As we will see, function `rtt_backoff()` is invoked by function `reset_rtx_timer(mild,backoff)` to double "`t_backoff_`."

Program 10.23 shows the details of function `rtt_backoff()`. If the current "`t_backoff_`" is less than 64 (Line 16), it will be doubled (i.e., shifted to the left by one bit in Line 17). Also, a large value of "`t_backoff_`" (e.g., > 8 in Line 18) implies a long interval between two RTT samples. In this case, smoothed RTT and RTT variation may not well represent the actual network RTT. In this case, RTT should be a function of the most recent RTT sample only. Therefore, Line 20 sets "`t_srtt_`" to zero. After this point, function `rtt_update(tao)` will invoke Lines 17 and 18 (rather than Lines 8–15) in Program 10.22 to estimate network RTT.

Program 10.24 Function `reset_rtx_timer` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
1 void TcpAgent::reset_rtx_timer(int mild, int backoff)
2 {
3     if (backoff)
4         rtt_backoff();
5     set_rtx_timer();
6     if (!mild)
7         t_seqno_ = highest_ack_ + 1;
8     rtt_active_ = 0;
9 }
```

10.6.9 Function `set_rtx_timer()` and Function `reset_rtx_timer(mild,backoff)`

Programs 10.21 and 10.24 show the details of functions `set_rtx_timer()` and `reset_rtx_timer(mild,backoff)`, respectively. From Line 14 in Program 10.21, function `set_rtx_timer()` simply sets the timer to expire at t seconds in future, where t is the timeout value returned from function `rtt_timeout()` (see also Program 10.23).

From Program 10.24, function `reset_rtx_timer(mild,backoff)` has four main tasks:

1. Restart the retransmission timer (Line 5)
2. Update the backoff multiplicative factor "`t_backoff_`," if the input argument "`backoff`" is nonzero (Lines 3 and 4)
3. Update the next transmitting sequence number. Store `highest_ack_+1` in "`t_seqno_`," if the input argument "`mild`" is zero (Lines 6 and 7)
4. Cancel the pending RTT sample collection process by setting "`rtt_active_`" to zero (Line 8).

10.6.10 Function `newtimer(pkt)`

Function `newtimer(pkt)` is invoked from within function `newack(pkt)` when a new ACK packet is received and the TCP sender is about to send out another packet. As shown in Program 10.25, it takes a pointer `pkt` to an ACK packet as an input argument. If the TCP sender still has data to transmit (i.e., Line 4 returns `true`), Line 5 will restart the retransmission timer by invoking `set_rtx_timer()`. Otherwise, Line 7 will cancel the timer by invoking `cancel_rtx_timer()`.

Program 10.25 Function `newtimer` of class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
 1 void TcpAgent::newtimer(Packet* pkt)
 2 {
 3     hdr_tcp *tcph = hdr_tcp::access(pkt);
 4     if (t_seqno_ > tcph->seqno() || tcph->seqno() < maxseq_)
 5         set_rtx_timer();
 6     else
 7         cancel_rtx_timer();
 8 }
```

10.6.11 Function `timeout(tno)`

Function `timeout(tno)` is invoked when a retransmission timer expires. It adjusts congestion window as well as slow-start threshold, and retransmits the lost packet. Again, function `expire(e)` is invoked when the timer expires. From Line 10 in Program 10.21, function `expire(e)` of class `RtxTimer` simply invokes function `timeout(TCP_TIMER_RTX)` of the associated `TcpAgent` object. As shown in Lines 1–19 of Program 10.26, function `timeout(tno)` takes a timer option (`tno`) as an input argument, where the possible values of "`tno`" are defined in Lines 20–25 of Program 10.26. In this section, we are interested in TCP Tahoe. Therefore, we will discuss the case where only `timeout(TCP_TIMER_RTX)` is invoked.

The basic operation of function `timeout(tno)` is to close the congestion window (Line 10), restart the retransmission timer (Lines 14 and 16), and retransmit the lost packet (Line 18). We will discuss the details of function `slowdown(...)` which closes the congestion window in Sect. 10.7. The retransmission timer is restarted using the function `reset_rtx_timer(mild, backoff)` (see Program 10.24). For zero value of "`mild`," this function sets "`t_seqno_`" to "`highest_ack_+1`". The non zero and zero values of the second input argument "`backoff`" inform function `reset_rtx_timer(mild, backoff)` to and not to (respectively) update the binary exponential backoff multiplicative factor (`t_backoff_`). Again, the TCP sender assumes that all packets with sequence number lower than "`highest_ack_`" are successfully transmitted. At a timeout event, it assumes that the first lost packet (i.e., the packet to be retransmitted) is the packet with sequence number `highest_ack_+1`. After preparing "`t_seqno_`" (i.e., set to `highest_ack_+1`) for retransmission, Line 18 executes `send_much(0, TCP_REASON_TIMEOUT, maxburst_)` to transmit the lost packet.

After a TCP sender transmits all the packets provided by an attached application, its variable "`t_seqno_`" is equal to variable "`curseq_`," and variable "`maxseq_`" stops increasing. After the last packet (with sequence number "`maxseq_`") is acknowledged, variable "`highest_ack_`" is equal to "`maxseq_`." At this point, the TCP sender enters an idle state. Its retransmission

Program 10.26 Function `timeout` of class `TcpAgent` and the possible values of its input argument "`tno`"

```
    //~/ns/tcp/tcp.cc
 1  void TcpAgent::timeout(int tno)
 2  {
        ...
 3      if (cwnd_ < 1) cwnd_ = 1;
 4      if (highest_ack_ == maxseq_ && !slow_start_restart_) {
 5      } else {
 6          recover_ = maxseq_;
 7          if (highest_ack_ < maxseq_) {
 8             ++nrexmit_;
 9             last_cwnd_action_ = CWND_ACTION_TIMEOUT;
10             slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_RESTART);
11          }
12      }
13      if (highest_ack_ == maxseq_)
14          reset_rtx_timer(0,0);
15      else
16          reset_rtx_timer(0,1);
17      last_cwnd_action_ = CWND_ACTION_TIMEOUT;
18      send_much(0, TCP_REASON_TIMEOUT, maxburst_);
19  }

    //~/ns/tcp/tcp.h
20  #define TCP_TIMER_RTX        0
21  #define TCP_TIMER_DELSND     1
22  #define TCP_TIMER_BURSTSND   2
23  #define TCP_TIMER_DELACK     3
24  #define TCP_TIMER_Q          4
25  #define TCP_TIMER_RESET      5
```

timer, however, does not stop at this moment. It keeps expiring for every period of RTO. From Line 14 of Program 10.26, function `timeout(tno)` will invoke `reset_rtx_timer(0,0)`, which stores the value of `highest_ack_+1` in variable "`t_seqno_`" but does not change the multiplicative factor "`t_backoff_`." Also, function `send_much(0, TCP_REASON_TIMEOUT, maxburst_)` will not send out any packet since "`t_seqno_`" is not less than "`curseq_`" (see Program 10.11).

When the application sends more user demand (i.e., data payload) by invoking `sendmsg(nbytes)`, variable "`curseq_`" is incremented and the TCP connection becomes active. In this case, function `send_much(0,0,maxburst_)` will send out packets, starting with the packet with sequence number `t_seqno_` = `max_seq_+1` = `highest_ack_` + 1.

There are two important details in function `timeout(tno)`. One is that regardless of whether connection is busy or idle, Line 17 sets the variable "`last_cwnd_action_`" which records the latest action imposed on the congestion window to be `CWND_ACTION_TIMEOUT`. Another is related to variable "`recover_`." Recall

that "recover_" contains the highest sequence number among all the transmitted TCP packets at the latest loss event (i.e., either timeout or Fast Retransmit). Line 6 hence records the highest TCP sequence number transmitted so far in the variable "recover_."

10.7 Window Adjustment Functions

From Sect. 9.1.2, a TCP sender dynamically adjusts congestion window to fully use the network resource. When the network is underutilized, a TCP sender increases transport-level transmission rate by opening the congestion window. In the slow-start phase, where the congestion window (cwnd_) is less than the slow-start threshold (ssthresh_), a TCP sender increases the congestion window by one for every received ACK packet. If "cwnd_" is not less than "ssthresh_," on the other hand, a TCP sender will be in the congestion avoidance phase, and the congestion window is increased by 1/cwnd_ for every received ACK packet.

When the network is congested, a TCP sender closes the congestion window to help relieve network congestion. As discussed in Sect. 9.1.2, TCP may decrease the window by half or may reset the congestion window size to one, depending on the situation.

Class TcpAgent provides two main functions, which can be used to adjust the congestion window:

- opencwnd(): Increases the size of the congestion window. The increasing method depends on "cwnd_" and "ssthresh_."
- slowdown(how): Decreases the size of the congestion window by the method specified in "how."

The possible values of "how" are defined in Program 10.27. All possible values of "how" contain 32 bits, and conform to the following format: 1 of "one" bit and 31 of "zero" bits. The difference among the values defined in Program 10.27 lies in the position of the "one" bit. This format acts as a simple identification of the input method "how" through an "AND" operator. For example, suppose the input argument "how" is set to CLOSE_CWND_ONE (=2). Let "x" be a variable which can be any value in Program 10.27. Then, how & x would be nonzero if and only if x=CLOSE_CWND_ONE. This assignment is also able to contain several "slowdown" methods in one variable using an "OR" operator. For example, let "how" be CLOSE_CWND_ONE|CLOSE_SSTHRESH_HALF. Then, how & x would be nonzero if and only if x=CLOSE_CWND_ONE or x=CLOSE_SSTHRESH_ HALF.

Program 10.27 Possible values of "how" – the input argument of function "slowdown"

```
    //~/ns/tcp/tcp.h
1   #define CLOSE_SSTHRESH_HALF      0x00000001
2   #define CLOSE_CWND_HALF          0x00000002
3   #define CLOSE_CWND_RESTART       0x00000004
4   #define CLOSE_CWND_INIT          0x00000008
5   #define CLOSE_CWND_ONE           0x00000010
6   #define CLOSE_SSTHRESH_HALVE     0x00000020
7   #define CLOSE_CWND_HALVE         0x00000040
8   #define THREE_QUARTER_SSTHRESH   0x00000080
9   #define CLOSE_CWND_HALF_WAY      0x00000100
10  #define CWND_HALF_WITH_MIN       0x00000200
11  #define TCP_IDLE                 0x00000400
12  #define NO_OUTSTANDING_DATA      0x00000800
```

Program 10.28 Function opencwnd of class TcpAgent

```
    //~/ns/tcp/tcp.cc
1   void TcpAgent::opencwnd()
2   {
3       if (cwnd_ < ssthresh_) {
4           cwnd_ += 1;
5       } else {
6           double increment = increase_num_ / cwnd_;
7           cwnd_ += increment;
8       }
9       if (maxcwnd_ && (int(cwnd_) > maxcwnd_))
10          cwnd_ = maxcwnd_;
11  }
```

10.7.1 Function opencwnd()

Function opencwnd() is invoked when a new ACK packet is received (see function recv_newack_helper() in Line 5 of Program 10.16). It opens the congestion window and allows the TCP sender to transmit more packets without waiting for acknowledgement. Program 10.28 shows the details of function opencwnd(). From Line 3, if "cwnd_" is less than "ssthresh_," the TCP sender will be in the slow-start phase and "cwnd_" will be increased by 1. Otherwise, the TCP sender must be in a congestion avoidance phase, and "cwnd_" will be increased by 1/cwnd_ (Lines 6 and 7), where "increase_num_" is usually set to 1. In both cases, Lines 9 and 10 bound "cwnd_" within "maxcwnd_," the predefined maximum congestion window size.

Program 10.29 Function "slowdown" of class TcpAgent

```
    //~/ns/tcp/tcp.cc
 1  void TcpAgent::slowdown(int how)
 2  {
 3      double win, halfwin, decreasewin;
 4      int slowstart = 0;
 5      if (cwnd_ < ssthresh_)
 6          slowstart = 1;
 7      halfwin = windowd() / 2; win = windowd();
 8      decreasewin = decrease_num_ * windowd();

 9      if (how & CLOSE_SSTHRESH_HALF)
10          if (first_decrease_ == 1||slowstart ||
                  last_cwnd_action_ == CWND_ACTION_TIMEOUT)
11              ssthresh_ = (int) halfwin;
12          else
13              ssthresh_ = (int) decreasewin;
14      else if (how & THREE_QUARTER_SSTHRESH)
15          if (ssthresh_ < 3*cwnd_/4) ssthresh_ = (int)
               (3*cwnd_/4);

16      if (how & CLOSE_CWND_HALF)
17          if (first_decrease_==1||slowstart||decrease_num_
               ==0.5){
18              cwnd_ = halfwin;
19          } else
20              cwnd_ = decreasewin;
21      else if (how & CWND_HALF_WITH_MIN) {
22           cwnd_ = decreasewin;
23           if (cwnd_ < 1) cwnd_ = 1;
24      } else if (how & CLOSE_CWND_RESTART) cwnd_=int(wnd_
          restart_);
25      else if (how & CLOSE_CWND_INIT) cwnd_ = int(wnd_init_);
26      else if (how & CLOSE_CWND_ONE) cwnd_ = 1;

27      if (ssthresh_ < 2) ssthresh_ = 2;
28      if (how & (CLOSE_CWND_HALF|CLOSE_CWND_RESTART|
                        CLOSE_CWND_INIT|CLOSE_CWND_ONE))
29          cong_action_ = TRUE;
30      if (first_decrease_ == 1) first_decrease_ = 0;
31  }
```

10.7.2 Function slowdown(how)

Function slowdown(how) closes the congestion window based on the method specified in the input argument "how." It is invoked from within function dupack_action() and timeout(tno) to decrease transport layer transmission rate. Function dupack_action() invokes function slowdown(how) feeding how = CLOSE_SSTHRESH_HALF | CLOSE_CWND_ONE (Line 7 in Program 10.19) as an input argument. From Program 10.29, this invocation halves the current slow-start threshold (Lines 9–13) and resets the congestion window to 1 (Line 26).

Function `timeout(tno)`, on the other hand, invokes function `slowdown(how)` with an input argument "`how`" = `CLOSE_SSTHRESH_HALF | CLOSE_CWND_RESTART` as an input argument (Line 10 in Program 10.26). From Program 10.29, this invocation halves the current slow-start threshold (Lines 9–13) and resets the congestion window to a predifined window-restart value (Line 24). In both cases, NS2 uses an "OR" operator to combine how to adjust the slow-start threshold and how to adjust the congestion window, and feed it as an input argument to function `slowdown(how)`.

The details of function `slowdown(how)` are shown in Program 10.29. In this function, Lines 4–6 first set a variable "`slowstart`" to one and zero when TCP is in the slow-start phase (i.e., `cwnd_< ssthresh_`) and in the congestion avoidance phase (i.e., `cwnd_>= ssthresh_`), respectively. Line 7 stores half of the window size in a variable "`halfwin`" and the window size in a variable "`win`." Variable "`decrease_num_`" in Line 8 is set to 0.5 by default. Therefore, the local variable "`decreasewin`" is half of the current congestion window. The variable "`decrease_num_`" provides an option for window decrement, where different TCP variants may set the value of `decrease_num_` differently (e.g., 0.3, 0.7). Lines 9–26 show different window closing method, which will be invoked according to the input argument "`how`." Line 27 ensures that the minimum slow-start threshold is 2. Line 29 sets the variable "`cong_action_`" to be `true` if the window adjustment method, "`how`," is either of `CLOSE_CWND_HALF`, `CLOSE_CWND_RESTART`, `CLOSE_CWND_INIT`, or `CLOSE_CWND_ONE`. Again, the variable "`cong_action_`" is used in function `output(seqno,reason)` to set the congestion flag of the transmitted packet. Finally, Line 32 sets "`first_decrease_`" to zero, indicating TCP has decreased the congestion window at least once.

Lines 9–15 adjust the slow-start threshold (`ssthresh_`) based on the value of "`how`":

- `CLOSE_SSTHRESH_HALF` (Lines 11 and 13): Sets the slow-start threshold "`ssthresh_`" to the half (halfwin or decreasewin) of the current congestion window size "`cwnd_`."
- `THREE_QUARTER_SSTHRESH` (Line 15): Sets the slow-start threshold "`ssthresh_`" to at least 3/4 of its current value.

Similarly, Lines 16–29 adjust the congestion window (`cwnd_`) based on the value of "`how`":

- `CLOSE_CWND_HALF` (Lines 17–20): Decreases the current congestion window size (i.e., "`cwnd_`") by half ("`halfwin`" "`decreasewin`").
- `CWND_HALF_WITH_MIN` (Lines 22 and 23): Sets the current congestion window size to "`decreasewin`" but not less than 1.
- `CLOSE_CWND_RESTART` (Line 24): Sets the current congestion window size to the predifined window-restart value `wnd_restart_`.
- `CLOSE_CWND_INIT` (Line 25): Sets the current congestion window size to "`wnd_init_`" (i.e., initial value of congestion window size).
- `CLOSE_CWND_ONE` (Line 26): Sets the current congestion window size to 1.

10.8 Chapter Summary

TCP is a reliable connection-oriented transport layer protocol. It provides a connection with end-to-end error control and congestion control. NS2 implements TCP senders and TCP receivers using C++ classes `TcpAgent` and `TcpSink`, which are bound to OTcl classes `Agent/TCP` and `Agent/TCPSink`, respectively. A TCP sender has four main responsibilities. First, based on user demand, it creates and forwards packets to a TCP receiver. Second, it provides an end-to-end connection with reliability by means of packet retransmission. Third, it implements timer-related components to estimate round trip time (RTT) and retransmission timeout (RTO), used to determine whether a packet is lost. Finally, it dynamically adjusts transport-level transmission rate to fully use the network resource without causing network congestion. A TCP receiver is responsible for creating (cumulative) ACK packets and forwards them back to the TCP sender.

10.9 Exercises

1. Which NS2 class is responsible for processing and creating TCP acknowledgment packets?
   a. What are the names and types of variables which store information about received packets?
   b. Upon receiving a TCP packet, what is the number this class puts in the acknowledgment packet? Explain the process. Draw a diagram if necessary.
   c. At a certain moment, what is the maximum number of TCP packets that this class needs to retain. How does NS2 engineer a data structure to keep this information about these packets?
2. How does an application tell TCP to start sending packets? When does TCP stop sending packets?
3. What are the functions of class `TcpAgent` related to packet transmission? What are their differences?
4. From within which function does class `TcpAgent` create a packet?
5. Which method does TCP use to transmit packets – immediate or delayed packet transmission? From within which function does it transmit packets? Explain the transmission process.
6. TCP usually transmits several packets without acknowledgment. Each packet can be used to collect round-trip time samples. What is the maximum number of round-trip time samples that a TCP agent can collect at a certain time? What are the variables that the TCP agent uses to collect these samples?

7. What is the purpose of the variable "`tcp_tick`" of class `TcpAgent`?
8. Explain how NS2 computes the smoothed round-trip time, round-trip time variation, and instantaneous retransmission timeout. Discuss the similarities/differences with that recommended by [35].

Chapter 11
Application: User Demand Indicator

Operating on top of a transport layer agent, an application models user demand for data transmission. A user is assumed to create bursts of data payload or application packets. These payload bursts are transformed into transport layer packets which are then forwarded to a transport layer receiving agent. Applications can be classified into *traffic generator* and *simulated application*. A traffic generator creates user demand based on a predefined schedule. A simulated application, on the other hand, creates the demand as if the application is running.

In the following, we first discuss the relationship between an application and a transport layer agent in Sect. 11.1. Class `Application` is introduced in Sect. 11.2. Sections 11.3 and 11.4 discuss the detailed implementation of traffic generators and simulated applications, respectively. Finally, the chapter summary is given in Sect. 11.5.

11.1 Relationship Between an Application and a Transport Layer Agent

From time to time, an application needs to exchange user demand information with a transport layer agent. An application declares a pointer "`agent_`" to an attached agent. Similarly, an agent defines a pointer "`app_`" to an attached application. The user demand information is exchanged between an application and an agent through these two pointers. Section 9.2.2 gives a four-step agent configuration method, which binds an application and a transport layer agent together. The details of these four steps are given below:

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_11, © Springer Science+Business Media, LLC 2012

Step 1: Create a Sending Agent, a Receiving Agent, and an Application

An agent and an application can be created using instproc new{. .} as follows:

```
set agent [new Agent/<agent_type>]
set app [new Application/<app_type>]
```

where <agent_type> and <app_type> denote the type of an agent (e.g., TCP or UDP) and an application (e.g., Traffic/CBR or FTP), respectively.

Step 2: Connect an Agent to an Application

A two way connection between an application and an agent can be created using an OTcl command of class Application whose syntax is shown below:

```
$app attach-agent $agent
```

where $app and $agent are Application and Agent objects. The details of instproc attach-app{s_type} are shown in Program 11.1. Line 7 stores an input Agent object in the variable "agent_." Line 12 invokes function attachApp(this) of class Agent, while Lines 19–22 create a connection from the Agent object to the Application object. From Line 21, function attachApp(app) stores an input Application object "app" in the variable "app_" of the Agent object. Since Line 12 feeds the pointer "this" to function attachApp(...) of the Application object, it simply sets the pointer agent_->app_ to point to the Application object.

Step 3: Attaching an Agent to a Low-Level Network

Here, we consider the case where an agent is connected to a node in a low-level network. As discussed in Sect. 6.5.3, an agent can be attached to a node using the instproc attach-agent{node agent} of class Simulator, where "node" and "agent" are the Node, and Agent objects, respectively. This instproc creates a two-way connection between a Node object "node" and an Agent object "agent." It sets variable agent::target_ to point to "node" and installs "agent" in the demultiplexer (i.e., instvar dmux_) of "node."

The process of attaching an agent to a node involves three OTcl classes: Simulator, Node, and RtModule. Figure 11.1 shows the main operation when "$ns attach-agent $node $agent" is executed:

1. Instproc attach-agent{node agent} of class Simulator invokes $node attach $agent.

Program 11.1 An OTcl command `attach-agent` of class `Application` and function `attachApp` of class `Agent`

```
    //~/ns/apps/app.cc
 1  int Application::command(int argc, const char*const* argv)
 2  {
 3      Tcl& tcl = Tcl::instance();
 4      ...
 5      if (argc == 3) {
 6          if (strcmp(argv[1], "attach-agent") == 0) {
 7              agent_ = (Agent*) TclObject::lookup(argv[2]);
 8              if (agent_ == 0) {
 9                  tcl.resultf("no such agent %s", argv[2]);
10                  return(TCL_ERROR);
11              }
12              agent_->attachApp(this);
13              return(TCL_OK);
14          }
15          ...
16      }
17      return (Process::command(argc, argv));
18  }

    //~/ns/common/agent.cc
19  void Agent::attachApp(Application *app)
20  {
21      app_ = app;
22  }
```

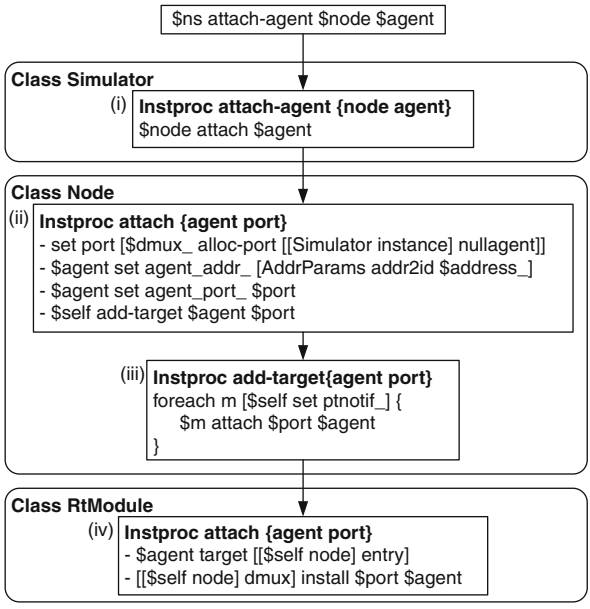

Fig. 11.1 Internal mechanism of instproc `attach-agent{node agent}` of class `Simulator`

2. Instproc attach{agent} of class Node allocates a port for an input agent $agent, configures the instvars "agent_addr_" and "agent_port_" of the input agent $agent, and invokes the instproc add-target{agent port} to ask every routing module stored in the instvar ptnotif_ to attach "agent" to the Node.
3. Instproc add-target{agent port} of class Node invokes instproc attach{ agent port} of each routing module (of class RtModule) stored in the instvar ptnotif_.
4. Instproc attach{agent port} of class RtModule creates a connection between a node and an agent. Here, $agent sets its $target_ to point to the entry of $node, while $node installs $agent in the slot "port" of its demultiplexer "dmux_." This connection is created for both sending and receiving agents.

Step 4: Associating a Sending Agent with a Receiving Agent

To associate a sending agent with a receiving agent, NS2 uses an instproc connect of class Simulator, whose syntax is shown below:

```
$ns connect $s_agent $r_agent
```

where $ns, $s_agent, and $r_agent are Simulator, sending Agent, and receiving Agent objects, respectively.

Program 11.2 shows the details of instproc connect{src dst}. Lines 3 and 4 invoke instproc simplex-connect{src dst}, which set up a connection from "src" to "dst,"[1] and simplex-connect{dst src} which creates a connection from "dst" back to "src."

Instvars "dst_addr_" and "dst_port_" are configured in Lines 9 and 10. When an agent creates a packet, it stores values in variables "dst_.addr_" and "dst_.port_" in the packet header. During a packet forwarding process, a low-level network delivers packets to the agent corresponding to whose address and port are specified in the packet header.

11.2 Applications

An application is defined in a C++ class Application as shown in Program 11.3. Class Application has only one key variable "agent_" which is a pointer to class Agent. Other two variables, "enableRecv_" and

[1]From Table 9.3, instvars "dst_addr_" and "dst_port_" are bound to the C++ variables "addr_" and "port_" of the object "dst_," respectively, in the C++ domain.

Program 11.2 Instprocs `connect` and `simplex-connect` of class `Simulator`

```
    //~/ns/tcl/lib/ns-lib.tcl
 1  Simulator instproc connect {src dst} {
 2      ...
 3      $self simplex-connect $src $dst
 4      $self simplex-connect $dst $src
 5      ...
 6  }

 7  Simulator instproc simplex-connect { src dst } {
 8      ...
 9      $src set dst_addr_ [$dst set agent_addr_]
10      $src set dst_port_ [$dst set agent_port_]
11      ...
12  }
```

Program 11.3 Declaration of class `Application`

```
    //~/ns/apps/app.h
 1  class Application : public Process {
 2  public:
 3      Application();
 4      virtual void send(int nbytes);
 5      virtual void recv(int nbytes);
 6      virtual void resume();
 7  protected:
 8      virtual int command(int argc, const char*const* argv);
 9      virtual void start();
10      virtual void stop();
11      Agent *agent_;
12      int enableRecv_;
13      int enableResume_;
14  };
```

"`enableResume_`," are flag variables, which indicate whether an `Application` object should react to functions `recv(nbytes)` and `resume()`, respectively. These two flag variables are set to zero by default.

11.2.1 Functions of Classes `Application` and `Agent`

After their connection is created, an application and an agent may invoke `public` functions of each other through the pointers "`agent_`" and "`app_`," respectively. The key `public` functions of class `Application` include functions `send(nbytes)`, `recv(nbytes)`, and `resume()`, while those of class `Agent` are functions `send(nbytes)`, `sendmsg(nbytes)`, `close()`, `listen()`, and `set_pkttype(pkttype)`.

Program 11.4 Implementation of functions `send`, `recv`, and `resume` of class `Application`

```
    //~/ns/apps/app.cc
1   void Application::send(int nbytes)
2   {
3       agent_->sendmsg(nbytes);
4   }

5   void Application::recv(int nbytes)
6   {
7       if (! enableRecv_)
8           return;
9       Tcl& tcl = Tcl::instance();
10      tcl.evalf("%s recv %d", name_, nbytes);
11  }

12  void Application::resume()
13  {
14      if (! enableResume_)
15          return;
16      Tcl& tcl = Tcl::instance();
17      tcl.evalf("%s resume", name_);
18  }
```

Apart from these `public` functions, class `Application` also provides `protected` functions `start()` and `stop()` to start and stop an `Application` object, respectively. Finally, there are five key OTcl commands for class `Application` which can be invoked from the OTcl domain: `start{}`, `stop{}`, `agent{}`, `send{nbytes}`, and `attach-agent{agent}`.

11.2.2 Public Functions of Class `Application`

Program 11.4 shows the details of the three following `public` functions of class `Application`:

- `send(nbytes)`: Inform the attached transport layer agent that a user needs to send "`nbytes`" of data payload. Line 3 sends the demand to the attached agent by executing "`agent_->sendmsg(nbytes)`."
- `recv(nbytes)`: Receive "`nbytes`" bytes from a receiving transport layer agent. A UDP agent specifies "`nbytes`" as the number of bytes in a received packet. In case of UDP, "`nbytes`" is equal to packet size; on the other hand, TCP specifies "`nbytes`" as the number of in-sequence received bytes. Due to possibility of out-of-order packet delivery, "`nbytes`" can be greater than the size of one packet in case of TCP.

Program 11.5 Declaration of class `TrafficGenerator`

```
    //~/ns/tools/trafgen.h
1   class TrafficGenerator : public Application {
2   public:
3       TrafficGenerator();
4       virtual double next_interval(int &) = 0;
5       virtual void init() {}
6       virtual double interval() { return 0; }
7       virtual int on() { return 0; }
8       virtual void timeout();
9       virtual void recv() {}
10      virtual void resume() {}
11  protected:
12      virtual void start();
13      virtual void stop();
14      double nextPkttime_;
15      int size_;
16      int running_;
17      TrafficTimer timer_;
18  };
```

- `resume()`: Invoked by a sending agent, this function indicates that the agent has sent out all data corresponding to the user demand. For a TCP sender, this function is invoked when it sends out all the packets regardless of whether the transmitted packets have been acknowledged.

Note that both functions `recv(nbytes)` and `resume()` will do nothing if "`enableRecv_`" = `0` and "`enableResume_`" = `0`, respectively. Otherwise, Lines 10 and 17 in Program 11.5 will invoke OTcl commands or instprocs `recv{nbytes}` and `resume{}` in the OTcl domain, respectively. By default, both "`enableRecv_`" and "`enableResume_`" are set to zero, and functions `recv(nbytes)` and `resume()` simply do nothing.

A user may specify actions to be done upon invocation of functions `recv(nbytes)` and `resume()` by

1. Setting "`enableRecv_`" and/or "`enableResume_`" to one.
2. Specifying the actions in

   a. Functions `recv(nbytes)` and/or `resume()`,
   b. Instprocs `recv{nbytes}` and/or `resume{}` in the OTcl domain, or
   c. OTcl commands `recv{nbytes}` and/or `resume{}` in the function `command()`.

It is important to perform both the steps above. Failing to perform the second step will result in a run-time error, since the OTcl commands or the instprocs `recv{nbytes}` and `resume{}` are undefined in class `Application`.

11.2.3 Related Public Functions of Class `Agent`

Class `Application` may invoke the following functions of class `Agent` through variable "`agent_`":

- `send(nbytes)`: Send "`nbytes`" of application payload (i.e., user demand) to a receiving agent. If `nbytes=-1`, the user demand would be infinite.
- `sendmsg(nbytes,flags)`: Similar to function `send(nbytes)`, remove this space, but also feed "`flags`" as an input variable.
- `close()`: Ask an agent to close the connection (applicable only to TCP)
- `listen()`: Ask an agent to listen to (i.e., wait for) a new connection (applicable only to Full TCP)
- `set_pkttype(pkttype)`: Set the variable "`type_`" of the attach agent to be "`pkttype`."

11.2.4 OTcl Commands of Class `Application`

Defined in the function `command`, OTcl commands associated with class `Application` are as follows:

- `start{}`: Invoke function `start()` to start the application.
- `stop{}`: Invoke function `stop()` to stop the application.
- `agent{}`: Return the name of the attached agent.
- `send{nbytes}`: Send "`nbytes`" bytes of user payload to the attached agent by invoking function `send(nbytes)`.
- `attach-agent{agent}`: Create a two-way connection between itself and the input `Agent` object (`agent`).

The details of the above OTcl command can be found in file *~ns*/apps/app.cc.

11.3 Traffic Generators

A traffic generator models user behavior which follows a predefined schedule. In particular, it sends a demand to transmit one burst of user payload to an attached agent at a time specified in the schedule, regardless of the state of the agent. In NS2, there are four main traffic generators:

- Constant Bit Rate (CBR): Send a fixed size payload to the attached agent. By default, the interval between two payloads (i.e., the sending rate) is fixed, but it can be optionally randomized.

- Exponential On/Off: Act the same as CBR during an ON period. Stop generating traffic during an OFF period. ON and OFF periods are exponentially distributed, and are alternated when one period terminates.
- Pareto On/Off: Similar to the Exponential On/Off traffic generator. However, the durations of ON and OFF periods follow a Pareto distribution.
- Traffic Trace: Generate traffic according to a given trace file, which contains a series of inter-burst transmission intervals and payload burst sizes.

11.3.1 An Overview of Class `TrafficGenerator`

NS2 implements traffic generators using class `TrafficGenerator`. Program 11.5 shows the declaration of the abstract class `TrafficGenerator`, where function `next_interval(size)` in Line 4 is pure virtual. Class `TrafficGenerator` consists of the following variables:

`timer_`	A `TrafficTimer` object, which determines when a new burst of payload is created.
`nextPkttime_`	Simulation time that the next payload will be passed to the attached transport layer agent
`size_`	Application payload size
`running_`	`true` if the `TrafficGenerator` object is running

Class `TrafficGenerator` derives and overrides the following four key functions of class `Application`. It derives functions `recv(nbytes)` and `resume()` (i.e., share the implementation) from class `Application`, and overrides functions `start()` and `stop()` of class `Application`. Functions `start()` and `stop()` inform the `TrafficGenerator` object to start and stop, respectively, generating user payload. In Program 11.6, function `start()` initializes the `TrafficGenerator` object by invoking function `init()`[2] in Line 3, and sets "`running_`" to 1 in Line 4. It computes and stores the time until the next payload burst is generated in the variable "`nextPkttime_`" in Line 5. Finally, it sets the "`timer_`" to expire at "`nextPkttime_`" seconds in future (Line 6). From Lines 8 to 13 in Program 11.6, function `stop()` does the opposite of function `start()`. It cancels the pending timer (if any) in Line 11, and sets "`running_`" to 0 in Line 12.

[2]In Line 5 of Program 11.5, function `init()` simply does nothing.

Program 11.6 Functions `start`, `stop`, and `timeout` of class `TrafficGenerator`

```
     //~/ns/tools/trafgen.cc
  1  void TrafficGenerator::start()
  2  {
  3      init();
  4      running_ = 1;
  5      nextPkttime_ = next_interval(size_);
  6      timer_.resched(nextPkttime_);
  7  }

  8  void TrafficGenerator::stop()
  9  {
  10     if (running_)
  11         timer_.cancel();
  12     running_ = 0;
  13 }

  14 void TrafficGenerator::timeout()
  15 {
  16     if (! running_)
  17         return;
  18     send(size_);
  19     nextPkttime_ = next_interval(size_);
  20     if (nextPkttime_ > 0)
  21         timer_.resched(nextPkttime_);
  22     else
  23         running_ = 0;
  24 }
```

Class `TrafficGenerator` also defines the following three new functions:

`next_interval(size)`	Takes payload size "`size`" as an input argument, and returns the delay time after which a new payload is generated (Line 4). This function is pure virtual and must be implemented by the instantiable derived classes of class `TrafficGenerator`.
`init()`	Initializes the traffic generator.
`timeout()`	Sends a user payload to the attached application and restart "`timer_`." This function is invoked when "`timer_`" expires.

The details of function `timeout()` are shown in Lines 14–24 of Program 11.6. Function `timeout()` does nothing if the `TrafficGenerator` object is not running (Lines 16–17). Otherwise, it will send "`size_`" bytes of user payload to the attached agent using function `send(nbytes)` (defined in Program 11.4). Then,

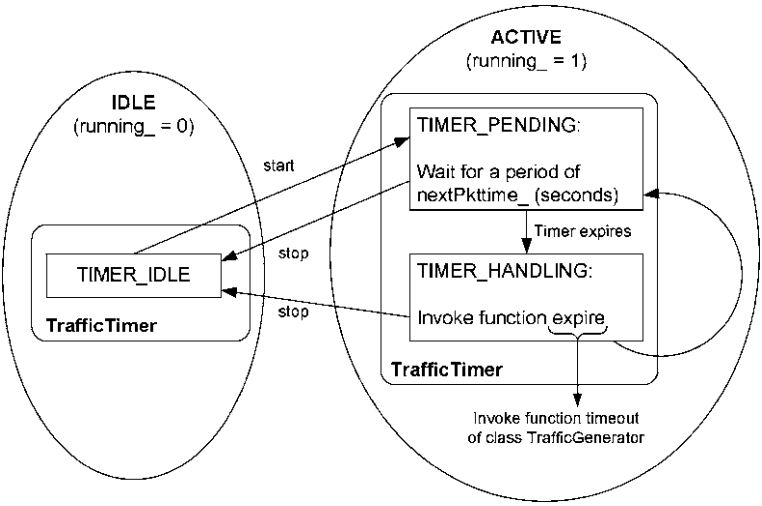

Fig. 11.2 Main mechanism of a traffic generator

Line 19 will compute "`nextPkttime_`." If "`nextPkttime_`" > `0`, Line 21 will inform "`timer_`" to expire after a period of "`nextPkttime_`." Otherwise, Line 23 will stop the `TrafficGenerator` by setting "`running_`" to zero.

11.3.2 Main Mechanism of a Traffic Generator

Figure 11.2 illustrates the main mechanism of a traffic generator, which relies heavily on the variable "`timer_`" whose class is `TrafficTimer` derived from class `TimerHandler`. As discussed in Sect. 15.1, class `TimeHandler` consists of three states: `TIMER_IDLE`, `TIMER_PENDING`, and `TIMER_HANDLING`. Each of these states corresponds to one of two `TrafficGenerator` states: Idle (i.e., `running_=0`) and Active (i.e., `running_=1`). While state `TIMER_IDLE` corresponds to the idle state of a `TrafficGenerator` object, the other two timer states are within the active state of a `TrafficGenerator` object.

Starting in an idle state, a traffic generator moves to active state when function `start()` is invoked. Here the "`timer_`" state is set to `TIMER_PENDING`. At the expiration, "`timer_`" moves to state `TIMER_HANDLING` and invokes function `timeout()` of class `TrafficGenerator`. After executing function `timeout()`, it changes the state to `TIMER_PENDING`, reschedules itself, and repeats the above process. When "`timer_`" state is `TIMER_PENDING` or `TIMER_HANDLING`, the traffic generator can be stopped by invoking function `stop()`.

Program 11.7 Declaration of class `TrafficTimer`, function `expire` of class `TrafficTimer`, and the constructor of class `TrafficGenerator`

```
    //~/ns/tools/trafgen.h
1   class TrafficTimer : public TimerHandler {
2   public:
3       TrafficTimer(TrafficGenerator* tg) : tgen_(tg) {}
4   protected:
5       void expire(Event*);
6       TrafficGenerator* tgen_;
7   };

    //~/ns/tools/trafgen.cc
8   void TrafficTimer::expire(Event *)
9   {
10      tgen_->timeout();
11  }

12  TrafficGenerator::TrafficGenerator() :
            nextPkttime_(-1), running_(0), timer_(this) {}
```

Program 11.7 shows the declaration of class `TrafficTimer`, which derives from class `TimerHandler` (see Sect. 15.1). Class `TrafficTimer` has a key variable "`tgen_`," a pointer to a `TrafficGenerator` object (Line 6). At the expiration, NS2 invokes function `expire(e)` of "`timer_`" (Lines 8–11), which in turn invokes function `timeout()` of the associated `TrafficGenerator` object (i.e., `*tgen_`).

A two-way connection between `TrafficGenerator` and `TrafficTimer` objects is created as follows. Class `TrafficGenerator` declares "`timer_`" as its pointer to a `TrafficTimer` object (Line 17 in Program 11.5). A `TrafficGenerator` object instantiates "`timer_`" by feeding a pointer to itself (i.e., "`this`") as an input argument (Line 12 in Program 11.7). The construction of variable "`timer_`" in turn assigns the input pointer (i.e., "`this`") to its pointer to a `TrafficGenerator` object, "`tgen_`" (Line 3 in Program 11.7), creating a connection back to the `TrafficTimer` object.

11.3.3 Built-in Traffic Generators in NS2

11.3.3.1 Constant Bit Rate (CBR) Traffic

A CBR traffic generator creates a fixed size payload burst for every fixed interval. As shown in Program 11.8, NS2 implements CBR traffic generators using a

Program 11.8 Class `CBRTrafficClass` which binds C++ class `CBR_Traffic` and OTcl class `Application/Traffic/CBR` together

```
   //~/ns/tools/cbr_traffic.cc
1 static class CBRTrafficClass : public TclClass {
2  public:
3    CBRTrafficClass() : TclClass("Application/Traffic/CBR") {}
4    TclObject* create(int, const char*const*) {
5        return (new CBR_Traffic());
6     }
7 } class_cbr_traffic;
```

Table 11.1 Instvars of a CBR traffic generator

Instvar	Default value	Description
`packetSize_`	210	Application payload size in bytes
`rate_`	488×10^3	Sending rate in bps
`random_`	0 (false)	If `true`, introduce a random time to the inter-burst transmission interval.
`maxpkts_`	16^7	Maximum number of application payload packets that CBR can send

C++ class `CBR_Traffic` which is bound to an OTcl class `Application/Traffic/CBR`, whose key instvars with their default values are shown in Table 11.1.

Note that, by default the inter-burst transmission interval, which is the interval between the beginning of two successive payload bursts, can be computed by dividing the payload burst size by the sending rate. By default, the inter-burst transmission interval is $210 \times 8/488.000 \approx 3.44$ ms. The detailed mechanism for class `CBR_Traffic` will be discussed in Sect. 11.3.4.

11.3.3.2 Exponential On/Off Traffic

An exponential on/off traffic generator acts as a CBR traffic generator during an ON interval and does not generate any payload during an OFF interval. ON and OFF periods are both exponentially distributed. As shown in Program 11.9, NS2 implements Exponential On/Off traffic generators using a C++ class `EXPOO_Traffic` which is bound to an OTcl class `Application/Traffic/Exponential`, whose key instvars with their default values are shown in Table 11.2.

11.3.3.3 Pareto On/Off Traffic

A Pareto On/Off traffic generator does the same as an Exponential On/Off generator but the ON and OFF periods conform to a Pareto distribution. As shown in Program 11.10, NS2 implements Pareto On/Off traffic generators using a C++

Program 11.9 Class `EXPTrafficClass` which binds C++ class `EXPOO_Traffic` and OTcl class `Application/Traffic/Exponential` together

```
    //~/ns/tools/expoo.cc
1 static class EXPTrafficClass : public TclClass {
2  public:
3     EXPTrafficClass() : TclClass("Application/
                                    Traffic/Exponential") {}
4     TclObject* create(int, const char*const*) {
5         return (new EXPOO_Traffic());
6     }
7 } class_expoo_traffic;
```

Table 11.2 Instvars of an exponential on/off traffic generator

Instvar	Default value	Description
`packetSize_`	210	Application payload size in bytes
`rate_`	64×10^3	Sending rate in bps during an ON period
`burst_time_`	0.5	Average ON period in seconds
`idle_time_`	0.5	Average OFF period in seconds

Program 11.10 Class `POOTrafficClass` which binds C++ class `POO_Traffic` and OTcl class `Application/Traffic/Pareto` together

```
    //~/ns/tools/pareto.cc
1 static class POOTrafficClass : public TclClass {
2  public:
3     POOTrafficClass() : TclClass("Application/Traffic/
          Pareto") {}
4     TclObject* create(int, const char*const*) {
5         return (new POO_Traffic());
6     }
7 } class_poo_traffic;
```

class `POO_Traffic` which is bound to an OTcl class `Application/Traffic/Pareto`, whose key instvars with their default values are shown in Table 11.3.

11.3.3.4 Traffic Trace

A traffic trace generates payload bursts according to a given trace file. As shown in Program 11.11, NS2 implements traffic trace using the C++ class `TrafficTrace` which is bound to an OTcl class `Application/Traffic/Trace`. Unlike other traffic generators described before, we need to specify a traffic trace file in the OTcl domain using the OTcl command `attach-tracefile` of class `Application/ Traffic/Trace` (see Example 11.1).

Table 11.3 Instvars of a Pareto/off traffic generator

Instvar	Default value	Description
packetSize_	210	Application payload in bytes
rate_	64×10^3	Sending rate in bps during an ON period
burst_time_	0.5	Average ON period in seconds
idle_time_	0.5	Average OFF period in seconds
shape_	1.5	A "Shape" parameter of a Pareto distribution

Program 11.11 Class TrafficTraceClass which binds C++ class TrafficTrace and OTcl class Application/Traffic/Trace together

```
   //~/ns/trace/traffictrace.cc
1 static class TrafficTraceClass : public TclClass {
2  public:
3     TrafficTraceClass() : TclClass("Application/Traffic/
      Trace") {}
4     TclObject* create(int, const char*const*) {
5             return(new TrafficTrace());
6     }
7 } class_traffictrace;
```

Example 11.1. A CBR traffic generator in Example 9.1 can be replaced with a traffic trace by substituting Lines 10–12 in Program 9.2 with the following lines:

```
set tfile [new Tracefile]
$tfile filename example-trace
set tt [new Applicaiton/Traffic/Trace]
$tt attach-tracefile $tfile
$tt attach-agent $udp
```

□

A traffic trace file is a pure binary file. A codeword in the binary file consists of two 32-bit fields. The first field indicates inter-burst transmission interval in microseconds, while the second is the payload size in bytes (see file *~ns*/tcl/ex/example-trace as an example traffic trace file).

11.3.4 Class CBR_Traffic: An Example Traffic Generator

This section presents a C++ implementation of class CBR_Traffic whose declaration is shown in Program 11.12. Class CBR_Traffic derives from class TrafficGenerator and has the following main variables:

rate_	CBR sending rate in bps
interval_	Packet inter-arrival time in seconds

Program 11.12 Declaration, function `start`, and function `init` of class `CBR_Traffic`

```
    //~/ns/tools/cbr_traffic.cc
 1  class CBR_Traffic : public TrafficGenerator {
 2   public:
 3      CBR_Traffic();
 4      virtual double next_interval(int&);
 5      inline double interval() { return (interval_); }
 6   protected:
 7      virtual void start();
 8      void init();
 9      double rate_;
10      double interval_;
11      double random_;
12      int seqno_;
13      int maxpkts_;
14 };

15 void CBR_Traffic::start()
16 {
17      init();
18      running_ = 1;
19      timeout();
20 }

21 void CBR_Traffic::init()
22 {
23      interval_ = (double)(size_ << 3)/(double)rate_;
24      if (agent_)
25          if (agent_->get_pkttype() != PT_TCP &&
                    agent_->get_pkttype() != PT_TFRC)
26              agent_->set_pkttype(PT_CBR);
27 }
```

`random_`	If `true`, the inter-arrival time will be random
`seqno_`	CBR sequence number
`maxpkts_`	Upper bound on the sequence number

Based on the main mechanism discussed in Sect. 11.3.2, NS2 activates a traffic generator by invoking function `start()`. When activated, a traffic generator invokes its function `timeout()`, which generates an application payload, periodically. An interval between two consecutive `timeout()` invocations is determined by the function `next_interval(size)`. The `timeout()` invocations occur repeatedly until the traffic generator is deactivated (by an invocation of function `close()`).

As shown in Program 11.12, function `start()` invokes function `init()` (Line 17) to initialize the traffic generator, sets "`running_`" to 1 (Line 18), and invokes function `timeout()` (Line 19). The details of function `init()` are shown in

Program 11.13 Function `next_interval` of class `CBR_Traffic`

```
    //~/ns/tools/cbr_traffic.cc
1   double CBR_Traffic::next_interval(int& size)
2   {
3       interval_ = (double)(size_ << 3)/(double)rate_;
4       double t = interval_;
5       if (random_)
6           t += interval_ * Random::uniform(-0.5, 0.5);
7       size = size_;
8       if (++seqno_ < maxpkts_)
9           return(t);
10      else
11          return(-1);
12  }
```

Lines 21–28 of Program 11.12. Line 23 computes the inter-burst transmission interval as transmission rate (`rate_`) divided by payload burst size "`size_<<3`" (in bits).[3] Function `init()` would also set the packet type of the attached agent to `PT_CBR`, unless the packet type has already been set to `PT_TCP` or `PT_TFRC` (Lines 25–26).

From Program 11.6, function `timeout()`, sends out "`size_`" bytes of application payload (Line 18), recomputes "`nextPkttime_`" as a value returned from the `next_interval(size_)` (Line 19), and schedules the timer "`timer_`" to expire at "`nextPkttime_`" seconds in future (Line 21). Again, the function `next_interval(size_)` is pure virtual and must be implemented by instantiable child classes of class `TrafficGenerator`. Class `CBR_Traffic` implements this function (Program 11.13), by returning the packet inter-arrival time converted from payload size "`size_`" and CBR transmission rate "`rate_`" (Lines 3 and 9). Optionally, Line 6 may add or subtract a random value to the computed interval if "`random_`" is set to `true`. Also, if the application payload is greater than "`maxpkts_`," Line 11 will return −1 rather than the computed interval.

11.4 Simulated Applications

Unlike traffic generators, a simulated application does not have a predefined schedule for payload generation. Rather, it acts as if an actual application is running. NS2 provides two built-in simulated applications: FTP and Telnet.

[3] Since the units of the variables "`size_`" and "`rate_`" are "bytes" and "bits per second," respectively, Line 9 multiplies "`size_`" with 8 by shifting "`size_`" to the left by 3 bits (see Sect. 15.4.2).

11.4.1 File Transfer Protocol

File Transfer Protocol (FTP) is a protocol which divides a given file into small pieces and transfers them to a destination host. Unlike a general FTP in practice, an NS2 FTP module does not need an input file. It simply informs an attached sending transport layer agent of file size in bytes. Upon receiving user demand (e.g., file size), the agent creates packets which can accommodate the file and forwards them to a connected receiving transport layer agent through a low-level network services. Also, an NS2 FTP module is not responsible for specifying a destination host. Destination host identification is the responsibility of a transport layer agent instead, the simulator employs the instproc `connect{src dst}` (Sect. 11.1) in order to associate a source with a destination.

Due to its simplicity, an FTP module is implemented in the OTcl domain only. Defined in class `Application/FTP`, which derives class `Application`, its main OTcl commands and instprocs include

`attach-agent{agent}`	Register the input `agent` as an attached agent.
`start{}`	Inform the attached agent of a demand to transmit a file with infinite size by executing "`send -1`."
`stop{}`	Stop the pending file transfer session.
`send{nbytes}`	Send a file with size `nbytes` bytes by invoking function `sendmsg(nbytes)` of the attached agent.
`produce{nbytes}`	Inform the attached agent to transmit until its sequence number has reached the minimum of `nbytes` and `maxseq_`.
`producemore {nbytes}`	Inform the attached agent to transmit `nbytes` more packets.

11.4.2 Telnet

Telnet is an interactive client-server text-based application. A Telnet client logs on to a server, and sends text messages to the server. The server in turn executes the received message and returns the result to the client. Clearly, Telnet is not implemented based on a predefined schedule, since its data traffic is created in response to user demand. However, NS2 models a Telnet application in the same way as it does for traffic generators: sending a fixed size packet for every randomized interval.

NS2 defines a Telnet application in C++ class `TelnetApp` and OTcl class `Application/Telnet`, which derives from class `Application`. It uses the

value stored in variable "size_" of the attached agent as the size of each Telnet packet, and computes the inter-burst transmission interval as follows:

- Case I: If "interval_" is nonzero, the inter-burst transmission interval is chosen from an exponential distribution with mean "interval_."
- Case II: If "interval_" is zero, the inter-burst transmission interval is chosen from an empirically generated distribution "tcplib" defined in file *~ns*/tcp/tcplib-telnet.cc.

Telnet has only one configurable variable "interval_." In common with other Application objects, it can be started and stopped using command start{} and stop{}, respectively.

11.5 Chapter Summary

Sitting on top of a transport layer agent, an application informs the attached agent of user demand. Applications can be classified into traffic generators and simulated applications. A traffic generator creates user demand based on a predefined schedule, while a simulated application does so as if the application is running.

Built-in traffic generators in NS2 include CBR (constant bit rate), exponential on/off, Pareto on/off, and Traffic Trace. A CBR traffic generator creates fixed size payloads for every fixed interval. Exponential on/off and Pareto on/off traffic generators create fixed size payloads during an ON period and create no payload during an OFF period. The ON and OFF durations are chosen from an exponential distribution and a Pareto distribution, respectively. Finally, payload size and inter-burst transmission interval for a traffic trace traffic generator are obtained from an input trace file.

NS2 has two built-in simulation application: FTP (File Transfer Protocol) and Telnet. FTP informs the attached agent of the file size (in bytes) to be transferred. The attached agent is responsible for creating packets which can accommodate a file, and choosing the destination of the FTP session. In practice, Telnet is a client–server application, whose traffic depends on the interaction between client and server. However, NS2 implements a Telnet as a traffic generator. In particular, it creates a fixed size payload for every random interval, whose distribution is either exponential or "tcplib" defined in *~ns*/tcp/tcplib-telnet.cc.

Class Application is the base class for all the above applications. It provides few key OTcl commands and instprocs to configure Application objects. An instproc attach-agent{agent} registers the input "agent" as an attached agent. Instprocs start{} and stop{} inform an application to start and stop generating data payload. Derived classes of class Application reuse these functionalities and define their own functionalities for their own purposes.

11.6 Exercises

1. Let `$n1` and `$n2` be two nodes and let `$ns` be the Simulator.
   a. Create an FTP agent to transfer a file with size 10 MBytes from Node `$n1` to Node `$n2`. Explain step-by-step how data are generated, traverse the network, arrive Node `$n2`, and are destroyed.
   b. Create a CBR traffic from Node `$n1` to Node `$n2`. Set the packet size to 100 bytes and bit rate to 4 kbps. Explain packet flow mechanism including timing mechanism (i.e., when and how much the application sends out data).
2. What are the differences of application and transport layers in the TCP/IP protocol stack and in the NS2 implementation?
3. How does an application send/receive user demand to/from an agent? Specify OTcl and/or C++ statements to do so.
4. Develop an application which has two stages. At the heavy-traffic stage, the application continuously provide data to the agent. At the light-traffic stage, it generates packets in the same way that CBR does. Design an experiment, and run simulation to test your developed application.
5. Modify class `Application` by forcing it to print out a message when its functions `recv(nbyte)` and `resume()` are invoked.

Chapter 12
Wireless Mobile Ad Hoc Networks

NS2 classifies communication networks into three main categories. First, wired networks are characterized by wired communication links. Chapter 7 shows a simple form of the wired links called `SimpleLinks` which can be used to connect regular `Node` discussed in Chap. 6.

The second category is pure wireless networks, which contain no wired links. All communications are carried out via "wireless" communication channels only. One category of wireless networks in which there is no central node or coordinator such as a base-station (BS) or an access-point (AP) is known as *wireless mobile ad hoc networks*. In this type of networks, there is no infrastructure and the mobile nodes generally communicate on a peer-to-peer basis. This is in contrast to an infrastructure-based network where mobile nodes communicate via the controller node (e.g., BS or AP), which is generally connected to a wired network infrastructure. Due to the absence of any physical wired communication links, all nodes in a wireless ad hoc network are able to move freely during simulation. NS2 incorporates both wireless communication and node mobility into regular nodes, and defines a new type of nodes called Mobile Nodes.

Finally, hybrid networks contain both wired and wireless communication links. In NS2 terminology, this type of networks is called wired-cum-wireless networks. Unfortunately, NS2 implementation is not as simple as including Regular Nodes and Mobile Nodes in a Tcl Simulation Script. The issues arise when nodes need to have both wired and wireless interfaces. NS2 designs a new node type called Base Station Nodes which act as gateways between wired and wireless domains.

This chapter focuses on the second type of wireless communication networks – wireless mobile ad hoc networks.[1] In particular, the emphasis is on *Mobile Nodes* associated with wireless mobile ad hoc networks. An overview of Mobile Nodes is given in Sect. 12.1. Sections 12.2–12.5 explain NS2 implementation for network

[1]The implementation of wired-cum-wired networks is out of the scope of this book. Interested readers are recommended to refer to [17].

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_12, © Springer Science+Business Media, LLC 2012

layer, data link layer, MAC layer, and physical layer, respectively. Implementation of node mobility is discussed in Sect. 12.6. Finally, the chapter summary is given in Sect. 12.7.

12.1 An Overview of Wireless Networking

In NS2, the central implementation of wireless networking lies in Mobile Nodes. Complying to the OOP concept, Mobile Nodes inherit all attributes and behaviors from its predecessor, Regular Node discussed in Chap. 6.

12.1.1 Mobile Node

In Fig. 12.1, Mobile Nodes are represented by a C++ class `MobileNode` which is bound to an OTcl class `Node/MobileNode`. The class hierarchy is defined in the C++ domain only, where the C++ class derives `MobileNode` from class `Node`, but the OTcl class `Node/MobileNode` is a top-level class.

Program 12.1 shows declaration of the C++ class `MobileNode`. In addition to attributes inherited from its parent class, the class `MobileNode` defines attributes (e.g., coordinate (`X_`,`Y_`,`Z_`) and `speed_`) to support node mobility.

12.1.2 Architecture of Mobile Node

Another fundamental characteristic of Mobile Nodes is the ability to communicate without physically wired channel. In NS2, this characteristic is contributed by several C++ components, put together in an OTcl object called a `Node/MobileNode` object.

Figure 12.2 shows the architecture of a Mobile Node, which consists of two main parts: regular node part and mobile extension part.

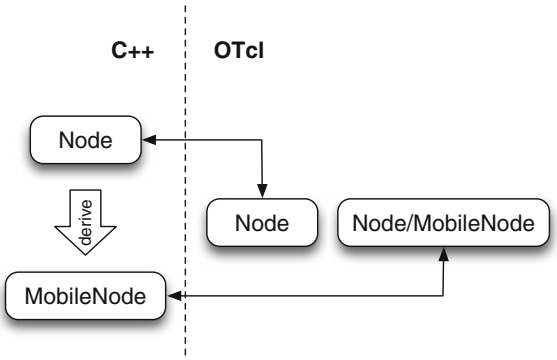

Fig. 12.1 Class hierarchy of mobile nodes

Program 12.1 A C++ class `MobileNode` which is bound to an OTcl class `Node/Mobile`

```
    //~ns/common/mobile-node.h
 1  class MobileNode : public Node
 2  {
 3  protected:
 4      double X_,Y_,Z_;      //The current location
 5      double speed_;        //In meters per second
 6      double dX_,dY_,dZ_;   //Units
 7      double destX_,destY_;//The destination
 8      Event  pos_intr_;
 9      double position_update_time_;      //Last updated
          position
10      double position_update_interval_; //Update interval
11      PositionHandler pos_handle_;       //For random-motion
                                             only
12  private:
13      int          random_motion_;  //Are we running random
                                        motion?
14      Topography   *T_;             //Define an area for
                                        moving
15      LIST_ENTRY(MobileNode) link_; //A global list of mobile
                                        nodes
16  };

    //~ns/common/mobile-node.cc
17  static class MobileNodeClass : public TclClass {
18  public:
19      MobileNodeClass() : TclClass("Node/MobileNode") {}
20      TclObject* create(int, const char*const*) {
21          return (new MobileNode);
22      }
23  } class_mobilenode;
```

12.1.2.1 Regular Node Part

This part is similar to that of regular nodes. The differences lie in classifier configuration and packet forwarding mechanism. The forwarding mechanism depends on types of Mobile Nodes as follows:

- *Source Node*: Packets are created by an application. They enter the node entry[2] and are passed to the address classifier. In this case, the destination address would be different from the address of this Node. The address classifier, therefore, forwards the packet to the routing agent via "`default_target_`."

[2]The node entry does not actually exist. It is an instproc of class `Node` which returns the first packet reception object. In Fig. 12.2, the node entry is the address classifier.

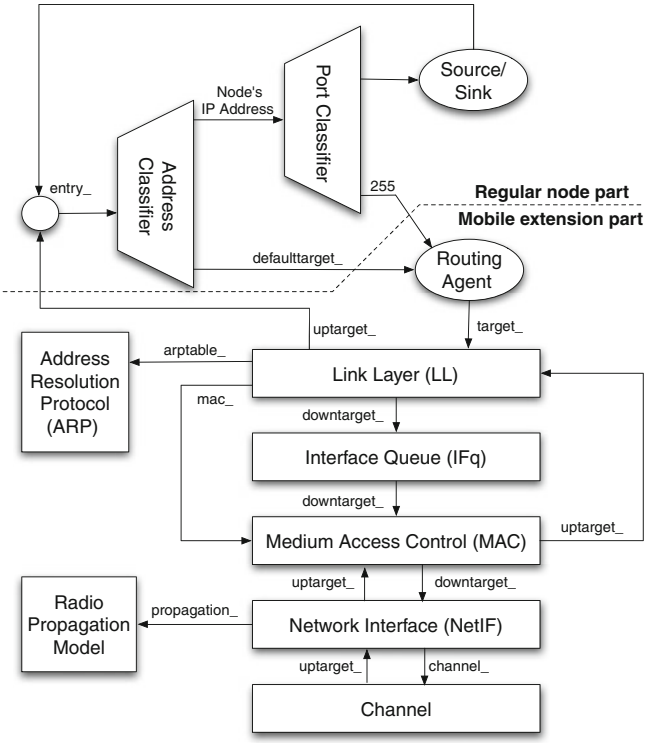

Fig. 12.2 The architecture of mobile nodes

- *Destination Node*: Packets enter a mobile node through node entry from the link layer. They are passed through an address classifier. In this case, the destination address is the same as the node address. The packets are therefore passed to the port classifier which then forwards the packets to the attached sink application.
- *Forwarding Node*: Packets are passed from the link layer to the node entry and the address classifier. In this case, the packet is forwarded to the routing agent, since the address in the packet header does not match with the node's address.

12.1.2.2 Mobile Extension Part

This part is an extension from the regular node part. It includes all the components in the lower part of Fig. 12.2. Each of these components is specified by the options

Table 12.1 Details of mobile node configuration: Options and corresponding instvars of the `Simulator` object, and their descriptions

Option	Instvar	Description
`-adhocRouting`	`routingAgent_`	Routing protocol (e.g., `AODV`).
`-llType`	`lltype_`	Link layer type (e.g., `LL`).
`-macType`	`macType_`	Medium access control type (e.g., `Mac/802_11`).
`-ifqType`	`ifqType_`	Type of the underlying buffer management discipline (e.g., `Queue/DropTail/PriQueue`)
`-ifqLen`	`ifqlen_`	Size of the buffer in packets (e.g., five packets)
`-antType`	`antType_`	Antenna type (e.g., `Antenna/OmniAntenna`)
`-propType`	`propType_`	Type of the underlying radio propagation models (e.g., `Propagation/TwoRayGround`)
`-phyType`	`phyType_`	Network interface type (e.g., `Phy/WirelessPhy`)
`-channelType`	`channelType_`	Channel type (e.g., `Channel/WirelessChannel`)
`-topoInstance`	`topoInstance_`	An OTcl instance which identify topography (e.g., `new Topology`)
`-agentTrace`	`agentTrace_`	Turning agent trace `ON` or `OFF`
`-routerTrace`	`routerTrace_`	Turning routing trace `ON` or `OFF`
`-macTrace`	`macTrace_`	Turning MAC trace `ON` or `OFF`
`-movementTrace`	`movementTrace_`	Turning movement trace `ON` or `OFF`

of instproc `node-config{args}`. The values and description for most frequently used options are shown in Table 12.1. The complete list of options can be found in the file *~ns*/tcl/lib/ns-lib.tcl.

12.1.3 General Packet Flow in a Wireless Network Implementation

In Fig. 12.2, objects in a Mobile Node are vertically connected. These objects refer to upper and lower objects using their C++ pointers "`uptarget_`" and "`downtarget_`" which can be configured using OTcl commands `down-target{...}` and `up-target{...}`, respectively (e.g., see example usage in Lines 22 and 29 of Program 12.6).

Due to the vertical architecture, packets generally move upward and downward. They move upward when they need to be processed locally. On the other hand, they move downward when they need to be transmitted over the air. NS2 determines the direction to which a packet is moving using a field "`direction_`" of the common packet header (see Program 8.6). Packets moving upward and downward have this field marked with "`UP`" and "`DOWN`," respectively.

12.1.4 Mobile Node Configuration Process

A process of creating Mobile Nodes consists of two key steps:

- **Step 1 – Mobile Node Configuration**: Store configuration information in instvars of the `Simulator` using the following instproc:

  ```
  $ns node-config -<option> <value>
  ```

  where `$ns` is the `Simulator` object. The available `<option>` and their `<value>` are given in Table 12.1. Among all options, `-adhocRouting` is mandatory. If configured, Step 2 will create Mobile Nodes, rather than Regular Nodes.
- **Step 2 – Mobile Node Construction**: Create a mobile node as in Fig. 12.2 using the OTcl statement

  ```
  $ns node
  ```

Example 12.1. The following is an excerpt from a built-in example (see the file *~ns*/tcl/ex/simple-wireless.tcl).

```
$ns node-config -adhocRouting AODV \
     -llType LL \
     -macType Mac/802_11 \
     -ifqType Queue/DropTail/PriQueue \
     -ifqLen 50 \
     -antType Antenna/OmniAntenna \
     -propType Propagation/TwoRayGround \
     -phyType Phy/WirelessPhy \
     -channelType Channel/WirelessChannel \
     -topoInstance $topo \
     -agentTrace ON \
     -routerTrace ON \
     -macTrace OFF \
     -movementTrace OFF
```

This OTcl statement configures all Mobile Nodes as follows. The routing protocol is AODV [37], the link layer type is `LL`, and the MAC protocol is IEEE 802.11 [39]. Each Mobile Node has a buffer which can hold upto 50 packets. The service discipline is based on prioritized queue which gives priority to routing packets. The type of antanna is omni-directional. The propagation model is two-ray propagation model [38]. The physical network transmission and the type of the shared channel are both wireless. Finally, the statement turns agent and router traces on, and turns MAC and movement traces off. □

Program 12.2 Instproc `node-config` and other variable setting instprocs of class `Simulator`

```
     //~ns/tcl/lib/ns-lib.tcl
  1  Simulator instproc node-config args {
  2      set args [eval $self init-vars $args]
  3      $self instvar addressType_ routingAgent_ propType_
           macTrace_ \
  4      ...
  5      if [info exists phyTrace_] {
  6          Simulator set PhyTrace_ $phyTrace_
  7      }
  8      if {[info exists propType_]} {
  9          set propInstance_ [new $propType_]
  10         Simulator set propInstCreated_ 1
  11     }
  12     ...
  13 }

  14 Simulator instproc adhocRouting {val} {
  15     $self set routingAgent_  $val
  16 }
  17 Simulator instproc llType {val} { $self set llType_  $val }
  18 Simulator instproc macType {val} { $self set macType_  $val}
       Object
```

12.1.4.1 Step 1: Mobile Node configuration

Discussed earlier in Sect. 6.1.3, the instproc `node-config{args}` creates and configures instvars of the `Simulator` object. The list of instvars of class `Simulator` corresponding to each option is shown in Table 12.1. The values in these instvars will later be used in Step 2 to create Mobile Nodes.

Program 12.2 shows the details of the instproc `node-config{args}` (Lines 1–13), and instprocs used to set node configuration instvars (Lines 14–18). The instproc `node-config{args}` takes as input arguments a `list` variable "`args`" which contains options and values for node configuration (see Example 12.1). The only main statement of this instproc is to invoke the instproc `init-var{args}` of class `Object` (Line 2) to initialize the instvars of class `Simulator`. The rest of this instproc (Lines 5–11) is just to set up variables and objects as configured by the instproc `init-var{...}`.

Program 12.3 shows the details of the instproc `init-var{args}`. This instproc takes node configuration options and values as its input arguments. Line 2 invokes the instproc `init-default-vars{classes}` to initialize default variables of the input class (i.e., `Simulator` in this case). Lines 4–14 are the main loop for this instproc. They iterate for every node option. Lines 5 and 6 store the first (i.e., the option) and the second (i.e., the value) entries of the input argument "`args`"

Program 12.3 Variable initialization instprocs

```
    //~tclcl/tcl-object.tcl
 1  Object instproc init-vars {args} {
 2      $self init-default-vars [$self info class]
 3      set shadow_args ""
 4      for {} {$args != ""} {set args [lrange $args 2 end]} {
 5          set key [lindex $args 0]
 6          set val [lindex $args 1]
 7          if {$val != "" && [string match {-[A-z]*} $key]} {
 8              set cmd [string range $key 1 end]
 9              if ![catch "$self $cmd $val"] {
10                  continue
11              }
12          }
13          lappend shadow_args $key $val
14      }
15      return $shadow_args
16  }

17  instproc init-default-vars {classes} {
18      foreach cl $classes {
19          if {$cl == "Object"} continue
20          $self init-default-vars "[$cl info superclass]"
21          foreach var [$cl info vars] {
22              if [catch "$self set $var"] {
23                  $self set $var [$cl set $var]
24              }
25          }
26      }
27  }
```

in local variables "key" and "val", respectively. Line 7 removes the prefixing hyphen "-" from the node configuration option string stored in the variable "key." Line 8 stores the result in a local variable "cmd." Then Line 9, invokes the instproc $cmd supplying $val as an input argument.[3] Examples of these instprocs $cmd are shown in Lines 14–18 of Program 12.2. Each of these instprocs stores a node configuration value in a corresponding instvar of class Simulator. Again, this information shall be later used in Step 2.

From within the instproc init-var{args}, Line 2 invokes the instproc init-default-vars{classes} whose details are shown in Lines 17–27. This instproc initializes default variables for all classes whose name is stored in the input arguments. Based on the OOP concept, the initialization must occur at the base class first. Line 20 recursively moves the process toward the top-level

[3]Note that, the command "catch" returns the execution result of the following Tcl statement. For example, the execution result is zero for successful execution. The set of possible execution results is shown in Program 3.8.

Program 12.4 Instproc `Node` of class `Simulator`: The wireless-related excerpt

```
    //~ns/tcl/lib/ns-lib.tcl
1   Simulator instproc node args {
2       $self instvar Node_ routingAgent_ wiredRouting_
          satNodeType_
3       if { [info exists routingAgent_] && ($routingAgent_
          != "") } {
4           set node [eval $self create-wireless-node $args]
5           ...
6           }
7           return $node
8       }
9       ...
10      return $node
11  }
```

class. Then, the variable initialization actions proceed downward from there. The variable initialization actions are defined in Lines 21–25. The statement "`$self set $var`" (Line 22) specifies `$var` as an instvar of this object. The statement repeats for every default instvar of class `$cl`. The list of instvars is given by the statement "`$cl info vars`" (Line 21). Also, the initialization actions repeats for every class in the input list "`classes`" (Line 21).

After the above process completes, the instvars in Table 12.1 would contain proper values. We are now ready to create mobile nodes.

12.1.4.2 Step 2: Mobile Node Construction

Similar to Regular Nodes, Mobile Nodes are created using the instproc `node {args}` of the `Simulator` object. This instproc reads the configuration stored in the instvars of the `Simulator` object, and creates a Mobile Node accordingly. Unless otherwise specified, the following explanation would be based on Example 12.1.

The Main Process

Program 12.4 shows the details of the instproc "`node`" of class `Simulator`. The aim of this instproc is to create a Mobile Node. Line 3 checks whether the instvar "`routingAgent_`" configured during Step 1 exists and is nonempty. If so, Line 4 creates a Mobile Node by invoking an instproc `create-wireless-node{args}`. The details of the instproc `create-wireless-node{args}` are shown in Program 12.5, where the main statements are Lines 3, 6, 18, 19, and 20, as shown in Fig. 12.3.

Program 12.5 Instprocs `create-wireless-node` and `create-node-instance` of class `Simulator`

```
     //~ns/tcl/lib/ns-lib.tcl
 1   Simulator instproc create-wireless-node args {
 2       ...
 3       set node [eval $self create-node-instance $args]
 4       switch -exact $routingAgent_ {
 5        AODV {
 6            set ragent [$self create-aodv-agent $node]
 7        }
 8           DSR {
 9               $self at 0.0 ``$node start-dsr"
10           }
11           ...
12           default {
13               eval $node addr $args
14               puts "Wrong node routing agent!"
15               exit
16           }
17       }
18       $node add-interface $chan $propInstance_ $llType_
               $macType_ $ifqType_ $ifqlen_ $phyType_ $antType_
               $topoInstance_
19       $node attach $ragent [Node set rtagent_port_]
20       $node topography $topoInstance_
21       return $node
22   }

23   Simulator instproc create-node-instance args {
24       set nodeclass Node/MobileNode
25       return [eval new $nodeclass $args]
26   }
```

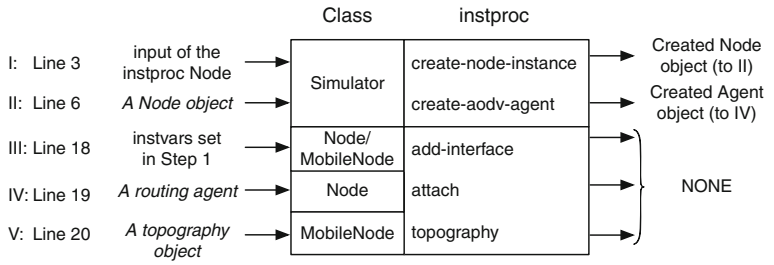

Fig. 12.3 Key OTcl statements invoked during an execution of the instproc `create-wireless-node` of class `Simulator` as shown in Program 12.5

I. **The instproc `create-node-instance` of class `Simulator` (Line 3)** creates a Mobile Node instance from an OTcl class "`Node/Mobile`" (see Lines 23–26 for the details).

Table 12.2 Classes, OTcl commands, and variables which connect the mobile node components together

C++ class	OTcl command	Name of variable	Type of the variable
LL,BiConnector	up-target	"uptarget_"	NsObject*
LL,BiConnector	down-target	"downtarget_"	NsObject*
LL	arptable	arptable_	ARPTable*
LL	mac	mac_	Mac*
Mac	netif	netif_	Phy*
Phy	channel	channel,downtarget_	Channel*,NsObject*
Phy	channel	channel,downtarget_	Channel*,NsObject*
Phy	"node"	node_	Node*
WirelessPhy	propagation	propagation_	Propagation*
WirelessPhy	antenna	ant_	Antenna*
Topography	channel	channel_	WirelessChannel*

II. **The instproc `create-aodv-agent` of class `Simulator` (Line 6)** creates an AODV routing agent (see Program 12.8).
III. **The instproc `add-interface` of class `Node/MobileNode` (Line 18)** creates and configures Mobile Node components as shown in Fig. 12.2 (see Program 12.6).
IV. **The instproc `attach` of class `Node` (Line 19)** attaches the input routing agent to the underlying `Node` object (see the details in Program 6.16).
V. **The OTcl command `topography` of C++ class `MobileNode` (Line 20)** specifies the input as a `topography` object for the Mobile Node.

Patching Components of Mobile Nodes

The mobile extension part in Fig. 12.2 consists of several components. These components are patched together by the instproc `add-interface{...}` of class `Node/MobileNode`.

Program 12.6 shows the details of the instproc `add-interface{args}` of class `Node/MobileNode`. Lines 6–10 create physical network interface, MAC, interface queue, link layer, and antenna objects, as specified by the input parameters. The created objects are stored in local variables in Lines 13–16. Line 17 creates and stores an ARP object in a local variable "`arptable_`." The rest of this instproc configures Mobile Node components as shown in Fig. 12.2. These statements use the following OTcl commands in Table 12.2, to store the input arguments in C++ variables.

Lines 18–19 and 26–27 create and configure a tracing object "`drpT_`," which traces packet drops according to wireless trace format (see Sect. 14.3.5). Line 25

Program 12.6 Instproc `add-interfaces` of class `Node/MobileNode`, which configures Mobile Node components as shown in Fig. 12.2

```
    //~ns/tcl/lib/ns-mobilenode.tcl
 1  Node/MobileNode instproc add-interface { channel pmodel
              lltype mactype qtype qlen iftype anttype topo } {
 2      $self instvar arptable_ nifs_ netif_ mac_ ifq_ ll_ ...
 3      set ns [Simulator instance]
 4      set t $nifs_
 5      incr nifs_
 6      set netif_($t) [new $iftype]    # interface
 7      set mac_($t) [new $mactype]     # mac layer
 8      set ifq_($t) [new $qtype]       # interface queue
 9      set ll_($t) [new $lltype]       # link layer
10      set ant_($t) [new $anttype]     # antanna
11      $ns mac-type $mactype
12      set nullAgent_ [$ns set nullAgent_]
13      set netif $netif_($t)
14      set mac $mac_($t)
15      set ifq $ifq_($t)
16      set ll $ll_($t)
17      set arptable_ [new ARPTable $self $mac]
18      set drpT [cmu-trace Drop "IFQ" $self]
19      $arptable_ drop-target $drpT
20      $ll arptable $arptable_
21      $ll mac $mac
22      $ll down-target $ifq
23      $ll up-target [$self entry]
24      $ifq target $mac
25      $ifq set limit_ $qlen
26      set drpT [cmu-trace Drop "IFQ" $self]
27      $ifq drop-target $drpT
28      $mac netif $netif
29      $mac up-target $ll
30      $mac down-target $netif
31      set god_ [God instance]
32      if {$mactype == "Mac/802_11"} {
33          $mac nodes [$god_ num_nodes]
34      }
35      $netif channel $channel
36      $netif up-target $mac
37      $netif propagation $pmodel    # Propagation Model
38      $netif node $self             # Bind node <---> interface
39      $netif antenna $ant_($t)
40      $channel addif $netif
41      $channel add-node $self
42      $topo channel $channel
43      $self addif $netif
44  }
```

sets the buffer size of the interface queue. Line 31 creates a GOD object (see Sect. 12.6.2). Finally, Lines 35–43 configure the physical layer of the Mobile Node (see Sect. 12.5).

12.2 Network Layer: Routing Agents and Routing Protocols

As discussed in Sect. 6.1.1, a routing protocol specifies how routing information is propagated to all related nodes. This section discusses wireless routing protocols via an example: Ad-hoc On-Demand Distance Vector Routing (AODV) routing protocol.

12.2.1 Preliminaries for the AODV Routing Protocol

12.2.1.1 Terminology

Before proceeding further, let us define the following terminology.

- *Active/inactive*: An active route entry of a Node is a route entry which is in use by the node or any of its neighbors. An active path consists of active routing entries from a source to a destination. Finally, active neighbors are nodes which have created or forwarded one or more packets to a given destination within the most recent *active timeout* period.
- *Fresh/stale*: This terminology is used to compare a pair of sequence numbers and/or route entries. A sequence number in greater value is said to be *fresh*. A route entry with fresh sequence number is said to be *fresh*. It is also said to be *fresh*, if it has equal sequence number and lower (better) number of hops to reach the destination. A node should always accept *fresh* route entries and discard *stale* routing information.
- *Route entries and routing table*: A route entry contains information of how to reach a destination. A node usually puts together all routing entries in a table-formatted database called a *routing table*.

  In AODV, each route entry contains the following fields:

  – Destination address, the next hop node, and the metric (i.e., the number of hops to the destination)
  – Sequence number corresponding to the destination which helps prevent a so-called routing loop problem
  – Active neighbors on this route entry
  – Expiry time which indicates the duration where this route entry is considered *fresh*.

12.2.1.2 Packet Types

AODV defines three main types of packets:

- *Route REQuest (RREQ)*: RREQ is originated and broadcasted to every neighbor of a source node during a route discovery process. RREQ contains the following information:

  – Source address (*src*)
  – Destination address (*dst*)
  – Broadcast ID (*bID*)
  – Source sequence number (SN_s)
  – Destination sequence number (SN_d)
  – No. of hops to destination (*hop_cnt*)

- *Route REPly (RREP)*: RREP is a packet replied by a node. It contains routing information for the destination specified in an RREQ. RREP contains the following information:

  – Source address (*src*)
  – Destination address (*dst*)
  – Time where this entry is considered valid (t_{exp})
  – No. of hops to destination (*hop_cnt*)
  – Destination sequence number (SN_d)

- *HELLO*: HELLO is a special unsolicited RREP packet. It probes neighbors within locality. HELLO contains only two pieces of information: address and sequence number of the sender.

12.2.2 The Principles of AODV

AODV is a proactive routing protocol which discovers a route to the destination as needed. It is designed to solve the network loop problem, where routing packets circulate indefinitely (see Fig. 12.4). AODV solves this problem by discarding packets with stale sequence number. In particular, every Mobile Node maintains three sequence number counters for three types of packets: a destination counter for RREP, a broadcast counter for RREQ, and a neighbor-probing counter for HELLO. Although used for different packet types, these counters work under the same principle.

Before creating a packet, a Mobile Node increases its counter value by one. Then it stamps the packet with the incremented sequence number and sends out the packet. Another Mobile Node receiving the packet determines the freshness of routing packets by comparing the sequence number. Again, a routing packet is said to be *fresh*.[4] In AODV, different sequence numbers are drawn from three different counters, depending on whether the packet type is RREQ, RREP, or HELLO.

[4]See the definition of "fresh" in Sect. 12.2.1.1.

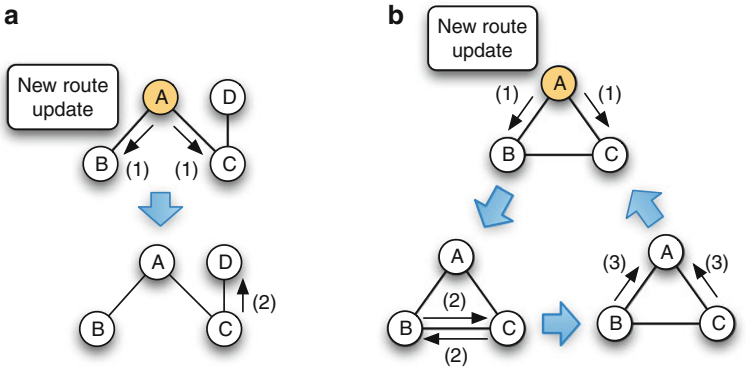

Fig. 12.4 A distance vector flooding protocol: (**a**) Tree topology – a route update process finishes in two steps, (**b**) Loop topology – a route update process continues indefinitely

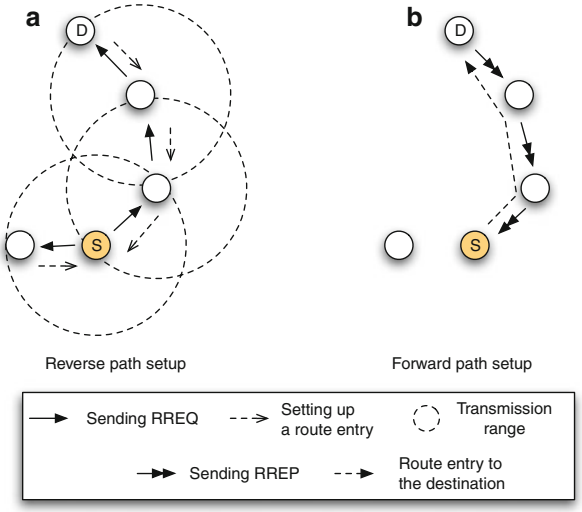

Fig. 12.5 AODV route discovery: Reverse path setup and forward path setup

12.2.2.1 Route Discovery: Identifying a Route to the Destination

Route discovery consists of two main steps (see Fig. 12.5). The first step is to locate the node which contains the required routing information. It starts when a Mobile Node needs routing information. In this case, the Mobile Node (e.g., Node "S" in Fig. 12.5) broadcasts an RREQ packet to all its neighboring nodes. Upon the detection, each of the neighboring nodes discards the packet if it is stale, and processes the packet if it is fresh.

Table 12.3 Various timers, their actions, and the triggering events

Timer	Timeout action	Start/reset by
RREQ timer	Remove the route entry	Insertion of a reverse path route entry
RREP timer	Remove the route entry	Insertion of a forward path route entry
HELLO timer	Send a HELLO packet	Broadcast HELLO
Transmission timer	Recognize a broken link	Transmission of a new packet

Suppose the packet is fresh. The node determines if it has the required routing information. If not, it will run the so-called *reverse path setup* process which records how to return to the source node in its routing table. Then, the node will increment the metric (i.e., the number of hops) in the RREQ packet by one, and will rebroadcast the RREQ packet. This process repeats until a node with the required routing information receives the RREQ packet, where the first Step completes.

The second step is to piggyback the required routing information back to the source node (e.g., Node "S" in Fig. 12.5). It begins by creating an RREP (i.e., route reply) packet. This RREP packet is carried back to the source node using the route identified during the reverse path setup. In the literature, the second Step is referred to as *forward path setup*.

12.2.2.2 Route and Neighbor Maintenance

Mobile Nodes in an ad hoc network are highly dynamic. As they move, the list of neighbors may change and the links may become broken. To keep up with the dynamic, AODV uses timers to regularly probe the network topology and status. These timers are shown in Table 12.3.

12.2.3 An Overview of AODV Implementation in NS2

NS2 implements a routing protocol using routing agents which create, transmit, receive, process, and destroy routing packets. In case of AODV, NS2 declare a C++ class `AODV` as shown in Program 12.7. Class `AODV` derives from class `Agent` and inherits three important attributes and behaviors: (1) a pointer "`target_`" which points to a link layer object (see Fig. 12.2), (2) a function `allocpkt()` which can be used to create packets, and (3) a packet reception function `recv(p,h)`. Among these attributes and behaviors, only the packet reception function is overridden by class `AODV`.[5]

[5]The details of packet reception function will be discussed in Sect. 12.2.6.

Program 12.7 Declaration of class AODV

```
    //~ns/aodv/aodv.h
1   class AODV: public Agent {
2       nsaddr_t          index;   //IP Address of this node
3       u_int32_t         seqno;   //Sequence Number
4       int               bid;     //Broadcast ID
5       aodv_rtable       rtable;  //Routing table
6       aodv_ncache       nbhead;  //A list of active neighbors
7       aodv_bcache       bihead;  //A list of seen Broadcast IDs
8       BroadcastTimer    btimer;  //Broadcast ID timer
9       HelloTimer        htimer;  //Hello timer
10      NeighborTimer     ntimer;  //Neighbor timer
11      RouteCacheTimer   rtimer;  //Route expiration timer
12      LocalRepairTimer  lrtimer; //Delay before route failure
                                       declaration
13      aodv_rqueue       rqueue;  //Store data packet during
                                       route discovery
14      PriQueue         *ifqueue; //A pointer to the interface
                                       queue
15      PortClassifier   *dmux_;   //A pointer to the
                                       demultiplexer
16  }
```

12.2.3.1 File and Class Structure

Main C++ Files for AODV Stored in the Directory ~*ns*/aodv/

aodv.cc,h	Main definition of AODV routing agents
aodv_packet.h	Packet header of AODV routing agents
aodv_rtable.cc,h	AODV route entry and routing table
aodv_rqueue.cc,h	Buffer which stores data packets during a route discovery process

AODV-Related C++ Classes

- *Agent* is responsible for creating, sending, receiving, processing, and destroying routing packets. AODV uses class AODV for these purposes.
- *Timer* takes care of time-driven actions. These classes are BroadcastTimer, HelloTimer, NeighborTimer, RouteCacheTimer, and LocalRepair Timer.
- *Routing information* is stored in route entries (classes aodv_rt_entry and aodv_rtable) and packet header format (classes hdr_aodv, hdr_aodv _error, hdr_aodv_request, and hdr_aodv_reply).

Table 12.4 AODV collections and their C++ implementation

Class name	Variable name	A member of	Data structure	Type of entry
aodv_bcache	bihead	AODV	bsd link list	BroadcastID
aodv_ncache	nbhead	AODV	bsd link list	AODV_Neighbor
	rt_nblist	aodv_rt_entry		
aodv_precursors	rt_pclist	aodv_rt_entry	bsd link list	AODV_Precursor
aodv_rthead	rthead	aodv_rtable	bsd link list	aodv_rt_entry
aodv_rqueue	rqueue	AODV	Link list	Packet
nsaddr_t	unreachable_dst[]	hdr_aodv_error	Array	N/A

- *Collections* contain items of the same type. These collections include seen broadcast IDs, active neighbors, precursors,[6] route entries, packets buffered during a route discovery process, and ID of unreachable destination nodes. The C++ implementation of these collections are shown in Table 12.4, where the details of *bsd link list* are given in Appendix C.1.

12.2.3.2 Route Entries and Packet Header

NS2 stores routing information in the following elements: Route entries (`aodv_rt_entry`) in routing tables (`aodv_rtable`), RREQ header (`hdr_aodv_request`), and RREP header (`hdr_aodv_reply`), and route error reporting header (`hdr_aodv_error`). These elements contain the information fields shown below:

- General fields in route entries (`aodv_rt_entry`):
  - `rt_dst` Destination address
  - `rt_nexthop` Next hop node
  - `rt_hops` Metric
  - `rt_seqno` Seq. No.
  - `rt_pclist` Precursors
  - `rt_nblist` Active neighbors
  - `rt_req_timeout` When I can send another request
  - `rt_req_cnt` Number of broadcasted RREQs
- RREQ-related fields in the packet header `hdr_aodv_request`:
  - `rq_src` The node which creates this RREQ
  - `rq_dst` The node which this RREQ is destined for
  - `rq_hop_count` Number of hops this RREQ has traveled
  - `rq_bcast_id` Broadcast ID

[6] Precursors are nodes associated with a route entry. They are the opposite of next hop nodes. Precursors of a node "`n`" associated with a destination "`dst`" are the nodes which have asked the node "`n`" to forward at least one packet to the destination "`dst`." Essentially, the route entry for the destination "`dst`" of the precursors indicates the next hop as the node "`n`."

Program 12.8 Instproc `create-aodv-agent` of class `Simulator` and OTcl command "`start`" of class `AODV`

```
    //~ns/tcl/lib/ns-lib.tcl
1   Simulator instproc create-aodv-agent { node } {
2       set ragent [new Agent/AODV [$node node-addr]]
3       $self at 0.0 "$ragent start"
4       $node set ragent_ $ragent
5       return $ragent
6   }

    //~ns/aodv/aodv.cc
7   int AODV::command(int argc, const char*const* argv) {
8       if(strncasecmp(argv[1], "start", 2) == 0) {
9             btimer.handle((Event*) 0);
10            htimer.handle((Event*) 0);
11            ntimer.handle((Event*) 0);
12            rtimer.handle((Event*) 0);
13            return TCL_OK;
14      }
15      ...
16      return Agent::command(argc, argv);
17  }
```

- RREP-related fields in the packet header `hdr_aodv_reply`:
  - `rp_src` The node which creates this RREP
  - `rp_dst` The node which this RREP is destined for
  - `rp_hop_count` Number of hops this RREP has traveled
  - `rp_lifetime` Duration between RREP creation and expiration
- Error fields in the packet header `hdr_aodv_error`:
  - `DestCount` Number of unreachable destinations
  - `unreachable_dst[]` Addresses of unreachable destinations
  - `unreachable_dst_seqno[]` Seq. No. of unreachable destinations

Note that fields belonging to routing table, RREQ header, and RREP header are prefixed with "`rt`," "`rq`," and "`rp`," respectively.

12.2.4 AODV Routing Agent Construction Process

An AODV routing agent is created by an OTcl statement "`create-aodv-agent{node}`," where `$node` is a Mobile Node which contains the routing agent (e.g., see Line 6 of Program 12.5).

Program 12.8 shows the details of instproc `create-aodv-agent{node}`. Line 2 creates an object from an OTcl class `Agent/AODV` which is bound to a C++ class `AODV`. As indicated in the file *~ns*/aodv/aodv.cc, the constructor of

class AODV takes node address as an input argument. Line 3 initializes the AODV routing agent by executing its OTcl command "start." In Lines 9–12, the OTcl command "start" initializes four key AODV timers by forcing expiration at the current time. The details of the AODV timers will be discussed later in Sect. 12.2.7. Finally, Lines 4 and 5 store the created routing agent in the instvar "ragent_" of the input Node object and returns the created routing agent to the caller.

12.2.5 General Packet Flow Mechanism in a Wireless Network

Packets can generally be classified into data packets and routing packets. While the former is created by transport layer agents, the latter is created by routing agents. Section 12.1.2 explains the data packet flow mechanism, assuming that the routing agent contains a route entry for the destination under interest. If the routing agent does not have the route entry, it will buffer the packet temporarily, initiate a route discovery process, and wait for the routing information. Once the information is available, it will transmit the data packets buffered earlier.

Routing packet flow mechanism begins from within a routing agent during a route discovery process. After the inception, a routing packet is configured according to the AODV protocol described in Sect. 12.2.2. Then, it is passed to the lower-layer objects for transmission over the air. In case of AODV, routing packets are marked with payload type PT_AODV. In addition, their header types can be AODVTYPE_RREQ for RREQ packets, AODVTYPE_RREP for RREP packets, AODVTYPE_RREQ for route error reporting packets, and AODVTYPE_HELLO for HELLO packets.

12.2.6 Packet Reception and Processing Function of AODV

Function recv(p,h) is central to AODV packet flow mechanism. It is executed to receive both data and routing packets in both upward and downward directions. Program 12.9 shows the details of this function. Line 4 determines whether the incoming packet (*p) is a routing packet (i.e., PT_AODV). If so, Line 6 processes the AODV packet by invoking the function recvAODV(p). Otherwise, the packet *p must be a data packet. In this case, Lines 9–21 pre-process the data packet and Lines 22–25 send off the packet.

The pre-processing Lines 9–21 include addition of an IP header[7] (Line 11), setting the TTL value (Line 13), and dropping packet if it circulates in a network loop (Lines 15–16) or if its TTL reaches zero (Line 19). To send a packet, Line 22

[7]TCP and its ACK also include IP header into the packet. We do not need to add IP header here. Otherwise, we would have added the IP header twice (See Line 10).

Program 12.9 Function `recv(p,h)` class `AODV`

```
    //~ns/aodv/aodv.cc
 1  void AODV::recv(Packet *p, Handler*) {
 2     struct hdr_cmn *ch = HDR_CMN(p);
 3     struct hdr_ip *ih = HDR_IP(p);
 4     if(ch->ptype() == PT_AODV) {
 5         ih->ttl_ -= 1;
 6         recvAODV(p);
 7         return;
 8     }
 9     if((ih->saddr() == index) && (ch->num_forwards() == 0)) {
10         if (ch->ptype() != PT_TCP && ch->ptype() != PT_ACK)
11             ch->size() += IP_HDR_LEN;
12         if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
13             ih->ttl_ = NETWORK_DIAMETER;
14     } else if(ih->saddr() == index) {
15         drop(p, DROP_RTR_ROUTE_LOOP);
16         return;
17     } else {
18         if(--ih->ttl_ == 0) {
19             drop(p, DROP_RTR_TTL); return;
20         }
21     }
22     if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
23         rt_resolve(p);
24     else
25         forward((aodv_rt_entry*) 0, p, NO_DELAY);
26 }
```

determines the packet type. If the packet is a broadcast packet, Line 25 will call the function `forward(rt,p,delay)` to send the packet. Otherwise, Line 23 will call the function `rt_resolve(p)` which invokes a route discovery process and send out the packet once the routing information is available.

When an incoming packet is an AODV routing packet, Line 6 of Program 12.9 invokes function `recvAODV(p)` the details of which are shown in Program 12.10. Line 3 inspects AODV header type of the incoming packet, and invokes one of the following functions: `recvRequest(p)`, `recvReply(p)`, `recvError(p)`, or `recvHello(p)`. These functions perform AODV operation as discussed earlier in Sect. 12.2.2. The readers are encouraged to see the details of these functions in the file *~ns*/aodv/aodv.cc.

12.2.7 AODV Time-Driven Actions

Another important part of the AODV protocol is to take time-based actions. These actions are to remove outdated information (e.g., broadcast IDs, route entries, active

Program 12.10 Function `recvAODV(p)` of class `AODV`

```
    //~ns/aodv/aodv.cc
 1 void AODV::recvAODV(Packet *p) {
 2  struct hdr_aodv *ah = HDR_AODV(p);
 3  switch(ah->ah_type) {
 4    case AODVTYPE_RREQ:
 5     recvRequest(p);
 6     break;
 7    case AODVTYPE_RREP:
 8     recvReply(p);
 9     break;
 10   case AODVTYPE_RERR:
 11    recvError(p);
 12    break;
 13   case AODVTYPE_HELLO:
 14    recvHello(p);
 15    break;
 16   default:
 17    fprintf(stderr, "Invalid AODV type (%x)\n", ah->ah_type);
 18     exit(1);
 19   }
 20 }
```

Table 12.5 Timers, their class variables, and their expiration actions

Class name	Variable name	Agent actions
`BroadcastTimer`	`btimer`	Call `agent->id_purge()`, i.e., purge all expired broadcast ID from the list.
`HelloTimer`	`htimer`	Call `agent->sendHello()`, i.e., send out a `HELLO` message.
`NeighborTimer`	`ntimer`	Call `agent->nb_purge()`, i.e., purge inactive neighbors from the list.
`RouteCacheTimer`	`rtimer`	Call `agent->rt_purge()`, i.e., purge stale route entries from the routing table.
`LocalRepairTimer`	`lrtimer`	– If the route is down, call `agent->rt_down(rt)`, where `rt` is a route entry whose destination is specified in the header of the incoming packet `*p` – Destroy the incoming packet `*p`.

neighbors), and are implemented using five C++ timers in Table 12.5, where "`agent`" is a pointer to the AODV agent containing the timer (see the timer implementation in Sect. 15.1).

All timers except for `LocalRepairTimer` are forced to expire at the node construction (see Lines 9–12 in Program 12.8). At the expiration, these timers take expiration actions as shown in Table 12.5 and restart themselves. The `LocalRepairTimer` object, on the other hand, is started when the routing agent sends out an RREQ packet. If the corresponding `RREP` packet is received before the timer expires, the timer will be stopped and nothing will happen. If, on the other hand, the

route entry is still down by the time the timer expires, the corresponding destination will be declared as unreachable by executing "`agent->rt_down(rt)`."

12.3 Data Link Layer: Link Layer Models, Address Resolution Protocols, and Interface Queues

This section focuses on the three NS2 modules located at the Data Link Layer: Link layer, Address Resolution Protocols (ARP), and Interface Queues (see Fig. 12.2).

12.3.1 Link Layer Objects

Link layer objects model several characteristics at the link layer such as link bandwidth, propagation delay, and packet framing (e.g., sequence number, acknowledge number). It bridges the routing layer (i.e., routing agent) to the MAC layer.

In NS2, link layer objects are implemented using a C++ class `LL` which is bound to the OTcl class with the same name. Program 12.11 shows its C++ class declaration and the details of its function `recv(p,h)`. The C++ class `LL` derives from class `LinkDelay` (see also Sect. 7.2). It inherits bandwidth and delay attributes from the class `LinkDelay`. It is also responsible for link-layer packet framing (e.g., embedding sequence number in Line 3 and acknowledge number in Line 4 into packet headers). It contains two important pointers – "`mac_`" in Line 7 pointing to a Medium Access Control (MAC) object and "`arptable_`" in Line 8 pointing to an Address Resolution Protocol (ARP) object. As discussed in Sect. 12.1.3, class `LL` has two pointers- "`downtarget_`" and "`uptarget_`" which connect to the lower and upper layer objects, respectively.

Lines 12–24 show the details of the function `recv(p,h)` of class `LL`. The process is quite straightforward for downward transmission using the function `sendDown(p)` (Lines 22 and 23). For upward transmission, class `LL` classifies packets into APR packets and non-ARP packets. Line 19 sends non-ARP packets up the hierarchy using `sendUp(p)`. If the incoming packet is an ARP packet, Line 17 will ask the ARP object to handle the packet by calling its function `arpinput(p,this)`.

12.3.2 Address Resolution Protocol

In practice, a Node needs to translate an IP address to a hardware address from time to time. To do so, it looks up the so-called ARP table for the address translation. If the required translation entry does not exist in the table, it may ask for the entry from

Program 12.11 Declaration and details of the function `recv(p,h)` of class `LL`

```
    //~ns/queue/priqueue.h
 1  class LL : public LinkDelay {
 2  protected:
 3      int seqno_;              // link-layer sequence number
 4      int ackno_;              // ACK received so far
 5      int macDA_;              // destination MAC address
 6      Queue* ifq_;             // interface queue
 7      Mac*   mac_;             // MAC object
 8      ARPTable*  arptable_;    // ARP table object
 9      NsObject* downtarget_;   // for outgoing packet
10      NsObject* uptarget_;     // for incoming packet
11  }

    //~ns/queue/priqueue.cc
12  void LL::recv(Packet* p, Handler*)
13  {
14      hdr_cmn *ch = HDR_CMN(p);
15      if(ch->direction() == hdr_cmn::UP) {
16          if(ch->ptype_ == PT_ARP)
17              arptable_->arpinput(p, this);
18          else
19              uptarget_ ? sendUp(p) : drop(p);
20          return;
21      }
22      ch->direction() = hdr_cmn::DOWN;
23      sendDown(p);
24  }
```

the other nodes. The protocol which asks for translation from an IP address to an hardware address is called an *Address Resolution Protocol (ARP)*, while the reversal protocol for translating hardware addresses to IP addresses is called Reverse ARP (RARP) [21].

NS2 implements ARPs using two C++ classes. Class `ARPEntry` models address translation record, while class `ARPTable` contains a link list of `ARPEntry` and takes all ARP-related actions (see the files *~ns*/mac/arp.h,cc). The address translation process proceeds as follows.

Suppose Node A would like to determine a hardware address of a given IP address. Node A calls function `arpresolve(dst,p,ll)` to put the hardware address corresponding to an IP Address "`dst`" in the packet `*p`. If Node A does not contain mapping information for the address "`dst`," it sends an *ARP request* packet using the function `arprequest(src,dst,ll)`. The packet is sent to the air via the down-target of the `LL` object, "`ll`."[8]

[8]The process bypasses the `LL` object to avoid any alteration (e.g., sequence number) at the link-layer object.

Suppose an ARP request packet arrives at Node B. The packet is delivered upward until it reaches the `LL` object. In this case, the packet direction would be "`UP`" and the payload type of the packet would be "`PT_ARP`." In this case, the statement "`arp_->arpinput(p,this)`" shall be executed (Line 17 in Program 12.11). If the packet `*p` is an ARP request packet and the hardware address to be translated is stored in the field "`arp_tpa`," the `ARPTable` object will create and transmit a ARP reply packet. If, on the other hand, the incoming packet `*p` is an ARP reply packet, the `ARPTable` object will look for the address mapping information in the ARP reply packet, embed the hardware address in the packet header, and transmit the constructed packet. The details of the above ARP operation can be found in *~ns*/mac/arp.h,cc.

Note that class `ARPTable` defines a pointer "`hold_`" to hold a packet while waiting for an ARP reply packet. If another packet enters the `ARPTable` before the ARP reply packet returns, the earlier buffered packet would be dropped (i.e., "`hold_`" will point to the new packet).

12.3.3 Interface Queues

Section 7.3 discusses the principles of queue and buffer management. In wired networks, a queue is installed in each of `SimpleLink` objects. In a wireless network, a queue is installed in each of the wireless physical interface. This is the reason NS2 calls a queue in a wireless network an *interface queue*.

The most widely used queue type in wireless networks is *prioritized queue*s. NS2 implements a prioritized queue in the C++ class `PriQueue`. This class derives from class `DropTail` and is bounded to the OTcl class whose name is `Queue/DropTail/PriQueue`. Prioritized queues operate fairly similar to drop-tail queues discussed in Sect. 7.3. But they enqueues high-priority and low-priority packets at the head and at the end of the queue, respectively. By default, high-priority packets include routing packets whose payload types are `PT_DSR`, `PT_TORA`, `PT_AODV`, `PT_AOMDV`, and `PT_MDART`. Packets with other payload types are treated as low-priority packets. The readers are encouraged to go through the details of class `PriQueue` in the files *~ns*/queue/priqueue.h,cc.

12.4 Medium Access Control Layer: IEEE 802.11

A Medium Access Control (MAC) protocol defines communications rules to which Mobile Nodes comply with to access a shared medium. Generally, MAC protocol can be classified into random MAC protocols and deterministic MAC protocols. NS2 implements TDMA (Time Division Multiple Access) MAC protocols and IEEE 802.11 MAC protocols for deterministic and random MAC protocols, respectively (see the directory *~ns*/mac).

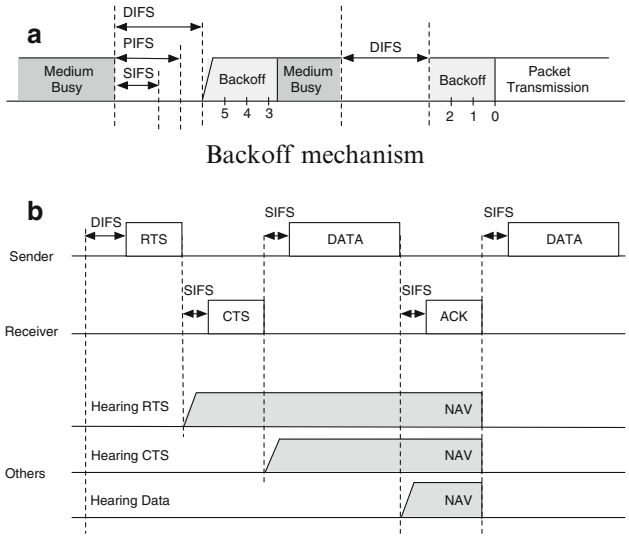

Fig. 12.6 Distributed coordination function in IEEE 802.11

This section explains MAC protocol implementation via an example, the standard IEEE 802.11-based MAC [39], which is used widely in wireless local area networks. The details for the TDMA MAC protocol can be found in the files *~ns*/mac/mac-tdma.h,cc.

12.4.1 Description of IEEE 802.11 MAC Protocol

IEEE 802.11 DCF consists of three main mechanisms: contention window adjustment, back-off mechanisms, and collision avoidance (CA). Once a Mobile Node is turned on, it initializes a state variable called *contention window* to its minimum value (`CWmin`). The contention window is doubled for every transmission failure until it reaches the maximum value (`CWmax`). If the transmission is successful or if the packet is dropped due to the retry limit, the contention window will be reset to `CWmin`.

Figure 12.6a illustrates the back-off mechanism in the IEEE 802.11 DCF MAC. After window adjustment, the DCF MAC protocol picks a random back-off value uniformly distributed between 0 and CW, where CW is the value of the current contention window. This back-off value is the number of *idle* time slots where a Mobile Node has to wait before commencing a transmission.

Another mechanism of IEEE 802.11 DCF is *Collision Avoidance* (CA), which uses two mechanisms: four-way handshake and InterFrame Space (IFS). In Fig. 12.6b, the four-way handshake method transmits RTS, CTS, DATA, and ACK

packets in sequence. Here, a sender intending to transmit data first transmits a Ready To Send (RTS) packet. Upon receiving an RTS packet, the receiver transmits a Clear To Send (CTS) packet back to the sender. Then, the sender can start sending a DATA packet. Finally, the receiver informs the sender of successful DATA packet reception by sending back an ACKnowledgment (ACK) packet. Despite its ability to handle *hidden node/exposed node* problem [40], this handshake mechanism incurs non-negligible overhead. IEEE 802.11 activates this mechanism for large packets only.

InterFrame Space (IFS) is an inactivity period at which a Mobile Node must sense before starting/resuming its backoff mechanism. The length of IFS depends on packet types. If the four-way handshake is activated, it will be DIFS (Distributed IFS) for RTS packets, and SIFS (Short IFS) for other packets. Otherwise, it will be DIFS and SIFS for DATA and ACK packets, respectively. Since SIFS is shorter than DIFS, the collision occurs when sending RTS packets only.

In IEEE 802.11, every packet contains a *Network Allocation Vector* (NAV) field. An NAV field contains the duration during which the Mobile Node is expected to take over the channel. Upon overhearing any of these packets, all Mobile Nodes, except the intended receiver, refrain from transmitting any packet until the end of period specified in the NAV.

12.4.2 NS2 Classes `Mac` and `Mac802_11`

In NS2, the IEEE 802.11 MAC is implemented using a C++ class `Mac802_11` which derives from an abstract class `Mac`. Program 12.12 shows the declarations of both the classes `Mac` and `Mac802_11`.

In class `Mac`, variables "`netif_`," "`ll_`," and "`channel_`" are three variables which connect to other objects in the Mobile Node architecture (see Fig. 12.2). The variable "`intr_`" is a dummy variable used with various timer objects. Pointers "`pktRx_`" and "`pktTx_`" point to packets which shall be received and transmitted later (see also Sect. 12.4.5).

In the derived class (`Mac802_11`), two other pointers – "`pktRTS_`" and "`pktCTRL_`" – are also used to store an RTS packet and an CTS packet or an ACK packet, respectively, before packet transmission (see Lines 38 and 39). The variables "`state_`" (Line 13), "`rx_state_`" (Line 32), and "`tx_state_`" (Line 33) indicate the current state of the `Mac` object, and the transmission and reception states of the `Mac802_11` object, respectively. The list of possible states is shown below:

```
MAC_IDLE   MAC_POLLING  MAC_RECV   MAC_SEND   MAC_RTS
MAC_BCN    MAC_CTS      MAC_ACK    MAC_COLL   MAC_MGMT
```

where `MAC_COLL` and `MAC_MGMT` stand for collision and management.

Program 12.12 Declaration of classes `Mac` and `Mac802_11`

```
    //~ns/mac/mac.h
 1  class Mac : public BiConnector {
 2  public:
 3      Mac();
 4      inline int addr() { return index_; }
 5  protected:
 6      int index_;           // MAC address
 7      double bandwidth_;    // Channel bitrate in bps
 8      double delay_;        // MAC overhead
 9      Phy *netif_;          // Network interface object
10      LL *ll_;              // Link layer object
11      Channel *channel_;    // Channel object
12      Event intr_;          // A dummy event
13      MacState state_;      // Current state
14      Packet *pktRx_;       // Cached incoming packet
15      Packet *pktTx_;       // Cached data packet to be TX
16  };

    //~ns/mac/mac-802_11.h
17  class Mac802_11 : public Mac {
18  public:
19      Mac802_11();
20  protected:
21      PHY_MIB phymib_;  //Physical layer MIB
22      MAC_MIB macmib_;  //MAC layer MIB
23  private:
24      RxTimer      mhRecv_;     // It's time to receive
25      TxTimer      mhSend_;     // ReTX timeout, if applicable
26      IFTimer      mhIF_;       // TX complete timer
27      BackoffTimer mhBackoff_;  // Backoff timer
28      DeferTimer   mhDefer_;    // Defer TX by IFS periods
29      NavTimer     mhNav_;      // Network Allocation Vector
30      double       dataRate_;   // Data rate
31      double       nav_;        // Time when NAV expires
32      MacState  rx_state_;      // Incoming state
33      MacState  tx_state_;      // Outgoing state
34      int       tx_active_;     // Is the transmitter ACTIVE?
35      u_int32_t cw_;            // Current contention Window
36      u_int32_t ssrc_;          // Retry Count for a short
                                       packet
37      u_int32_t slrc_;          // Retry Count for a long
                                       packet
38      Packet    *pktRTS_;       // Cached outgoing RTS packet
39      Packet    *pktCTRL_;      // Cached outgoing non-RTS
                                       packet
40  };
```

Lines 21 and 22 declare two Management Information Base (MIB) variables which contain the basic information about the IEEE 802.11 module:

- Class `PHY_MIB` defines physical layer MIB, with the following key variables:

`(CWMin,CWMax)`	The smallest and the largest contention window size
`SlotTime`	The length of a time slot
`SIFSTime`	The length of an SIFS interval

- Class `MAC_MIB` defines MAC layer MIB, with the following key variables:

`RTSThreshold`	Packets whose length is greater than `RTSThreshold` bytes are considered as long packets.
`ShortRetryLimit`	Max. *re*-transmissions for a short packet
`LongRetryLimit`	Max. *re*-transmissions for a long packet

Lines 24–29 define the following six timers which help carry out six time-based actions: Packet reception timer (`mhRecv_`), Retransmission timer (`mhSend_`), Transmission complete timer (`mhIF_`), Backoff timer (`mhBackoff_`), Medium sensing timer (`mhDefer_`), and NAV timer (`mhNav_`).

The variables "`dataRate_`" and "`nav_`" (Lines 30 and 31) define physical data rate in bps, and the duration in seconds during which the node needs to refrain from transmitting and/or receiving. The variable "`tx_active_`" (Line 34) indicates whether the node is currently engaged in packet transmission. A wireless node cannot receive any packet while transmitting due to its self interference. Finally, the variables "`cw_`," "`ssrc_`," and "`slrc_`" store the current contention window size, and the retransmission counters for short and long packets, respectively (Lines 35–37).

12.4.3 Basic Functions of NS2 Classes `Mac` and `Mac802_11`

Classified into five categories, these functions are shown below:

Main functions:

`recv(p,h):`
: Receive a packet `*p` with a handler `*h`.

`send(p,h):`
: Send a packet `*p` with a handler `*h` in a downward direction.

`transmit(p,timeout):`
: Send a packet `*p` to the "`downtarget_`." Start the retransmission timer with a "`timeout`" period.

`collision(p):`
: Called when the packet `*p` collides with the packet being received "`pktRx_`," this function takes actions according to Sect. 12.4.1.

Packet preparation functions:

Functions `sendRTS(dst)`, `sendCTS(dst,dur)`, `sendDATA(p)`, and `sendACK(dst)` prepare and store the RTS, CTS, DATA, and ACK packets in variables `*pktRTS_`, `*pktCTRL_`, `*pktTx_`, and `*pktCTRL_`, respectively.

Packet transmission functions:

Functions `check_pktCTRL()`, `check_pktRTS()`, and `check_pktTx()` inspect the relevant cached CTS/ACK/RTS/DATA packet, and invokes `transmit(p,timeout)` to transmit the packet if the medium is free. Otherwise, it starts the IEEE 802.11 backoff process.

Packet retransmission functions:

Functions `RetransmitRTS()` and `RetransmitDATA()` increment and check whether the retransmission counter (i.e., "`ssrc_`" or "`slrc_`") exceeds the limit. If so, they will drop the packet, and reset the counter and contention window size. Otherwise, it will increment the contention window size, and start the backoff timer.

Packet reception functions:

Helper functions `recvRTS()`, `recvCTS()`, `recvDATA()`, and `recvACK()` are invoked to take necessary actions when one of the RTS/CTS/DATA/ACK packets is received.

Resume functions:

`tx_resume()`: Resume pending transmission;

- *Case 1 (new transmission)*: This function is invoked after the channel is sensed idle for a DIFS period of time. It initiates a backoff process using the backoff timer.
- *Case 2 (continuing transmission)*: This function from within one of the above packet reception functions. It starts the medium sensing timer with a parameter SIFS.

`rx_resume()`: (Pending reception)

This function is invoked when the packet arrives. It sets the state to `MAC_IDLE`, i.e., the Mobile Node is ready to receive another packet.

Idleness functions:

Function `is_idle()` will return 1 (*idle*), if both "`rx_state_`" and "`tx_state_`" are `MAC_IDLE`, and the NAV value stored in the variable "`nav_`" is less than the current time (i.e., medium is idle). Otherwise, it will return 0 (*busy*). In NS2 implementation, *idleness* refers not only to the medium but also to the entire `Mac802_11` object including the transmitting as well as the receiving states.

12.4.4 Timer Concepts for Implementation of IEEE 802.11

IEEE 802.11 implementation relies heavily on timers. To perform an action (e.g., send or receive a packet), a `Mac802_11` records what to do in its variables and starts relevant timers. The action will later be performed when the timer expires. As discussed in Sect. 12.4.2, class `Mac802_11` uses six timers. Table 12.6 shows the implementation details of these timers. Here, the first column shows the class names as well as the names of the `Mac802_11` class variables for each timer. The second column shows the functions of class `Mac802_11` which starts each timer as well as the corresponding timeout. The final column shows the implication of timer expiration as well as the main expiration actions. At the expiration, all the timers reset their variables and invoke a function associated with the attached `Mac802_11` object. Names of the functions as well as the key actions for each timer are shown in this column.

12.4.5 Packet Reception Mechanism of IEEE 802.11

The mechanism of the function `recv(p,h)` of class `Mac802_11` in Program 12.13 proceeds based on the following three cases:

Case 1: Transmitting Packet to Lower Layer Objects Using `send(p,h)`

In this case, the Mobile Node intends to transmit a packet. The packet arrives the `Mac802_11` object from the higher layer. The packet direction would be `DOWN` (Line 4 of Program 12.13), and Line 5 invokes the function `send(p,h)` to send out the packet.

In Program 12.14, the function `send(p,h)` contains instructions for sending a packet `*p` received from the upper layers. This implies that the packet (`*p`) is a new data packet. During packet preparation process, Line 5 reconfigures the packet `*p` and stores it in the variable `*pktTx_`, and Line 6 creates and stores an RTS packet in the variable `*pktRTS_`. Finally, Lines 7–13 start the backoff timers. The transmission of the prepared packet will be carried out once the timer expires.

In Table 12.6, at the expiration, a backoff timer executes the function `backoffHandler()` of the associated `Mac802_11` object (see Lines 15–20 in Program 12.14). Again, at the expiration of a backoff timer, the `Mac802_11` object is allowed to transmit exactly one packet. If either CTS or ACK is waiting to be transmitted in `*pktCTRL_`, this function would do nothing, since their transmission can commence after a period of SIFS, without backing off.[9] In this

[9] If this is the case, the packet would have been transmitted before the timer expiration.

Table 12.6 IEEE 802.11 timers and their starter functions, timeout, and expiration implication and actions

Timer	Starter function and timeout	Expiration implication
Packet reception timer: – Class = `RxTimer` – Variable = `mhRecv_`	Case 1. `collision(p)` Timeout = the max. of transmission time of two colliding packets. Case 2. `recv(p)` Timeout = Transmission time of packet `*p`.	*Implication*: The last bit of the packet has arrived. *Actions*: – `recvHandler()` → `recv_timer()` – Call the relevant packet reception function.
Retransmission timer: – Class = `Txtimer` – Variable = `mhSend_`	`transmit(p,timeout)` Timeout = `timeout`	*Implication*: The packet is lost. *Actions*: – `sendHandler()` →`send_timer():` – If CTS or ACK was transmitted, free the packet. Otherwise, retransmit the RTS or DATA packet using a retransmission function.
Transmission completion timer: – Class = `IFtimer_` – Variable = `mhIF_`	`transmit(p,timeout)` Timeout = Transmission time of packet `*p`	*Implication*: The last bit of the packet is transmitted. *Actions*: – txHandler() → Set `tx_active_` to 0
Medium sensing timer: – Class = `DeferTimer` – Variable = `mhDefer_`	`tx_resume()` Timeout = SIFS	*Implication*: The medium has been idle for a period of SIFS. *Actions*: – `deferHandler()` – Try to transmit a packet using one of the packet transmission functions.
Backoff timer: – Class = `BackoffTimer` – Variable = `mhBackoff_`	Case 1. `send(p,h)` Case 2. `check_pktRTS()`, Case 3. `check_pktTx ()` Case 4. `tx_resume()` Case 5. `recvACK(p)` Timeout = Backoff value	*Implication*: Backoff reaches its zero value. *Actions*: – `backoffHandler()` – Try to transmit a packet using one of the packet transmission functions.
NAV timer: – Class = `NavTimer` – Variable = `mhNav_`	`recv_timer()` Timeout = Case 1: EIFS (transmission error or packet collision) Case 2: NAV (normal packet reception)	*Implication*: The medium is idle. *Actions*: – `navHandler()` – Resume the backoff timer.

Program 12.13 Function `recv(p,h)` of class `Mac802_11`

```
    //~ns/mac/mac-802_11.cc
 1  void Mac802_11::recv(Packet *p, Handler *h)
 2  {
 3      struct hdr_cmn *hdr = HDR_CMN(p);
 4      if(hdr->direction() == hdr_cmn::DOWN) {
 5          send(p,h); return;
 6      }
 7      if(tx_active_ && hdr->error() == 0)
 8          hdr->error() = 1;
 9      if(rx_state_ == MAC_IDLE) {
10          setRxState(MAC_RECV);
11          pktRx_ = p;
12          mhRecv_.start(txtime(p));
13      } else
14          collision(p);
15  }
```

Program 12.14 Functions `send(p,h)` and `backoffHandler()` of class `Mac802_11`

```
    //~ns/mac/mac-802_11.cc
 1  void Mac802_11::send(Packet *p, Handler *h)
 2  {
 3      double rTime;
 4      struct hdr_mac802_11* dh = HDR_MAC802_11(p);
 5      sendDATA(p);
 6      sendRTS(ETHER_ADDR(dh->dh_ra));
 7      if(mhBackoff_.busy() == 0)
 8          if(is_idle())
 9              if (mhDefer_.busy() == 0)
10                  mhBackoff_.start(cw_, is_idle(),
11                                   phymib_.getDIFS());
12          else
13              mhBackoff_.start(cw_, is_idle());
14  }

15  void Mac802_11::backoffHandler()
16  {
17      if(pktCTRL_)  return;
18      if(check_pktRTS() == 0)  return;
19      if(check_pktTx() == 0)   return;
20  }
```

case, Line 17 would return here. On the other hand, if the `Mac802_11` object contains neither CTS nor ACK packets, Lines 18 and 19 will, in sequence, check and try to transmit either an RTS packet or a DATA packet, respectively.

Case 2: Receiving a Packet from a Lower Layer Object

In this case, a packet reaches the *idle* Mobile Node from the air interface. The packet direction would be `up`. From Program 12.13, after checking for idleness in Line 9, Line 10 sets the receiving states to `MAC_RECV`, Line 11 stores the incoming packet in the variable `*pktRx_`, and starts the packet reception timer "`mhRecv_`" with the timeout being packet transmission time (i.e., `txttime(p)` in Line 12).

From Table 12.6, the expiration action is to execute function `recv_timer()` whose details are shown in Program 12.15. Lines 23–38 determine the packet type and invoke relevant packet reception functions. These functions clean up the current transmission variables and prepare the next packet for transmission, if any. Finally, Line 40 clears the variable "`pktRx_`" and calls the function `rx_resume()` which sets receiving state to `MAC_IDLE` (Line 44).

Case 3: Self-Interference and/or Collision

This case is complementary to Case 2. Packets arrive from the air interface but are unsuccessfully received. There are two causes of error: Self-interference and packet collision. Consider Program 12.13. Lines 7 and 8 mark the packet to be in error due to self-interference,[10] if the transmitter of the `Mac802_11` object is active, while receiving a packet.

Packet collision occurs since the `Mac802_11` is busy while receiving a packet. Here, a packet `*p` collides with the packet under reception `*pktRx_`. Under a noncapturing model, both the packets would be lost. But the loss would not be realized immediately. It would be realized once the `Mac802_11` object receives the entire packet and is unable to understand the contaminated packet.

Program 12.16 shows the details of function `collision(p)`. In Lines 7–15, this function drops the shorter packet, keeps the longer packet in `*pktRx_`, and sets "`rx_state_`" to `MAC_COLL`. Again, once the packet reception timer expires, the function `recv_timer()` would be executed, and Lines 11–15 of Program 12.15 will drop the packet `*pktRx_` since the receiving state was set to `MAC_COLL`.

12.4.6 Implementation of Packet Retransmission in NS2

Class `Mac802_11` uses a retransmission timer, (i.e., `mhSend_` in Table 12.6) stored in a class variable "`mhSend_`" to control packet retransmission. This timer starts every time a packet is transmitted using the function `transmit(p,timeout)`. It is stopped upon the reception of expected packets using functions `recvCTS(p)`, `recvDATA(p)`, and `recvACK(p)`. If the retransmission

[10] Self-interference is a wireless property, where the transmitting signal interferes with the receiving signal.

Program 12.15 Functions `recv_timer()` and `rx_resume()` of class `Mac802_11`

```
    //~ns/mac/mac-802_11.cc
 1  void Mac802_11::recv_timer()
 2  {
 3      hdr_cmn *ch = HDR_CMN(pktRx_);
 4      hdr_mac802_11 *mh = HDR_MAC802_11(pktRx_);
 5      u_int32_t dst = ETHER_ADDR(mh->dh_ra);
 6      u_int8_t  type = mh->dh_fc.fc_type;
 7      u_int8_t  subtype = mh->dh_fc.fc_subtype;
 8      if(tx_active_) {
 9          Packet::free(pktRx_); goto done;
10      }
11      if(rx_state_ == MAC_COLL) {
12          discard(pktRx_, DROP_MAC_COLLISION);
13          set_nav(usec(phymib_.getEIFS()));
14          goto done;
15      }
16      if( ch->error() ) {
17          Packet::free(pktRx_);
18          set_nav(usec(phymib_.getEIFS()));
19          goto done;
20      }
21      if(dst != (u_int32_t)index_)
22          set_nav(mh->dh_duration);
23      switch(type) {
24          case MAC_Type_Control:
25              switch(subtype) {
26                  case MAC_Subtype_RTS:
27                      recvRTS(pktRx_);break;
28                  case MAC_Subtype_CTS:
29                      recvCTS(pktRx_);break;
30                  case MAC_Subtype_ACK:
31                      recvACK(pktRx_);break;
32              }; break;
33          case MAC_Type_Data:
34              switch(subtype) {
35                  case MAC_Subtype_Data:
36                      recvDATA(pktRx_);break;
37              }; break;
38      }
39      done:
40      pktRx_ = 0;
41      rx_resume();
42  }

43  void Mac802_11::rx_resume()
44   setRxState(MAC_IDLE);
45  }
```

Program 12.16 Function `collision()` of class `Mac802_11`

```
    //~ns/mac/mac-802_11.cc
 1  void Mac802_11::collision(Packet *p)
 2  {
 3      switch(rx_state_) {
 4          case MAC_RECV:
 5              setRxState(MAC_COLL);
 6          case MAC_COLL:
 7              if(txtime(p) > mhRecv_.expire()) {
 8                  mhRecv_.stop();
 9                  discard(pktRx_, DROP_MAC_COLLISION);
10                  pktRx_ = p;
11                  mhRecv_.start(txtime(pktRx_));
12              }
13              else {
14                  discard(p, DROP_MAC_COLLISION);
15              }
16      }
17  }
```

Program 12.17 Function `send_timer()` of class `Mac802_11`

```
    //~ns/mac/mac-802_11.cc
 1  void Mac802_11::send_timer(){
 2     switch(tx_state_) {
 3        case MAC_RTS:
 4           RetransmitRTS(); break;
 5        case MAC_CTS:
 6           Packet::free(pktCTRL_);
 7           pktCTRL_ = 0; break;
 8        case MAC_SEND:
 9           RetransmitDATA(); break;
10        case MAC_ACK:
11           Packet::free(pktCTRL_);
12           pktCTRL_ = 0; break;
13        case MAC_IDLE:
14           break;
15     }
16     tx_resume();
17  }
```

is not stopped before its expiration, function `send_timer()` will be invoked to retransmit the packet.

Program 12.17 shows details of the function `send_timer()`. Lines 2–15 prepares a packet to transmit, and Line 16 calls the function `tx_resume()` to resume the pending backoff process. The details of the function `tx_resume()` are shown in Program 12.18.

For the packet preparation process, Line 2 determines the value stored in the variable "`tx_state_`." If the value of `tx_state_` is either `MAC_RTS`

Program 12.18 Function `tx_resume()` of class `Mac802_11`

```
    //~ns/mac/mac-802_11.cc
 1  void Mac802_11::tx_resume()
 2  {
 3     double rTime;
 4     if(pktCTRL_)
 5        mhDefer_.start(phymib_.getSIFS());
 6     else if(pktRTS_)
 7        if (mhBackoff_.busy() == 0)
 8           mhBackoff_.start(cw_, is_idle(),
                              phymib_.getDIFS());
 9     else if(pktTx_)
10        if (mhBackoff_.busy() == 0) {
11           hdr_cmn *ch = HDR_CMN(pktTx_);
12           struct hdr_mac802_11 *mh = HDR_MAC802_11(pktTx_);
13           if ((u_int32_t) ch->size() <
               macmib_.getRTSThreshold()
             ||(u_int32_t) ETHER_ADDR(mh->dh_ra) ==
               MAC_BROADCAST)
14              mhBackoff_.start(cw_, is_idle(),
                                 phymib_.getDIFS());
15           else
16              mhDefer_.start(phymib_.getSIFS());
17        };
18     setTxState(MAC_IDLE);
19  }
```

or `MAC_SEND` – meaning either RTS or DATA packets were transmitted and not acknowledged – Lines 4 and 9 will invoke the relevant functions for packet retransmission. These two functions first increment the retransmission counter (i.e., either "`ssrc_`" or "`slrc_`") by one, check its value against the retry limit, and retransmit/drop the packet. Note that CTS and ACK packets require no acknowledgment. Therefore, Lines 6–7 and 11–12 simply destroy the packet stored in `*pktCTRL_`.

12.4.7 Implementation of Carrier-Sensing, Backoff, and NAV

12.4.7.1 Basic Carrier Sensing

NS2 implements medium sensing and backoff mechanism using timers stored in variables "`mhDefer_`" and "`mhBackoff_`," respectively. From Table 12.6, these two timers can be started from within one of five functions of class `Mac802_11`. The details about five timer-starter functions are shown in Table 12.7.

Carrier sensing in NS2 takes one of the following three values. First, for a new transmission, the duration is the addition of DIFS and the current content window

Table 12.7 Initiation details for backoff and medium sensing timers

Functions where the timer is started	Timer type	Timer duration when the MAC layer is		Functions invoked before the timer is started
		IDLE	BUSY	
`send(p,h)`	`mhBackoff_`	`cw_`+DIFS	`cw_`	None
`check_pktTx()` `check_pktRTS()`	`mhBackoff_`	0	`cw_`	`inc_cw()`
`recvAck()`	`mhBackoff_`	`cw_`	`cw_`	`rst_cw()`
`tx_resume()`				
– CTS/ACK	`mhDefer_`	SIFS	SIFS	None
– RTS	`mhBackoff_`	`cw_`+DIFS	`cw_`+DIFS	None
– DATA (short)	`mhBackoff_`	`cw_`+DIFS	`cw_`+DIFS	None
– DATA (long)	`mhDefer_`	SIFS	SIFS	None

(`cw_`). This is the case for a transmission of RTS or short DATA packet where the RTS/CTS handshake is not required before packet transmission. Second, after the first packet transmission, the `Mac802_11` takes over the medium by reducing the medium sensing time from DIFS to SIFS. This is the case for CTS packets, ACK packets, and long DATA packets. Finally, after a successful packet transmission, an ACK packet is received. Here, the `Mac802_11` resets the contention window using the function `rst_cw()`, and starts backing off for a period of "`cw_`" before being able to commence another packet transmission.

The expiration of the above two timers signifies the end of the medium sensing and backoff periods. At the expiration, the `Mac802_11` object is allowed to transmit one packet. From Table 12.6, at the expiration, backoff and medium sensing timers execute the functions `backoffHandler()` and `deferHandler()`, respectively of class `Mac802_11`.

12.4.7.2 Pausing and Resuming Backoff Timer

IEEE 802.11 decreases the backoff counter for every idle time slot. Therefore, the C++ class `BackoffTimer` implements the concept of *pausing* and *resuming* as follows:

- Before backoff: The second input argument of function `start(cw,idle, dur)` of class `BackoffTimer` indicates whether the associated `Mac802_11` is idle. If so, the backoff process would proceed as normal. Otherwise, the `BackoffTimer` would just pause the timer, but would not place a timer expiration event on the simulation time line.
- Periodic backoff status check: Class `Mac802_11` defines a function `checkBackoffTimer()` which pauses and resumes the backoff timer "`mhBackoff_`," if the `Mac802_11` object is busy and idle, respectively. This

function is called for every time the receiving (i.e., `setRxState(s)`) and sending (i.e., `setTxState(s)`) states of the `Mac802_11` change.[11]

12.4.7.3 Network Allocation Vector

Network Allocation Vector is the duration during which the medium is expected to be busy. This duration takes one of the two following values:

- An advertised NAV value: Upon overhearing a packet intended to other nodes, the `Mac802_11` object can extract and use a value in the field "`dh_duration`" as its NAV value (see Lines 21 and 22 in Program 12.15).
- EIFS: If the received packet is in error or the collision has occurred, the `Mac802_11` cannot extract an NAV value from the packet. In this case, it uses EIFS as a default NAV value (see Line 13 and 18 in Program 12.15).

Class `Mac802_11` defines a function `set_nav(nav)` to set its NAV value and to start the `NavTimer` object. Once expired, the `NavTimer` object checks whether the `Mac802_11` is idle, and resume the `BackoffTimer`, if so.

12.5 Physical Layer: Physical Network Interfaces and Channel

Located at the bottom of Fig. 12.2, these two Mobile Node components represent the physical layer. Physical network interfaces are the hardware (e.g., radio modem, antenna) which creates and sends out data bits, while channels model the medium shared by all Mobile Nodes.

12.5.1 Physical Network Interface

Program 12.19 shows declaration of an abstract class `Phy` and its derived class, namely, `WirelessPhy`. These two classes model transmitting and receiving hardware. The class `Phy` transmits/receives data from a `*node_` object in Line 8 to the `*channel_` object in Line 10, using the bit rate specified by "`bandwidth_`" in Line 9. Deriving from class `BiConnector`, class `Phy` contains two pointers "`uptarget_`" and "`downtarget_`." While the "`uptarget_`" points to the upper Mobile node component, "`downtarget_`" is not in use. Class `Phy` instead uses its pointer "`channel`" to refer to the attached channel.

[11] The state changes from within the functions `recv(p,h)`, `collision(p)`, `rx_resume()`, `tx_resume()`, and all packet transmission functions.

Program 12.19 Declaration of classes `Phy` and `WirelessPhy`

```
    //~ns/mac/phy.h
 1  class Phy : public BiConnector {
 2  public:
 3     Phy();
 4     void    recv(Packet* p, Handler* h);
 5     virtual void sendDown(Packet *p)=0;
 6     virtual int sendUp(Packet *p)=0;
 7  protected:
 8     Node*   node_;       // The owner of this netif
 9     double  bandwidth_; // Bit rate in bps
10     Channel *channel_;  // The channel for output
11  };

    //~ns/mac/wireless-phy.h
12  class WirelessPhy : public Phy {
13  public:
14     void sendDown(Packet *p);
15     int  sendUp(Packet *p);
16  protected:
17     double  Pt_;        // TX signal power (W)
18     double  freq_;      // Signal frequency
19     double  lambda_;    // Signal wavelength (m)
20     double  L_;         // System loss factor
21     double  CSThresh_; // Carrier sense threshold (W)
22     double  RXThresh_; // Receive power threshold (W)
23     Antenna *ant_;      // Antenna
24     Propagation *propagation_;  // Propagation Model
25     Modulation  *modulation_;   // Modulation module
26  };
```

The packet reception function `recv(p,h)` (Line 4) is invoked to receive packets coming from both upper and lower layers. The class `Phy` contains two pure virtual functions – `sendDown(p)` and `sendUp(p)` – which shall be overridden by its derived class `WirelessPhy` (Lines 5 and 6).

Class `WirelessPhy` derives from class `Phy`. It overrides the functions `sendDown(p)` and `sendUp(p)`, which send the packets downward and upward (Lines 14 and 15), respectively. It also defines several wireless properties such as transmission power (`Pt_`) or transmitting signal frequency (`freq_`) as shown in Lines 17 and 18, respectively.

Class `WirelessPhy` defines two important thresholds: a carrier-sensing threshold (`CSThresh_` in Line 21) and a packet reception threshold (`RXThresh_` in Line 22). If the received signal is below "`CSThresh_`," it is considered undetectable. If, on the other hand, the signal is greater than "`CSThresh_`" but still below "`RXThresh_`," it is detectable but the received packet is considered to be in error. Only when the signal is greater than "`RXThresh_`" will the packet be considered successfully received.

Program 12.20 Declaration of classes `Channel` and `WirelessChannel`

```
    //~ns/mac/channel.h
 1  class Channel : public TclObject {
 2  public:
 3     Channel(void);
 4     struct if_head   ifhead_; // Listening phy. netif.
 5     virtual void recv(Packet* p, Handler*);
 6  };

 7  class WirelessChannel : public Channel{
 8  public:
 9     WirelessChannel(void);
10  private:
11     MobileNode *xListHead_; //Listening nodes
12     int numNodes_;          //Number of listening nodes
13     void sendUp(Packet* p, Phy *txif);
14     MobileNode **getAffectedNodes(MobileNode *mn,
                      double radius, int *numAffectedNodes);
15     void addNodeToList(MobileNode *mn);
16     void removeNodeFromList(MobileNode *mn);
17  };
```

12.5.2 Wireless Channels

C++ class `Channel` and its derived class `WirelessChannel` model share physical medium. As shown in Program 12.20, class `Channel` has one important variable "`if_head`" (Line 4) which is the head of a link list containing all physical network interface listening on this channel. It has one important function `recv(p)`.

Class `WirelessChannel` receives packets via the function `recv(p,h)`. Upon receiving a packet, it configures and returns the packet to all listening nodes using function `sendUp(p,txif)`, where `*txif` is a wireless transmitting physical interface object (Lines 13).

Class `WirelessChannel` stores all listening nodes in a link list whose head is stored in its variable "`xListHead_`" (Line 11). The total number of listening nodes is denoted by the variable "`numNodes_`" (Line 12). Class `WirelessChannel` also provides a public function `getAffectedNodes(mn,radius,num_an)` returns a pointer to a list of Mobile Node affected by transmission of a Mobile Node `*mn` whose transmission range is "`radius`." The resulting number of affected nodes are stored in `*numAffectedNodes`.

12.5.3 Sender Operations at the Physical Layer

On the sender side, a higher-layer Mobile Node component (a `Mac` object in most cases) sends a packet to the physical network interface. The network interface sets up physical layer parameters and forwards the packet to the channel. This operation begins with the execution of function `recv(p,h)` of the Class `Phy`.

Program 12.21 Function `recv(p)` of class `Phy`, function `sendDown(p)` of class `WirelessPhy`, and function `recv(p,h)` of class Channel

```
    //~ns/mac/phy.cc
 1  void Phy::recv(Packet* p, Handler*)
 2  {
 3     struct hdr_cmn *hdr = HDR_CMN(p);
 4     switch(hdr->direction()) {
 5        case hdr_cmn::DOWN :
 6           sendDown(p); return;
 7        case hdr_cmn::UP :
 8           if (sendUp(p) == 0) {
 9              Packet::free(p);
10              return;
11           } else
12              uptarget_->recv(p, (Handler*) 0);
13     }
14  }

    //~ns/mac/wireless-phy.cc
15  void WirelessPhy::sendDown(Packet *p)
16  {
17     p->txinfo_.stamp((MobileNode*)node(), ant_->copy(),
          Pt_, lambda_);
18     channel_->recv(p, this);
19  }

    //~ns/mac/channel.cc
20  void Channel::recv(Packet* p, Handler* h)
21  {
22      sendUp(p, (Phy*)h);
23  }
```

Program 12.21 shows the details of the function `recv(p,h)`. Lines 3 and 4 determine the packet direction. For downward packet transmission, Line 6 executes "`sendDown(p)`" and returns. As shown in Lines 15–19, the function `sendDown(p)` embeds various information – including the sending node, antenna, transmission power, and signal wavelength – in the packet header (Line 17), and sends the packet to the attached "`channel_`" (Line 18). Upon receiving a packet, the "`channel_`" starts executing receiver operations.

12.5.4 Receiver Operations at the Physical Layer

On the receiving side, the process starts when the channel receives a packet via its function `recv(p)`. From Lines 20–23 of Program 12.21, the function `recv(p)` simply calls function `sendUp(p,h)` of class `WirelessChannel` to send the packet upward.

Program 12.22 Function `sendUp(p,tifp)` of class `WirelessChannel`

```
    //~ns/mac/channel.cc
 1  void WirelessChannel::sendUp(Packet* p, Phy *tifp)
 2  {
 3      Scheduler &s = Scheduler::instance();
 4      Phy *rifp = ifhead_.lh_first;
 5      Node *tnode = tifp->node();
 6      Node *rnode = 0;
 7      Packet *newp;
 8      double propdelay = 0.0;
 9      struct hdr_cmn *hdr = HDR_CMN(p);
10      MobileNode *mtnode = (MobileNode *) tnode;
11      MobileNode **affectedNodes;// **aN;
12      int numAffectedNodes = -1, i;
13      hdr->direction() = hdr_cmn::UP;
14      affectedNodes = getAffectedNodes(mtnode, distCST_ +
          5, &numAffectedNodes);
15      for (i=0; i < numAffectedNodes; i++) {
16          rnode = affectedNodes[i];
17          if(rnode == tnode)
18              continue;
19          newp = p->copy();
20          propdelay = get_pdelay(tnode, rnode);
21          rifp = (rnode->ifhead()).lh_first;
22          for(; rifp; rifp = rifp->nextnode())
23              s.schedule(rifp, newp, propdelay);
24      }
25      delete [] affectedNodes;
26      Packet::free(p);
27  }
```

Shown in Program 12.22, the function `sendUp(p,tifp)` sends the packet `*p` to the all affected nodes. Line 13 changes the packet direction to upward. Line 14 retrieves the list of affected nodes and stores the head pointer in a variable "`affectedNodes`." Lines 15–24 configure and copy the incoming packets to all applicable network interfaces. Finally, after sending copies of the incoming packet to all the affected nodes, Line 26 destroys the incoming packet.

The packet copying and forwarding process in Lines 15–24 are executed for each of the affected nodes. Lines 16–18 skip the loop if the current affected node (`rnode`) is the node which sent this packet (`tnode`). Line 19 creates a copy of the incoming packet. Line 20 computes propagation delay from the sending node to the affected node. Lines 21–23 send the copied packet to all physical network interface of the affected node with propagation delay "`propdelay`" seconds.

Line 23 does not immediately send out the packet. Rather, it schedules a packet reception event at the `WirelessPhysical` object at "`propdelay`" seconds in future. When the packet reception event is dispatched, the copied packet is delivered to the physical network interface via the function `recv(p)` defined in the class `Phy` (see Program 12.21).

Program 12.23 Function `sendUp(p)` of class `WirelessPhy`

```
    //~ns/mac/wireless-phy.cc
 1  int WirelessPhy::sendUp(Packet *p)
 2  {
 3      PacketStamp s;
 4      double Pr; int pkt_recvd = 0;
 5      Pr = p->txinfo_.getTxPr();
 6      if(propagation_) {
 7          s.stamp((MobileNode*)node(), ant_, 0, lambda_);
 8          Pr = propagation_->Pr(&p->txinfo_, &s, this);
 9          if (Pr < CSThresh_) {
10              pkt_recvd = 0;
11              goto DONE;
12          }
13          if (Pr < RXThresh_) {
14              hdr_cmn *hdr = HDR_CMN(p);
15              hdr->error() = 1;
16          }
17      }
18      if(modulation_) {
19          hdr_cmn *hdr = HDR_CMN(p);
20          hdr->error() = modulation_->BitError(Pr);
21      }
22      pkt_recvd = 1;
23  DONE:
24      return pkt_recvd;
25  }
```

On the receiver side, the direction of incoming packet is upward. From Program 12.21, Line 8 executes function `sendUp(p)` to prepare the packet `*p` for reception. The function `sendUp(p)` returns the number of received packets whose signal is sufficiently strong. If the number is greater than zero, Line 12 will deliver the packet to the upper layer objects (e.g., a `Mac` object). Otherwise, Lines 9 and 10 will destroy the packet and return.

Program 12.23 shows the details of function `sendUp(p)` of class `Wireless Phy`. Line 5 first retrieves the transmission power from the packet header and stores the retrieved power in a local variable "`Pr`." If the propagation model exists, Line 8 computes the received power and stores the result in a local variable "`Pr`." Lines 9 and 13 check whether the received signal power exceeds the predefined thresholds. If below the carrier sense threshold (i.e., `CSThresh_`), the signal is considered undetectable. In this case, Line 10 would set the number of received packets to be zero. If, on the other hand, the signal is detectable but still less than a receiving threshold (i.e., `RXThresh_`), the packet will be marked as in error (Line 15). Next, if the modulation scheme exists, Line 20 will again check whether the assumed modulation scheme can tolerate the signal weakness, and update the error flag on the packet header. If detectable, regardless of whether they are in error, the packet will be delivered to and handled later by upper layer objects. In this case, the variable "`pkt_recvd`" is set to 1.

Program 12.24 Mobility configuration of mobile nodes

```
     //~ns/tcl/ex/simple-wireless.tcl
  1  set topo [new Topography]
  2  $topo load_flatgrid 500 500
  3  create-god $val(nn)
  4  for {set i 0} {$i < $val(nn) } {incr i} {
  5      set node_($i) [$ns_ node]
  6      $node_($i) random-motion 0;# disable random
                                                motion
  7  }
  8  $node_(0) set X_ 5.0
  9  $node_(0) set Y_ 2.0
 10 $node_(0) set Z_ 0.0
 11 $node_(1) set X_ 390.0
 12 $node_(1) set Y_ 385.0
 13 $node_(1) set Z_ 0.0
 14 $ns_ at 50.0 "$node_(1) setdest 25.0 20.0 15.0"
 15 $ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"
 16 $ns_ at 100.0 "$node_(1) setdest 490.0 480.0 15.0"
```

12.6 An Introduction to Node Mobility

Introduced earlier in Sect. 12.1.1, class `MobileNode` defines several attributes to support mobility. In Program 12.1, the variables "`X_`," "`Y_`," and "`Z_`" (Lines 4) represent the current node coordinate. The variable "`speed_`" (Line 5) is the node speed in meters per second. These four variables are bound to the instvars in the OTcl domain with the same name. By default, a Mobile Node updates its position for every interval of "`position_update_interval_`" seconds (Line 10). Line 14 defines a pointer "`T_`" to a `Topography` object which defines the area where the node is moving. Finally, Line 15 stores the head pointer "`link_`" of a global link list which contains pointers to all active Mobile Nodes. Note that although defined, the "Z" dimension is not used in NS2.

12.6.1 Basic Mobility Configuration

As shown in Program 12.24, the basic mobility configuration consists of four main steps:

- *Step 1 – Topology creation*: Line 1 creates a `Topology` object and Line 2 identifies the area where the Mobile Node will move during the simulation.

- *Step 2 – GOD configuration*: Line 3 creates a GOD object, and informs the GOD object that the simulation contains `$val(nn)` nodes.[12]
- *Step 3 – Location initialization*: Lines 8–13 specify node location in all three dimensions.
- *Step 4 – Mobility pattern specification*: Line 6 indicates that the mobility model in this simulation will be deterministic. Lines 14–16 specify how the Mobile Nodes move using the instproc `setdest{...}` of the OTcl class `MobileNode` whose syntax is as follows:

  ```
  $node setdest <dest_x> <dest_y> <speed>
  ```

  This OTcl command tells the Mobile Node `$node` to move toward the destination (`<dest_x>`,`<dest_y>`) with speed "`<speed>`" meters per second, by setting the variables "`destX_`," "`destY_`," and "`speed_`" of the C++ `MobileNode` object, respectively.

12.6.2 General Operation Director

GOD stands for General Operation Director (GOD) is an omniscient entity which knows all the environmental information, especially that which should not made available to simulation objects. GOD helps simplify system analysis in some case where we assume perfect information (e.g., perfect channel state information (CSI)).

In mobile networking, GOD precomputes the movement of all the nodes and the distance (in hops) between two nodes before simulation. At run-time, simulation objects request the distance between a pair of nodes, only when necessary. This reduces the need to simulate node movement in real-time, thereby greatly decreasing the usage of simulation resource (e.g., CPU time and memory). Despite its usefulness, programmers should be cautious, not to make information available to simulation objects. For example, Mobile Nodes should not retrieve the network topology map from GOD. Rather, they should use the underlying routing protocol for this purpose.

Again, GOD is configured using the following global procedure:

```
create-god <num_nodes>
```

where `<num_nodes>` is the number of mobile nodes in a simulation (see the details in Lines 1–8 Program 12.25).

Lines 9–19 in Program 12.25 show a part of class `God` declaration. From Line 16, the variable "`instance_`" is declared as `static` to guarantee the uniqueness of the one and only one GOD object in a simulation. The variable "`num_nodes`"

[12] See GOD description in Sect. 12.6.2.

Program 12.25 The global procedure `create-god` and the declaration of a C++ class `God`

```
    //~ns/tcl/mobility/com.tcl
 1  proc create-god { nodes } {
 2      set god [God info instances]
 3      if { $god == "" } {
 4          set god [new God]
 5      }
 6      $god num_nodes $nodes
 7      return $god
 8  }

    //~ns/mobile/god.h
 9  class God : public BiConnector {
10  public:
11      int hops(int i, int j){
12          return min_hops[i * num_nodes + j];
13      };
14      int nodes() { return num_nodes; }
15  private:
16      static God* instance_;
17      int num_nodes;
18      int* min_hops; //for i*num_nodes+j
19  };
```

(Line 17) stores the number of Mobile Nodes in a simulation. The number of hops between nodes "`i`" and "`j`" is stored in the (`i`×`num_nodes`)+`j`th element of the array "`min_hop`" (Line 17).

Class `God` contains two important public functions. One is function `nodes()` which returns the number of Mobile Nodes in the simulation (Line 14). Another is function `hop(i,j)` which returns the distance (in hops) between node "`i`" and node "`j`" (Lines 11–13). At run-time, NS2 may invoke these two functions as needed.

It is important to know, that despite not in use, GOD is a mandatory object for simulation of a mobile network. During the Network Configuration Phase, an MAC object reads the number of nodes from the GOD object (see Line 33 in Program 12.6). If the GOD object is not initialized, a run-time error message will appear on the screen.

12.6.3 Random Mobility

NS2 supports deterministic and random mobility. Section 12.6.1 shows how deterministic mobility can be set up using an OTcl command `setdest{...}` of class `MobileNode`. This section focuses on the other approach: Random mobility.

Program 12.26 Enabling random motion: Function `start()` of class `MobileNode` and function `handle(e)` of class `PositionHandler`

```
    //~ns/common/mobile-node.cc
1   void MobileNode::start()
2   {
3       Scheduler& s = Scheduler::instance();
4       random_position();
5       random_destination();
6       s.schedule(&pos_handle_,&pos_intr_,
          position_update_interval_);
7   }

8   void PositionHandler::handle(Event*)
9   {
10      Scheduler& s = Scheduler::instance();
11      node->update_position();
12      node->random_destination();
13      s.schedule(&node->pos_handle_, &node->pos_intr_,
                              node->position_update_interval_);
14  }
```

Random mobility can be deactivated/activated using the following two OTcl commands of class `MobileNode`:

```
$node random-motion <flag>
$node start
```

The upper OTcl command stores `<flag>` in the variable "`random_motion_`" of class `MobileNode`. The lower invokes the function `start()` of the `MobileNode` object, whose details are shown in Program 12.26.

From within the function `start()`, Lines 4 and 5 invoke functions `random_position()` and `random_destination()`, respectively. These two functions randomize the current location and the destination, respectively, of the Mobile Node. Line 6 sets the position handler "`pos_handle_`" (see also Line 11 in Program 12.1) to expire after a period of "`position_update_interval_`." At the expiration, the process repeats by updating node position, computing a randomized destination, and setting the position handler to expire after the same interval (Lines 8–14).

12.6.4 Mobility and Traffic Generators: Standalone Helper Utility

The benefits of deterministic mobility is that programmers have full control over where and how the Mobile Nodes move during simulation. But as the number of Mobile Nodes increases, it becomes increasingly tedious to specify destinations for all the nodes. Although random mobility could solve this problem, it does not allow programmers to review or control how the Mobile Nodes move.

12.6.4.1 Mobility Generation Utility "setdest"

NS2 provides a "setdest" shell utility[13] which creates deterministic mobility statements, which can be inserted into a Tcl simulation script.

Written in C++, the executable "setdest" is located in the directory ~*ns*/indep-utils/cmu-sen-gen/setdest. NS2 provides two versions of the utility:

- Version 1 (developed by Carnegie Mellon University):

```
>>./setdest -v <version>    -n <num_nodes>  -t <sim_time>
            -M <max_speed>  -p <pause_time>
            -x <max_x>      -y <max_y>
```

- Version 2 (developed by University of Michigan)

```
>>./setdest -v <version>    -n <num_nodes> -t <sim_time>
            -s <speed_type> -m <min_speed> -M <max_speed>
            -P <pause_type> -p <pause_time>
            -x <max_x>      -y <max_y>
```

  - Speed: Here <speed_type> can be either "uniform" or "normal," for uniform or normal distributions, respectively. In case of "normal," the randomized speed values are taken from a truncated normal distribution with mean $\overline{s}$ and standard deviation σ_s, where

$$\overline{s} = \frac{\text{max\_speed+max\_speed}}{2}, \sigma_s = \frac{\text{max\_speed-max\_speed}}{4}$$

  - Pause time: Here <pause_type> can be "constant" or "uniform." These two cases set the pause time to be a constant value of pause_time and the value uniformly distributed within [0,pause_time], respectively.

Example 12.2. Consider the following shell statement:

```
>>./setdest -n 2 -p 10 -M 10 -t 1000 -x 500 -y 500
# nodes: 2, pause: 10.00, max speed: 10.00,
                  max x: 500.00, max y: 500.00
#
$node_(0) set X_ 278.612941841477
$node_(0) set Y_ 236.713393413552
$node_(0) set Z_ 0.000000000000
$node_(1) set X_ 279.050029009209
$node_(1) set Y_ 458.932826402138
$node_(1) set Z_ 0.000000000000
$god_ set-dist 0 1 1
$ns_ at 10.00 "$node_(0) setdest 173.450 234.76 5.03"
$ns_ at 10.00 "$node_(1) setdest 13.85 99.32 9.88"
```

[13]This utility has the same name as the OTcl command setdest{...} of class Node/MobileNode.

```
$ns_ at 30.87 "$node_(0) setdest 173.45 234.76 0.00"
...
$ns_ at 992.49 "$node_(0) setdest 160.96 161.81 0.00"
```

where the first line is provided by the user, and the subsequent lines are the mobility-related NS2 statements created by the utility "`setdest`."

This statement creates mobility statements for two Mobile Nodes with pause time of 10 s (constant). The maximum speed is 10 m per second, and the simulation time is 1,000 s. The topology ranges from 0 to 500 (in meter) on both the X-axis and the Y-axis. □

12.6.4.2 Traffic Generation Utility "`cbrgen.tcl`"

NS2 also provides another independent utility "`cbrgen.tcl`," written in Tcl, to create traffic-related OTcl statements:

```
>>ns cbrgen.tcl -type <cbr|tcp> -nn <num_nodes>
                -seed <seed> -mc <max_conn> -r <rate>
```

Unlike the mobility generation utility, this utility "`cbrgen.tcl`" is an NS script and needs to be invoked through the interpreter, which, in this case, is the executable "`ns`." Despite its name, this utility can create both TCP and CBR traffic by specifying either "`cbr`" or "`tcp`" after the option `-type`. Other options include the number of Mobile Nodes in the simulation (`<num_nodes>`), seed (`<seed>`), the maximum of connections that will be generated (`<max_conn>`), and the data rate in bps for CBR traffic (`<rate>`). An example use of this utility is shown below.

Example 12.3. Consider the following shell statement:

```
>>ns cbrgen.tcl -type cbr -nn 10 -seed 1.0 -mc 45
                                  -rate 4.0
# nodes: 10, max conn: 45, send rate: 0.25, seed: 1.0
# 1 connecting to 2 at time 2.5568388786897245
set udp_(0) [new Agent/UDP]
$ns_ attach-agent $node_(1) $udp_(0)
set null_(0) [new Agent/Null]
$ns_ attach-agent $node_(2) $null_(0)
set cbr_(0) [new Application/Traffic/CBR]
$cbr_(0) set packetSize_ 512
$cbr_(0) set interval_ 0.25
$cbr_(0) set random_ 1
$cbr_(0) set maxpkts_ 10000
$cbr_(0) attach-agent $udp_(0)
$ns_ connect $udp_(0) $null_(0)
$ns_ at 2.5568388786897245 "$cbr_(0) start"
#
# 4 connecting to 5 at time 56.333118917575632
```

```
set udp_(1) [new Agent/UDP]
...
$ns_ connect $udp_(8) $null_(8)
$ns_ at 31.464945688594575 "$cbr_(8) start"
#
#Total sources/connections: 6/9
```

where, again, the first and the subsequent lines are supplied by the users and the resulting traffic-related NS2 statements are created by the utility "`cbrgen.tcl`."

The first statement specifies CBR traffic for ten Mobile Nodes. The seed is set to `1.0`. The maximum number of connections is `45`. The CBR rate is `4.0` bps. □

12.6.4.3 Working with Scenario Files

While the above two standalone utilities are useful, it is quite hard to use them because the results are displayed on the screen. It is more convenient to redirect the output to a scenario file using ">" or ">>" instructions (see Sect. A.3.5). For example, the following two shell statements can be executed from the directory *~ns*/indep-utils/cmu-sen-gen/setdest.

```
>>./setdest -n 2 -p 10 -M 10 -t 1000 -x 500 -y 500 >
      myfile
>>ns cbrgen.tcl -type cbr -nn 2 -seed 1.0 -mc 4
                                    -rate 4.0 > myfile
```

These two statements create and store NS2 mobility and traffic statements in *scenario* files whose name is "`myfile`." Examples of built-in scenario files can be found in the directory *~ns*/tcl/mobility/scene.

We conclude this section with the summary of how to configure mobility/traffic using scenario files.

Step 1: Run the stand-alone utilities to creates scenario files (e.g., `myfile`).
Step 2: Configure a Tcl Simulation Script as usual.
Step 3: Source the created scenario files into the Tcl Simulation Script using the procedure "`source`" (e.g., "`source myfile`," see also Sect. 3.7).[14]

12.7 Chapter Summary

This chapter covers another network simulation domain: wireless networking. Central to this domain are Mobile Nodes characterized by packet forwarding mechanism using wireless channels and node mobility. In the OTcl domain,

[14]Caution: It is important to validate the path to the scenario files. Failing to do will result in a "file not found" error.

a Mobile Node (class `Node/MobileNode`) is a composite object – consisting of the following key objects: routing (e.g., AODV), link layer, interface queue (e.g., prioritize queues), Medium Access Control (MAC) protocol (e.g., IEEE 802.11), physical network interface, and shared channel objects. NS2 supports both deterministic and random mobility. NS2 also provides two standalone utility – namely `setdest` and `cbrgen.tcl` – to facilitate the network configuration process.

12.8 Exercises

1. Consider Regular Nodes and Mobile Nodes. What are their differences? Where do Mobile Nodes implement wireless links and mobility? Draw network diagrams to support your answer.
2. Explain the packet flow in a wireless network.
   a. When and who is responsible for packet creation/destruction?
   b. Explain the sequence of objects which receive and forward the packets.
   c. What happens when the packet reaches a wireless channel?
   d. What do the source node, the destination node, and the intermediate nodes do?
3. What are the two key steps to create a Mobile Node? What are the purposes of these two key steps?
4. What is a routing loop problem? Why is it a problem? Suggest a way to solve this problem.
5. Explain forward path setup and backward path setup in AODV.
6. How does NS2 determine transmission power, received power, and whether the received packets are in error? Show the related OTcl/C++ statements.
7. What are the roles of timers in the NS2 IEEE 802.11 module?
8. What are the purposes of the variables "`pktRx_`" and "`pktTx_`" of class `Mac` and the variables "`pktRTS_`" and "`pktCTRL_`" of class `Mac802_11`.
9. How does NS2 implement self-interference in the IEEE 802.11 MAC protocol?
10. Show few examples of OTcl statements which set up deterministic/random mobility for a given Mobile Node `$n1`. Explain the OTcl statements you provide.
11. Use the NS2 independent utilities to create the following OTcl statements:
    a. Five Mobile Nodes, each with speed uniformly distributed between 10 and 20 m s^{-1}, and constant pause time of 20 s. The geographical area is set to be 700 on both X-axis and Y-axis.
    b. For the above settings, create at most four TCP connections.

    Test your OTcl statements by running simulation.

Chapter 13
Developing New Modules for NS2

So far, we have explained the details of the basic components of NS2 including their functionalities, internal mechanisms, and configuration methods. In this chapter, we demonstrate how new NS2 modules are created, configured, and incorporated through two following examples. One is an Automatic Repeat reQuest (ARQ) protocol, which is a mechanism to improve transmission reliability of a communication link by means of packet retransmission. Another is a packet scheduler which arranges the transmission sequence of packets from multiple incoming data flows.

13.1 Automatic Repeat reQuest

ARQ is a method of handling communication errors by packet retransmission. An ARQ transmitter (i.e., a transmitting node which implements an ARQ protocol) is responsible for transmitting data packets and retransmitting the lost packets. An ARQ receiver (i.e., a receiving node which implements an ARQ protocol), on the other hand, is responsible for receiving packets and (implicitly or explicitly) informing the transmitter of the transmission result. It returns an ACK (acknowledgment) message and/or a NACK (negative acknowledgment) message to the transmitter if a packet is successfully or unsuccessfully (respectively) received. Based on the received ACK/NACK pattern, the ARQ transmitter decides whether to retransmit the lost packet or to transmit a new packet.

This section focuses on a limited-persistence stop-and-wait ARQ protocol. This type of ARQ protocols is characterized by the two following properties. With limited-persistence, an ARQ transmitter gives up the retransmission if the transmission fails consecutively for a certain number of times. Another property is "stop-and-wait." Here, an ARQ transmitter transmits a packet and waits for an acknowledgment from the corresponding ARQ receiver before commencing another (lost or new) packet transmission.

In the following, we first design the NS2 modules for a limited-persistence stop-and-wait ARQ protocol with an error-free and delay-free (i.e., immediate) feedback

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_13, © Springer Science+Business Media, LLC 2012

Program 13.1 Binding codes for ARQ transmitters, ARQ ACK transmitter, and ARQ NACK transmitter

```
// arq.cc
1  #include "arq.h"
2  static class ARQTxClass: public TclClass {
3  public:
4      ARQTxClass() : TclClass("ARQTx") {}
5      TclObject* create(int, const char*const*) {
6          return (new ARQTx);
7      }
8  } class_arq_tx;
9  static class ARQAckerClass: public TclClass {
10 public:
11     ARQAckerClass() : TclClass("ARQAcker") {}
12     TclObject* create(int, const char*const*) {
13         return (new ARQAcker);
14     }
15 } class_arq_acker;
16 static class ARQNackerClass: public TclClass {
17 public:
18     ARQNackerClass() : TclClass("ARQNacker") {}
19     TclObject* create(int, const char*const*) {
20         return (new ARQNacker);
21     }
22 } class_arq_nacker;
```

channel in Sect. 13.1.1. Sections 13.1.2 and 13.1.3 demonstrate the C++ and OTcl implementations, respectively. Finally, in Sect. 13.1.4, we extend the ARQ modules for an error-free feedback channel with nonzero processing and propagation delay. Implementation of an ARQ protocol with an error prone feedback channel is left as an exercise for the readers.

13.1.1 The Design

13.1.1.1 Architecture

ARQ aims at improving transmission reliability on a lossy link. In this section, we shall build ARQ modules on top of a lossy link defined in Sect. 15.3 (see also Fig. 15.7). In particular, we shall incorporate three following ARQ modules (see class binding codes in Program 13.1) into a `SimpleLink` object as shown in Fig. 13.1.

- **ARQ Transmitter** (`tARQ_`): Keeps track of transmission result (by waiting for messages from ARQ ACK/NACK transmitter) and retransmit the lost packet if necessary. ARQ transmitters are defined in C++ class `ARQTx` bound to the OTcl class with the same name.

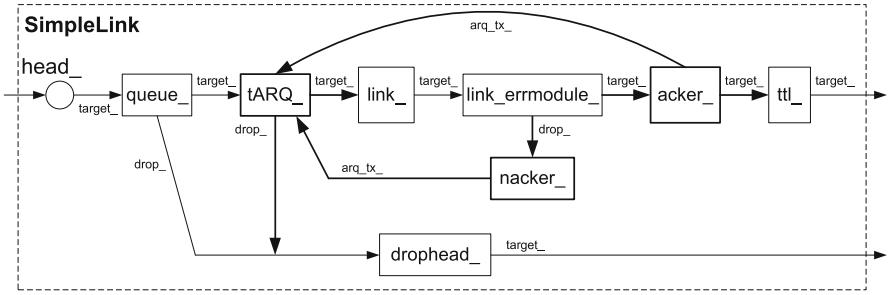

Fig. 13.1 Architecture of a `SimpleLink` object with ARQ-related modules

- **ARQ ACK transmitter (`acker_`)**: Sends an ACK message to the ARQ transmitter if the packet is successfully transmitted. ARQ ACK transmitters are defined in C++ class `ARQAcker` bound to the OTcl class with the same name.
- **ARQ NACK transmitter (`nacker_`)**: Sends a NACK message to the ARQ transmitter if the packet is not successfully transmitted. ARQ NACK transmitters are defined in C++ class `ARQNacker` bound to the OTcl class with the same name.

13.1.1.2 Packet Forwarding Mechanism

Consider Fig. 13.1. When "`tARQ_`" receives a packet from "`queue_`," it passes the packet to its downstream objects. When "`link_errmodule_`" receives the packet, it simulates packet error. If the packet is in error, it will be forwarded to "`nacker_`." Otherwise, it will be forwarded to "`acker_`."

Upon receiving a packet, "`acker_`" sends an ACK message to the ARQ transmitter, and forwards the packet to its downstream object, "`ttl_`." On the other hand, "`nacker_`," upon receiving a packet, sends an NACK message to the ARQ transmitter. To simplify implementation, we let the "`nacker_`" send the packet (which was simulated to be in error) back to the ARQ transmitter.

When the ARQ transmitter receives an ACK message or an NACK message, it prepares a new packet for (re)transmission. If an ACK message is received, it will fetch another packet from "`queue_`." On the other hand, if an NACK message is received, it will decide whether to retransmit or drop the packet. In the latter case, a new packet will be fetched from the "`queue_`." After the preparation is complete, the packet forwarding mechanism proceeds as discussed above.

13.1.1.3 Callback Mechanism

In Sect. 7.3.3, we discuss the callback mechanism of a `SimpleLink` object. The `LinkDelay` object "`link_`" asks the Scheduler to transmit a callback message to "`queue_`" when the packet departure process is complete.

Program 13.2 Declaration of classes ARQTx and ARQHandler

```
// arq.h
23 #include "connector.h"
24 Class ARQTx;
25 enum ARQStatus {IDLE,SENT,ACKED,RTX,DROP};
26 class ARQHandler : public Handler {
27 public:
28     ARQHandler(ARQTx& arq) : arq_tx_(arq) {};
29     void handle(Event*);
30 private:
31     ARQTx& arq_tx_;
32 };
33 class ARQTx : public Connector {
34 public:
35     ARQTx();
36     void recv(Packet*, Handler*);
37     void nack(Packet*);
38     void ack();
39     void resume();
40   protected:
41     ARQHandler arqh_;
42     Handler* handler_;
43     Packet* pkt_;
44     ARQStatus status_;
45     int blocked_;
46     int retry_limit_;
47     int num_rtxs_;
48 };
```

With the introduction of ARQ modules, we will have "link_" call back to "tARQ_" rather than "queue_." In this case, a callback signal resumes the pending retransmission process, rather than the packet departure process. When the retransmission completes, "tARQ_" fetches another packet from "queue_," and continues the regular packet transmission process.

13.1.2 C++ Implementation

We implement the ARQ mechanism using five C++ classes. Apart from three main classes – ARQTx, ARQAcker, and ARQNacker – introduced in the previous section, two helper classes include

- Class ARQHandler: A handle for callback mechanism.
- Class ARQRx: The base class for classes ARQAcker and ARQNacker.

The declaration of all five classes are shown in Programs 13.2 and 13.3.

Program 13.3 Declaration of classes ARQRx, ARQAcker, and ARQNacker

```
// arq.h
49 class ARQRx : public Connector {
50 public:
51     ARQRx()  {arq_tx_=0; };
52     int command(int argc, const char*const* argv);
53 protected:
54     ARQTx* arq_tx_;
55 };
56 class ARQAcker : public ARQRx {
57 public:
58     ARQAcker() {};
59     virtual void recv(Packet*, Handler*);
60 };
61 class ARQNacker : public ARQRx {
62 public:
63     ARQNacker() {};
64     virtual void recv(Packet*, Handler*);
65 };
```

13.1.2.1 Class ARQTxs

Class ARQTx represents ARQ transmitters. Derived from class Connector, it can be used to connect two NsObjects.[1] The main C++ variables of class ARQTx are shown below:

num_rtxs_	Current number of packet retransmissions; It is increased by one for every transmission failure, and is reset to zero when a new packet arrives (e.g., due to a packet drop or a transmission success).
retry_limit_	The retry limit; The ARQ protocol will retransmit the lost packet as long as num_rtxs_<=retry_limit_.
blocked_	Indicates whether the ARQTx object is blocked. If blocked, the ARQTx object will not transmit any packet.
arqh_	A handler which is passed to the downstream object
handler_	A handler of an upstream object (which is a QueueHandler object in our case)
status_	Current status of the ARQTx object defined in Line 25 of Program 13.2
pkt_	A pointer to the packet which is being transmitted.

Class ARQTx defines the four following functions:

[1] In Fig. 13.1, we use an ARQTx object "tARQ_" to connect a Queue object "queue_" with a LinkDelay object "link_."

`recv(p,h)`	Receive packet `*p` from an upstream object.
`ack()`	Process an ACK message; This function invoked by an `ARQAcker` object.
`nack(p)`	Process a NACK message; This function invoked by an `ARQNacker` object.
`resume()`	Resume the operation from the "blocked" state; This function is called by the downstream `LinkDelay` object when the pending packet transmission is complete.

13.1.2.2 Class `ARQHandler`

This helper class facilitates the callback mechanism. From Line 31 in Program 13.2, it has only one variable "`arq_tx_`" which is the reference to the `ARQTx` object. During the callback process, an `ARQHandler` object uses this reference to tell the `ARQTx` object to resume the pending retransmission process.

13.1.2.3 Classes `ARQRx`, `ARQAcker`, and `ARQNacker`

Another part of ARQ implementation is an ARQ receiver, which is responsible for reacting to the ARQ transmitter. Represented by a C++ class `ARQRx`, an ARQ receiver contains a pointer "`arq_tx_`" (see Line 54 in Program 13.3) to an ARQ transmitter (i.e., an `ARQTx` object). This pointer is initialized to zero at the object construction (Line 51), and is associated with an ARQ transmitter by the OTcl command `attach-ARQTx` (Program 13.6).

There are two classes derived from class `ARQRx`: classes `ARQAcker` and `ARQNacker`. These two classes are responsible for sending ACK and NACK messages, respectively, to the associated ARQ transmitter.

13.1.2.4 Callback Mechanism

Consider the program related to a callback process in Program 13.4. The process begins when a `Queue` object sends a packet to an `ARQTx` object via function `recv(p,h)` (Lines 72–76). The `ARQTx` object stores the handler "`h`" in the class variable "`handler_`." This handler will be used to fetch another packet from the `Queue` object when the retransmission process completes.

Next, the `ARQTx` object sends the packet as well as its own handler "`arqh_`" to its downstream `LinkDelay` object. Again, the `LinkDelay` object will put this handler "`arqh_`" on the simulation timeline at the time when the packet transmission completes. At the firing time, the Scheduler calls the default action (i.e., the function `handle(e)`) of the handler "`arqh_`." From Lines 102–105, the `ARQHandler` object invokes function `resume()` of the associated `ARQTx` object

Program 13.4 Functions of classes ARQTx and ARQHandler

```
   // arq.cc
66 ARQTx::ARQTx() : arqh_(*this)
67 {
68     num_rtxs_ = 0;  retry_limit_ = 0; handler_ = 0;
69     pkt_ = 0; status_ = IDLE; blocked_ = 0;
70     bind("retry_limit_", &retry_limit_);
71 }
72 void ARQTx::recv(Packet* p, Handler* h)
73 {
74     handler_ = h; status_ = SENT; blocked_ = 1;
75     send(p,&arqh_);
76 }
77 void ARQTx::ack()
78 {
79     num_rtxs_ = 0; status_ = ACKED;
80 }
81 void ARQTx::nack(Packet* p)
82 {
83     num_rtxs_++; pkt_=p;
84     if( num_rtxs_ <= retry_limit_)
85         status_ = RTX;
86     else
87         status_ = DROP;
88 }
89 void ARQTx::resume()
90 {
91     blocked_ = 0;
92     if ( status_ == ACKED ) {
93          status_ = IDLE; handler_->handle(0);
94      } else if ( status_ == RTX ) {
95          status_ = SENT; blocked_ = 1;
96          send(pkt_,&arqh_);
97      } else if ( status_ == DROP ) {
98          status_ = IDLE; drop(pkt_);
99          handler_->handle(0);
100     }
101 }
102 void ARQHandler::handle(Event* e)
103 {
104    arq_tx_.resume();
105 }
```

(i.e., ARQ transmitter) "arq_tx_." When the function resume() is invoked, the ARQTx object checks the retransmission status. If the ACK message is received (i.e., ACKED in Line 92) or the limit on the number of allowable retransmissions is exceeded (i.e., DROP in Line 97), the ARQTx object will fetch another packet

Program 13.5 Functions of classes ARQRx, ARQAcker, and ARQNacker

```
// arq.cc
106 void ARQAcker::recv(Packet* p, Handler* h)
107 {
108     arq_tx_->ack();
109     send(p,h);
110 }
111 void ARQNacker::recv(Packet* p, Handler* h)
112 {
113     arq_tx_->nack(p);
114 }
```

by executing handler_->handle(0) (Lines 93 and 99).[2] Otherwise, it will retransmit the packet by executing send(pkt_, &arqh_) (Line 96).

13.1.2.5 Packet Forwarding Mechanism

Consider Fig. 13.1. The main packet forwarding mechanism is to pass the packet from "queue_," to "tARQ_," to "link_," and to "link_errmodule_," respectively. Then, if the packet is simulated to be and not to be in error, it will be forwarded to nacker_ and acker_ whose classes are ARQNacker and ARQAcker, respectively.

Consider Program 13.5. That an ARQAcker object receives a packet implies successful packet transmission. In this case, the ARQAcker object sends an ACK message to the associated ARQTx object "arq_tx_" (by executing arq_tx_->ack() in Line 108). Then, it forwards the packet to its downstream object (by executing send(p,h) in Line 109).

On the other hand, that an ARQNacker object receives a packet implies a packet loss. In this case, the Nacker object sends a NACK message to the associated ARQTx object "arq_tx_" (by executing arq_tx_->nack(p) in Line 113), passing the packet in error "p" as an input argument. In the next section, we shall see what an ARQTx would do when receiving an ACK message or a NACK message.

13.1.2.6 Processing ACK and NACK Messages

Figure 13.2 and Program 13.4 show details of how an ARQ transmitter processes ACK/NACK messages. Upon receiving an ACK message (via function ack() in Lines 77–80), the ARQTx object resets the retransmission limit "num_rtxs_" to zero and sets the status to ACK. If an NACK message is received (via function

[2]The variable "handle_" stores the QueueHandler object associated with the upstream Queue object.

Program 13.6 Defintion of the OTcl command `attach-ARQTx`

```
115 int ARQRx::command(int argc, const char*const* argv)
116 {
117     Tcl& tcl = Tcl::instance();
118     if (argc == 3) {
118         if (strcmp(argv[1], "attach-ARQTx") == 0) {
119             if (*argv[2] == '0') {
120                 tcl.resultf("Cannot attach NULL ARQTx\n");
121                 return(TCL_ERROR);
122             }
123              arq_tx_ = (ARQTx*)TclObject::lookup(argv[2]);
124              return(TCL_OK);
125         }
126
127     } return Connector::command(argc, argv);
128 }
```

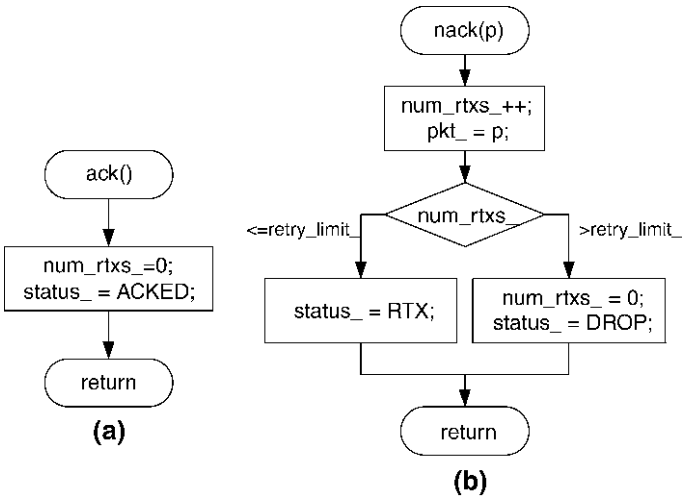

Fig. 13.2 Flowcharts of functions (**a**) `ack()` and (**b**) `nack(p)` of class `ARQTx`

`nack(p)` in Lines 81–88), the `ARQTx` object increments the number of retransmissions (i.e., "`num_rtxs_`") attempted so far. If the number of retransmissions is within the limit, it will set its status to `RTX`. Otherwise, the status will be set to `DROP`. In both the cases, a pointer to the packet "`p`" simulated to be in error is stored in the class variable "`pkt_`" for future use.

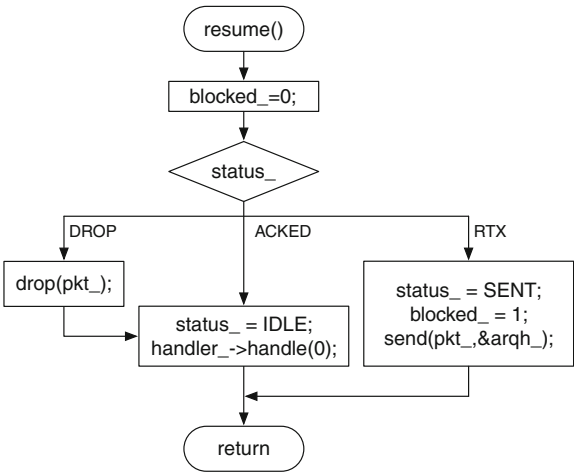

Fig. 13.3 Flowchart of the function `resume()` of class `ARQTx`

13.1.2.7 Actions and Status of ARQ Transmitters

Upon receiving ACK/NACK messages, the `ARQTx` object does not immediately transmit, retransmit, or drop the packet, since there might be packets in transit. At the moment, it records the transmission result in the class variable "`status_`," and waits for a callback signal from the `LinkDelay` object before taking further actions.

Figure 13.3 shows three possibilities taken from within the function `resume()`, when a callback signal is received (see Lines 89–101 in Program 13.4).

If the "`status_`" was set to `RTX`, the lost packet is retransmitted here.[3] If the "`status_`" was set to `DROP`, the packet "`pkt_`" is destroyed here. If the "`status_`" was set to either `ACK` or `DROP`, the `ARQTx` will fetch another packet from the `Queue` object by executing `handler_->handle(0)`.[4]

13.1.3 OTcl Implementation

In the OTcl domain, we need to create `ARQTx`, `ARQAcker`, and `ARQNack` objects–"`tARQ_`", "`acker_`," and "`nacker_`," respectively, and insert them into a `SimpleLink` object as shown in Fig. 13.1. Program 13.7 shows two OTcl instprocs developed for this purpose.

[3] The packet in error was earlier stored in the class variable "`pkt_`" (see Line 83 in Program 13.4).

[4] Again, the variable "`handle_`" stores a pointer to the `QueueHandler` object associated with the upstream `Queue` object.

Program 13.7 OTcl Instprocs for an ARQ Module

```
     //~ns/tcl/lib/ns-link.tcl
  1  SimpleLink instproc link-arq { limit } {
  2      $self instvar link_ link_errmodule_ queue_ drophead_
  3      $self instvar tARQ_ acker_ nacker_
  4      set tARQ_ [new ARQTx]
  5      set acker_ [new ARQAcker]
  6      set nacker_ [new ARQNacker]
  7      $tARQ_ set retry_limit_ $limit
  8      $acker_ attach-ARQTx $tARQ_
  9      $nacker_ attach-ARQTx $tARQ_
 10      $queue_ target $tARQ_
 11      $tARQ_ target $link_errmodule_
 12      $link_errmodule_ target $acker_
 13      $acker_ target $link_
 14      $tARQ_ drop-target $drophead_
 15      $link_errmodule_ drop-target $nacker_
 16  }

     //~ns/tcl/lib/ns-lib.tcl
 17  Simulator instproc link-arq {limit from to} {
 18      set link [$self link $from $to]
 19      $link link-arq $limit
 20  }
```

13.1.3.1 Instproc `SimpleLink::link-arq{limit}`

This instproc creates the ARQ-related objects and configures the `SimpleLink` object as shown in Fig. 13.1. Lines 4–6 create instvars "`tARQ_`," "`acker_`," and "`nacker_`." Line 7 stores the input argument "`limit`" in the instvar "`retry_limit_`" of "`tARQ_`." From Line 70 in Program 13.4, the instvar "`retry_limit_`" is bound to the C++ class variable with the same name. This variable indicates the maximum number of retransmissions for an erroneous packet.

Lines 8 and 9 associate "`acker_`" and "`nacker_`," respectively, with "`tARQ_`," using the OTcl command `attach-ARQTx`. Defined in C++ class `ARQRx`, this OTcl command stores the input argument (i.e., "`tARQ_`" in our case) in the C++ class variable "`arq_tx_`" (see Program 13.6). Finally, Lines 10–15 configure the rest of the components as shown in Fig. 13.1.

13.1.3.2 Instproc `Simulator::link-arq{limit from to}`

From the Tcl simulation script, the `Simulator` object is readily accessible, while `SimpleLink` objects are not. Acting as a programming interface from the Tcl simulation script, this instproc (Lines 17–20) invokes the instproc `link-arq{...}`

of the SimpleLink object. In particular, it creates and configures ARQ modules of the link connecting Node "from" to Node "to." The input argument "limit" here is used as the retry limit of the ARQ module.

Example 13.1. We now setup an experiment to show the impact of a limited-persistence stop-and-wait ARQ protocol on TCP throughput. Our experiment is based on Sect. 10.1. We insert an error module with 0.3 error probability in the link connecting Node n1 to Node n3, implement a limited-persistence ARQ over this lossy link, vary the retry limit from 0 to 3, and plot TCP throughput versus the retry limit.

Tcl Simulation Script

We insert the following codes in the Tcl simulation script file "tcp.tcl" in Sect. 10.1:

```
    //tcp.tcl
1   set em [new ErrorModel]
2   $em set rate_ 0.3
3   $em unit pkt
4   $em ranvar [new RandomVariable/Uniform]
5   $em drop-target [new Agent/Null]
6   $ns link-lossmodel $em $n1 $n3

7   $ns link-arq 3 $n1 $n3

8   proc show_tcp_seqno {} {
9       global tcp
10      puts "The final tcp sequence number is
                                  [$tcp set t_seqno_]"
11  }

12  $ns at 0.0 "$ftp start"
13  $ns at 100.0 "show_tcp_seqno"
14  $ns at 100.1 "$ns halt"
15  $ns run
```

Here, Lines 1–6 create an error module with packet error probability 0.3, and insert the created error module immediately after the instvar "link_" of the SimpleLink object connecting Node n1 and Node n3. Line 7 creates and configures ARQ-related components with retry limit of 3. We run the simulation for 100.1 s and collect the results when the simulation time is 100.0 s. After running the script file "tcp.tcl" above, the following result appears on the screen:

```
>> ns tcp.tcl
>> The final tcp sequence number is 37587
```

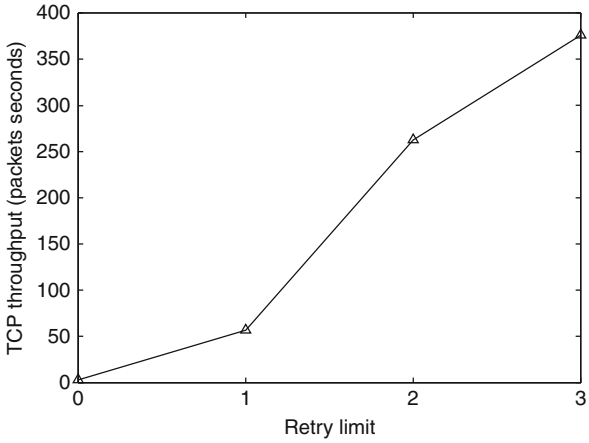

Fig. 13.4 Impact of the retry limit of a limited persistent ARQ protocol on TCP throughput

TCP throughput in packets per second is computed as the final TCP sequence number divided by the simulation time. We vary the retry limit (in Line 7 above) to $\{0, 1, 2, 3\}$, and plot TCP throughput in Fig. 13.4. Clearly, increasing retry limit increases link reliability and therefore increases TCP throughput. □

13.1.4 ARQ Under a Delayed (Error-Free) Feedback Channel

We have developed an NS2 module for an ARQ protocol with an immediate and error-free feedback. This section extends the modules developed earlier for a *non-immediate* error-free feedback channel. The extension for a non-immediate and *error-prone* feedback channel is left for the reader as an exercise.

We modify the ARQ modules in Sects. 13.1.2–13.1.3 using the Scheduler. The idea is to delay the transmission of ACK/NACK messages for a certain amount of time. The modification is shown in Program 13.8.

13.1.4.1 Function `recv(p,h)` of ARQ Receivers

We move the function `recv(p,h)` from classes `ARQAcker` and `ARQNacker` to their base class `ARQRx`. The new function `recv(p,h)` is defined in Lines 1–8.

The delay is implemented by scheduling packet reception events on the simulation timeline after "`delay_`" seconds (see Sect. 4.3 for details about the function `schedule(...)`). At the firing time, the Scheduler will invoke function `handle(e)` associated with the first input argument of the function `schedule(...)`, i.e., the pointer "`this`" (see Line 5). Here, the pointer "`this`" points to either an `ARQAcker` object or an `ARQNacker` object, whose details are

Program 13.8 Modification in file `arq.cc` for ARQ with error-free delay feedback channels

```
     //arq.cc
  1  void ARQRx::recv(Packet* p, Handler* h)
  2  {
  3      pkt_ = p; handler_ = h;
  4      if (delay_ > 0)
  5          Scheduler::instance().schedule(this, &event_,
                 delay_);
  6      else
  7          handle(&event_);
  8  }
  9  void ARQAcker::handle(Event* e)
  10 {
  11     arq_tx_->ack();
  12     send(pkt_,handler_);
  13 }
  14 void ARQNacker::handle(Event* e)
  15 {
  16     arq_tx_->nack(pkt_);
  17 }
  18 ARQRx::ARQRx()
  19 {
  20     pkt_ = 0; handler_ = 0;  delay_ = 0;
  21     bind("delay_", &delay_);
  22 }
```

shown in Lines 9–13 and 14–17, respectively. These two functions perform the same actions as those in the function `recv(p,h)` in Sect. 13.1.2.

13.1.4.2 Binding Variable `delay_`

In the constructor, we bind the variable "`delay_`" to the OTcl instvar with the same name (see Lines 18–22).

13.1.4.3 Configuration in the OTcl Domain

In the OTcl domain, we also need to include the two following lines into the instproc `link-arq{limit}` of class `SimpleLink` (e.g., after Line 6 in Program 13.7):

```
$acker_ set delay_ [$self delay]
$nacker_ set delay_ [$self delay]
```

Here, the link delay in the forward direction (returned from `$self delay`) is used as the ARQ feedback delay for both ACK and NACK generators (i.e., "`acker_`" and "`nacker_`," respectively).

Example 13.2. Compare the TCP throughputs for the cases with an immediate feedback channel and a delayed feedback channel in the link layer ARQ protocols.

Here, we use the results in Example 13.1 as a benchmark. When rerunning the Tcl simulation script in Example 13.1 under the ARQ protocol with a delayed feedback channel, the following result should appear on the screen:

```
>> ns tcp.tcl
>> The final tcp sequence number is 20596
```

which is less than 37587 in Example 13.1. The readers are encouraged to experiment with different input parameters (e.g., feedback delay or retry limit) to gain more insights into the impact of link layer ARQ protocols on TCP performance. □

13.2 Packet Scheduling for Multi-Flow Data Transmission

Packet scheduling is a mechanism to arrange transmission sequence of incoming packets. For example, a *round-robin* (RR) packet scheduler transmits packets from different flows in sequence. This section shows the implementation of a round-robin packet scheduler in NS2.

13.2.1 The Design

We modify the packet scheduler by adding few components into a `SimpleLink` object as shown in Fig. 13.5.

13.2.1.1 Architecture of a `SimpleLink` with a Packet Scheduler

The key modifications are as follows:

- *Source traffic*: We assume that each traffic source tags packets with its unique flow ID.
- *Flow classifiers*: Packets from different sources are mixed as they enter a `SimpleLink` object. These unclassified packets get classified by a flow classifier. Packets with the same flow ID are transmitted to the same place.
- *A queue array*: A dedicated queue is provided for packets whose flow IDs are the same (i.e., generated from the same source).
- *The packet scheduler*: The packet scheduler fetches and transmits packets from the queue array. It selects the queue to fetch a packet based on the underlying scheduling policy.

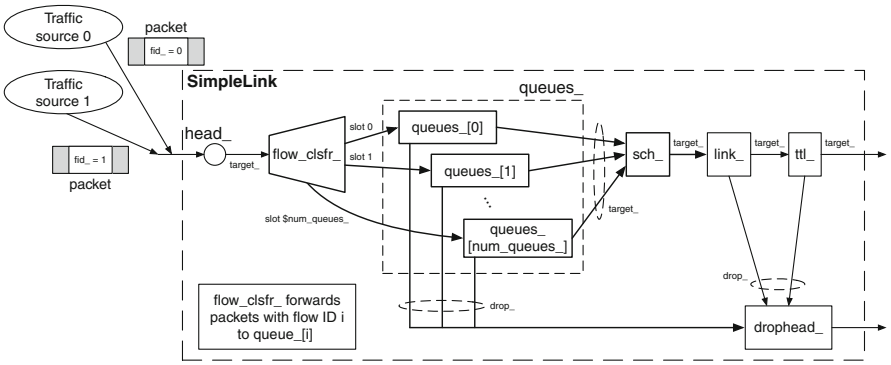

Fig. 13.5 Architecture of a `LinkSch` object

13.2.1.2 Packet Forwarding and Callback Mechanism

The process begins when unclassified packets enters the `SimpleLink` object and proceeds as follows:

1. The flow classifier forwards packets with the same flow ID to the same queue.
2. When not blocked, a queue sends a packet to its downstream packet scheduler.
3. The packet scheduler selects a flow, transmits a packet from the selected flow, and fetches another packet from the queue corresponding to the selected flow.
4. After a transmission, the packet scheduler blocks itself until it receives a callback message, indicating that the pending packet transmission is complete.
5. When receiving a packet, a `LinkDelay` schedules a transmission of a callback message to the upstream packet scheduler when the packet transmission process is complete.
6. At the firing time, the process goes back to step (3).

13.2.2 C++ Implementation

We develop three following C++ classes: `FlowClassifier`, `PktScheduler`, and `RRScheduler`

13.2.2.1 Class `FlowClassifier`

Class `FlowClassifier` represents flow classifiers discussed earlier. The C++ code for this class is shown in Program 13.9. From Lines 10–16, this class is bound to the OTcl class `Classifier/Flow`. Again, flow classifiers are responsible for classifying packets based on flow ID, as implemented in Lines 17–21.

Program 13.9 C++ Implementation of class `FlowClassifier`

```
// classifier-flow.h
1  #include "packet.h"
2  #include "ip.h"
3  #include "classifier.h"
4  class FlowClassifier : public Classifier {
5  protected:
6      int classify(Packet *p);
7  };

// classifier-flow.cc
8  #include "classifier-flow.h"
9  #include "ip.h"
10 static class FlowClassifierClass : public TclClass {
11 public:
12     FlowClassifierClass() : TclClass("Classifier/Flow") {}
13     TclObject* create(int, const char*const*) {
14         return (new FlowClassifier());
15     }
16 } class_flow_classifier;
17 int FlowClassifier::classify(Packet *p)
18 {
19     int flow = hdr_ip::access(p)->flowid();
20     return flow;
21 }
```

13.2.2.2 Class `PktScheduler`

The main responsibility of a packet scheduler is to determine transmission sequence of the attached upstream `Queue` objects. In this section, we assume that each `Queue` object holds packets of the same flow ID and the packet scheduler determines the transmission sequence based on the flow ID only.

Programs 13.10 and 13.11 show the declaration and implementation, respectively, of C++ class `PktScheduler`. From Program 13.10, class `PktScheduler` has one constant and three key variables:

`MAX_FLOW`	The maximum number of queues which can be attached to the packet scheduler.
`blocked_`	Set to "1" if the packet scheduler is transmitting a packet, and set to "0" otherwise.
`pkt_[i]`	The packet from flow "`i`" waiting to be transmitted
`handler_[i]`	The `QueueHandler` object of the queue corresponding to flow "`i`"

Program 13.11 shows function implementation of class `PktScheduler`. In Lines 38–44, the function `recv(p,h)` determines and stores the flow ID of the incoming packet in a local variable "`pid`." Line 41 stores the packet as well as the input handler in its class variables `pkt_[pid]` and `qh_[pid]`, respectively. If not

Program 13.10 Declaration of C++ class `PktScheduler`

```
// pkt-sched.h
1  #include "connector.h"
2  #include "ip.h"
3  #define MAX_FLOWS 10
4  class PktScheduler : public Connector {
5  public:
6      PktScheduler();
7      virtual void handle(Event*);
8      virtual void recv(Packet*, Handler*);
9    protected:
10     void sendNextPkt();
11     int getPktID(Packet* p) { return hdr_ip::access(p)->
       flowid(); }
12     virtual int nextID()=0;
13     Handler* qh_[MAX_FLOWS];
14     Packet* pkt_[MAX_FLOWS];
15     int blocked_;
16 };
```

Program 13.11 Functions of C++ class `PktScheduler`

```
// pkt-sched.cc
17 #include "pkt-sched.h"
18 PktScheduler::PktScheduler()
19 {
20     for (int i=0;i<MAX_FLOWS;i++){
21         pkt_[i] = 0; qh_[i]=0;
22     }
23     blocked_ = 0;
24 }
25 void PktScheduler::handle(Event*)
26 {
27     blocked_ = 0; sendNextPkt();
28 }
29 void PktScheduler::sendNextPkt()
30 {
31         int nid = nextID();
32         if (nid >=0) {
33             send(pkt_[nid],this);
34             blocked_ = 1; pkt_[nid] = 0;
35             qh_[nid]->handle(0);
36         }
37 }
38 void PktScheduler::recv(Packet* p, Handler* h)
39 {
40     int pid = getPktID(p);
41     pkt_[pid] = p; qh_[pid] = h;
42     if (!blocked_)
43         sendNextPkt();
44 }
```

blocked (i.e., no pending packet transmission), the packet transmitter will select and send out one packet to its downstream object, using function `sendNextPkt()`.

In Lines 29–37, the function `sendNextPkt()` first determines the next flow which will be allowed to transmit a packet using function `nextID()`. Being pure virtual, the function `nextID()` forces the derived classes of class `PktScheduler` to provide how the next flow is selected based on underlying scheduling policy. This function returns the flow ID of the next flow, or "-1" if all the flows do not have packets to transmit.

Suppose there is at least one flow which has packets to transmit. The packet scheduler sends out the packet (Line 33), blocks itself (Line 34), resets the variable `pkt_[nid]` (Line 34), and fetches a new packet from the upstream `Queue` object (Line 35).

This example shows another type of callback mechanism which does not use a handler dedicated to the packet scheduler. From Line 33, the packet scheduler sends the pointer to itself (i.e., "`this`") along with the packet to its downstream object. When this pointer reaches a `LinkDelay` object, it is placed on the simulation timeline. At the firing time, the function `handle(e)` of the packet scheduler is executed at the firing time.

13.2.2.3 Class `RRScheduler`

Program 13.12 shows the details of class `RRScheduler`. We derive a C++ class `RRScheduler` from class `PktScheduler`, and bind this class to an OTcl class `PktScheduler/RR` (Lines 52–58). Class `RRScheduler` contains only one variable "`current_id_`" which stores the ID of the flow whose head of the line packet is being transmitted.

Class `RRScheduler` overrides the pure virtual function `nextID()` defined in its base class. This function returns the next ID which has a packet to transmit, and returns −1 if all flows have no packet to transmit (Lines 63–75).

13.2.3 OTcl Implementation

In the OTcl domain, we insert three following components into class `SimpleLink`, as shown in Fig. 13.5:

`sch_`	A round-robin scheduler whose class is `PktScheduler/RR`
`flow_clsfr_`	A flow classifier
`queues_`	A queue array which stores packets from different flows classified by the flow classifier

From Program 13.13, we develop two instprocs to configure packet schedulers in the OTcl domain:

Program 13.12 C++ implementation of C++ Class `RRScheduler`

```
// pkt-sched.h
45 class RRScheduler : public PktScheduler {
46 public:
47     RRScheduler() ;
48 private:
49     virtual int nextID();
50     int current_id_;
51 };

// pkt-sched.cc
52 static class RRSchedulerClass: public TclClass {
53 public:
54     RRSchedulerClass() : TclClass("PktScheduler/RR") {}
55     TclObject* create(int, const char*const*) {
56         return (new RRScheduler());
57     }
58 } class_rr_scheduler;
59 RRScheduler::RRScheduler()
60 {
61     current_id_ = -1;
62 }
63 int RRScheduler::nextID()
64 {
65     int count = 0;
66     current_id_ = (current_id_ + 1 ) % MAX_FLOWS;
67     while( (pkt_[current_id_] == 0) && (count<MAX_FLOWS) ) {
68         current_id_ = (current_id_ + 1 ) % MAX_FLOWS;
69         count++;
70     }
71     if (count == MAX_FLOWS)
72         return -1;
73     else
74         return current_id_;
75 }
```

13.2.3.1 Instproc `insert-sched` {`num_queues`} of class `SimpleLink`

This instproc inserts packet-scheduler-related components into a `SimpleLink` object. Lines 4 and 5 create a flow classifier "`flow_clsfr_`" and a packet scheduler "`sch_`," respectively. Lines 6–12 create a queue array with "`num_queues`" components to store packets from different flows. The `i`th queue is created and installed in the `i`th slot of the flow classifier (Lines 7 and 8). The target and drop-target of every queue are directed to the packet scheduler and the dropping point (i.e., "`drophead_`"), respectively (Lines 9 and 10). The queue is then initialized by its instproc `reset`. Finally, Lines 13–15 insert the above components between "`head_`" and "`link_`."

Program 13.13 OTcl implementation of a link with a round-robin packet scheduler

```
//~pkt-sched.tcl
1  SimpleLink instproc insert-sched {num_queues} {
2       $self instvar link_ queues_ head_ drophead_
3       $self instvar sch_ flow_clsfr_

4       set flow_clsfr_ [new Classifier/Flow]
5       set sch_ [new PktScheduler/RR]
6       for {set i 0} {$i < $num_queues} {incr i} {
7           set queues_($i) [new Queue/DropTail]
8           $flow_clsfr_ install $i $queues_($i)
9           $queues_($i) target $sch_
10          $queues_($i) drop-target $drophead_
11          $queues_($i) reset
12      }
13      $head_ target $flow_clsfr_
14      $sch_ target $link_
15      $sch_ drop-target $drophead_
16 }

17 Simulator instproc insert-sched-to-SL {from to num_queues} {
18      set link [$self link $from $to]
19      $link insert-sched $num_queues
20 }
```

13.2.3.2 Instproc `insert-sched-to-SL{from to num_queues}` of class `Simulator`

Readily accessible to the Tcl simulation script, this instproc configures the link connecting Node "`from`" and Node "`to`," as shown in Fig. 13.5. In Example 13.3, we shall use this instproc to insert packet-scheduler-related components into a `SimpleLink` object.

Example 13.3. Consider Sect. 10.1 and Fig. 9.3. Replace the TCP flow with "`num_queues`" TCP flows whose flow ID are 0, 1, 2, and so on. Apply a round-robin packet scheduling discipline to the link connecting the Node `n1` and the Node `n3`.

Tcl Simulation Script

```
//pkt-sched.tcl
21 set num_queues [lindex $argv 0]
22 set ns [new Simulator]
23 set n1 [$ns node]
24 set n2 [$ns node]
25 set n3 [$ns node]
26 $ns duplex-link $n1 $n2 5Mb 2ms DropTail
```

```
27 $ns duplex-link $n2 $n3 5Mb 2ms DropTail
28 $ns duplex-link $n1 $n3 5Mb 2ms DropTail
29 $ns insert-sched-to-SL $n1 $n3 $num_queues
30 for {set i 0} {$i < $num_queues} {incr i} {
31     set tcp($i) [new Agent/TCP]
32     set sink($i) [new Agent/TCPSink]
33     set ftp($i) [new Application/FTP]
34     $tcp($i) set fid_ $i
35     $ns attach-agent $n1 $tcp($i)
36     $ns attach-agent $n3 $sink($i)
37     $ftp($i) attach-agent $tcp($i)
38     $ns connect $tcp($i) $sink($i)
39     $ns at 0.0 "$ftp($i) start"
40 }
41 $ns at 100.1 "$ns halt"
42 $ns run
```

The above Tcl simulation script "`pkt-sched.tcl`" takes the number of TCP flows as an input argument, and simulates the transmission of these TCP flows under a round-robin packet scheduler.

Line 21 takes an input argument from the shell and stores it in a local variable "`num_queues`" (see the syntax for inputting parameters in Tcl in a local Sect. A.1.1). Line 29 inserts components related to a packet scheduler into the `SimpleLink` object connecting Node `$n1` and Node `$n3`. The "`for`" loop in Lines 30–40 creates and configures TCP flows, where packets created by the "`i`th" element of "`tcp`" are tagged with flow ID "`i`" (by Line 34).

By running the simulation for 1 TCP flow and 3 TCP flows, the following results are shown on the screen.

```
>> ns pkt-sched.tcl 1
The final tcp(0) sequence number is 60110

>> ns pkt-sched.tcl 3
The final tcp(0) sequence number is 20052
The final tcp(1) sequence number is 20051
The final tcp(2) sequence number is 20051
```

The TCP throughput is computed by the final sequence number divided by the simulation time. Since the simulation time here is 100 s (see Line 28), the throughput of TCP flow "`0`" is 610.1 packets/s and 200.52 packets/s when the number of TCP flows is 1 and 3, respectively.

With a round-robin scheduler, each element of the array "`queue_`" has equal chance to transmit packets. In principle, every TCP flow should have the same throughput performance (as shown above). Also, the per-flow throughput in case of n TCP flows should be approximately n times less than that in the case of single TCP flow.

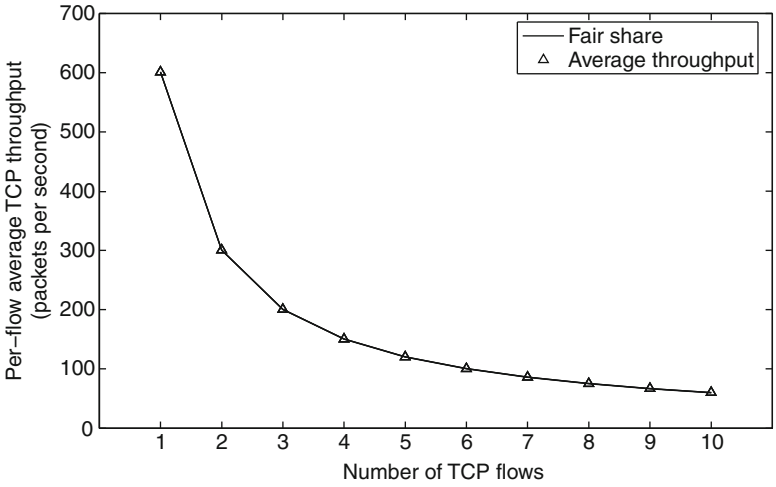

Fig. 13.6 Impact of number of TCP flows on per-flow throughput under round-robin packet scheduling

From the results shown above, the per-flow TCP throughput of all the flows varies very little from each other. Also, the throughput is approximately one-third of TCP throughput in case of single TCP flow (i.e., (60110/100)/3 = 601.10/3 = 200.37).

Next, we run the above Tcl simulation script for 1–10 TCP flows. We compare the average TCP throughput and the *fair share* TCP throughput in Fig. 13.6. Here, we define the *fair share* TCP throughput for n TCP flows as γ/n, where γ is the TCP throughput in the case of single TCP flow. We observe that both average and fair share throughput are almost inline with each other. We also observe that TCP throughput for each flow is very similar to each other. These two observations validate the round-robin operation, which treats every TCP flow equally. □

13.3 Chapter Summary

This chapter demonstrates how new modules are created, configured, and incorporated into NS2. Two examples are provided here on Automatic Repeat reQuest (ARQ)-based error recovery modules and packet scheduling modules. In most of the cases, we need to develop NS2 program in both C++ and OTcl domains. In the C++ domain, the main task is to define the internal mechanisms of the new NS2 components. The main tasks in the OTcl domain, on the other hand, are to integrate the developed NS2 components into the existing NS2 modules, and to instantiate and configure the newly developed modules from a Tcl simulation script.

13.4 Exercises

1. The implementation of a delayed feedback channel makes use of the function `schedule(...)` of class `Scheduler`. Can we use class `TimerHandler` for the implementation instead? Discuss the pros and cons.
2. Consider the C++ classes `ARQTx` and `PktScheduler` in Sects. 13.1.2 and 13.2.2. Do we need the class variable "`blocked_`"? Why or why not?
3. In Sect. 13.1.4, we show how the file `arq.cc` can be modified to incorporate feedback delay.
   a. Modify the file `arq.h` and regenerate the results.
   b. Do we need variables "`pkt_`," "`handler_`," "`event_`," "`delay_`"? Why or why not?
4. Consider the ARQ modules developed in this chapter.
   a. Can you remove class `ARQHandler`? Why and why not?
   b. Regenerate the result of Example 13.2 with the retry limit being 0, 1, 2, and 3. Plot the result on Fig. 13.6. Discuss the impact of the delay on the feedback channel.
   c. Develop a module for an ARQ protocol with an *error prone* delayed feedback channel.
   d. Modify the ARQ to be timer-based retransmission mechanism: A packet is assumed to be lost unless an ACK message is received within a timeout period.
   e. The developed ARQ modules are associated with a `SimpleLink` object. Redesign the modules such that it appear as a component in a Node. Test your module in both wired and wireless networks.
   f. This chapter implements ARQ at the link level. Redesign the modules to operate at connection level (i.e., end-to-end). Incorporate your modules with UDP to test your modules.
5. Design a link with a round-robin packet scheduler and an ARQ-based error control mechanism.
6. Redesign the packet schedulers developed in this chapter. Implement the packet scheduler as a component in a node. Test your implementation with wired and wireless networks.
7. A Weighted Fair Queue (WFQ) packet scheduler gives fair access to every data flow. Under a WFQ packet scheduler, each data flow gains channel access in proportion to its weight. The algorithm for WFQ-based packet scheduling can be found in [28]. Develop a module for a WFQ packet scheduler. Validate the module by plotting the results in a graph.

Chapter 14
Postsimulation Processing: Debugging, Tracing, and Result Compilation

A typical NS2 simulation consists of three main steps: (1) simulation design, (2) configuring and running simulation, and (3) postsimulation processing (see Fig. 2.3 in Sect. 2.5.2). The former two aspects were discussed extensively in the previous chapters, while the last aspect will be discussed in this chapter.

Postsimulation processing encompasses *debugging*, *tracing*, and *compilation of simulation results*. Debugging is a process of removing programming errors. Variable tracing tracks changes in variables under consideration. Packet tracing records the details of packets passing through network checkpoints. Simulation result compilation collects information and computes relevant performance measures from the simulation. This chapter discusses the details of debugging, variable tracing, packet tracing, and result compilation in Sects. 14.1, 14.2, 14.3, and 14.4, respectively. Finally, the chapter summary is given in Sect. 14.5.

14.1 Debugging: A Process to Remove Programming Errors

A programming error is usually referred to as a *bug*. The process of locating and fixing the error is usually called *debugging*. This section discusses two types of programming errors (i.e., bugs) and provides guidelines for debugging in NS2.

14.1.1 Types of Programming Errors

Based on the NS2 architecture, programming errors can be classified into compilation errors and runtime errors.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_14, © Springer Science+Business Media, LLC 2012

14.1.1.1 Compilation Errors (C++ Only)

This type of errors occurs during a compilation process, which consists of two phases. The first phase converts C++ files (with extension "`.cc,h`") into object files (with extension "`.o`"). In this phase, errors may occur if the compiler does not understand the C++ codes. In this case, the compiler will show error messages on the screen, indicating the location and the reason of the errors. Examples of C++ compilation errors include:

- Incorrect C++ syntax
- Using undefined variables and/or functions.

In the second phase, the compiler links the created object files and creates an executable "`ns`" file. An error in this phase is caused by improper linkage of C++ files. Examples of C++ linking errors include:

- *Instantiate an object from an abstract class*: During a linking process, an error will occur if an object is instantiated from an abstract class, which leaves at least one pure virtual function unimplemented.

  A proper solution to this error is to provide implementation for the pure virtual function. However, for simplicity (but not for appropriateness), a user may provide empty implementation for the pure virtual function to remove the error.
- *Modifying a base class without creating the object files of the child classes*: This error usually occurs when the dependency in the "`Makefile`" is not properly defined. When a certain class is modified, the compiler does not recreate object files of the child classes. The solution is to define the dependency in the "`Makefile`" properly, or to remove all related object files before compiling the codes.

Note that OTcl is a scripting language. There is no need to compile OTcl program before the execution. Therefore, compilation errors do not occur in the OTcl domain.

14.1.1.2 Runtime Errors

This type of errors occurs during NS2 simulation. It is caused by improper OTcl and/or C++ programming. Since the OTcl domain implements error message trapping mechanism, an OTcl error message contains detailed and useful information. Each error message indicates where and why the error occurred. The C++ domain, on the other hand, does not implement error trapping. Generally the error messages in this case (e.g., segmentation fault) are fairly short and do not contain much information. Examples of OTcl runtime errors include

- Incorrect OTcl syntax
- Referring to instvars, instprocs, or commands which do not exist.

Examples of C++ runtime errors include

- *Segmentation fault*: This is usually caused by an invalid access to a memory content. For example, trying to access "`a[6]`" would cause a segmentation fault if "`a`" was declared as "`int a[3];`."
- *Not implementing a mandatory (non-pure virtual) function*: Apart from using a pure virtual function, NS2 provides another way to force a child class to implement a mandatory function. Here, NS2 may implement error-like actions (e.g., print out an error message) in the base class. If a child class does not implement this mandatory function, the function of the base class will be invoked and the error-like actions will be taken. Examples of this type of errors are the implementation of functions `sendmsg(...)` and `sendto(...)` of class `Agent` in the file *~ns*/common/agent.cc.

14.1.2 Debugging Guidelines

After identifying types of programming errors (i.e., bugs), the next steps are to locate the programming codes which cause the errors and to fix the errors. This section provides guidelines which facilitate the debugging process.

In general, two useful debugging tools are breakpoints and variable viewers. A breakpoint is a place where a program is intentionally stopped during an execution. By strategically placing breakpoints in a program, programmers can easily find out the statement(s) responsible for an error. A variable viewer, on the other hand, allows the programmers to determine the values of variables and analyze the cause of an error.

There are two debugging methods in NS2. The first method is to use debugging tools. For Tcl, NS2 supports Don Libs' debugger [26], while the standard GNU debugger [27] can be used to debug the C++ codes. The second method is to manually debug the program. Table 14.1 shows a list of OTcl and C++ commands which can be used for manual debugging.

Example 14.1. To debug NS2 codes, it is usually useful to identify objects and/or the types of objects. Consider two OTcl commands – namely `show-target-class` and `show-target-address` in Program 14.1. These two OTcl commands show the class and address, respectively, of the target of a `TcpAgent` object.

In Example 10.1 which implements the network in Fig. 9.3, insert the following lines immediately before Line 15:

```
1 puts "The reference string for $tcp is $tcp"
2 puts "Press RETURN to start the simulation!!"
3 gets stdin
4 $ns at 3.1 "$tcp show-target-class"
5 $ns at 5.1 "$tcp show-target-address"
6 $ns at 10.1 "$ns halt"
7 $ns run
```

Table 14.1 Debugging commands in OTcl and C++ domains

Tools	OTcl	C++
Breakpoints	"gets stdin"	"getchar()","cin"
Variable viewer	"puts"	"printf(...)","cout"
Simulation time	"Simulator::now"	"Scheduler::clock()"
Cross-domain function invocation	Tcl/OTcl Commands	"Tcl::evalf(...)"
Cross-domain variable retrieval	Bound variables	Bound variables
Simulator object retrieval	"Simulator::instance"	"Simulator::instance()"
Scheduler object retrieval	N/A	"Scheduler::instance()"
Passing an OTcl value to the C++ domain	N/A	"TclObject::result(...)"
C++ to OTcl variable conversion	N/A	"TclObject::name()"
OTcl to C++ variable conversion	N/A	"TclObject::lookup(...)"

```
>>ns tcp-dbg.tcl
The reference string for $tcp is _o55
Press RETURN to start the simulation!!
<RETURN>
3.1 OTcl class of TcpAgent::target is Classifier/
  Hash/Dest
Press RETURN to continue!!
<RETURN>
5.1 [$tcp target] returns OTcl reference string _o12
  and C++
 address 0xd655c0
5.1 Variable TcpAgent::target_ corresponds to OTcl
  reference string _o12 and C++ address 0xd655c0
Press RETURN to continue!!
<RETURN>
```

Here, the lines with <RETURN> are actually blank lines, where the program is paused and waits for a <RETURN> keystroke.

Lines 5–11 in Program 14.1 show the details of OTcl commands show-target -class{}. Line 6 retrieves the Simulator object and stores it in a variable "sim." In Line 7, the function evalf(...) of class Tcl evaluates the Tcl statement in the same manner as printf(...) (see also Fig. 14.1). It puts the values stored in target_->name() and sim->name() as the first and second

Program 14.1 The OTcl commands `show-target-class` and `show-target-address` of C++ class `TcpAgent`

```
    //~/ns/tcp/tcp.cc
1   int TcpAgent::command(int argc, const char*const* argv)
2   {
3      ...
4      Tcl& tcl = Tcl::instance();
5      if (strcmp(argv[1], "show-target-class") == 0) {
6         Simulator& sim = Simulator::instance();
7         tcl.evalf("puts [format \"%1.1f OTcl class of
              TcpAgent::target is [%s info class]\" [%s now] ]",
              target_->name(),sim.name());
8         cout<<"Press RETURN to continue!!\n\n";
9         getchar();
10        return (TCL_OK);
11     }
12     if (strcmp(argv[1], "show-target-address") == 0) {
13        Scheduler& sch = Scheduler::instance();
14        tcl.evalf("%s target",this->name());
15        Connector *conn=(Connector*)TclObject::lookup
            (tcl.result());
16        cout<<sch.clock()<<" [$tcp target] returns OTcl
            reference string "<<tcl.result()<<" and C++ address
            "<<conn <<"\n";
17        cout<<sch.clock()<<" Variable TcpAgent::target_
              corresponds to OTcl reference string "<<target_->
                name() <<" and C++ address "<<target_<<"\n";
18        cout<<"Press RETURN to continue!!\n\n";
19        getchar();
20        return (TCL_OK);
21     }
22     ...
23  }
```

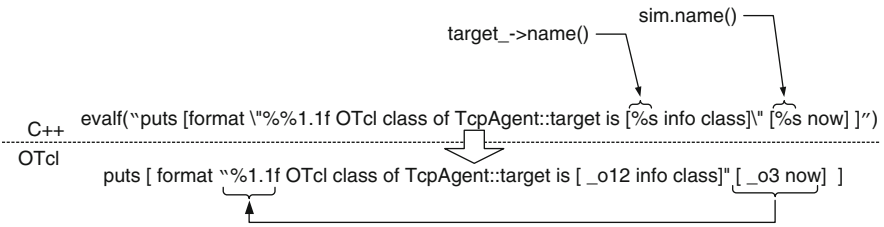

Fig. 14.1 Details of Line 7 in Program 14.1

arguments, respectively, and passes the entire statement to the Tcl interpreter. Here, the function `name()` defined in class `TclObject` is used to translate the C++ variables "`target_`" and "`sim`" to the OTcl reference strings.

Lines 12–21 in Program 3.1 show the details of the OTcl command `show-target-address{}`. Line 13 first retrieves the `Scheduler` object and stores it in a variable "`sch`." Line 14 asks the Tcl to interpret "`_o55 target`," where `_o55` is the OTcl reference string corresponding to the current TCP object. Line 15 uses function `result()` to obtain an OTcl reference string of the target of the `TcpAgent` object. It retrieves a pointer to the C++ object corresponding to a given string using the function `lookup(...)`. The retrieved pointer to the C++ object is then cast to a pointer to a `Connector` object and stored in a local variable "`conn`." Finally, Lines 16 and 17 display the target information on the screen. □

14.2 Variable Tracing

Variable tracing records changes in instvars of TclObjects under consideration.

14.2.1 Activation Process for Variable Tracing

Variable tracing can be activated in the OTcl domain using the following three steps:

- *Step 1:* Create a trace file whose name is `<filename>` and attach the file to a Tcl file channel variable whose name is `<fch>`, using the following syntax (see the detail for Tcl file channels in Sect. A.1.7):

```
set <fch> [open "<filename>" w]
```

- *Step 2:* Specify the instvar to be traced using the following syntax:

```
<obj> trace <var_name> [<tracer>]
```

  where `<obj>` and `<var_name>` refer to an OTcl object and the name of its instvar which needs to be traced. Optionally, programmers can provide a dedicated tracer object, as indicated by `<tracer>`, which keeps track of changes in the instvar `$<var_name>`. If a tracer is not given, NS2 will use `<obj>` as a tracer object.
- *Step 3:* Attach the created Tcl file channel to the tracer using the following syntax.

```
<obj> attach $<fch>,  or
<tracer> attach $<fch>
```

  where the upper line is used when no tracer is specified, and the lower line is used when a dedicated tracer is explicitly specified.

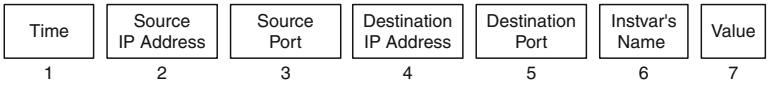

Fig. 14.2 Trace format defined in class `TcpAgent`

Example 14.2. Suppose we would like to trace the variable "`t_seqno_`" of a `TcpAgent` object `$tcp` in Example 10.1, and store the trace strings in a file "`trace.txt`." We may include the following statements into the Tcl simulation script:

```
1 $tcp trace t_seqno_
2 set trace_ch [open "trace.txt" w]
3 $tcp attach $trace_ch
```

Here, Line 1 tells the `Agent/TCP` object `$tcp` to trace its instvar "`t_seqno_`." Line 2 creates a trace file "`trace.txt`." Line 3 tells the tracer (i.e., `$tcp`) to send all its tracing information to the Tcl file channel `$trace_ch` attached to the created traced file. These three lines inform NS2 to record all the changes in the instvar "`t_seqno_`" associated with `$tcp` in the file "`trace.txt`." After simulation, the following trace file whose format complies with Fig. 14.2 is created:

```
...
4.06820 0 0 2 0 t_seqno_ 14
4.06986 0 0 2 0 t_seqno_ 15
4.07153 0 0 2 0 t_seqno_ 9
4.07153 0 0 2 0 t_seqno_ 10
4.08468 0 0 2 0 t_seqno_ 14
...
```

□

14.2.2 Traceable Variable

Variable tracing is applicable to *traceable* variables only. Traceable variables are the members of classes derived from class `TracedVar` – namely classes `TracedInt` and `TracedDouble`.

Traceable variables can be used as if they are regular `int` or `double` variables. However, if tracing is activated, all their changes will be recorded in a given trace file. The mechanism to track changes will be discussed later in the next section.

Program 14.2 shows examples of traceable and non-traceable variables of class `TcpAgent`. The variables "`t_seqno_`" (Line 3) and "`cwnd_`" (Line 4) are member of class `TracedInt` and `TracedDouble`, respectively. Therefore, they are traceable. The variable "`last_ack`" (Line 5), on the other hand, is of type `double`, and therefore not traceable.

Program 14.2 Example of traceable variables: `t_seqno_` and `cwnd_`

```
    //~/ns/tcp/tcp.h
1   class TcpAgent : public Agent {
2   protected:
3       TracedInt t_seqno_;
4       TracedDouble cwnd_;
5       int last_ack_;
6   }
```

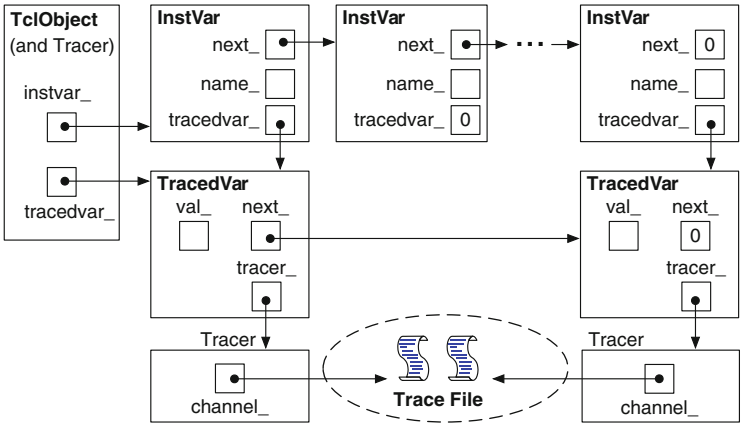

Fig. 14.3 An architecture of components involving in variable tracing

14.2.3 Components and Architecture for Variable Tracing

Variable tracing consists of five main components, where their relationship is shown in Fig. 14.3:

1. A TclObject: An object that contains traceable variables.
2. A traceable variable: A variable that needs to be traced. Its tracing capability is defined in a C++ class `TracedVar`, while its OTcl linkage is defined in a C++ class `InstVar`.
3. A tracer: An object that records changes in the traceable variable.
4. A Tcl channel: An object that attaches to a tracer to a trace file.
5. A trace file: A file that records trace strings.

14.2.3.1 TclObjects

Program 14.3 shows the declaration of class `TclObject`. In general, a TclObject may consist of several variables which are bound to `instvars` in the OTcl domain.

Program 14.3 Declaration of classes `TclObject` and `IntsVar`

```
    //~/tclcl/tclcl.h
 1  class TclObject {
 2  public:
 3      ....
 4      virtual void trace (TracedVar*);
 5  protected:
 6      int traceVar(const char* varName, TclObject* tracer);
 7      InstVar*   instvar_;
 8      TracedVar* tracedvar_
 9  }

    //~/tclcl/Tcl.cc
10  class InstVar {
11  protected:
12      InstVar(const char* name);
13      const char* name_;
14      TracedVar* tracedvar_;
15  public:
16      InstVar* next_;
17      inline const char* name() { return name_; }
18      inline TracedVar* tracedvar() { return tracedvar_; }
19      inline void tracedvar(TracedVar* v) { tracedvar_ = v; }
20  };
```

These variables are modeled by class InstVar. They together form a link list, and let the TclObject maintains only the pointer "`instvar_`" (Line 7) to the head of the link ed-list.

C++ class `TracedVar` provides a traceable variable with tracing capability. From Fig. 14.3, each traceable variable is associated with a `TracedVar` object. These `TracedVar` objects form a link list, whose head (`tracedvar_` in Line 8) is maintained by the associated TclObject. In summary, a TclObject maintains two pointers to two heads of link lists which model OTcl linkage (`instvar_`) and traceablility (`tracedvar_`).

14.2.3.2 Traceable Variables: OTcl Linkage

Shown in Lines 10–20 of Program 14.3, class `InstVar` binds a C++ class variable to an OTcl instvar. Class `InstVar` has three main variables: "`next_`," "`name_`," and "`tracedvar_`." The pointer "`next_`" provides a support to create a link list. The variable "`name_`" contains the OTcl instvar name. The pointer "`tracedvar_`" points to a `TracedVar` object, responsible for tracking changes. Class InstVar has three main functions: `name()`, `tracedvar()`, and `tracedvar(v)`. These functions are used to configure the class variables "`name`" and "`tracedvar_`" (Lines 17–19).

We note here that not all OTcl instvars need to be traceable. As a result, not all `InstVar` object has it pointer "`tracedvar_`" configured. From Fig. 14.3, the second `InstVar` object in the link list does not have its pointer point to a `TracedVar` object, and therefore not traceable.

14.2.3.3 Traceable Variables: Tracing Capability

Class `TracedVar` provides tracing capability to traceable variables. It overloads basic operators – including "`+`," "`-`," "`*`," and "`/`." When a `TracedVar` object executes one of these operators, its overloading operator executes the basic operation and reports the change in value to a tracer.

NS2 implements `TracedVar` objects through an abstract class `TracedVar`. In Program 14.4, class `TracedVar` has four main variables:

`next_`	A pointer to the next `TracedVar` object (Line 12)
`name_`	OTcl name of the instvar associated with this `TracedVar` object (Line 15)
`owner_`	A pointer to a TclObject which contains this `TracedVar` object (Line 16)
`tracer_`	A pointer to the TclObject responsible for keeping track of the value changes (Line 17)

Class `TracedVar` has one pure virtual function and six regular functions. The pure virtual function `value(...)` returns the value of the `TracedVar` object[1] (Line 5). The other six functions act as interface functions to set and retrieve variables of class `TracedVar` (Lines 6–11).

Class `TracedInt` declares an `int` variable "`val_`" (Line 28) to store its current value. It also defines a value assignment function, `assign(newval)`, in Lines 30–37. This function stores the input argument "`newval`" in the class variable "`val_`" (Line 34), and asks the associated tracer to track the value change (Line 36). By forcing all value assignment – including all overloading operators – to use this function, class `TracedInt` ensures that all the changes would be reported to the tracer.

14.2.3.4 Tracers

A tracer is a TclObject responsible for recording changes in value of traceable variables. Program 14.5 shows the details of function `trace(v)`. The implementation of this function at the base class aims at error reporting. From Lines 1–4, the error messages would be shown on the screen, if the function `trace(v)` is

[1]Since the base class `TracedVar` does not define a variable to store the value, the function `value(...)` must be declared as pure virtual.

Program 14.4 Declaration of classes `TracedVar` and `TracedInt`, and the function `assign` of class `TracedInt`

```
    //~/tclcl/tracedvar.h
 1  class TracedVar {
 2  public:
 3      TracedVar();
 4      virtual ~TracedVar() {}
 5      virtual char* value(char* buf, int buflen) = 0;
 6      inline const char* name() { return (name_); }
 7      inline void name(const char* name) { name_ = name; }
 8      inline TclObject* owner() { return owner_; }
 9      inline void owner(TclObject* o) { owner_ = o; }
10      inline TclObject* tracer() { return tracer_; }
11      inline void tracer(TclObject* o) { tracer_ = o; }
12      TracedVar* next_;
13  protected:
14      TracedVar(const char* name);
15      const char* name_;
16      TclObject* owner_;
17      TclObject* tracer_;
18  };

19  class TracedInt : public TracedVar {
20  public:
21      TracedInt() : TracedVar() {}
22      TracedInt(int v) : TracedVar(), val_(v) {}
23      virtual ~TracedInt() {}
24      inline int operator++() { assign(val_ + 1); return
        val_; }
25      inline int operator=(int v) { assign(v); return val_; }
26  protected:
27      virtual void assign(const int newval);
28      int val_;
29  };

    //~/tclcl/tracedvar.cc
30  void TracedInt::assign(int newval)
31  {
32      if (val_ == newval)
33          return;
34      val_ = newval;
35      if (tracer_)
36          tracer_->trace(this);
37  }
```

invoked. This implementation forces all the derived classes to override the function `trace(v)`. Otherwise, an error message will be shown on the screen at runtime when the function `trace(v)` of class `TclObject` is executed.[2]

Program 14.5 The Functions `trace` of classes `TclObject` and `TcpAgent`, and the function `traceVar` of class `TcpAgent`

```
    //~/tclcl/Tcl.cc
1   void TclObject::trace(TracedVar*)
2   {
3       fprintf(stderr, "SplitObject::trace called in
                                the base class of %s\n",name_);
4   }

    //~/ns/tcp/tcp.cc
5   void TcpAgent::trace(TracedVar* v)
6   {
7           traceVar(v);
8   }
9   void TcpAgent::traceVar(TracedVar* v)
10  {
11      Scheduler& s = Scheduler::instance();
12      char wrk[TCP_WRK_SIZE];
13      double curtime = &s ? s.clock() : 0;
14      if (v == &cwnd_)
15          ...
16      else if (v == &t_rtt_)
17          ...
18      else
19          snprintf(wrk, TCP_WRK_SIZE,
                "%-8.5f %-2d %-2d %-2d %-2d %s %d\n",
                curtime, addr(), port(), daddr(), dport(),
                v->name(), int(*((TracedInt*) v)));
20      (void)Tcl_Write(channel_, wrk, -1);
21  }
```

Consider class `TcpAgent` in Lines 5–21 of Program 14.5, as an example. Class `TcpAgent` overrides the function `trace(v)` by invoking its own function `traceVar(v)` in Line 7. The function `traceVar(v)` creates and stores a trace string in a local variable "`wrk`" (Line 19), and invokes function `Tcl_Write(...)` to write the string "`wrk`" to the trace channel "`channel_`" whose class is `Tcl_Channel` (Line 20).

[2]This is one of the common errors mentioned in Sect. 14.1.1.

14.2.3.5 Trace Channels

A trace channel is a Tcl channel[3] used specifically to transport trace strings to a trace file. In most case, it is attached to a trace file and sends all received trace stings to the attached trace file. For example, Line 20 in Program 14.5 simply prints the trace string to the attached trace file via the statement "`Tcl_Write(channel_, wrk, -1)`."

14.2.3.6 Trace Files and Trace Format

A trace file is a text file which collects all trace strings throughout the simulation. The format of trace strings is usually defined by the tracer under the function `trace(v)` For example, Fig. 14.2 shows the format defined by a tracer whose class is `TcpAgent`. The trace format of class `TcpAgent` is defined in its function `trace(v)`, which in turn invokes function `TraceVar(v)` to print out a trace string as shown in Line 19 of Program 14.5.

14.2.4 Tracing in Action: An Example of Class `TcpAgent`

This section demonstrates via an example how variable tracing actually takes place at runtime. Consider a C++ statement "`t_seqno_++`" invoked from within the class `TcpAgent`. The key procedures are shown in Fig. 14.4.

First, the operator "++" of a `TracedVar` object is overloaded by a function `operator++()` defined in Line 24 of Program 14.4. The function `operator++()` executes "`assign(val_+1)`" to increment the value stored in the variable "`val_`" by 1 and record the change. From within the function `assign(newval)`, a `TracedInt` "`t_seqno_`" executes "`tracer_->trace(this)`", where "`tracer_`" and "`this`" are the associated tracer and the address of the `TracedVar` object, respectively (Line 36 in Program 14.4). Since, in this case, the variable "`tracer_`" is a `TcpAgent` object, the function `trace(v)` in Lines 5–8 of Program 14.5 is invoked, and a trace string is printed to the trace file according to the format specified in Line 19 of Program 14.5.

14.2.5 Setting Up Variable Tracing

As we have seen, in Sect. 14.2.1, that an activation of variable tracing has three major steps: (1) creating a trace file, (2) specifying traceable variables, and (3) attaching the trace file to a tracer. Since Step 1 is discussed in Sect. A.1.7, this section focuses on the latter two steps.

[3] See the details of Tcl channel in Sect. A.1.7.

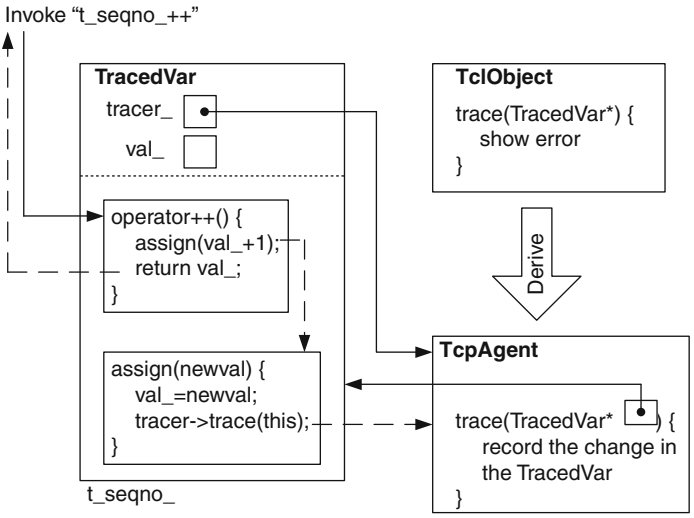

Fig. 14.4 The mechanism of function `operator++()` of class `TcpAgent`

14.2.5.1 Specifying Traceable Variables

Again, NS2 specifies an active traceable variable whose name is `<var_name>` via the following OTcl statement:

```
$obj trace <var_name> [$tracer]
```

where `$obj` is a TclObject, and the optional `$tracer` is a tracer. If the argument `$tracer` is not present, the TclObject `$obj` will be used as a tracer.

As shown in Program 14.6, the OTcl command `trace{...}` informs a TclObject (i.e., `$obj`) to trace its instvar `<var_name>`. Line 7 stores the third input argument (i.e., `argv[3]`), if any, in a local variable "`tracer`." Then, Line 8 feeds the second input argument (e.g., `<var_name>`) as well as the variable "`tracer`" to the function `traceVar(...)`. If the caller did not provide the third input argument, the pointer "`this`" would instead be used (Line 5).

The details of function `traceVar(varName,tracer)` are shown in Lines 13–24 of Program 14.6.[4] This function creates a connection from a `TracedVar` object whose OTcl instvar name is "`varName`" to a pointer which points to a tracer object "`tracer`." Lines 15 and 16 locate an entry of the `InstVar` link list whose variable "`name_`" matches with the string "`varName`." When the

[4]This function is different from the function `TraceVar(v)` of class `TcpAgent` in Program 14.5, since the number and types of input arguments are different.

Program 14.6 OTcl command `trace` and the function `TraceVar` of class `TclObject`

```
    //~/tclcl/Tcl.cc
 1  int TclObject::command(int argc, const char*const* argv)
 2  {
 3      if (argc > 2) {
 4          if (strcmp(argv[1], "trace") == 0) {
 5              TclObject* tracer = this;
 6              if (argc > 3)
 7                  tracer = TclObject::lookup(argv[3]);
 8              return traceVar(argv[2], tracer);
 9          }
10      }
11      return (TCL_ERROR);
12  }

13  int TclObject::traceVar(const char* varName, TclObject*
    tracer)
14  {
15      for (InstVar* p = instvar_; p != 0; p = p->next_) {
16          if (strcmp(p->name(), varName) == 0) {
17              if (p->tracedvar()) {
18                  p->tracedvar()->tracer(tracer);
19                  tracer->trace(p->tracedvar());
20                  return TCL_OK;
21              }
22          }
23      }
24  }
```

entry is found, Line 17 ensures that the matching instvar contains a reference to a `TracedVar` object. Then, Line 18 informs the matching instvar to use the input tracer object `*tracer` as its tracer (see Line 11 in Program 14.4). Line 19 informs the "`tracer`" (i.e., the input argument) to create a trace string for the matched instvar and write the string in the attached traced file via the trace channel, "`channel_`" (see Lines 5–20 in Program 14.5).

14.2.5.2 Attaching a Trace File to a Tracer

NS2 attaches a trace file whose associated Tcl file channel is `$fch` to a tracer `$tracer` through the following OTcl statement:

```
$tracer attach $fch
```

The tracing channel variable "`channel_`" whose class is `Tcl_Channel` is defined in class `Agent` (Line 4 in Program 14.7). Furthermore, the OTcl command `attach` is defined in the class `TcpAgent` whose details are shown in Lines 10–19

Program 14.7 Declaration of class `Agent` and its OTcl command `attach`

```
    //~/ns/common/agent.h
 1  class Agent : public Connector {
 2  ...
 3  protected:
 4      Tcl_Channel channel_;
 5  ...
 6  }

    //~/ns/tcp/tcp.cc
 7  int TcpAgent::command(int argc, const char*const* argv)
 8  {
 9      ...
10      if (strcmp(argv[1], "attach") == 0) {
11          int mode;
12          const char* id = argv[2];
13          channel_ = Tcl_GetChannel(tcl.interp(), (char*)id,
              &mode);
14          if (channel_ == 0) {
15              tcl.resultf("trace: can't attach %s for
                  writing", id);
16              return (TCL_ERROR);
17          }
18          return (TCL_OK);
19      }
20      ...
21  }
```

in Program 14.7. Here, Line 12 converts the input file name to a string "`id`". Line 13 retrieves a Tcl file channel corresponding to "`id`", and stores it in the variable "`channel_`". After this point, a connection to a trace file is created from within a tracer, and the tracer is able to write trace strings to the attached trace file through its variable "`channel_`".

14.3 Packet Tracing

Packet tracing records the details when packets pass through network checkpoints, where Trace objects are installed. Packet tracing can be activated using the following OTcl statement:

```
$ns trace-all $file
```

where `$ns` and `$file` are the `Simulator` object and a Tcl file channel, respectively.

Example 14.3. Consider Example 9.1. Insert the following OTcl statements after Line 2 in Example 9.1:

```
set f [open out.tr w]
$ns trace-all $f
```

where the upper line creates a Tcl file channel attached to a file "`out.tr`" and stores it in a variable "`f`." The lower line informs NS2 to activate the packet tracing mechanism and to record trace strings to the trace file "`out.tr`" via the Tcl file channel `$f`.

After adding the following statements at the end of the Tcl scripting file and running the simulation:

```
$ns at 0.0 "$cbr start"
$ns at 10.0 "$ns halt"
$ns run
```

the trace result will be stored in the file "`out.tr`." The following lines are a part of the file "`out.tr`":

```
 //out.tr
+ 0 0 2 cbr 210 ------- 0 0.0 2.0 0 0
- 0 0 2 cbr 210 ------- 0 0.0 2.0 0 0
r 0.002336 0 2 cbr 210 ------- 0 0.0 2.0 0 0
+ 0.00375 0 2 cbr 210 ------- 0 0.0 2.0 1 1
- 0.00375 0 2 cbr 210 ------- 0 0.0 2.0 1 1
...
```

□

In the following, Sects. 14.3.1 and 14.3.2 discuss the OTcl configuration method and C++ internal mechanism implementation, respectively, of packet tracing. Sections 14.3.3 discusses the details of the packet tracing helper class `BaseTrace`. Various types of packet tracing objects are presented in Sect. 14.3.4. Finally, the packet trace formats are shown in Sect. 14.3.5.

14.3.1 OTcl Configuration Interfaces

This section demonstrates how NS2 sets up packet tracing using the instproc `trace-all{}`. For brevity, the following discussion will be based mainly on a simplex-link with a droptail queue only. The readers are encouraged to look through the NS2 program and find out more about packet tracing. As shown in Fig. 14.5, packet tracing can be set up in two main steps:

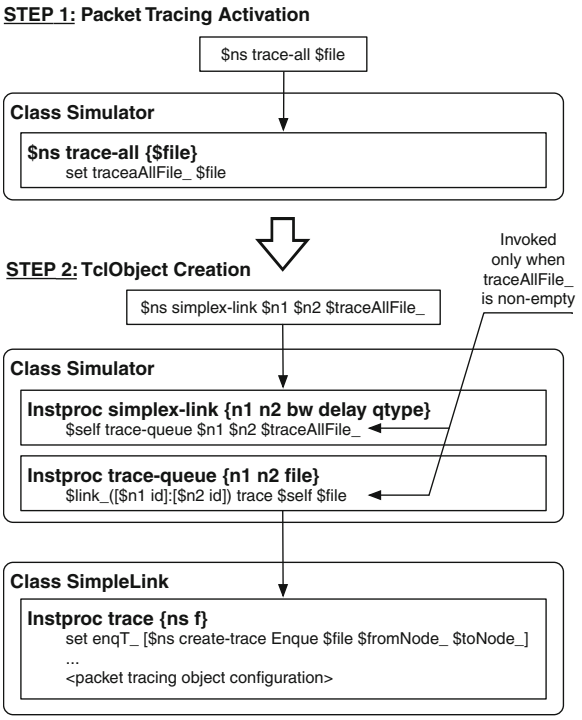

Fig. 14.5 The packet tracing configuration process of a `SimpleLink` object

Program 14.8 The instproc `trace-all` of class `Simulator`

```
     //~/ns/tcl/lib/ns-lib.tcl
1 Simulator instproc trace-all file {
2     $self instvar traceAllFile_
3     set traceAllFile_ $file
4 }
```

Step 1: Packet Tracing Activation The Instproc `trace-all{file}` of Class `Simulator`

Again, packet tracing can be activated using the instproc `trace-all{file}` of class `Simulator` whose details are shown in Program 14.8. This instproc stores the Tcl file channel "`file`" in a flag instvar "`traceAllFile_`" (Line 2), which indicates that packet tracing is enabled. When a TclObject is created, packet tracing objects are inserted if the instvar "`traceAllFile_`" is nonempty.

Program 14.9 The instprocs `simplex-link` and `trace-queue` of class `Simulator`

```
   //~/ns/tcl/lib/ns-lib.tcl
 1 Simulator instproc simplex-link { n1 n2 bw delay qtype args }
     {
 2     $self instvar link_
 3     set sid [$n1 id]
 4     set did [$n2 id]
 5     ...
 6     set q [new Queue/$qtype]
 7     set link_($sid:$did) [new SimpleLink $n1 $n2 $bw
         $delay $q]
 8     ...
 9     set trace [$self get-ns-traceall]
 10    if {$trace != ""} {
 11           $self trace-queue $n1 $n2 $trace
 12    }
 13 }

 14 Simulator instproc trace-queue { n1 n2 {file ""} } {
 15    $self instvar link_ traceAllFile_
 16    $link_([$n1 id]:[$n2 id]) trace $self $file
 17 }
```

Step 2: TclObject Creation

Step 2.1: The instproc `simplex- link{...}` of Class Simulator

The instproc `simplex-link{...}` of class `Simulator` creates a link between two nodes. Program 14.9 shows a part of the instproc `simplex-link{...}` related to packet tracing. Lines 6 and 7 create a `SimpleLink` object connecting node `n1` to node `n2`. Line 9 stores the instvar `traceAllFile_` in a local variable `trace`, and Lines 10–12 execute "`$self trace-queue{n1 n2 trace}`", if packet tracing was activated earlier.

Step 2.2: The instproc `trace-queue{n1 n2 file}` of Class Simulator

Lines 14–17 in Program 14.9 show the details of function `trace-queue{n1 n2 file}`. Again, class `Simulator` has an instvar "`link_`"– an associative array that contains the instances of all links in a simulation. The index of the instvar "`link_`" is of format "`sid:did`," where "`sid`" and "`did`" are IDs of the beginning and ending nodes of the instance "`$link_[sid:did]`."

The instproc `trace-queue{n1 n2 file}` invokes the instproc `trace{ns file}` associated with the `SimpleLink` object `link_([$n1 id]:[$n2 id])` (see Line 16), to create and configure packet tracing components of the `SimpleLink` object.

Program 14.10 The instproc `trace` of class `SimpleLink` and the instproc `create-trace` of class `Simulator`

```
     //~/ns/tcl/lib/ns-link.tcl
  1  SimpleLink instproc trace { ns f {op ""} } {
  2      $self instvar enqT_ deqT_ drpT_ queue_ link_ fromNode_
           toNode_
  3      $self instvar rcvT_ ttl_ trace_
  4      $self instvar drophead_
  5      set trace_ $f
  6      set enqT_ [$ns create-trace Enque $f $fromNode_
           $toNode_ $op]
  7      set deqT_ [$ns create-trace Deque $f $fromNode_
           $toNode_ $op]
  8      set drpT_ [$ns create-trace Drop $f $fromNode_
           $toNode_ $op]
  9      set rcvT_ [$ns create-trace Recv $f $fromNode_
           $toNode_ $op]
  10     $self instvar drpT_ drophead_
  11     set nxt [$drophead_ target]
  12     $drophead_ target $drpT_
  13     $drpT_ target $nxt
  14     $queue_ drop-target $drophead_
  15     $deqT_ target [$queue_ target]
  16     $queue_ target $deqT_
  17     $self add-to-head $enqT_
  18     $rcvT_ target [$ttl_ target]
  19     $ttl_ target $rcvT_
  20 }

     //~/ns/tcl/lib/ns-lib.tcl
  21  Simulator instproc create-trace { type file src dst
        {op ""} } {
  22      $self instvar alltrace_
  23      set p [new Trace/$type]
  24      $p set src_ [$src id]
  25      $p set dst_ [$dst id]
  26      lappend alltrace_ $p
  27      if {$file != ""} {
  28          $p attach $file
  29      }
  30     return $p
  31 }
```

Step 2.3: The instproc `trace{ns f}` of Class SimpleLink

As shown in Program 14.10, the instproc `trace{ns f}` of class `SimpleLink` takes two input arguments: the Simulator "`ns`" and a Tcl file channel "`f`." Line 5 stores the input Tcl file channel `$f` in the instvar "`trace_`." Lines 6–9 create packet tracing objects "`enqT_`," "`deqT_`," "`drpT_`," and "`rcvT_`," using the instproc `create-trace{...}` associated with the input `Simulator` object "`ns`." Lines 11–19 configure the created packet tracing objects as indicated in Fig. 7.1.

Step 2.4: The instproc `create-trace{type file src dst}` of Class Simulator

From Lines 21–31 in Program 14.10, the instproc `create-trace{type file src dst}` creates and configures a packet tracing object whose type is "`type`." Line 23 first creates a packet tracing object with type specified in "`type`." Lines 24 and 25 configure the member variables "`src_`" and "`dst_`," respectively, of the created packet tracing object "`p`." Line 26 stores the created packet tracing object in an instvar "`alltrace_`" of the Simulator. Lines 27–29 attach a Tcl file channel "`file`" to the created packet tracing object. Finally, Line 30 returns the created packet tracing object to the caller.

14.3.2 C++ Main Packet Tracing Class `Trace`

In C++, packet tracing objects are implemented using class `Trace` declared in Program 14.11, which is bound to an OTcl class with the same name (see Lines 15–23). From Line 1, class `Trace` derives from class `Connector` and can be inserted between two NsObjects to record the details of traversing packets. As a connector, a packet tracing object receives a packet "`*p`" by having its upstream object invoke its function `recv(p,h)`. Upon receiving a packet, it records the details of the packet in a trace file and forwards the packet to its downstream object.

14.3.2.1 Main C++ Variable of Class `Trace`

Class `Trace` consists of four main variables: "`src_`," "`dst_`," "`type_`," and "`pt_`." The variables "`src_`" (Line 3) and "`dst_`" (Line 4) store addresses of the upstream and downstream nodes, respectively, of this `Trace` object. The variable "`type_`" in Line 10 indicates the type of the `Trace` object. Despite its `int` type, the true meaning of this variable is `char` equivalent. For example, the types of objects which trace packet enquing and dequing are "`+`" and "`-`," which correspond to decimal values of 43 and 45, respectively. Finally, the pointer "`pt_`" in Line 9 is a reference to a `BaseTrace` object, which provides the basic functionalities for packet tracing. We shall discuss the details of class `BaseTrace` later in Sect. 14.3.3.

14.3.2.2 Main C++ Functions of Class `Trace`

Class `Trace` has three following main functions: the constructor, function `recv(p,h)`, and function `format(tt,s,d,p)`.

Program 14.11 Declaration of class `Trace` and their constructors

```
    //~/ns/trace/trace.h
 1  class Trace : public Connector {
 2  protected:
 3      nsaddr_t src_;
 4      nsaddr_t dst_;
 5      virtual void format(int tt, int s, int d, Packet* p);
 6  public:
 7      Trace(int type);
 8      ~Trace();
 9      BaseTrace *pt_;
10      int type_;
11      int command(int argc, const char*const* argv);
12      static int get_seqno(Packet* p);
13      void recv(Packet* p, Handler*);
14  };

    //~/ns/trace/trace.cc
15  class TraceClass : public TclClass {
16  public:
17      TraceClass() : TclClass("Trace") { }
18      TclObject* create(int argc, const char*const* argv) {
19          if (argc >= 5)
20              return (new Trace(*argv[4]));
21          return 0;
22      }
23  } trace_class;

24  Trace::Trace(int type) : Connector(), pt_(0), type_(type)
25  {
26      bind("src_", (int*)&src_);
27      bind("dst_", (int*)&dst_);
28      pt_ = new BaseTrace;
29  }

    //~/ns/tcl/lib/ns-trace.tcl
30  Trace instproc init type {
31      $self next $type
32      $self instvar type_
33      set type_ $type
34  }
```

The Constructors

Lines 24–29 and 30–34 show the constructors of a C++ class `Trace` and the bound OTcl class `Trace`, respectively. The OTcl constructor simply stores the input argument in its instvar "`type_`" (Line 33). Similarly, the C++ constructor stores the input argument in variable "`type_`" (Line 24). It also binds variables "`src_`" and "`dst_`" to the instvars with the same name (Lines 26 and 27) and creates a new `BaseTrace` object "`*pt_`" (Line 28).

Program 14.12 Function `recv` of class `Trace`

```
    //~/ns/trace/trace.cc
1 void Trace::recv(Packet* p, Handler* h)
2 {
3     format(type_, src_, dst_, p);
4     pt_->dump();
5     if (target_ == 0)
6         Packet::free(p);
7     else
8         send(p, h);
9 }
```

Function `recv(p,h)`

Function `recv(p,h)` is the main packet reception function whose details are shown in Program 14.12. Line 3 invokes function `format(type_,src_, dst_,p)` to store the details of the packet "`*p`" in the internal variable "`wrk_`" of the associated `BaseTrace` object "`*pt_`." Line 4 executes "`pt_->dump()`" which tells the attached `BaseTrace` object to print packet details to its attached trace file. If the `Trace` object contains a valid downstream object, "`target_`," Line 8 will forward the packet "`*p`" to the downstream object. Otherwise, Line 6 will destroy the packet "`*p`."

Function Format `(tt,s,d,p)`

Shown in Programs 14.13 and 14.14, function `format(tt,s,d,p)` stores the packet details in the internal variable "`wrk_`" of the associated `BaseTrace` object "`*pt_`" (Lines 26–45). This function takes, as input arguments, the packet tracing type "`tt`," a source node ID "`s`," a destination node ID "`d`," and a pointer to an incoming packet "`*p`." Line 7 stores the packet type in a local variable "`name`." Lines 9–21 create a flag string and store it in a local variable "`flag`." Lines 22–25 retrieve addresses and ports of the source and the destination nodes. Finally, Lines 26–45 print out a packet trace string to variable "`pt_->wrk_`."[5] The packet trace format will be discussed in greater detail in Sect. 14.3.5.

14.3.2.3 Main OTcl Commands of a Packet Tracing Object

There are three OTcl main commands associated with the OTcl class `Trace`: `flush{}`, `detach{}`, and `attach{file}`. In Program 14.15, the OTcl command

[5] As we shall see in Sect. 14.3.3, the function `buffer()` of class `BaseTrace` simply returns the variable "`wrk_`."

Program 14.13 Function `format` of class `Trace`

```
    //~/ns/trace/trace.cc
 1  void Trace::format(int tt, int s, int d, Packet* p)
 2  {
 3      hdr_cmn *th = hdr_cmn::access(p);
 4      hdr_ip *iph = hdr_ip::access(p);
 5      hdr_tcp *tcph = hdr_tcp::access(p);
 6      packet_t t = th->ptype();
 7      const * name = packet_info.name(t);
 8      int seqno = get_seqno(p);
 9      char flags[NUMFLAGS+1];
10      for (int i = 0; i < NUMFLAGS; i++)
11          flags[i] = '-';
12      flags[NUMFLAGS] = 0;
13      hdr_flags* hf = hdr_flags::access(p);
14      flags[0] = hf->ecn_ ? 'C' : '-';
15      flags[1] = hf->pri_ ? 'P' : '-';
16      flags[2] = '-';
17      flags[3] = hf->cong_action_ ? 'A' : '-';
18      flags[4] = hf->ecn_to_echo_ ? 'E' : '-';
19      flags[5] = hf->fs_ ? 'F' : '-';
20      flags[6] = hf->ecn_capable_ ? 'N' : '-';
21      flags[7] = 0;
22      char *src_nodeaddr = Address::instance().
                                 print_nodeaddr(iph->saddr());
23      char *src_portaddr = Address::instance().
                                 print_portaddr(iph->sport());
24      char *dst_nodeaddr = Address::instance().
                                 print_nodeaddr(iph->daddr());
25      char *dst_portaddr = Address::instance().
                                 print_portaddr(iph->dport());
        ...
```

`flush{}` (Lines 5–10) clears the buffer of the attached Tcl channel by invoking `pt_->flush(ch)`, where "`ch`" is the attached Tcl channel. The OTcl command `detach{}` does not clear the channel buffer, but simply sets the pointer which points to the attached Tcl channel to Null (see Line 12). Finally, the OTcl command `attach{file}` sets the input "`file`" as the Tcl file channel (Lines 19 and 20).

14.3.3 C++ Helper Class BaseTrace

One of the main variables of class `Trace`, "`pt_`," is a pointer to a `BaseTrace` object. Class `BaseTrace` acts as an interface from a packet tracing object to a Tcl channel. Shown in Program 14.16, class `BaseTrace` is bound to an OTcl class with the same name. It has two main variables: "`channel_`" (Line 14) and "`wrk_`" (Line 15). While "`channel_`" models a Tcl channel, "`wrk_`" is a buffer which

Program 14.14 Function `format` of class `Trace` (cont.)

```
    //~/ns/trace/trace.cc
        ...
26      sprintf(pt_->buffer(),
27          "%c "TIME_FORMAT" %d %d %s %d %s %d %s.%s %s.%s
28          tt,
29          pt_->round(Scheduler::instance().clock()),
30          s,
31          d,
32          name,
33          th->size(),
34          flags,
35          iph->flowid(),
36          src_nodeaddr,
37          src_portaddr,
38          dst_nodeaddr,
39          dst_portaddr,
40          seqno,
41          th->uid(),
42          tcph->ackno(),
43          tcph->flags(),
44          tcph->hlen(),
45          tcph->sa_length() );
46      delete [] src_nodeaddr;
47      delete [] src_portaddr;
48      delete [] dst_nodeaddr;
49      delete [] dst_portaddr;
50  }
```

stores a trace string. At the construction, the Tcl channel "`channel_`" is set to Null (Line 24), and the trace string "`wrk_`" is allocated with memory space which can hold upto 1026 characters (Line 26).

The key functions of class `BaseTrace` include `channel(...)`, `buffer()`, `flush(channel)`, and `dump()`. The operations of the first three functions are fairly straightforward and are omitted for brevity. The function `dump()` (Lines 28–37 of Program 14.16) is responsible for dumping a trace string stored in the variable "`wrk_`" to the Tcl channel. Here, Line 30 retrieves and stores the length of the string "`wrk_`" in a local variable "n." Line 32 attaches an end-of-line character to "`wrk_`." Line 33 attaches zero to "`wrk_`" indicating the end of the string. Line 34 writes "`wrk_`" to the Tcl channel "`channel_`" using a function `Tcl_Write(...)`.

In common with class `Trace`, class `BaseTrace` has three main OTcl commands: `flush{}`, `detach{}`, and `attach{file}`. These three commands perform the same action as those in class `Trace`. We will omit the details of these three OTcl commands for brevity.

Program 14.15 Function `command` of class `Trace`

```
    //~/ns/trace/trace.cc
1   int Trace::command(int argc, const char*const* argv)
2   {
3       Tcl& tcl = Tcl::instance();
4       if (argc == 2) {
5           if (strcmp(argv[1], "flush") == 0) {
6               Tcl_Channel ch = pt_->channel();
7               if (ch != 0)
8                   pt_->flush(ch);
9               return (TCL_OK);
10          }
11          if (strcmp(argv[1], "detach") == 0) {
12              pt_->channel(0) ;
13              return (TCL_OK);
14          }
15      } else if (argc == 3) {
16          if (strcmp(argv[1], "attach") == 0) {
17              int mode;
18              const char* id = argv[2];
19              Tcl_Channel ch = Tcl_GetChannel(tcl.interp(),
                    (char*)id,&mode);
20              pt_->channel(ch);
21              if (pt_->channel() == 0) {
22                  tcl.resultf("trace: can't attach %s
                        for writing", id);
23                  return (TCL_ERROR);
24              }
25              return (TCL_OK);
26          }
27      }
28      return (Connector::command(argc, argv));
29  }
```

14.3.4 Various Types of Packet Tracing Objects

NS2 uses different types of packet tracing objects to trace packets at different places. For example, a `Trace/Enque` object is placed immediately before a queue to trace packets which enter the queue. The type (i.e., variable "`type_`") of a `Trace/Enque` object is "+," which is equivalent to 43 in decimal. When a packet passes through a `Trace/Enque` object, a line beginning with "+" is written onto the Tcl Channel.

Among all built-in OTcl packet tracing classes, the most common ones include:

- `Trace/Enque` ("+"): Trace packet arrival (usually at a queue)
- `Trace/Deque` ("-"): Trace packet departure (usually at a queue)

Program 14.16 Declaration, an OTcl binding class, the constructor of class `BaseTrace`, and the function `dump()` of class `BaseTrace`

```
    //~/ns/trace/basetrace.h
 1  class BaseTrace : public TclObject {
 2  public:
 3      BaseTrace();
 4      ~BaseTrace();
 5      virtual int command(int argc, const char*const* argv);
 6      virtual void dump();
 7      inline Tcl_Channel channel() { return channel_; }
 8      inline void channel(Tcl_Channel ch) {channel_ = ch; }
 9      inline char* buffer() { return wrk_ ; }
10      void flush(Tcl_Channel channel) { Tcl_Flush(channel); }
11      #define PRECISION 1.0E+6
12      #define TIME_FORMAT "%.15g"
13  protected:
14      Tcl_Channel channel_;
15      char *wrk_;
16  };

    //~/ns/trace/basetrace.cc
17  class BaseTraceClass : public TclClass {
18  public:
19         BaseTraceClass() : TclClass("BaseTrace") { }
20         TclObject* create(int argc, const char*const* argv) {
21            return (new BaseTrace());
22         }
23  } basetrace_class;

24  BaseTrace::BaseTrace() : channel_(0),
25  {
26    wrk_ = new char[1026];
27  }

28  void BaseTrace::dump()
29  {
30      int n = strlen(wrk_);
31      if ((n > 0) && (channel_ != 0)) {
32          wrk_[n] = '\n';
33          wrk_[n + 1] = 0;
34          (void)Tcl_Write(channel_, wrk_, n + 1);
35          wrk_[n] = 0;
36      }
37  }
```

- `Trace/Drop` ("d"): Trace packet drop (delivered to a drop-target)
- `Trace/Recv` ("r"): Trace packet reception at a certain node

where the characters in the parentheses are attributed to the variable "`type_`" defined in the C++ class `Trace`.

Program 14.17 Constructors of classes `Trace/Enque` and `Trace/Deque`, and the C++ binding class for the OTcl class `Trace/Deque`

```
    //~/ns/tcl/lib/ns-trace.tcl
1   Class Trace/Enque -superclass Trace
2   Trace/Enque instproc init {} {
3       $self next "+"
4   }
5   Trace/Deque instproc init {} {
6       $self next "-"
7   }

    //~/ns/trace/trace.h
8   static class DequeTraceClass : public TclClass {
9   public:
10      DequeTraceClass() : TclClass("Trace/Deque") { }
11      TclObject* create(int args, const char*const* argv) {
12      if (args >= 5)
13              return (new DequeTrace(*argv[4]));
14          return NULL;
15      }
16  } dequetrace_class;
```

Among these four classes, only class `Trace/Deque` has an implementation in the C++ domain. The main difference among the above four packet tracing objects lies in their constructors. As shown in Program 14.17, the OTcl class `Trace/Enque` derives from an OTcl class `Trace` (Line 1), while the OTcl class `Trace/Deque` is mapped to the C++ class `DequeTrace` (Lines 8–16). Lines 3 and 6 show that the classes `Trace/Enque` and `Trace/Deque` are constructed with characters "+" and "-," respectively. From Line 24 of Program 14.11, these characters are stored in its C++ variable "`type_`" defined at the base class `Trace`.

As an example, consider the process of creating a `Trace/Enque` object in Fig. 14.6. The process starts when a statement "`new Trace/Enque`" is executed (e.g., Line 23 in Program 14.10). From within the OTcl constructor, the type "+" is repeatedly fed to the constructor up the hierarchy by the statement "`$self next "+"`." When class `SplitObject` is reached, the interpreter executes "`create-shadow "+"`," which in turn invokes the function `create()` of class `TraceClass` in the C++ domain. From Line 24 in Program 14.11, the constructor of class `Trace` is invoked, and the type "+" is fed as an input argument. Since the constructor takes an integer as an input argument, the ASCII code "+" is converted into a decimal value "43." Finally, the constructor stores the input argument (i.e., "43" in this case) in its variable "`type_`."

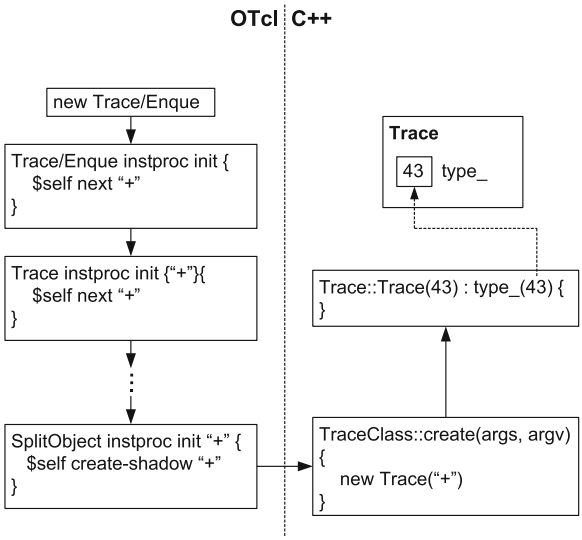

Fig. 14.6 Construction of a `Trace/Enque` object

14.3.5 Format of Trace Strings for Packet Tracing

The final product of packet tracing is usually a trace file. Each line in a trace file – usually called a trace record – follows a predefined packet trace format. This section discusses three main types of packet trace format. First, normal packet trace format is associated with regular wired network simulation. Second, wireless packet trace format is the default format when running a wireless network simulation. This format is sometimes referred to as old wireless trace format or *CMU wireless trace format*. Finally, the new wireless trace format is the most comprehensive built-in packet tracing in NS2. Among these formats, the former two follow predefined structures. The interpretation of these traced formats is mainly based on the positions of strings on a trace record. The last type, on the other hand, is structureless. The interpretation is based on the label located before a trace string.

14.3.5.1 Normal Packet Trace Format

Packet trace format is defined in the function `format(...)` of class `Trace` (Programs 14.13 and 14.14). In a normal case, each trace record (i.e., each line in a trace file) follows the format shown in Fig. 14.7, where each box and the space between two boxes represent a trace string and a space, respectively, in a trace record. A trace record for the normal trace format contains 12 fields, each of which is indicated by the column number below each box.

Fig. 14.7 Packet trace format

- Event Type: The type (i.e., variable "`type_`") of the `Trace` object which generates the trace string. Most widely used event types are shown below. The complete list of event types is given in file *~ns*/tcl/lib/ns-trace.tcl.
  - "`+`" which represents a packet enque event,
  - "`-`" which represents a packet deque event,
  - "`r`" which represents a packet reception event,
  - "`d`" which represents a packet drop (e.g., sent to "`dropHead_`") event, and
  - "`c`" which represents a packet collision event at the MAC level.
- Time: When the packet trace record is created.
- Sending Node and Receiving Node: IDs of the nodes located before and after, respectively, the tracing object which creates this trace record.
- Payload Type: Name of the payload type, as specified in Program 8.9.
- Packet Size: Size of the packet in bytes.
- Flags: A 7-digit flag string is defined in Lines 9–21 of Program 14.13. Each flag digit is set to "`-`," when disabled. Otherwise, it will be set as follows:
  - 1st: Set to "`E`" if an ECN (Explicit Congestion Notification) echo is enabled.
  - 2nd: Set to "`P`" if the priority in the IP header is enabled.
  - 3rd: Not in use.
  - 4th: Set to "`A`" if the corresponding TCP takes an action on a congestion (e.g., closes the congestion window).
  - 5th: Set to "`E`" if the congestion has occurred.
  - 6th: Set to "`F`" if the TCP fast start is used.
  - 7th: Set to "`N`," when the transport layer protocol is capable of using Explicit Congestion Notification (ECN).
- Flow ID: The field "`fid_`" of the IP packet header.
- Source Address and Destination Address: The source and destination addresses of a packet specified in an IP packet header. For a flat addressing scheme, the format of these two fields is "`a.b`," where "`a`" is the address and "`b`" is the port.
- Sequence Number: The sequence number corresponding to the protocol specified in the packet payload type.
- Packet Unique ID: A unique ID stored in the common packet header.

Example 14.4. Consider the following trace record from a trace file:

```
- 1.257849 0 2 tcp 1040 ------- 2 0.1 3.2 4 118
```

The interpretation for this trace line is as follows. At "`1.257849`" second, a packet with unique packet ID is "`118`" exits the queue ("`-`") of the link connecting node

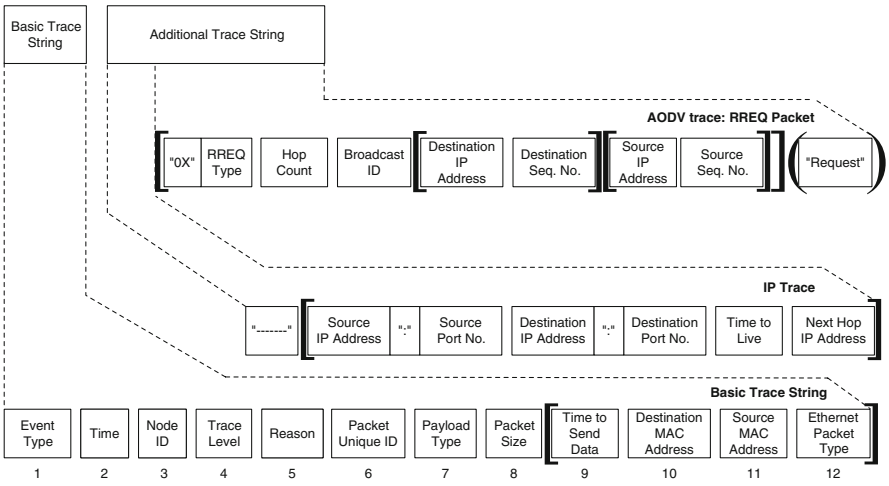

Fig. 14.8 Wireless trace format

"0" to node "2". The packet is a "tcp" packet whose size is "1040" bytes. This packet belongs to the TCP flow number "2" and is tagged with sequence number "4". The source and destination sockets of the packet are "0.1" and "3.2", respectively, where the address and port are separated by a dot symbol (".").

14.3.5.2 Wireless Packet Trace Format

Wireless packet trace format is activated automatically when running a wireless network (see Chap. 12 for wireless networking in NS2). As shown in Fig. 14.8, wireless packet trace format consists of two main parts. The former – *Basic trace string* – is mandatory. It appears on every wireless trace record. The latter – *Additional trace string* – is protocol-specific. Its format depends on the entity being traced. While NS2 supports several traceable entity (e.g., TCP, CBR, TORA), Fig. 14.8 shows only two examples of additional trace string format – namely IP trace and AODV-RREQ trace.[6] A more comprehensive list of trace format can be found in the files *~ns*/trace/cmu-trace.h,cc, in [17], and in [11].

The following notations are also adopted in addition to those used in Fig. 14.7. Strings in quotation marks are those that appear in a trace record as they are. Also, a trace record contains both square brackets and parentheses. Their locations in a trace record are shown in Fig. 14.8. The column numbers are given for the basic tracing only, since the field location for additional trace string is protocol-specific.

Fields in a wireless trace record are shown below:

[6] See the details of AODV in Sect. 12.2.

- Trace level: The common levels are `AGT` for agent trace, `RTR` for routing trace, `MAC` for MAC trace. See the list of possible values in the file *~ns*/trace/cmu-trace.cc.
- Reason: The reason for this trace (e.g., "`NRTE`" for No RouTe Entry).
- Time to Send Data: Expected duration required to transmitted this packet over the wireless channel as indicated by the underlying MAC protocol.
- Ethernet Packet Type: Currently, there are only two Ethernet packet types:
  - A general IP packet: The value is "`ETHERTYPE_IP`" defined as "`0x0800`."
  - An ARP packet: The value is "`ETHERTYPE_ARP`" defined as "`0x0860`."
- RREQ Type: Type as indicated in the field "`rq_type`" of the `hdr_aodv_request struct` data type. By default, the value is "`AODVTYPE_RREQ`" defined as "`0x02`."

Example 14.5. Consider the following wireless trace record:

```
s 21.500275000 _0_ MAC  --- 0 AODV 106 [0 ffffffff 0
  800] ------ [0:255 -1:255 30 0] [0x2 1 4 [1 0]
  [0 10]](REQUEST)
```

The interpretation for the basic trace format is as follows. The node "`_0_`" sends (i.e., "`s`") at time "`21.500275`" second. The trace level is at the "`MAC`" layer. The packet has the unique ID of "`0`," contains an "`AODV`" payload type, and is "`106`" bytes in size. The MAC protocol assumes that the delay over the underlying wireless channel is zero "`0`." Its source and destination MAC addresses are "`0`" and "`ffffffff`," respectively. Finally, this packet is an IP packet running over an Ethernet network (i.e., "`800`").

For the IP trace format, this packet is tagged with source and destination addresses of "`0`" and "`1`," respectively. The ports for both source and destination are "`255`." The time to live and the address of the next hop node are "`30`" hops and "`0`," respectively.

Finally, for AODV trace format, this packet is an RREQ packet (i.e., "`0x2`.") The number of hop counts is "`1`" and the broadcast ID is "`4`." The destination IP address and sequence number are `1` and 0, respectively. The source IP address and sequence number are `0` and 10, respectively. Finally, the string "`(REQUEST)`" confirms that this is an RREQ packet. □

14.3.5.3 New Wireless Trace Format

New wireless trace format is the most comprehensive trace format in NS2. It can be activated in a simulation of wireless networks by the following OTcl statement:

```
$ns_ use-newtrace
```

where `$ns_` is the `Simulator` object. The above OTcl statement must be placed before the statement "`$ns_ trace-all <ch>`." Otherwise the new wireless trace will not be activated.

Again, new wireless trace format is structureless. Like the former two trace format, the new wireless trace begins with an event type – which can be "send (s)," "receive (r)," "drop (d)," or "forward (f)." However, subsequent trace strings follow the following syntax:

```
-<tag> <tg_value> [-<subtab> <stg_value>]
```

where <tag> is a one-letter or two-letter option tag, indicating the meaning of the following <tg_value>. In addition, some tags require an optional <subtag> whose value is specified in <stg_value>. The list of option tags and sub-tags as well as their meaning is given below:

General information:

-t	Time

Node information (-N?):

-Ni	Node ID	-Nx, -Ny, -Nz	Node coordinate
-Nl	Trace level: AGT/RTR/MAC	-Nw	Reason
-Ne	Energy level (default = -1, i.e., not tracing energy level)		

IP Information (-I?):

-Is	Source (addr.port)	-Id	Destination (addr.port)
-It	Packet type	-Il	Packet size
-If	Flow ID	-Ii	Unique ID
-Iv	Time to live		

Next hop (-H?):

-Hs	ID of this node	-Hs	ID of the next hop node

MAC level information (-M?):

-Ms	Source Ethernet address	-Md	Destination Ethernet address
-Mt	Ethernet type	-Ma	Packet transmission time

Application information–ARP (-P arp -P?):

-Ps	Source IP address	-Pd	Destination IP address
-Pm	Source MAC address	-Pa	Destination MAC address
-Po	Either "REQUEST" or "REPLY"		

Application information–CBR (-P cbr -P?):

-Pi	Sequence number
-Pf	The number of time this packet has been forwarded
-Po	Min. number of hops to reach the destination as indicated by GOD. The default value when GOD is not in use is -1.

Application information–TCP (`-P tcp -P?`):

`-Ps`	Sequence number	`-Pa`	Acknowledgement number
`-Pf`	Same as that of CBR	`-Po`	Same as that of CBR

where the character "?" represents a letter. Again, the complete list of tracing tags and subtags can be found in the files *~ns*/trace/cmu-trace.h,cc, in [17], and [11].

Example 14.6. Consider the following new wireless trace record:

```
s -t 31.000000000 -Hs 0 -Hd -2
  -Ni 0 -Nx 19.36 -Ny 17.32 -Nz 0.00 -Ne -1.000000
  -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0
  -Is 0.0 -Id 1.0 -It tcp -Il 40 -If 2 -Ii 3 -Iv 32
  -Pn tcp -Ps 0 -Pa 0 -Pf 0 -Po 0
```

The interpretation is as follows. At 31 s, this record traces Node `0` at the agent `AGT` level. The Node is located at the coordinate `(19.36,17.32,0)`. The reasons string (i.e., `-Nw`) is empty. The record does not trace the node energy level (`-Ne -1`). The packet is sent from the address 0 port 0 to the node with the address 1 port 0. The packet type is "`tcp`." Its size is 40 bytes. The flow ID is `2`. The packet unique ID is `3`. The time to live is `32`. This packet has never been forwarded before. The optimal number forwards is zero (i.e., GOD is not active). □

14.4 Compilation of Simulation Results

Compilation of simulation results refers to a process of collecting information from simulation and compute performance measures under consideration. There are three main approaches to collect simulation data for result compilation: through C++ through, variable tracing, and through packet tracing.

- *Through C++ program*: This refers to an approach which inserts C++ statements into the original NS2 program. As mentioned earlier in this book, the modification of C++ program results in a quick simulation. This approach also provides great flexibility in which most information would be accessible. However, programmers require a fair amount of knowledge about the C++ architecture to collect results from the simulation.
- *Through variable tracing*: This method is perhaps the most convenient way to collect the results. The programmers do not need to know the details of the C++ architecture. They only need to know the traceability and/or binding structure of OTcl instvars under consideration. However, the range of collectable information is limited to traceable variables only.
- *Through packet tracing*: Packet tracing is easy to set up, and it provides a great deal of detailed packet movements. The downside of packet tracing is that it significantly drains computational power (e.g., memory, CPU time), and

dramatically slows down simulation. The great amount of collected information can also overwhelm researchers who need to compute performance measure such as throughput. Ironically, this great amount of information does not necessarily contain the required information such as the number of error correction bits in packet headers. In most cases, packet tracing proceeds in two steps. The first step is to write all information at the network checkpoints into a trace file using an OTcl statement "`$ns trace-all $fch`." The second step is to filter out unnecessary information and compute performance measures of interest. In this step, a scripting language such as AWK can be used (see Appendix A).

Example 14.7. Consider Example 10.1 which creates the network in Fig. 9.3. Insert an error model with error probability 0.5% into the link connecting Node 1 and Node 3. Suppose the maximum TCP transmission window size is set to 20. Run simulation for 10 seconds, and perform the following task.

- *Through C++ codes*: Find out the number of times TCP transmission window is reduced.
- *Through variable tracing*: Plot the dynamic variation of TCP transmission window.
- *Through packet tracing*: Compute the average interval between two TCP packets entering the link layer buffer.

Constructing a Network

An error model can be inserted into the network by inserting the following OTcl codes immediately after Line 7 of Example 10.1:

```
set em [new ErrorModel]
$em set rate_ 0.005
$em unit pkt
$em ranvar [new RandomVariable/Uniform]
$em drop-target [new Agent/Null]
$ns lossmodel $em $n1 $n3
```

The maximum TCP transmission window is set to 20 by the following statement after Line 10 in Example 10.1: "`$tcp set window_ 20`."

Result Compilation Through C++ Codes

TCP shrinks its transmission window when the function `slowdown(how)` of class `TcpAgent` is invoked. Therefore, we may declare a variable "`num_slowdowns_`" of class `TcpAgent` in file *~ns*/tcp/tcp.h, initialize it to zero in the constructor, and add the two following lines in the function `slowdown(how)`:

```
num_slowdowns_++;
printf("Total number of TCP window shrinking events
          is %d \n", num_slowdowns_);
```

After recompiling NS2, we run the script "`tcp.tcl`" and obtain the following results:

```
>> ns tcp.tcl
Total number of TCP window shrinking events is 1
Total number of TCP window shrinking events is 2
Total number of TCP window shrinking events is 3
...
Total number of TCP window shrinking events is 36
```

In this simulation, TCP shrinks its transmission window 36 times.

Result Compilation Through Variable Tracing

Variable tracing refers to the methods for tracking changes in OTcl instvars. Section 14.2 discussed a built-in mechanism to do so. Alternatively, programmers may use manual variable tracing shown below.

Transmission window size of a TCP connection is the minimum of instvars "`cwnd_`" and "`window_`" of a `Agent/TCP` object. Since these two variables are available in the OTcl domain, we may collect samples of TCP window size by inserting the following Tcl script after Line 14 in Example 10.1.

```
1  set f_cwnd [open cwnd.tr w]
2  proc plot_tcp { } {
3      global f_cwnd tcp ns
4      if { [$tcp set cwnd_] < [$tcp set window_] } {
5              puts $f_cwnd "[$ns now] [$tcp set cwnd_]"
6      } else {
7              puts $f_cwnd "[$ns now] [$tcp set window_]"
8      }
9      $ns at [expr [$ns now] + 0.2] plot_tcp
10 }
11 $ns at 0.01 "plot_tcp"
```

The above statements put time and TCP transmission window size in the file "`cwnd.tr`" every 0.2 s. Line 1 above creates a Tcl channel "`f_cwnd`" which is attached to the file "`cwnd.tr`." Lines 2–10 define a procedure `plot_tcp{}`. Lines 11 invokes the procedure `plot_tcp{}` at 0.01 s. Within the procedure `plot_tcp{}`, Lines 5 and 7 print instvars "`cwnd_`" and "`window_`," whichever is less, on the Tcl channel "`f_cwnd`." Line 9 schedules an invocation of the procedure `plot_tcp{}` at 0.2 s in future. This invocation continuously prints out simulation time and TCP transmission window size to the Tcl channel until the simulation terminates.

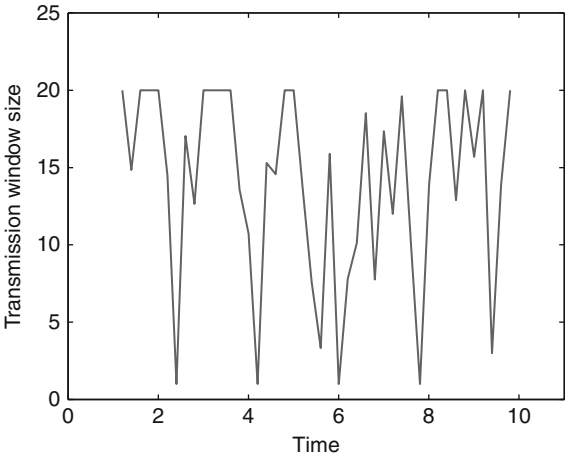

Fig. 14.9 Dynamics of TCP transmission window for Example 14.7

After running the above Tcl simulation script, the file "`cwnd.tr`" is created. The first and the second columns of the file "`cwnd.tr`" are the time and the corresponding TCP transmission window, respectively. Figure 14.9 plots simulation time (1st column) against transmission window size (2nd column). Since we set the instvar "`window_`" to be 20, TCP transmission window can never exceed 20. We can also observe frequent decreases in TCP transmission window size due to packet losses.

Result Compilation Through Packet Tracing

The first step in this approach is to enable tracing in the Tcl simulation script. Again, this step can be carried out by inserting the following statements after Line 4 in Example 10.1.

```
set f_trace [open trace.tr w]
$ns trace-all $f_trace
```

The second step is to process the trace file. In this case, there is only one TCP flow in the simulation and we can measure the interval between two TCP packets entering a queue, which connect Node 1 (with ID 0) to Node 3 (with ID 1), using the AWK script file "`avg.awk`" in Program 14.18. By executing the AWK script, we will see the following result on the screen:

```
>> awk -f avg.awk trace.tr
Average TCP packet inter-arrival time is 0.001703
```

Line 1 in Program 14.18 initializes variable "`started`" to zero. Lines 2–14 collect samples of the inter-arrival time of TCP packets. Line 2 filter out the trace records which do not begin with "+." From Line 5, the samples are collected only

Program 14.18 An AWK script which computes the average interval between two TCP packets entering a link layer buffer of Node 1

```
 //avg.awk
1 BEGIN{ started = 0 }
2 /^+/ {
3     time = $2;
4     if (started == 1) {
5         if ($3==0 && $4==2 && $5 == "tcp") {
6             interval = time-old_time;
7             old_time = time;
8             cum_interval += interval;
9             total_samples ++;
10         }
11     } else {
12         started = 1; old_time = time;
13     }
14 }
15 END {
16     avg_interval = cum_interval/total_samples;
16     printf("Average TCP packet inter-arrival time
               is %f\n", avg_interval);
17 }
```

for the source node `0`, the destination node `2`, and protocol "`tcp`" (see Fig. 14.7). Finally, Lines 15–17 compute and print the average TCP packet inter-arrival time on the screen. □

14.5 Chapter Summary

This chapter focuses on debugging, tracing, and compilation of simulation results. Debugging refers to a process of removing compilation and run-time errors in both C++ and OTcl domains. This chapter provides guidelines and necessary commands for debugging. Although originally designed to facilitate the understanding of network dynamics, NS2 tracing could also be useful in the debugging process. NS2 supports two types of tracing. Variable tracing records the changes in value of a variable (in most cases in a file), while packet tracing stores the details of packets passing through network checkpoints (again in most cases in a file).

There are three major result compilation approaches. The first approach is through C++ program. It is quick at runtime and gives programmers an access to most of the NS2 components. On the other hand, the programmers may require a fair amount of knowledge on the C++ architecture. Also, since this method involves the modification of C++ code, it could mess up the original NS2 source codes.

The second approach is through variable tracing. This approach allows programmers to collect the results from the OTcl domain in a simple way. Using this

approach, programmers do not need to understand the entire architecture of NS2, but the range of collectable information is fairly limited.

The last approach is through packet tracing, which consists of two steps: (1) recording trace strings in a trace file and (2) processing the trace file. Although simple and informative, this method drains computation resource and can overwhelm programmers with the deluge of information. Programmers would need to learn a scripting language such as AWK to extract required information from the trace file. The above three approaches for compilation of results have their own strengths and weaknesses. Programmers need to choose the right one based on their main simulation objectives.

14.6 Exercises

1. Explain the difference between variable tracing and packet tracing.
2. What are the possible causes of segmentation fault error?
3. What are the differences between compilation error and runtime error? In which domain (C++ or OTcl) do these errors occur?
4. Write down the statements which retrieve the current simulation time from both C++ and OTcl domains?
5. Write down a C++ statement which converts a C++ object "`c_obj`" to an OTcl referencing string and a C++ statement which converts a OTcl referencing string "`str`" to a C++ object.
6. What are attributes of a traceable variable?
7. Explain the following components and their relationship for variable tracing: TclObjects, traceable variables, tracers, Tcl channels, and trace files. Draw a diagram and give examples to support your answer.
8. Set up the packet tracing network diagram in Fig. 2.6. What are the OTcl statements which set up packet tracing? What type of packet tracing format appear in the trace file?
9. From Exercise 8, write AWK scripts to do the following:
   a. Compute the throughput in bps of the link connecting nodes `N0` and `N2`.
   b. Compute the throughput in bps corresponding to `TCP0`.
   c. Compute average packet delivery delay for the link connecting nodes `N2` and `N3`.
10. In Fig. 2.6, trace changes in packet sequence number of `TCP0` over time. Plot your results in a graph. Explain in time sequence how NS2 sets up the trace and how it traces the variable in realtime.

11. Set up a simulation for a wireless network with ten Nodes which runs AODV as the routing protocol.
    - Using wireless (i.e., CMU) tracing, compute end-to-end throughput averaged over all the nodes.
    - Using new wireless tracing, plot the energy level of node `1`.

Chapter 15
Related Helper Classes

Helper classes generally not a part of a network, but are used in NS2 as an internal mechanism and/or are network components in a special case. This chapter discusses the details of three main NS2 helper classes. In Sect. 15.1, we first discuss class `Timer`, which is widely used by other NS2 modules such as TCP to implement conditional time-based actions. In Sect. 15.2, we demonstrate a random number generation process in NS2. In Sect. 15.3, we explain the details of class `ErrorModel`, which can be used to simulate packet error. Section 15.4 discusses bit masking and bit shifting operations used in various places in NS2. Finally, the chapter summary is given in Sect. 15.5.

15.1 Timers

Timer is a component that can be used to delay actions. Unless cancelled or restarted, a timer takes actions after it has been started for a given period of time (i.e., at the expiration). For example, a sender starts a retransmission timer as soon as it transmits a packet. Unless cancelled by a reception of an acknowledgment packet, the timer assumes packet loss and asks the sender to retransmit the lost packet at the timer expiration.

15.1.1 Implementation Concept of Timer in NS2

As shown in Fig. 15.1, a timer consists of four following states: `IDLE`, `SET WAITING TIME`, `WAITING`, and `EXPIRED`. A transition from one state to another occurs immediately when the operation in the current state is complete (i.e., by default), or when the timer receives a start message, a restart message, or a cancel message.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_15, © Springer Science+Business Media, LLC 2012

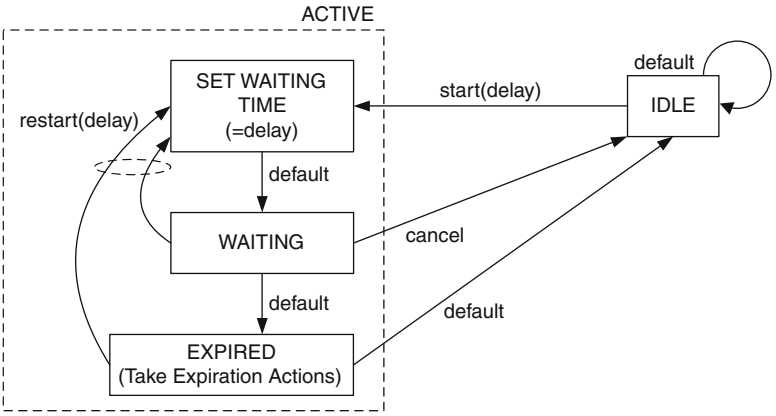

Fig. 15.1 Timer life cycle

When a timer is created, it sets the state to be `IDLE`. Upon receiving a start message, the timer moves to the state `SET WAITING TIME`, where it sets its waiting time to be "`delay`" seconds and moves to the state `WAITING`. The timer stays in the state `WAITING` for "`delay`" seconds and moves to the state `EXPIRED`. At this point, the timer takes predefined expiration actions and moves back to the state `IDLE`. Hereafter, we will say that the timer *expires* as soon as it enters the state `EXPIRED`. Also, we shall refer to the actions taken in state `EXPIRED` as *expiration actions*.

The above timer life cycle occurs by default when the message "start" is received. When a "cancel" messages is received, the timer will stop waiting and move back to the state `IDLE`. If a restart message is received, the timer will restart the waiting process in the state `SET WAITING TIME`.

Implementation of timer in NS2 is a very good example of the *inheritance* concept in OOP. Each timer needs to implement the three following mechanisms: (1) waiting mechanism, (2) interface functions to start, restart, and cancel the waiting process, and (3) expiration actions. The first two mechanisms are common to all timers; however, the last mechanism (i.e., expiration actions) is what differentiates one timer from another. From an OOP point of view, the timer base class must define the waiting mechanism and message receiving interfaces, and leave the implementation of the expiration actions to the derived classes.

In NS2, timers are implemented in both C++ and OTcl. However, both C++ and OTcl timer classes are standalone (i.e., not bound together by `TclClass`). Relevant functions and variables in both domains are shown in Table 15.1. In both domains, NS2 implements the waiting process by utilizing the Scheduler. Upon entering the state `SET WAITING TIME`, NS2 places a timer expiration event on the simulation timeline. When the Scheduler fires the expiration event, the timer enters the state `EXPIRED` and executes the expiration actions.

Table 15.1 Timer implementation in C++ and OTcl domains

Components of a timer	C++ components	OTcl components
State IDLE	status_=TIMER_IDLE	"id_" unset
State SET WAITING TIME	status_=TIMER_PENDING	"id_" set
State WAITING	status_=TIMER_PENDING	"id_" set
State EXPIRATION	status_=TIMER_HANDLING	"id_" set
Message start	Function sched	Instprocs sched and resched
Message restart	Function resched	Instprocs sched and resched
Message cancel	Function cancel	Instprocs cancel and destroy
Action at the expiration	Function expire	Instproc timeout

15.1.2 OTcl Implementation

In the OTcl domain, NS2 implements timers using an OTcl class Timer. The implementation of class timer consists of three parts. First, the waiting mechanism is implemented by placing a timer expiration event on the simulation timeline using the instproc at{...} of class Simulator (See Lines 9 and 15 in Program 15.1). Second, the interface of class Timer is defined in the instprocs sched{delay}, resched{delay}, cancel{}, and destroy{}. Finally, the expiration actions are specified in the instproc timeout{}, which is implemented in child classes of class Timer (see class ConnTimer in file *~ns*/tcl/webcache/webtraf.tcl, for example).

Program 15.1 shows details of various instprocs of OTcl class Timer. Class Timer has two key instvars: "ns_" in Line 6 and "id_" in Line 7. The instvar "ns_" is a reference to the Simulator. It is configured at the construction of a Timer object (see Lines 2–4). The constructor of class Timer takes the Simulator as its input argument and stores the input instance in its instvar "ns_." The instvar "id_" (Line 7) indicates the state of the timer. If the timer is idle, "id_" will not exist (since it is unset). If the timer is active, "id_" will contain the unique ID of the timer expiration event on the simulation timeline.

The instprocs sched{delay} (Lines 5–10) and resched{delay} (Lines 11–13) are NS2 implementation for receiving a start message and a restart message, respectively. They take one input argument "delay" and set the timer to expire after "delay" seconds. Regardless of the timer state, the instproc sched{delay} cancels the timer using the instproc cancel{} in Line 8. In Line 9, it tells the timer to expire at "delay" seconds in future by invoking the instproc after{ival args} of class Simulator. Shown in Lines 14–16, the instproc after{...} uses an OTcl command at of class Simulator to place another OTcl command in future.[1] From Line 9, the instproc sched{delay} schedules an invocation of

[1]As discussed in Sect. 4.2.3, the OTcl command "at{...}" places an AtEvent object on the simulation timeline, and returns the unique ID of the scheduled event to the caller.

Program 15.1 Timer related OTcl instprocs

```
    //~/ns/tcl/mcast/timer.tcl
 1  Class Timer
 2  Timer instproc init { ns } {
 3      $self set ns_ $ns
 4  }
 5  Timer instproc sched delay {
 6      $self instvar ns_
 7      $self instvar id_
 8      $self cancel
 9      set id_ [$ns_ after $delay "$self timeout"]
10  }
11  Timer instproc resched delay {
12      $self sched $delay
13  }

    //~/ns/tcl/lib/ns-lib.tcl
14  Simulator instproc after {ival args} {
15          eval $self at [expr [$self now] + $ival] $args
16  }

    //~/ns/tcl/mcast/timer.tcl
17  Timer instproc cancel {} {
18      $self instvar ns_
19      $self instvar id_
20      if [info exists id_] {
21          $ns_ cancel $id_
22          unset id_
23      }
24  }
25  Timer instproc destroy {} {
26      $self cancel
27  }
```

the instproc "`timeout{}`" at "`delay`" seconds in future and stores the unique ID corresponding to the timer expiration event in the instvar "`id_`."

Lines 17–27 in Program 15.1 show details of the instprocs `cancel{}` and `destroy{}` of class `Timer`. Both the instprocs act as an interface to receive a cancel message. Note that "`id_`" exists only when a timer expiration event is on the simulation timeline. Timer is cancelled only when "`id_`" exists (i.e., the condition in Line 20 is `true`). In this case, Line 21 feeds "`id_`" to the instproc `cancel{id_}` (see Program 15.2) of the Simulator instance to remove the timer expiration event from the timeline. Finally, Line 22 unsets the instvar "`id_`" to indicate that the expiration event is no longer on the simulation timeline.

Program 15.2 shows details of the instproc `cancel{...}` of class `Simulator` and the OTcl command `cancel{uid}` of class `Scheduler`. The instproc `cancel{...}` takes one input argument "`uid`," which is the unique ID of an

Program 15.2 Instproc `cancel` of class `Simulator` and an OTcl command `cancel` of class `Scheduler`

```
     //~/ns/tcl/lib/ns-lib.tcl
  1  Simulator instproc cancel args {
  2      $self instvar scheduler_
  3      return [eval $scheduler_ cancel $args]
  4  }

     //~/ns/common/scheduler.cc
  5  int Scheduler::command(int argc, const char*const* argv)
  6  {
  7      ...
  8      if (strcmp(argv[1], "cancel") == 0) {
  9          Event* p = lookup(STRTOUID(argv[2]));
 10          if (p != 0) {
 11              cancel(p);
 12              AtEvent* ae = (AtEvent*)p;
 13              delete ae;
 14          }
 15      }
 16      ...
 17 }
```

event to be cancelled. Line 3 invokes the OTcl command `cancel{uid}` of the Scheduler (stored in an instvar "`scheduler_`" of the Simulator), which removes the timer expiration event whose unique ID is "`uid`" (see Lines 9–13).

15.1.3 C++ Class Implementation

This section explains the C++ implementation of a timer. We first show the life cycle of a C++ timer based on C++ functions (in Table 15.1). Second, we briefly discuss the declaration of C++ abstract class `TimerHandler`, which represents timers in the C++ domain. Third, we describe the details of three main components of a timer: (1) internal waiting mechanism, (2) interface functions, and (3) expiration actions. Fourth, we demonstrate how a timer is cross-referenced with another object. Finally, we conclude this section by providing guidelines for implementing timers in NS2.

15.1.3.1 Timer Life Cycle

Based on Fig. 15.1 and Table 15.1, we redraw the life cycle of a `TimerHandler` object (i.e., a C++ timer object) in Fig. 15.2. The default state of a timer is `TIMER_IDLE`. Upon invoking functions `sched(delay)` or `resched(delay)`,

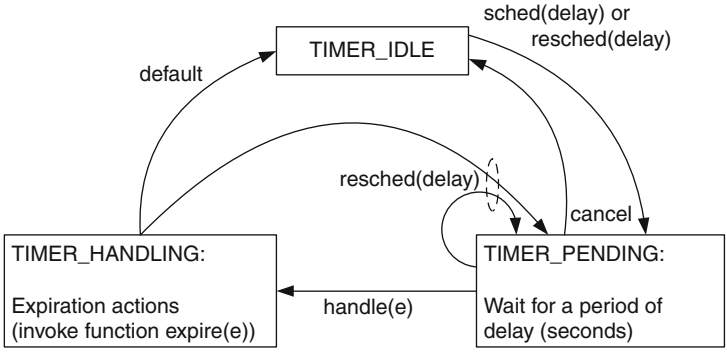

Fig. 15.2 Life cycle of a `TimerHandler` object

the timer moves from the state `TIMER_IDLE` to another state `TIMER_PENDING`, where the timer starts a waiting period of "`delay`" seconds. When the timer expires, it moves to the state `TIMER_HANDLING` and takes expiration actions by invoking the function `expire(e)`. After taking expiration actions, the timer moves to the state `TIMER_IDLE`, and the cycle starts over again. Regardless of the state, function `resched(delay)` cancels the pending timer and restarts the timer. In the state `TIMER_PENDING`, we may cancel the timer by invoking function `cancel()`, which stops the active timer and changes the state of the timer to `TIMER_IDLE`.

15.1.3.2 Brief Overview of Class `TimerHandler`

Program 15.3 shows the declaration of a C++ abstract class `TimerHandler`, which represents timers. Line 7 defines three states of a `TimerHandler` object as members of `TimerStatus enum` data type: `TIMER_IDLE`, `TIMER_PENDING`, and `TIMER_HANDLING`. Class `TimerHandler` contains only two member variables: "`status_`" in Line 12 and "`event_`" in Line 13. The variable "`status_`" stores the current timer state (or status). It takes a value in $\{0, 1, 2\}$, which corresponds to the values of the `TimerStatus enum` type shown in Line 7. The default state of a timer is `TIMER_IDLE`. Therefore, variable "`status_`" is set to `TIMER_IDLE` at the timer construction (see Line 3). Another variable "`event_`" (of class `Event`) represents a timer expiration event. It acts as a glue between a `TimerHandler` object and the Scheduler. The details of variable "`event_`" will be discussed in the next section.

The key functions of class `TimerHandler` along with their descriptions are given below.

Program 15.3 Declaration of class `TimerHandler`

```
    //~/ns/common/timer-handler.h
 1  class TimerHandler : public Handler {
 2  public:
 3     TimerHandler() : status_(TIMER_IDLE) { }
 4     void sched(double delay);    // cannot be pending
 5     void resched(double delay); // may or may not be pending
 6     void cancel();               // must be pending
 7     enum TimerStatus { TIMER_IDLE, TIMER_PENDING,
         TIMER_HANDLING};
 8     int status() { return status_; };
 9  protected:
10     virtual void expire(Event *) = 0;
11     virtual void handle(Event *);
12     int status_;
13     Event event_;
14  private:
15     inline void _sched(double delay) {
16        (void)Scheduler::instance().schedule(this, &event_,
17        delay); }
18     inline void _cancel() {
19         (void)Scheduler::instance().cancel(&event_);
20     }
21  };
```

`sched(delay)`	Start the timer and set the timer to expire at "`delay`" seconds in future.
`_sched(delay)`	Place a timer expiration event on the simulation time line at "`delay`" seconds in future.
`resched(delay)`	Restart the timer and set the timer to expire at "`delay`" seconds in future.
`cancel()`	Cancel the pending timer.
`_cancel()`	Remove a timer expiration event from the simulation time line.
`status()`	Return the variable "`status_`," the current state of the timer.
`handle(e)`	Invokes the function `expire(e)`. It is used by the Scheduler to dispatch a timer expiration event (see Chap. 4).
`expire(e)`	Take expiration actions. It is a pure virtual function, and must be implemented by child instantiable classes of class `TimerHandler`.

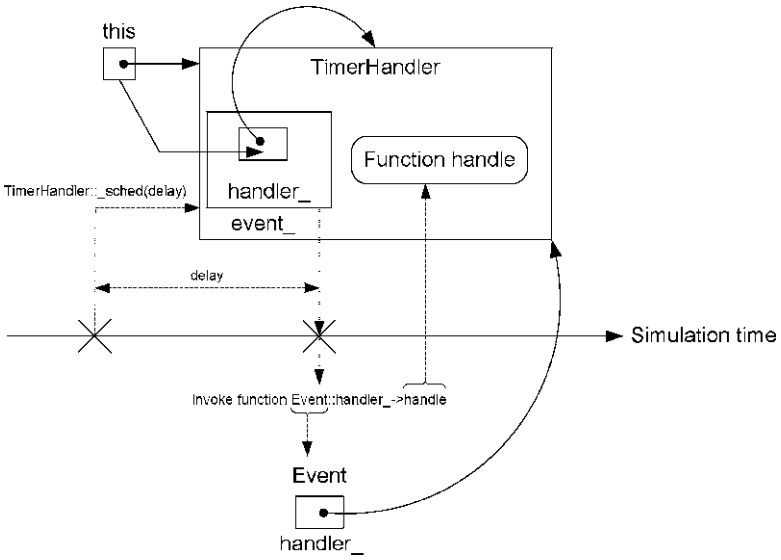

Fig. 15.3 A diagram which represents the timer waiting process (i.e., function `_sched(delay)`)

15.1.3.3 Internal Waiting Mechanism

Class `TimerHandler` implements waiting mechanism through functions `_sched(delay)` and `_cancel(delay)`. Basically, these two functions place and remove "`event_`" on the simulation timeline. In Line 16 of Program 15.3, the function `_sched(delay)` executes "`schedule(this,&event_,delay)`," where "`this`" is the timer address, "`event_`" is an expiration dummy event (see Sect. 4.3.7), and "`delay`" is the duration until the timer expires. The function `schedule(...)` stores the address of the timer "`this`" in the variable "`handler_`" of the `Event` pointer "`event_`," essentially setting `event_->handler_` to point to the `TimerHandler` object. Then, it places the object "`* event_`" on the simulation timeline at "`delay`" seconds in future. At the firing time, the Scheduler invokes the function `dispatch(e)`, which in turn executes `event_->handler_->handle(...)`. Since the variable "`handle_`" of the dispatched "`event_`" points to the `TimerHandler` object (see Fig. 15.3), NS2 invokes the function `handle(e)` associated with the `TimerHandler` object at the firing time. The function `handle(e)` of class `TimerHandler` in turn invokes the function `expire(e)` (Line 6 of Program 15.4) which takes expiration actions specified by the derived classes of class `TimerHandler`.

Function `_cancel()` does the opposite of what function `_sched(delay)` does. It removes the timer expiration event from the simulation timeline. From Line 19 in Program 15.3, it invokes function `cancel(&event_)` of class `Scheduler` to remove the event "`event`" from the simulation timeline.

15.1.3.4 Expiration Actions

At the firing time, the Scheduler dispatches a timer expiration event by invoking function `handle(e)` of the associated timer (see also Fig. 15.3). Details of the function `handle(e)` are shown in Program 15.4. Line 3 first checks whether the current "`status_`" is `TIMER_PENDING`. If so, Line 5 will change the variable "`status_`" to `TIMER_HANDLING`, and Line 6 will invoke function `expire(e)` to take expiration actions. After returning from the function `expire(e)`, the variable "`status_`" is set by default to `TIME_IDLE` (Line 8). However, if "`status_`" has already changed (e.g., when the timer is rescheduled; "`status_`"≠`TIMER_HANDLING` in Line 7), function `handle(e)` will not change variable "`status_`."

Program 15.4 Function `handle` of class `TimerHandler`

```
   //~/ns/common/timer-handler.cc
1  void TimerHandler::handle(Event *e)
2  {
3      if (status_ != TIMER_PENDING)
4          abort();
5      status_ = TIMER_HANDLING;
6      expire(e);
7      if (status_ == TIMER_HANDLING)
8          status_ = TIMER_IDLE;
9  }
```

In Line 10 of Program 15.3, the function `expire(e)` is pure virtual. Therefore, derived instantiable classes of class `TimerHandler` are responsible for providing expiration actions by overriding this function. For example, class `MyTimer` below derives from class `TimerHandler` and overrides function `expire(e)`:

```
void MyTimer::expire(Event *e)
{
    printf("MyTimer has just expired!!\n");
}
```

which prints the statement "`MyTimer has just expired!!`" on the screen upon timer expiration.

15.1.3.5 Interface Functions to Start, Restart, and Cancel a Timer

The details of function `sched(delay)` of class `TimerHandler` is shown in Program 15.5. Function `sched(delay)` takes one input argument "`delay`," and sets the timer to expire at "`delay`" seconds in the future by feeding "`delay`" into

Program 15.5 Function `sched` of class `TimerHandler`

```
    //~/ns/common/timer-handler.cc
1   void TimerHandler::sched(double delay)
2   {
3       if (status_ != TIMER_IDLE) {
4           fprintf(stderr,"Couldn't schedule timer");
5           abort();
6       }
7       _sched(delay);
8       status_ = TIMER_PENDING;
9   }
```

function `_sched(delay)` (Line 7). Note that the function `sched(delay)` must be invoked when the "`status_`" of the timer is `TIMER_IDLE`. Otherwise, Lines 4 and 5 will show an error message and exit the program.

Program 15.6 shows the details of functions `resched(delay)` and `cancel()` of class `TimerHandler`. Function `resched(delay)` is very similar to function `sched(delay)`. In fact, when invoked with "`status_`" $\neq$ `TIMER_PENDING`, it does the same as function `sched(delay)` does (i.e., starts the timer). However, when `status_=TIMER_PENDING` (Line 3) – meaning "`event_`" was placed on the simulation timeline before the invocation – it removes the timer expiration event from the simulation time line, by invoking function `_cancel()`, and (re)starts the timer (Lines 4 and 5, respectively).

Program 15.6 Functions `resched` and `cancel` of class `TimerHandler`

```
    //~/ns/common/timer-handler.cc
1   void TimerHandler::resched(double delay)
2   {
3       if (status_ == TIMER_PENDING)
4           _cancel();
5       _sched(delay);
6       status_ = TIMER_PENDING;
7   }

8   void TimerHandler::cancel()
9   {
10      if (status_ != TIMER_PENDING) {
11          ...
12          abort();
13      }
14      _cancel();
15      status_ = TIMER_IDLE;
16  }
```

Lines 8–16 of Program 15.6 show the details of function `cancel()` of class `TimerHandler`. Function `cancel()` invokes function `_cancel()` in Line 14 to remove the pending timer expiration event from the simulation timeline. Function `cancel()` must not be invoked, when "`event_`" is not on the simulation timeline (i.e., "`status_`" is either `TIMER_IDLE` or `TIMER_HANDLING`). Otherwise, NS2 will show an error message on the screen and exit the program (Lines 11 and 12).

15.1.3.6 Cross-Referencing a Timer with Another Object

In most cases, the usefulness of a timer stands out when it is cross-referenced with another object. In this case, the object uses a timer as a waiting tool, which starts, restarts, and cancels the waiting process as necessary. The timer, on the other hand, informs the object of timer expiration, upon which the object may take expiration actions.

A typical cross-reference between a timer and an object can be created as follows:

1. Declare the timer as a variable of the object class.
2. Declare a pointer to the object as a member of the timer class.
3. Define a non-default constructor for the timer class. Store the input argument of the constructor in its member pointer variable (which points to the associated object).
4. Instantiate a timer object from within the constructor of the associated object. Use the non-default constructor of the timer class defined above. Feed the pointer "`this`" (i.e., the pointer to the object) as an input argument to the constructor of the timer.

We now conclude this section with a simple timer example.

Example 15.1. Consider a process of counting the number of customers who enter a store during a day. Let class `Store` represent a convenience store (i.e., an object class), and let class `StoreHour` represent the number of opening hours of a day (i.e., a timer class). The opening hours is specified when the store is opened. The objective here is to count the number of visiting customers during a day, and print out the result when the store is closed.

Classes `Store` and `StoreHour`

From Program 15.7, class `Store` also has three variables. First, "`hours_`" (Line 17) contains opening hours of the store and is set to zero at the construction. Second, "`count_`" (Line 18) records the number of customers who have entered the store so far and is set to zero at the construction. Finally, variable "`timer_`" is a `StoreHour` object. Function `close()` (Lines 12 and 13) of class `Store` is invoked when the store is being closed. It prints out the opening hours and

Program 15.7 Declaration of classes `Store` and `StoreHour`

```
    //store.h
 1  class Store;
 2  class StoreHour : public TimerHandler {
 3  public:
 4      StoreHour(Store *s) { store_ = s; };
 5      virtual void expire( Event *e );
 6  protected:
 7      Store *store_;
 8  };

 9  class Store : public TclObject {
10  public:
11      Store() : timer_(this) { hours_ = -1; count_ = 0; };
12      void close(){
13          printf("The number of customers during
                    %2.2f hours is %d\n", hours_,count_);
14      };
15      int command(int argc, const char*const* argv);
16  protected:
17      double hours_;
18      int count_;
19      StoreHour timer_;
20  }
```

number of visiting customers for today on the screen. Declared in Lines 1–8, class `StoreHour` has only one variable "`store_`" (Line 7) which is a pointer to a `Store` object.

Cross-Referencing `Store` and `StoreHour` Objects

The process of cross-referencing a `Store` object and a `StoreHour` object is shown in Fig. 15.4. The constructor of class `Store` constructs its variable "`timer_`" with the pointer "`this`" to the `Store` object (see Line 11). The constructor of class `StoreHour` stores the input pointer in its variable "`store_`." Since the input argument is the pointer to the `Store` object, the constructor of the `StoreHour` object essentially sets the variable "`store_`" to point back to the `Store` object.

Due to the cross-referencing, the compiler needs to recognize one of these two classes when declaring another. Line 1 helps the compiler recognize class `Store` when compiling Line 7. After compiling Line 2, the compiler recognizes class `StoreHour` and can compile Line 19 without error.

It is also important to note that when compiling Lines 2–8, the compiler recognizes only `Store` class name. Any attempt to invoke functions (e.g., `close()`) of class `Store` will result in a compilation error. This is the reason why we need

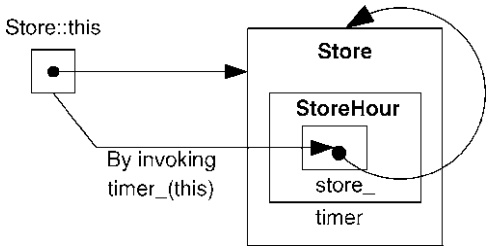

Fig. 15.4 A diagram which represents the process of cross-referencing a `Store` object and a `StoreHour` object

Program 15.8 Function `expire` of class `StoreHour` as well as OTcl Commands `open` and `new-customer` of class `Store`

```
    //store.cc
1   void StoreHour::expire(Event*) {
2       store_->close();
3   };

4   int Store::command(int argc, const char*const* argv)
5   {
6       if (argc == 3) {
7           if (strcmp(argv[1], "open") == 0) {
8               hours_ = atoi(argv[2]);
9               count_ = 0;
10              timer_.sched(hours_);
11              return (TCL_OK);
12          }
13      } else  if (argc == 2) {
14          if (strcmp(argv[1], "new-customer") == 0) {
15              count_++;
16              return (TCL_OK);
17          }
18      }
19      return TclObject::command(argc,argv);
20  }
```

to separate C++ programs into header and C++ files. Again, since a header file is included at the top of a C++ file, the compiler first goes through the header file and recognizes all the variables and functions specified in the header file. With this knowledge, the compiler can compile the C++ file without error.

Defining Expiration Actions

Derived from class `TimerHandler`, class `StoreHour` overrides function `expire(e)` as shown in Lines 1–3 of Program 15.8. At the expiration, the timer (i.e., `StoreHour` object) simply invokes function `close()` of the associated `Store` object.

Creating OTcl Interface

We bind the C++ class `Store` to an OTcl class with the same name using a mapping class `StoreClass` shown in Program 15.9. Lines 4–20 in Program 15.8 also show OTcl interface commands `open{hours}` and `new-customer{}`. With opening hours "`hours`" as an input argument, the OTcl command `open{hours}` (Lines 8–11) is invoked when the store is opened. Line 8 stores the opening hours in variable "`hours_`," Line 9 resets the number of visiting customers to zero, and Line 10 tells "`timer_`" to expire at "`hours_`" hours in future. The OTcl command `new-customer{}` is invoked as a customer enters the store. In Line 15, this command simply increases "`count_`" by one. Again, at the timer expiration, the timer invokes function `close()` through the pointer "`store_`" and prints out the opening hours (i.e., "`hours_`") as well as the number of visiting customers (i.e., "`count_`") for today (see function `expire(e)` in Line 2 of Program 15.8).

Program 15.9 A mapping class `StoreClass` which binds C++ and OTcl classes `Store`

```
     //store.cc
  1 static class StoreClass : public TclClass {
  2  public:
  3     StoreClass() : TclClass("Store") {}
  4     TclObject* create(int, const char*const*) {
  5         return (new Store);
  6     }
  7 } class_store;
```

Testing the Codes

After defining files `store.cc` and `store.h`, we include `store.o` to the `MakeFile` and run "`make`" at NS2 root directory to include classes `Store` and `StoreHour` into NS2 (see Sect. 2.7).

Define a test Tcl simulation script in a file `store.tcl`.

```
 //store.tcl
1 set ns [new Simulator]
2 set my_store [new Store]
3 $my_store open 10.0
4 $ns at 1 "$my_store new-customer"
5 $ns at 5 "$my_store new-customer"
6 $ns at 6 "$my_store new-customer"
7 $ns at 8 "$my_store new-customer"
8 $ns at 11 "$my_store new-customer"
9 $ns run
```

We run the script `store.tcl` and obtain the following results:

```
>>ns store.tcl
The number of customers during 10.0 hours is 4
```

From the above script, when Line 2 creates a `Store` object, NS2 automatically creates a shadow C++ `Store` Object. Line 3 invokes an OTcl command `open` with input argument 10.0, essentially opening the store for 10.0 h. From Program 15.8, the OTcl command `open{10.0}` and tells the associated timer to expire at 10.0 h in future, and clears the variable "`count_`." Lines 4–8 invoke command `new-customer{}` at 1st, 5th, 6th, 8th, and 11th hours. Each of these lines increases the number of visiting customers (i.e., "`count_`") by one. By the end of 11th hour in future, variable "`count_`" should be 5. However, the program shows that the number of visiting customers is 4. This is because the timer expires and invokes the function `close()` at the 10th hour. □

15.1.4 Guidelines for Implementing Timers in NS2

We now summarize the process of defining a new timer. Suppose that we would like to define a new timer class `StoreHour`. Suppose further that a `Store` object is responsible for starting, restarting, and canceling the `StoreHour` object, and for taking expiration actions. Then, the implementation of the above timer classes proceeds as follows:

From Class `StoreHour`

- *Step 1:* Design class structure:
  - Derive class `StoreHour` from class `TimerHandler`.
  - Declare a pointer (e.g., "`store_`") to class `Store`. The public function of class `Store` is accessible through the above pointer (e.g., "`store_`")
- *Step 2:* Bind the reference to class `Store` in the constructor.
- *Step 3:* Define expiration actions in the function `expire(e)`.

From Class `Store`

- *Step 1:* Design class structure:
  - Derive class `Store` from class `TclObject` *only if* an interface to OTcl is necessary.
  - Declare a `StoreHour` variable (e.g., "`timer_`") as a member variable.

- *Step 2:* From within the constructor instantiate the above `StoreHour` variable (e.g., "`timer_`") with the pointer "`this`."

At runtime, we only need to instantiate a `Store` object. The internal mechanism of class `Store` will automatically create and configure a `StoreHour` object. Also, we do not need any global (or OTcl) reference to the `StoreHour` object, since it is usually manipulated by class `Store`.

15.2 Implementation of Random Numbers in NS2

This section demonstrates implementation of random number generators in NS2. In principle, NS2 uses so-called Random Number Generator (RNG) to generate random numbers. An RNG sequentially picks numbers from a stream of pseudo-random numbers. A set of generated random numbers is characterized by the point where the RNG starts picking the numbers – called "seed." By default, NS2 sets the seed to 1. Therefore, the results obtained from every run are essentially the same.

Random numbers can also be transformed to conform to a given distribution. Such a transformation is carried out through instprocs in the OTcl domain, and through classes derived from class `RandomVariable` in the C++ domain. We will discuss the details of RNGs and the seeding mechanism in Sects. 15.2.1 and 15.2.2, respectively. Section 15.2.3 shows the implementation of RNGs in NS2. Section 15.2.4 discusses different simulation scenarios, where RNGs are set differently. Section 15.2.5 explains the implementation of a C++ class `RandomVariable` which transforms random numbers according to a given distribution. Finally, Sect. 15.2.6 gives a guideline to define a new RNG and a new random variable in NS2.

15.2.1 Random Number Generation

NS2 generates random numbers by sequentially picking numbers from a stream of pseudo-random number (as discussed in Sect. 1.3.1). It uses the *combined multiple recursive generator* (MRG32k3a) proposed by L'Ecuyer (1999) as a pseudo-random number generator. Generally speaking, an MRG32k3a generator contains streams of pseudo-random numbers from which the numbers picked sequentially seem to be random. In Fig. 15.5, an MRG32k3a generator provides 1.8×10^{19} independent streams, each of which consists of 2.3×10^{15} substreams. Each substream contains 7.6×10^{22} random numbers (i.e., the period of each substream is 7.6×10^{22}). In summary, an MRG32k3a generator can create 3.1×10^{57} numbers which appear to be random.

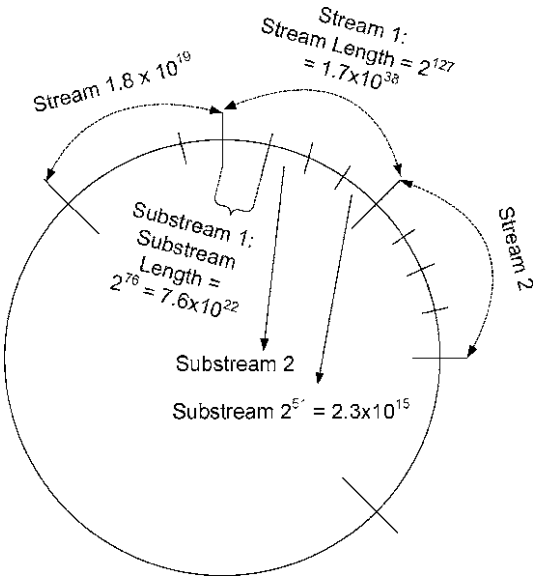

Fig. 15.5 Streams and substreams of an MRG32k3a generator

15.2.2 Seeding a Random Number Generator

As mentioned in Sect. 1.3.1, "seed" is one of the main ingredients of Random Number Generator (RNG). Loosely speaking, a seed specifies the location on a stream of pseudo-random numbers, where an RNG starts picking random numbers sequentially. When seeded differently, two RNGs start picking pseudo-random numbers from different locations, and therefore generate two distinct sets of random numbers. On the other hand, if seeded with the same number, two RNGs will start picking random numbers from the same location, and therefore generate the same set of random numbers.

By default, NS2 always uses only one OTcl variable `defaultRNG` as a default RNG, and always seeds the `defaultRNG` with 1. Therefore, the simulation results for every run are essentially the same. To collect independent simulation results, we must seed different runs differently.

Example 15.2. In the following, we run NS2 for three times to show NS2 seeding mechanism.

```
1 >>ns
2 >>$defaultRNG seed
3 1
4 >>$defaultRNG next-random
5 729236
6 >>$defaultRNG next-random
7 1193744747
8 >>exit
```

```
### RESTART NS2 ###
9 >> ns
10 >>$defaultRNG seed
11 1
12 >>$defaultRNG next-random
13 729236
14 >>$defaultRNG next-random
15 1193744747
16 >>exit

### RESTART NS2 ###
17 >>ns
18 >>$defaultRNG seed 101
19 >>$defaultRNG next-random
20 72520690
21 >>$defaultRNG next-random
22 308637100
23 >>exit
```

In the first run (Lines 1–8), the variable `defaultRNG` (i.e., the default RNG) is used to generate two random numbers. In Line 2, the instproc `seed` returns the current seed which is set (by default) to 1. Lines 4 and 6 use the instproc `next-random{}` to generate two random numbers, 729236 and 1193744747, respectively. Finally, Line 8 exits the NS2 environment.

Lines 9–16 repeat the process in Lines 1–8. In Lines 10 and 11, we can observe that the seed is still 1. As expected, the first and the second random numbers generated are 729236 and 1193744747, respectively. These two numbers are the same as those in the first run. Essentially, the first run and the second run generate the same results. To generate different results, we need to seed the simulation differently.

Lines 17–22 show the last run, where the seed is set differently (to 101). The first and the second random number generated in this case are 72520690 and 308637100, respectively. These two numbers are different from those in the first two runs, since Line 15 sets the seed of the `defaultRNG` to 101. □

The key points about seeding the mechanism in NS2 are as follows:

- A seed specifies the starting location on a stream of pseudo-random numbers, and hence characterizes an RNG.
- To generate two independent simulation results, each simulation must be seeded differently.
- At initialization, NS2 creates a variable `defaultRNG` as the default RNG, and seeds the `defaultRNG` with 1. By default, NS2 generates the same simulation result for every run.

- When seeded with zero, an RNG replaces the seed with current time of the day and counter. Despite their tendency to be independent, two runs may pick the same seed and generate the same result. To ensure independent runs, we must seed the RNG manually.
- NS2 seeds a *new* `RNG` object to the beginning of the next random stream. Therefore, every `RNG` object is independent of each other.

15.2.3 OTcl and C++ Implementation

NS2 uses a C++ class `RNG` (which is bound to an OTcl class with the same name) to generate random numbers (see Program 15.10). In most cases, it is not necessary to understand the details of the MRG32k3a generator. This section shows only the key configuration and implementation in the OTcl and C++ domains. The readers may find the detailed implementation of an MRG32k3a generator in files ~*ns*/tools/rng.cc,h.

Program 15.10 A mapping class `RNGClass` which binds OTcl and C++ classes `RNG`

```
    //~/ns/tools/rng.cc
1 static class RNGClass : public TclClass {
2 public:
3     RNGClass() : TclClass("RNG") {}
4     TclObject* create(int, const char*const*) {
5         return(new RNG());
6     }
7 } class_rng;
```

15.2.3.1 OTcl Commands and Instprocs

In the OTcl domain, class `RNG` defines the following OTcl commands:

`seed{}`	Return the seed of RNG.
`seed{n}`	Set the the seed of RNG to be "n."
`next-random{}`	Return a random number.
`next-substream{}`	Advance to the beginning of the next substream.
`reset-start-substream{}`	Return to the beginning of the current substream.

`normal{avg std}`	Return a random number normally distributed with average "`avg`" and standard deviation "`std`."
`lognormal{avg std}`	Return a random number log-normally distributed with average "`avg`" and standard deviation "`std`."

Defined in file ~*ns*/tcl/lib/ns-random.tcl, the following instprocs generate random numbers:

`exponential{mu}`	Return a random number exponentially distributed with mean "`mu`."
`uniform{min max}`	Return a random number uniformly distributed in `[min,max]`.
`integer{k}`	Return a random integer uniformly distributed in {`0,1, ..., k-1`}.

15.2.3.2 C++ Functions

In the C++ domain, the key functions of class `RNG` include (see the details in files ~*ns*/tools/rng.cc,h):

`set_seed(n)`	If `n = 0`, set the the seed of the RNG to be current time and counter. Otherwise, set the seed to be "`n`."
`seed()`	Return the seed of the RNG.
`next()`	Return a random `int` number in {`0,1,...,MAX_INT`}.
`next_double()`	Return a random `double` number in [0,1].
`reset_start_substream()`	Move to the beginning of the current substream.
`reset_next_substream()`	Move to the beginning of the next substream.
`uniform(k)`	Return a random `int` number uniformly distributed in {`0,1, ..., k-1`}.
`uniform(r)`	Return a random `double` number uniformly distributed in [0,`r`].
`uniform(a,b)`	Return a random `double` number uniformly distributed in [`a`,`b`].

`exponential(k)`	Return a random number exponentially distributed with mean "`k`."
`normal(avg,std)`	Return a random number normally distributed with average "`avg`" and standard deviation "`std`."
`lognormal(avg,std)`	Return a random number log-normally distributed with average "`avg`" and standard deviation "`std`."

15.2.4 Randomness in Simulation Scenarios

In most cases, a simulation falls into one of the following three scenarios.

15.2.4.1 Deterministic Setting

This type of simulation is usually used for debugging. Its purpose is to locate programming errors in the simulation codes or to understand complex behavior of a certain network. In both cases, it is convenient to run the program under a deterministic setting and generate the same result repeatedly. By default, NS2 seeds the simulation with 1. The deterministic setting is therefore the default setting for NS2 simulation.

15.2.4.2 Single-Stream Random Setting

The simplest form of statistical analysis is to run a simulation for several times and compute statistical measures such as average and/or standard deviation. By default, NS2 always uses `defaultRNG` with seed "1" to generate random numbers. To statistically analyze a system, we need to generate several distinct sets of results. Therefore, we need to seed different runs differently. In a single-stream random setting, we need only one RNG. Hence, we may simply introduce the diversity to each run by seeding different runs with different values `<n>` (e.g., in Example 15.2, Line 18 seeds the default RNG with 101).

```
$defaultRNG seed <n>
```

which seeds the default RNG with a number `<n>`.

15.2.4.3 Multiple-Stream Random Setting

In some cases, we may need more than one independent random variable for a simulation. For example, we may need to generate random values of packet

inter-arrival time as well as packet size. These two variables should be independent and should not share the same random stream. We can create two independent RNG using "`new RNG`." Since NS2 seeds each RNG with different random stream (see Sect. 15.2.2), the random processes with different RNGs are independent of each other.

Example 15.3. Suppose that the inter-arrival time and packet size are exponentially distributed with mean 5 and uniformly distributed within [100, 5000], respectively. Print out the first five random values of inter-arrival time and packet size.

Tcl Simulation Script

```
1 $defaultRNG seed 101
2 set arrivalRNG [new RNG]
3 set sizeRNG [new RNG]

4 set arrival_ [new RandomVariable/Exponential]
5 $arrival_ set avg_ 5
6 $arrival_ use-rng $arrivalRNG

7 set size_ [new RandomVariable/Uniform]
8 $size_ set min_ 100
9 $size_ set max_ 5000
10 $size_ use-rng $sizeRNG

11 puts "Inter-arrival time Packet size"
12 for {set j 0} {$j < 5} {incr j} {
13 puts [format "%-8.3f %-4d" [$arrival_ value] \
                        [expr round([$size_ value])]]
14 }
```

Results on the Screen

```
Inter-arrival time   Packet size
1.048                1880
7.919                116
8.061                3635
4.675                2110
7.201                1590
```

The details of the above Tcl simulation script are as follows. Lines 4 and 7 create an exponentially random variable[2] "`arrival_`" and a uniformly distributed

[2]We will discuss the details of random variables in the next section.

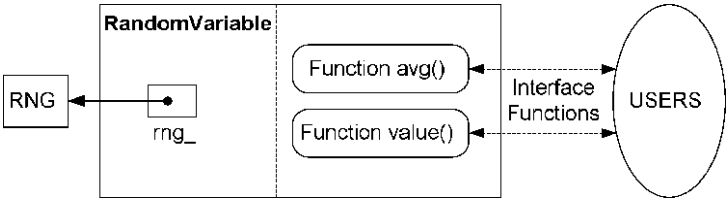

Fig. 15.6 A schematic diagram of class `RandomVariable`

random variable "`size_`" whose parameters are defined in Lines 5 and 6 and Lines 8–10, respectively. Lines 11–14 print out five random numbers generated by "`arrival_`" and "`size_`." In Sect. 15.2.5, we will see that the OTcl command "`value`" of class `RandomVariable` returns a random number, and the OTcl command "`use-rng`" is used to specify an RNG for a random variable.

By default, `defaultRNG`[3] is used to generate random numbers for both "`arrival_`" and "`size_`." In this case, Lines 2 and 3 create two independent RNGs: "`arrivalRNG`" and "`sizeRNG`." NS2 specifies these two variables as RNGs for "`arrival_`" and "`size_`" using an OTcl command `use-rng` in Lines 6 and 10, respectively. Since the created RNG objects are independent, random variable "`arrival_`" and "`size_`" are independent of each other. □

15.2.5 Random Variables

In NS2, a random variable is a module which generates random values whose statistics follow a certain distribution. It uses an RNG to generate random numbers and transforms the generated numbers to values which conform to a given distribution. This implementation is carried out in C++ abstract class `RandomVariable` whose diagram and declaration are shown in Fig. 15.6 and Program 15.11, respectively.

Consider the declaration of class `RandomVariable` in Program 15.11. Class `RandomVariable` contains a pointer "`rng_`" (Line 9) to an `RNG` object (used to generate random numbers), and two pure virtual interface functions: `value()` in Line 3 and `avg()` in Line 4. Function `value()` generates random numbers, transforms the generated numbers to values conforming to the underlying distribution, and returns the transformed values to the caller. Function `avg()` returns the average value of the underlying distribution. Since these two functions are pure virtual, they must be overridden by all derived instantiable classes of class `RandomVariable`. The list of key built-in instantiable C++ classes as well as their bound OTcl classes is given in Table 15.2.

[3]Line 1 sets the seed of `defaultRNG` to be 101. But we do not use `defaultRNG` in this example.

Program 15.11 Declaration of class `RandomVariable`

```
    //~/ns/tools/ranvar.h
1   class RandomVariable : public TclObject {
2    public:
3       virtual double value() = 0;
4       virtual double avg() = 0;
5       int command(int argc, const char*const* argv);
6       RandomVariable();
7       int seed(char *);
8    protected:
9       RNG* rng_;
10  };
```

Table 15.2 Built-in C++ and OTcl random variable classes

C++ class	OTcl class
`UniformRandomVariable`	`RandomVariable/Uniform`
`ExponentialRandomVariable`	`RandomVariable/Exponential`
`ParetoRandomVariable`	`RandomVariable/Pareto`
`ParetoIIRandomVariable`	`RandomVariable/ParetoII`
`NormalRandomVariable`	`RandomVariable/Normal`
`LogNormalRandomVariable`	`RandomVariable/LogNormal`
`ConstantRandomVariable`	`RandomVariable/Constant`
`HyperExponentialRandomVariable`	`RandomVariable/HyperExponential`
`WeibullRandomVariable`	`RandomVariable/Weibull`
`EmpiricalRandomVariable`	`RandomVariable/Empirical`

15.2.5.1 Random Number Generator

A `RandomVariable` object uses its variable "`rng_`" to generate random numbers. By default, every random variable uses the `defaultRNG` as its RNG. As shown in Program 15.12, the constructor (Lines 1–4) of class `RandomVariable` stores the default RNG returned from the statement `RNG::defaultrng()` in the variable "`rng_`."

To create multiple *independent* random variables, the variable "`rng_`" of each random variable must be independent of each other. From Example 15.3, this can be achieved by creating and binding a dedicated RNG to each random variable. As will be discussed in the next section, the process of binding an RNG to a random variable is carried out using the OTcl command `use-rng` associated with a `RandomVariable` object.

15.2.5.2 OTcl Commands

Shown in Program 15.12, class `RandomVariable` defines the following two OTcl commands, which can be invoked from the OTcl domain:

- `value{}`: Returns a random number by invoking the function `value()` (Lines 9–12).

Program 15.12 The constructor, OTcl command `value`, and OTcl command `use-rng` of class `RandomVariable`

```
    //~/ns/tools/ranvar.cc
1   RandomVariable::RandomVariable()
2   {
3       rng_ = RNG::defaultrng();
4   }

    //~/ns/tools/ranvar.cc
5   int RandomVariable::command(int argc, const char*const* argv)
6   {
7       ...
8       if (argc == 2) {
9           if (strcmp(argv[1], "value") == 0) {
10              tcl.resultf("%6e", value());
11              return(TCL_OK);
12          }
13      }
14      if (argc == 3) {
15          if (strcmp(argv[1], "use-rng") == 0) {
16              rng_ = (RNG*)TclObject::lookup(argv[2]);
17              ...
18              return(TCL_OK);
19          }
20      }
21      ...
22  }
```

- `use-rng{rng}`: Casts the input argument "`rng`" to type `RNG*`, and stores the cast object in the variable "`rng_`" (Lines 15–19).

  Note that an example use of the OTcl command `use-rng{rng}` is shown in Lines 6 and 10 in Example 15.3.

Since class `RandomVariable` is abstract, it is not bound to the OTcl domain. However, all its derived classes are bound to the OTcl domain. Table 15.2 lists ten built-in C++ and OTcl random variable classes.

15.2.5.3 Exponential Random Variable

As an example, consider implementation of an exponentially distributed random variable in Program 15.13. From Table 15.2, NS2 implements an exponentially distributed random variable using the C++ class `ExponentialRandomVariable` and the OTcl class `RandomVariable/Exponential`.

Since an exponential random variable is completely characterized by an average value, class `ExponentialRandomVariable` has only one member variable "`avg_`" (Line 9), which stores the average value. At the construction (see Lines 18–20), class `ExponentialRandomVariable` binds its variable "`avg_`" to an

Program 15.13 An implementation of class `ExponentialRandomVariable`

```
    //~/ns/tools/ranvar.h
 1  class ExponentialRandomVariable : public RandomVariable {
 2   public:
 3      virtual double value();
 4      ExponentialRandomVariable();
 5      double* avgp() { return &avg_; };
 6      virtual inline double avg() { return avg_; };
 7      void setavg(double d) { avg_ = d; };
 8   private:
 9      double avg_;
10 };

    //~/ns/tools/ranvar.cc
11 static class ExponentialRandomVariableClass : public
   TclClass {
12 public:
13      ExponentialRandomVariableClass() : TclClass(
  "RandomVariable/Exponential") {}
14      TclObject* create(int, const char*const*) {
15          return(new ExponentialRandomVariable());
16      }
17 } class_exponentialranvar;

18 ExponentialRandomVariable::ExponentialRandomVariable(){
19      bind("avg_", &avg_);
20 }

21 double ExponentialRandomVariable::value(){
22      return(rng_->exponential(avg_));
23 }
```

instvar "`avg_`" in the OTcl domain. Functions `avg()` in Line 6 and `avgp()` in Line 5 return the value stored in "`avg_`" and the address of "`avg_`," respectively. Function `setavg(d)` in Line 7 stores the value in "`d`" into variable "`avg_`." Function `value()` in Lines 21–23 returns a random number exponentially distributed with mean "`avg_`." It invokes function `exponential(avg_)` of variable "`rng_`," feeding variable "`avg_`" as an input argument to obtain an exponentially distributed random number.

15.2.6 Guidelines for Random Number Generation in NS2

We conclude this section by providing the following guidelines for implementing randomness numbers in NS2:

1. Determine the type of simulation: deterministic setting, single-stream random setting, or multi-stream random setting.
2. Create RNG(s) according to the simulation type.
3. If needed, create a random variable
   - Define the inheritance structure: C++, OTcl, and mapping classes.
   - Define function `avg()` which returns the average value of the distribution to the caller.
   - Define function `value()` which returns a random number conforming to the specified distribution.
4. Specify an RNG for each random variable using an OTcl command `use-rng` of class `RandomVariable`.

15.3 Built-in Error Models

An error model is an NS2 module which imposes error on packet transmission. Derived from class `Connector`, it can be inserted between two NsObjects. An error model simulates packet error upon receiving a packet. If the packet is simulated to be in error, the error model will either drop the packet or mark the packet with an error flag. If the packet is simulated not to be in error, on the other hand, the error model will forward the packet to its downstream object. An error model can be used for both wired and wireless networks. However, this section discusses the details of an error model through a wired class `SimpleLink` only.

Program 15.14 Class `ErrorModelClass` which binds C++ and OTcl classes `ErrorModel`

```
    //~/ns/queue/errmodel.cc
1   static class ErrorModelClass : public TclClass {
2   public:
3       ErrorModelClass() : TclClass("ErrorModel") {}
4       TclObject* create(int, const char*const*) {
5           return (new ErrorModel);
6       }
7   } class_errormodel;
```

NS2 implements error models using a C++ class `ErrorModel` which is bound to an OTcl class with the same name (see Program 15.14). Class `ErrorModel` simulates Bernoulli error, where transmission is simulated to be either in error or not in error. NS2 also provides `ErrorModel` classes with more functionalities such as two-state error model. Tables 15.3 and 15.4 show NS2 built-in error models whose implementation is in the C++ and OTcl domain, respectively.

Table 15.3 Built-in error models which contain C++ and OTcl implementation

C++ class	OTcl class	Description
`TwoStateErrorModel`	`ErrorModel/TwoState`	Error-free and error-prone states
`ComplexTwoState MarkovModel`	`ErrorModel/Complex TwoStateMarkov`	Contain two objects of class `TwoStateErrorModel`
`MultiStateErrorModel`	`ErrorModel/MultiState`	Error model with more than two states
`TraceErrorModel`	`ErrorModel/Trace`	Impose error based on a trace file
`PeriodicErrorModel`	`ErrorModel/Periodic`	Drop packets once every *n* packets
`ListErrorModel`	`ErrorModel/List`	Specify the a list of packets to be dropped
`SelectErrorModel`	`SelectErrorModel`	Selective packet drop
`SRMErrorModel`	`SRMErrorModel`	Error model for SRM
`MrouteErrorModel`	`ErrorModel/Trace/ Mroute`	Error model for multicast routing
`ErrorModule`	`ErrorModule`	Send packets to classifier rather than "`target_`"
`PGMErrorModel`	`PGMErrorModel`	Error model for PGM
`LMSErrorModel`	`LMSErrorModel`	Error model for LMS

Table 15.4 Built-in OTcl error models defined in file ~*ns*/tcl/lib/ns-errmodel.tcl

OTcl class	Base class	Description
`ErrorModel/Uniform`	`ErrorModel`	Uniform error model
`ErrorModel/Expo`	`ErrorModel/TwoState`	Two state error model; Each state is represented by an exponential random variable.
`ErrorModel/Empirical`	`ErrorModel/TwoState`	Two state error model; Each state is represented by an empirical random variable.
`ErrorModel/TwoStateMarkov`	`ErrorModel/Expo`	`ErrorModel/Expo` model where the state residence time is exponential

15.3.1 OTcl Implementation: Error Model Configuration

In common with those of most objects, configuration interfaces of an error model are defined in the OTcl domain. Such a configuration includes parameter configuration and network configuration.

15.3.1.1 Parameter Configuration

There are two ways to configure an error model object: through bound variables and through OTcl commands. Class `ErrorModel` binds the following C++ variables to OTcl instvars with the same name:

`enabled_`	Set to 1 if this error model is active, and set to 0 otherwise.
`rate_`	Error probability
`delay_pkt_`	If set to `true`, the error model will delay (rather than drop) the transmission of corrupted packets.
`delay_`	Delay time in case that `delay_pkt_` is set to `true`.
`bandwidth_`	Used to compute packet transmission time
`markecn_`	If set to `true`, the error model will mark error flag (rather than drop) in flag header of the corrupted packet.

The second configuration method is through the following OTcl commands whose input arguments are stored in `args`:

`unit{arg}`	Store `arg` in C++ variable "`unit_`."
`ranvar{arg}`	Store `arg` in C++ variable "`ranvar_`."
`FECstrength{arg}`	Store `arg` in C++ variable `FECstrength_`.
`datapktsize{arg}`	Store `arg` in C++ variable "`datapktsize_`."
`cntrlpktsize{arg}`	Store `arg` in C++ variable "`cntrlpktsize_`."
`eventtrace{arg}`	Store `arg` in C++ variable "`et_`."

Among the above OTcl commands, `unit{}`, `ranvar{}`, and `FEC strength{}`, when taking no input argument, return values stored in "`unit_`," "`ranvar_`," and "`FECstrength_`," respectively.

15.3.1.2 Network Configuration

As a `Connector` object, an error model can be inserted into a network to simulate packet errors. OTcl defines two pairs of instprocs to insert an error model into a `SimpleLink` object (see Sect. 7.1). Each pair consists of one instproc from class `SimpleLink` and one instproc from class `Simulator` as shown below (see Fig. 15.7):

- `SimpleLink::errormodule{em}`: Inserts an error model "`em`" right after the head of a `SimpleLink` object.

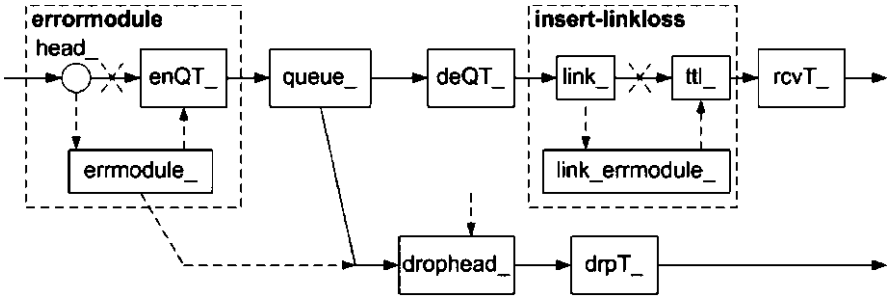

Fig. 15.7 Instprocs `errormodule` and `insert-linkloss` of class `SimpleLink`

- `Simulator::lossmodel{lossobj from to}`: Executes "`errormodule`"from within the `SimpleLink` object which connects node "`from`" to node "`to`."
- `SimpleLink::insert-linkloss{em}`: Inserts an error model "`em`" right after instvar "`link_`" of the `SimpleLink` object.
- `Simulator::link-lossmodel{lossobj from to}`: Executes the instproc "`insert-linkloss{...}`" from within the `SimpleLink` object which connects node "`from`" to node "`to`."

Program 15.15 shows the details of instproc `errormodule{em}` of class `SimpleLink`, which inserts the input error model (e.g., "`em`") immediately after the link's head. Lines 6 and 7 store the input error model (i.e., "`em`") in instvar "`errmodule_`." Line 8 inserts the input error model next to the link's head by invoking instproc `add-to-head{em}`, and Line 9 sets the drop target of the input error model "`em`" to "`drophead_`."

Program 15.15 Instproc `errormodule` of class `SimpleLink`, and instproc `add-to-head` of class `Link`

```
    //~/ns/tcl/lib/ns-link.tcl
 1  SimpleLink instproc errormodule args {
 2      $self instvar errmodule_ queue_ drophead_
 3      if { $args == "" } {
 4          return $errmodule_
 5      }
 6      set em [lindex $args 0]
 7      set errmodule_ $em
 8      $self add-to-head $em
 9      $em drop-target $drophead_
10  }

11  Link instproc add-to-head { connector } {
12      $self instvar head_
13      $connector target [$head_ target]
14      $head_ target $connector
15  }
```

In Lines 11–15 of Program 15.15, instproc `add-to-head{connector}` inserts the input argument "`connector`" between link's head (i.e., the instvar "`head_`") and target of the link's head (see Lines 13 and 14).

Program 15.16 shows the details of instproc `insert-linkloss{em}`, which inserts the input error model after the instvar "`link_`." Line 6 stores the input error model in a local variable "`em`." Lines 7–9 delete the instvar "`link_errmodule_`" if it exists. Then Line 10 stores the variable "`em`" in the instvar "`link_errmodule_`." Lines 11 and 12 insert the variable "`em`" immediately after the instvar "`link_`." Finally, Line 13 sets the drop target of the variable "`em`" to be the instvar "`drophead_`."

Program 15.16 An instproc `insert-linkloss` of class `SimpleLink`

```
    //~/ns/tcl/lib/ns-link.tcl
1   SimpleLink instproc insert-linkloss args {
2       $self instvar link_errmodule_ queue_ drophead_ link_
3       if { $args == "" } {
4           return $link_errmodule_
5       }
6       set em [lindex $args 0]
7       if [info exists link_errmodule_] {
8           delete link_errmodule_
9       }
10      set link_errmodule_ $em
11      $em target [$link_ target]
12      $link_ target $em
13      $em drop-target $drophead_
14  }
```

In most cases, a `SimpleLink` object is inaccessible from a Tcl simulation script. Therefore, class `Simulator` provides interface instprocs `lossmodel{...}` and `link-lossmodel{...}` to invoke instprocs `errormodule{em}` and `insert- linkloss{em}`, respectively, of class `SimpleLink`.[4]

The details of both the instproc `lossmodel{lossobj from to}` and the instproc `link-lossmodel{lossobj from to}` of class `Simulator` are shown in Program 15.17, where they insert an error model "`lossobj`" into the link which connect a node "`from`" to a node "`to`." Lines 2 and 6 invoke instproc `link{from to}` of class `Simulator`. In Line 18, this instproc returns the `Link` object which connects a node "`from`" to a node "`to`." Lines 3 and 7 then insert an error model into the returned `Link` object, by executing `errormodule{em}` and `insert-linkloss{em}`, respectively.

[4]Caution: The details of instproc `insert-linkloss` has been changed slightly since NS version 2.35. The configuration (as in Fig. 15.7) might look different under different versions of NS2.

Program 15.17 Instprocs `lossmodel`, `link-lossmodel`, and `link` of class `Simulator`

```
     //~/ns/tcl/lib/ns-lib.tcl
  1  Simulator instproc lossmodel {lossobj from to} {
  2      set link [$self link $from $to]
  3      $link errormodule $lossobj
  4  }

  5  Simulator instproc link-lossmodel {lossobj from to} {
  6      set link [$self link $from $to]
  7      $link insert-linkloss $lossobj
  8  }

  9  Simulator instproc link { n1 n2 } {
  10     $self instvar Node_ link_
  11     if { ![catch "$n1 info class Node"] } {
  12     set n1 [$n1 id]
  13     }
  14     if { ![catch "$n2 info class Node"] } {
  15     set n2 [$n2 id]
  16     }
  17     if [info exists link_($n1:$n2)] {
  18         return $link_($n1:$n2)
  19     }
  20     return ""
  21 }
```

15.3.2 C++ Implementation: Error Model Simulation

The internal mechanism of an error model is specified in the C++ domain. As shown in Program 15.18, C++ class `ErrorModel` derives from class `Connector`. It uses packet forwarding/dropping capabilities (e.g., a variable "`target_`" and a function `recv(p,h)`) inherited from class `Connector`, and define error simulation mechanism.

15.3.2.1 Variables

The key variables of class `ErrorModel` are given below:

`enable_`	Set to 1 if this error model is active, and set to 0 otherwise
`rate_`	Error probability
`delay_`	Time used to delay (rather than dropping) a corrupted packet
`bandwidth_`	Transmission bandwidth used to compute packet transmission time

Program 15.18 Declaration of class `ErrorModel`

```
     //~/ns/queue/errmodel.h
  1  enum ErrorUnit { EU_TIME=0, EU_BYTE, EU_PKT, EU_BIT };

  2  class ErrorModel : public Connector {
  3  public:
  4      ErrorModel();
  5      virtual void recv(Packet*, Handler*);
  6      virtual void reset();
  7      virtual int corrupt(Packet*);
  8      inline double rate() { return rate_; }
  9      inline ErrorUnit unit() { return unit_; }
 10  protected:
 11      int enable_;
 12      ErrorUnit unit_;
 13      double rate_;
 14      double delay_;
 15      double bandwidth_;
 16      RandomVariable *ranvar_;
 17      int FECstrength_;
 18      int datapktsize_;
 19      int cntrlpktsize_;
 20      double *cntrlprb_;
 21      double *dataprb_;
 22      Event intr_;
 23      virtual int command(int argc, const char*const* argv);
 24      int CorruptPkt(Packet*);
 25      int CorruptByte(Packet*);
 26      int CorruptBit(Packet*);
 27      double PktLength(Packet*);
 28      double* ComputeBitErrProb(int);
 29  };

     //~/ns/queue/errmodel.cc
 30   ErrorModel::ErrorModel() : firstTime_(1), unit_(EU_PKT),
      ranvar_(0), FECstrength_(1)
 31   {
 32       bind("enable_", &enable_);
 33       bind("rate_", &rate_);
 34       bind("delay_", &delay_);
 35  }
```

`unit_`	Error unit (`EU_TIME`, `EU_BYTE`(default), `EU_PKT`, or `EU_BIT`)
`ranvar_`	Random variable which simulates error
`FECstrength_`	Number of bits in a packet which can be corrected
`datapktsize_`	Number of bytes in a data packet
`cntrlpktsize_`	Number of bytes in a control packet

dataprb_	An array whose ith entry is the probability of having at most i corrupted data bits
cntrlprb_	An array whose ith entry is the probability of having at most most i corrupted control bits
firstTime_	Indicate whether an error has occurred.
intr_	A queue callback object (see Sect. 7.3.3).

The variable "rate_" specifies the error probability, while the variable "unit_" indicates the unit of "rate_." If "unit_" is packets (i.e., EU_PKT), "rate_" will represent packet error probability. If "unit_" is bytes (i.e., EU_BYTE) or bits (i.e., EU_BIT), "rate_" will represent byte error probability or bit error probability, respectively.

15.3.2.2 Functions

The key functions of class ErrorModel are given below:

rate()	Return the error probability stored in variable "rate_."
unit()	Return the error unit stored in variable "unit_."
PktLength(p)	Return the length (in error units) of the packet "p."
reset()	Set the variable "firstTime_" to 1.
recv(p,h)	Receive a packet "p" and a handler "h."
corrupt(p)	Return 1/0 if the transmission is in error/not in error.
CorruptPkt(p)	Return 1/0 if the transmission is in error/not in error.
CorruptByte(p)	Return 1/0 if the transmission is in error/not in error.
CorruptBit(p)	Return the number of corrupted bits.
ComputeBitErrProb(size)	Computes the cumulative distribution of having i= $\{0, \cdots,$ FECstrength_$\}$ corrupted bits.

Program 15.19 Function `recv(p,h)` of class `ErrorModel`

```
    //~/ns/queue/errmodel.cc
1   void ErrorModel::recv(Packet* p, Handler* h)
2   {
3       hdr_cmn* ch = hdr_cmn::access(p);
4       int error = corrupt(p);
5       if (h && ((error && drop_) || !target_)) {
6           double delay = Random::uniform(8.0*ch->size()/
        bandwidth_);
7           if (intr_.uid_ < 0)
8               Scheduler::instance().schedule(h, &intr_,
        delay);
9       }
10      if (error) {
11          ch->error() |= error;
12          if (drop_) {
13                  drop_->recv(p);
14                  return;
15          }
16      }
17      if (target_) {
18              target_->recv(p, h);
19      }
20  }
```

15.3.2.3 Main Mechanism

The main mechanism of an `ErrorModel` object lies within the packet reception function `recv(p,h)` shown in Program 15.19. When receiving a packet, an `ErrorModel` object simulates packet error (by invoking function `corrupt(p)` in Line 4 of Program 15.19), and reacts to the error based on the underlying configuration. If an error occurs, Line 11 will mark an error flag in the common packet header. Then if "`drop_`" exists, Lines 13 and 14 will drop the packet and terminate the function. If the packet is not in error, on the other hand, function `recv(p,h)` will skip Lines 11–15 and will forward the packet to "`target_`" if it exists. A cautionary note: since a corrupted packet will also be forwarded to "`target_`" if "`drop_`" does not exist, *NS2 will not show any error but the simulation results might not be correct!*

Lines 6–8 in Program 15.19 are related to NS2 callback mechanism discussed in Sect. 7.3.3. Callback mechanism is an NS2 technique to have a downstream object invoke an upstream object along a downstream path. For example, after transmitting a packet, a queue needs to wait until the packet leaves the queue (i.e., wait for a *callback* signal to release the queue from the blocked state), before commencing another packet transmission. From Sect. 7.2, a `LinkDelay` object uses the Scheduler to inform the queue of packet departure (i.e., send a release signal) at the packet departure time.

A callback process is implemented by passing the handler (h) of an upstream object (e.g., the queue) along with packet (p) to a downstream object through function recv(p,h). Upon receiving the handler, an NsObject reacts by either (1) passing the handler to its downstream object and hoping that the handler will be dealt with somewhere along the downstream path, or (2) immediately scheduling a callback event at a certain time.

Condition (1) occurs when an upstream object passes down the handler "h," and is waiting for a callback signal. Condition (2) indicates the case where the ErrorModel object is responsible for sending a callback signal.[5] Condition (2) consists of the two following subconditions. One is the case where the packet will be dropped. Another is when "target_" does not exist. In these cases, the ErrorModel will be the last object in a downstream path which can deal with the packet, and is therefore responsible for the callback mechanism.

According to Line 5 in Program 15.19, the ErrorModel object chooses to call back when both of the following conditions are satisfied:

1. Handler "h" exists (i.e., nonzero), and
2. Either
   (a) Packet is in error and the variable "drop_" exists, and/or
   (b) The variable "target_" does not exist.

When choosing to callback, Line 8 schedules a callback event after a delay time of "delay" seconds. NS2 assumes that an error can occur in any place in a packet with equal probability. Correspondingly, the time at which an error is materialized is uniformly distributed in $[0, txt]$, where txt is the packet transmission time (Line 6).

15.3.2.4 Simulating Transmission Errors

In the previous section, we discussed how class ErrorModel forwards or drops (or marks with an error flag) packets based on the simulated error. This discusses the details of function corrupt(p) which simulates transmission error. Taking a packet pointer "p" as an input argument, the function corrupt(p) returns zero and one if the transmission is simulated not to be and to be in error, respectively.

Program 15.20 shows the details of function corrupt(p). The function corrupt(p) always returns zero if the ErrorModel object is disabled (i.e., enable_=0; see Lines 4 and 5). Given that the ErrorModel object is enabled, the function corrupt(p) returns a logic value (i.e., true or false) depending on whether the value returned from the functions CorruptPkt(p) in Line 16, CorruptByte(p) in Line 10, CorruptBit(p) in Lines 13 and 14, and CorruptTime(p) in Line 8 is zero, when "unit_" is equal to

[5] If not, the ErrorModel object will assign the responsibility to its downstream object. In this case, the handler "h" should be passed to the downstream object, by invoking target_->recv(p,h).

Program 15.20 Functions `corrupt` `CorruptPkt`, `CorruptByte`, and `PktLength` of class `ErrorModel`

```
    //~/ns/queue/errmodel.cc
 1  int ErrorModel::corrupt(Packet* p)
 2  {
 3      hdr_cmn* ch = HDR_CMN(p);
 4      if (enable_ == 0)
 5          return 0;
 6      switch (unit_) {
 7      case EU_TIME:
 8          return (CorruptTime(p) != 0);
 9      case EU_BYTE:
10          return (CorruptByte(p) != 0);
11      case EU_BIT:
12          ch = hdr_cmn::access(p);
13          ch->errbitcnt() = CorruptBit(p);
14          return (ch->errbitcnt() != 0);
15      default:
16          return (CorruptPkt(p) != 0);
17      }
18      return 0;
19  }

20  int ErrorModel::CorruptPkt(Packet*)
21  {
22      double u = ranvar_ ? ranvar_->value() : Random::
    uniform();
23      return (u < rate_);
24  }

25  int ErrorModel::CorruptByte(Packet* p)
26  {
27      double per = 1 - pow(1.0 - rate_, PktLength(p));
28      double u = ranvar_ ? ranvar_->value() : Random::
    uniform();
29      return (u < per);
30  }

31  double ErrorModel::PktLength(Packet* p)
32  {
33      if (unit_ == EU_PKT)
34          return 1;
35      int byte = hdr_cmn::access(p)->size();
36      if (unit_ == EU_BYTE)
37          return byte;
38      if (unit_ == EU_BIT)
39          return 8.0 * byte;
40      return 8.0 * byte / bandwidth_;
41  }
```

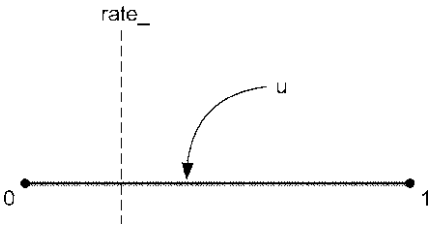

Fig. 15.8 Transforming uniform distribution to Bernoulli distribution

EU_PKT, EU_BYTE, EU_BIT, and EU_TIME, respectively. Similar to the function corrupt(p), these functions return a zero and a nonzero value if the packet is not in error and is in error, respectively.

In some cases, the packet error process in a communication link can be modeled as having Bernoulli distribution. Suppose that "ranvar_" (Line 16 in Program 15.18) is a random variable which generates uniformly distributed random numbers "u" in the range [0,1]. From Fig. 15.8, "u" could be any point "×" in [0,1] with equal probability. Given a threshold "rate_," "u" will be in [0,rate_) with probability "rate_." In other words, to have probability of "rate_" for an event (e.g., packet error), we need to generate a uniformly distributed random number "u," and assume the occurrence of the event if and only if u < rate_.

Lines 20–41 of Program 15.20 show the details of functions CorruptPkt(p), CorruptByte(p), and pktLength(p). Function CorruptPkt(p) in Lines 20–24 uses the above method (see Fig. 15.8) to simulate packet error. In other words, it generates uniformly distributed random numbers "u" and assumes that a packet is in error if and only if u < rate_.

For function CorruptByte(p), the variable "rate_" represents byte error probability. Line 27 translates byte error probability to packet error probability (per)[6] and simulates packet error in the same way as the function CorruptPkt(p) does.

Function PktLength(p) in Lines 31–40 of Program 15.20 computes the length of a packet in the corresponding "unit_." In particular, if "unit_" is

- EU_PKT, function PktLength(p) will return 1 (see Line 34).
- EU_BYTE, function PktLength(p) will return the number of bytes in the packet stored in field "size_" of the common packet header (see Lines 35–37).
- EU_BITS, function PktLength(p) will return the number of bits in the packet (see Line 39).
- EU_TIME (if none of the above matches), function PktLength(p) will return the transmission time of the packet (see Line 40).

Program 15.21 shows the details of function CorruptBit(p) of class ErrorModel. When this function is called for the first time (i.e., "firstTime_" is 1),

[6]Packet error probability is $1 - (1-\texttt{rate\_})^n$, where "rate_" is byte error probability and n = PktLength(p) is number of bytes in a packet.

Lines 5 and 6 precompute error probabilities for a control packet and a data packet and store the probabilities in "cntrlprb_" and "dataprb_," respectively. The computation is achieved via function ComputeBitErrProb(size) which takes the size of a control packet (i.e., size=cntrlpktsize_) or a data packet (i.e., size=datapktsize_) as its input argument. The values stored in cntrlprb_[i] and dataprb_[i] denote the probability that at most i bits are in error. Line 7 then sets "firstTime_" to zero so that function CorruptBit will skip Lines 5–7 when it is invoked again.

Function CorruptBit(p) computes packet error probability based on either "dataprb_" or "cntrlprb_." Any packet whose size is at least as large as "datapktsize_" is considered a data packet. In this case, Line 10 stores "dataprb_" in "dpfr", later used to compute bit error probability. If, on the other hand, the packet size is smaller than "datapktsize_," it will be considered a control packet, and "cntrlpb_" will be stored in "dptr" as bit error probability.

Program 15.21 Functions CorruptBit and ComputeBitErrProb of class ErrorModel

```
     //~/ns/queue/errmodel.cc
  1  int ErrorModel::CorruptBit(Packet* p)
  2  {
  3      double u, *dptr; int i;
  4      if (firstTime_ && FECstrength_) {
  5          cntrlprb_ = ComputeBitErrProb(cntrlpktsize_);
  6          dataprb_ = ComputeBitErrProb(datapktsize_);
  7          firstTime_ = 0;
  8      }
  9      u = ranvar_ ? ranvar_->value() : Random::uniform();
  10     dptr = (hdr_cmn::access(p)->size() >= datapktsize_)
                                          ? dataprb_ : cntrlprb_;
  11     for (i = 0; i < (FECstrength_ + 2); i++)
  12         if (dptr[i] > u) break;
  13     return(i);
  14 }

  15 double* ErrorModel::ComputeBitErrProb(int size)
  16 {
  17     double *dptr; int i;
  18     dptr = (double *)calloc((FECstrength_ + 2), sizeof
                                                    (double));
  19     for (i = 0; i < (FECstrength_ + 1) ; i++)
  20          dptr[i] = comb(size, i) * pow(rate_,
                (double)i) * pow(1.0 - rate_, (double)
                                                  (size - i));
  21     for (i = 0; i < FECstrength_ ; i++)
  22         dptr[i + 1] += dptr[i];
  23     dptr[FECstrength_ + 1] = 1.0;
  24     return dptr;
  25 }
```

Since the value stored in `dptr[i]` is the probability that at most "`i`" bits are in error, Lines 11 and 12 increment "`i`" until the probability exceeds "`u`" and returns "`i`" to the caller. In this case, the variable "`i`" is the number of corrupted bits.

The details of function `ComputeBitErrProb(size)` are shown in Program 15.21. This function takes the packet size as an input argument and returns an array "`dptr`" of `double` whose i^{th} entry contains the probability of having at most `i` corrupted bits. Given a packet size "`size`," the probability of having exactly `i` corrupted bits is $\binom{\texttt{size}}{\texttt{i}}(\texttt{rate\_})^{\texttt{i}}(1-\texttt{rate\_})^{\texttt{size-i}}$, as shown in Line 20, where "`rate_`" is the bit error probability. Lines 21–23 compute the cumulative summation of "`dprt`." Note that Line 23 sets `dptr[FECstrength_ + 1]` to 1.0 since a packet is considered to be in error if the number of corrupted bits is greater than `FECstrength_`.

15.3.3 Guidelines for Implementing a New Error Model in NS2

In order to implement a new error model in NS2, we need to follow the three steps below:

1. Design and create an error model class in OTcl, C++, or both domains.
2. Configure the parameters of the error model object such as error probability (`rate_`), error unit (`unit_`), and random variable (`ranvar_`).
3. Insert an error model into the network (e.g., using instproc `lossmodel {lossobj from to}` or instproc `link-lossmodel{lossobj from to}` of class `Simulator`).

Example 15.4. Consider the simulation script in Program 9.1, which creates a network as shown in Fig. 9.3. Include an error model with packet error probability 0.1 for the link connecting nodes `n1` and `n3`.

Tcl Simulation Script

```
1  set ns [new Simulator]
2  set n1 [$ns node]
3  set n2 [$ns node]
4  set n3 [$ns node]
5  $ns duplex-link $n1 $n2 5Mb 2ms DropTail
6  $ns duplex-link $n2 $n3 5Mb 2ms DropTail
7  $ns duplex-link $n1 $n2 5Mb 2ms DropTail

8  set em [new ErrorModel]
9  $em set rate_ 0.1
10 $em unit pkt
```

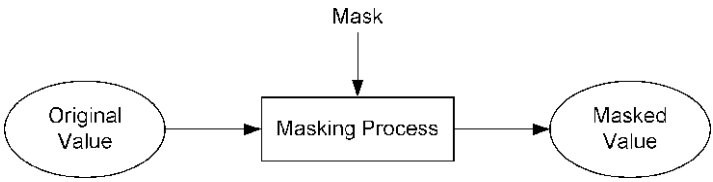

Fig. 15.9 Bit masking

```
11 $em ranvar [new RandomVariable/Uniform]
12 $em drop-target [new Agent/Null]
13 $ns link-lossmodel $em $n1 $n3

14 set udp [new Agent/UDP]
15 set null [new Agent/Null]
16 set cbr [new Application/Traffic/CBR]
17 $ns attach-agent $n1 $udp
18 $ns attach-agent $n3 $null
19 $cbr attach-agent $udp
20 $ns connect $udp $null
21 $ns at 1.0 "$cbr start"
22 $ns at 100.0 "$cbr stop"
23 $ns run
```

where Lines 8–13 are included (into the simulation script in Program 9.1) to impose error on packet transmission. Note that the OTcl command `unit{u}` sets variable "`unit_`" to the value corresponding to the input argument "`u`." The possible values of "`u`" include "`time`," "`byte`," "`pkt`," and "`bit`." □

15.4 Bit Operations in NS2

15.4.1 Bit Masking

Bit masking is a bit transformation technique, which can be used for various purposes. Given a *mask*, a *bit masking process* transforms an *original value* to a *masked value* (see Fig. 15.9). In this section, we will show two examples of bit masking: subnet masking and modulo masking.

15.4.1.1 Subnet Masking

A 4-byte IP address can be divided into host address and network address. While a host address identifies a host (e.g., a computer), a network address characterizes a

group of hosts. A host is given a host IP address as its identification and a 4-byte *subnet mask* which identifies its network. A subnet mask consists of all-one upper bits and all-zero lower bits (i.e., of format "$1\cdots10\cdots0$"). For a given host IP address and a subnet mask, the network IP address can be determined as follows:

$$\text{Network IP Address} = \text{Host IP Address} \,\&\, \text{Subnet Mask} \tag{15.1}$$

where & is a bitwise "AND" operator.

Example 15.5. A class-C (i.e., subnet mask = 255.255.255.0) host IP address 10.1.2.3 has the network IP address of

$$(10.1.2.3)\&(255.255.255.0) = (10\&255).(1\&255).(2\&255).(3\&0) = 10.1.2.0 \tag{15.2}$$

In fact, all class-C IP addresses whose first three bytes are 10.1.2 have the same network address. Correspondingly, a class-C network address corresponds to 256 IP addresses. □

From the above example, the original value (i.e., host IP address) 10.1.2.3 is *masked* (using bitwise "and") with a *mask* 255.255.255.0 (i.e., class C subnet mask) such that the *masked value* (i.e., network IP address) is 10.1.2.0.

15.4.1.2 Modulo Masking

Modulo is a remainder computation process. Suppose $a = b \times c + d$. Then $a\%c = d$, where % is a modulo operator. Bit masking can also be used as a modulo operator with $c = 2^n$ where n is a positive integer.

To implement a modulo masking, the upper and lower bits of a modulo mask are set to contiguous zeros and contiguous ones, respectively (i.e., of format "$0\cdots01\cdots1$"), and the masking operation is a bitwise "AND" operation. Suppose, an original value is of format $xx..xx$, where x can be zero or one. The modulo masking applies bitwise "AND" to an original value and the modulo mask, and obtains the masked value as follows:

$$\begin{array}{lcl} \text{original value} & = & x\cdots xx\cdots x \\ \text{upper-bound mask} & = & 0\cdots01\cdots1 \\ \text{masked value} & = & 0\cdots0x\cdots x. \end{array}$$

Suppose the number of one-bits of a modulo mask is n. The bits whose positions are greater than n are removed during a masking process, and the masked value is bounded by $2^n - 1$. On the other hand, the bits whose positions are not greater than n are kept unchanged. These lower order bits in fact represent the remainder when the original value is divided by 2^n. Modulo masking is therefore equivalent to a modulo operation.

Table 15.5 Components of subnet masking and modulo masking

Masking components	Subnet masking	Modulo masking
The mask	$1\cdots10\cdots0$	$0\cdots01\cdots1$
The mask operation	Bitwise "AND"	Bitwise "AND"
Masked value	Network IP address	Remainder

We summarize the masking components of subnet masking and modulo masking in Table 15.5. Note that both subnet masking and modulo masking use a bitwise "AND" as their mask operation. Since their masks are different, the implications for their masked value are different.

15.4.2 Bit Shifting and Decimal Multiplication

Another important bit operation is bit shifting which is equivalent to decimal multiplication. If a binary value is shifted to the left by n bits, the corresponding decimal value will increase by 2^n times. Similarly, a binary number right shifted by n bits returns the quotient of the decimal value divided by 2^n.

To prove the above statement, consider an arbitrary value $y = \sum_{m=0}^{M} x_m 2^m$, where $x_m \in \{0, 1\}, m = \{0, \cdots, M\}$. Let $y << n$ denote the value of y after being shifted to the left by n bits. Then

$$\begin{aligned} y << n &= \left(\sum_{m=0}^{M} x_m 2^m\right) << n = (\underbrace{x x \cdots x}_{M \text{ bits}}) << n \\ &= \left(\sum_{m=0}^{M} x_m 2^{m+n}\right) \quad = (\underbrace{x x \cdots x}_{M \text{ bits}} \underbrace{0 0 \cdots 0}_{n \text{ bits}}). \end{aligned} \tag{15.3}$$

Suppose $y = \sum_{m=0}^{M} x_m 2^m$. We have

$$y \times 2^n = \left(\sum_{m=0}^{M} x_m 2^m\right) \times 2^n = \left(\sum_{m=0}^{M} x_m 2^{m+n}\right) \tag{15.4}$$

which is the same as (15.3). This proves the first part (i.e., left shifting) of the above statement. The second part of (i.e., right shifting) the statement can be proven similarly and is omitted for brevity.

The relationship between bit shifting and decimal multiplication can be summarized as follows:

- An n-bit left shift results in multiplication of the decimal value by 2^n.
- An n-bit right shift returns the quotient when the decimal value is divided by 2^n.

15.5 Chapter Summary

This chapter presents three major helper classes: timers, random number generators, and error models. The first helper class is `Timer`. Unless restarted or cancelled, class `Timer` waits for a certain time and takes expiration actions. Class `Timer` provides three main interface functions to start, restart, and cancel the waiting process. Class `Timer` is usually cross-referenced to another object, which contains an instruction on how to perform expiration actions. At the expiration (i.e., when `expire(e)` is invoked), the timer informs the object to execute the expiration actions. The object, on the other hand, may start, restart, or cancel the timer through its reference to the timer.

The second part of this chapter demonstrates how NS2 implements Random Number Generator (RNG) to generate random numbers. By default, NS2 always seeds the simulation with 1 – meaning NS2 is *deterministic* by default. To introduce randomness into simulation, we need to seed `defaultRNG` differently.

The last helper class is class `ErrorModel` which is a packet error simulation class. Derived from class `Connector`, it can be inserted into a network using OTcl instprocs (e.g., `lossmodel{...}` and `insert-lossmodel{...}`). Class `ErrorModel` simulates packet error upon a packet reception. If the packet is simulated to be in error, it will either drop or mark the corrupted packet with an error flag. Otherwise, it will forward the packet to its downstream object.

This chapter also presents two main bit operations: bit masking and bit shifting. Bit masking is a bit transformation process which can be used for various purposes. This chapter gives two examples of bit basking. One is subnet masking, which is a process to determine a network for an IP address. Another is a modulo masking, which can be used as a modulo operation. As another bit operation, bit shifting can be used for decimal multiplication or division. Shifting an original value to the left and right by n bits is equivalent to multiplying and dividing the original value by 2^n, respectively.

15.6 Exercises

1. In Example 15.3,
   a. Change the seed to "999." Rerun the script for a couple of times. Observe and explain the output.
   b. Change the seed to "0." Rerun the script for a couple of times. Observe and explain the output.
   c. Print out the values of "`arrival_`" and "`size_`" for (a) and (b), and show that they are exponentially and uniformly distributed. (Hint: Set the seed properly.)
   d. Change the mean of "`arrival_`" to 10 and the interval of "`size_`" to [400, 2000], and repeat (c).

e. Remove Line 6 and repeat (c). Observe and explain the output.
f. Remove Lines 6 and 10 and repeat (c). Observe and explain the output.

2. Write a simulation script which generates random numbers exponentially distributed with mean 1.0. To verify the script, plot the probability density function.
3. Write a simulation script which generates a random number normally distributed with mean 1.0 and standard deviation 0.05. To verify the script, plot the probability density function.
4. Develop a new class for a discrete random variable whose probability mass function is $(0.1, 0.3, 0.3, 0.2, 0.1)$. Test the code by generating random numbers and verify the probability mass function.
5. In Example 15.4, collect statistics for packets which are in error and not in error. Verify that the packet error probability is 0.1. Adjust the simulation time if necessary. How long must your simulation be to ensure the convergence of 0.1 error probability ?

   a. Initially set link bandwidth to 5 Mbps.
   b. Change the bandwidth to 500 kbps. What happens to the measured convergence time ? Explain why.

6. Consider a two state error model, which consists of *good* and *bad* states. Packet transmission in a good state is always error free, while packet transmitted in a bad state is always corrupted. The time that an error model stays in good and bad states is exponentially distributed with means t_{good} and t_{bad}, respectively. Write a simulation script for the above two state error model with $t_{\text{good}} = 10$ sand $t_{\text{bad}} = 1$ s. Verify the results and show the convergence time.
7. Let a modulo mask be 64. Show that the modulo masking and modulo operation are equivalent for the following original values: 63, 64, 65, 127, 128, and 129.
8. Consider a ball color-number matching experiment, where balls are fed one-by-one to an observer. Each ball is masked with a color and a number. The color can be either black or white, while the unique number is increased one-by-one as the balls are fed to the observer. From time to time, the observer is given a number and is asked to identify the color of one of the 64 most recently observed balls. Design a memory-friendly approach for the observation.
9. What are the values of 2, 3, 31, 45, and 56, when shifted to the left and right by 1, 2, and 3 bits?

A
Programming Essentials

This appendix covers the programming languages, which are essential for developing NS2 simulation programs. These include Tcl/OTcl which is the basic building block of NS2 and AWK which can be used for postsimulation analysis.

A.1 Tcl Programming

Tcl is an interpreted language, whose strength is its simplicity. Most Tcl statements can be contained in one line [36]. At runtime, Tcl translates and executes Tcl statements line by line without the need for program compilation. This section digests [36] down to only what necessary to understand NS2. Readers may refer to [36] for more details on Tcl.

A.1.1 Program Invocation

There are three approaches for executing a Tcl program:

1. *Interpret one Tcl statement*: At the command prompt, execute

```
>>tclsh <Tcl Statement>
>>ns <Tcl Statement>
```

   where "`tclsh`" is an executable file which invokes the Tcl interperter.[1] Since NS2 understands Tcl, an alternative to enter into the Tcl environment is to enter NS2 environment by executing "`ns`" (i.e., the lower line) instead of "`tclsh`" (i.e., the upper line).

[1]For NS2 version 2.35, the file `tclsh.exe` is located in the directory `ns-allinone-2.35/tcl8.5.8/unix`.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_16, © Springer Science+Business Media, LLC 2012

2. *Enter into the Tcl environment and interpret Tcl statement line by line*: The first step for this approach is to enter a Tcl environment by executing "`tclsh`" or "`ns`" at the command prompt. Then, write down the Tcl statements line by line. Finally, exit the Tcl environment by the *command* "`exit`" or a control key "`Ctrl + C`."
3. *Interpret a Tcl Scripting File*: This approach is similar to the first one. At the command prompt, execute

```
>> tclsh <filename> [<arg0> <arg1> ...]
>> ns <filename> [<arg0> <arg1> ...]
```

Here, Tcl interprets the statements in the file `<filename>` line by line.

Note that "`<arg0> <arg1> ...`" are the optional input arguments, which can be used within a Tcl program via the following two special variables:

`argv`	A list variable containing all input arguments provided at the program invocation
`argc`	The number of elements in "`argv`"

A.1.2 Syntax

Unlike keyword-based languages such as C++ or Java, Tcl is a position-based language. There is no reserved word (e.g., `printf` in C++) in Tcl. Tcl differentiates user-defined words from *command* words by looking at the position of the word. More specifically,

- The first word in a statement is always a *command* name.
- Each word in a statement is separated by a white space.[2]
- A statement is terminated with a semicolon (i.e., "`;`") or an end of the line.
- The following symbols can be used to compose complex Tcl statements:

Substitution

`$`	Followed by a variable name; a dollar sign (i.e., `$`) tells Tcl to replace the entire word with the value stored in the variable.
`[]`	Group words as a Tcl statement (i.e., the first within "`[]`" is treated as a *command* word). Tcl statements enclosed within "`[]`" shall be executed before those lying outside "`[]`."

[2] A white space is one or more space and/or tab characters.

Grouping

`" "`	Group words as a string. Substitute variables with their values.
{}	Group words as a string. Treat all special characters as a normal character. *Do not* perform value substitution.

Others

`()`	Indicate an array index or input arguments of mathematical functions.
\	Treat the following special character as a string.
#	Mark the beginning of a comment.
`;`	Mark the end of a Tcl statement.
`EndOfLine`	Mark the end of a Tcl statement.

A.1.3 Variables and Basic Operations

By default, everything in Tcl is string, which can be manipulated using the following five basic operations:

Assignment: `set <name> <value>` (e.g., `set x 0`)
Store `<value>` in the variable whose name is `<name>`.

Deassignment: `unset <name>` (e.g., `unset x`)
Remove the variable `<name>` from the list of known variables.

Input: `gets <channel> [<varname>]` (e.g., `gets stdin x`)
Retrieve a line from what attached to a Tcl `<channel>`. The retrieved value is stored in the variable `<varname>`. The details of Tcl channels will be discussed later in Sect. A.1.7. In case of the standard input, the `<channel>` is "`stdin`." During program debugging, a statement "`gets stdin`" asks the program to wait for an end-of-line character from the user.

The *command* "`gets`" returns the number of characters it reads from the channel. However, if it cannot read from `<channel>` (e.g., reaching the end of the file), it will return "−1."

Output: `puts [-nonewline] [<channel>] <str>` (e.g., `puts $x`)
Output `<str>` to `<channel>`. By default, `<channel>` is the standard output (i.e., screen), and a new line character would be attached to `<str>`. However, the new line character can be suppressed by the option `-nonewline`. Also, the output can be sent to some other place by specifying `<channel>`.

Error reporting: `error <str>` (e.g., `error "Fatal Error!!"`)
Report the error and send `<str>` to the associated application (e.g., monitor) and returning "`TCL_ERROR`".

Example A.1. The following Tcl script, "`convert.tcl`," converts temperatures from Fahrenheit to Celsius. The conversion starts at 0° (Fahrenheit), proceeds with a step of 50° (Fahrenheit), and stops when the temperature exceeds 140° (Fahrenheit). The program prints out the converted temperature in Fahrenheit as long as it does not exceed 140°.

```
# convert.tcl
# Fahrenheit to Celsius Conversion
1  set lower 0
2  set upper 140
3  set step 50
4  set fahr $lower
5  while {$fahr < $upper} {
6     set celsius [expr 5*($fahr - 32)/9]
7     puts -nonewline "Fahrenheit / Celsius :
                       $fahr / $celsius"
8     set fahr [expr $fahr + $step]
9     puts "\t Enter \"y\" to continue ..."
10    flush stdout
11    set cont 0
12    while { $cont != y } {
13        gets stdin cont
14    }
15 }
```

When executing the Tcl scripting file "`convert.tcl`," the following results should be shown on the screen:

```
>>tclsh convert.tcl
Fahrenheit / Celsius : 0 / -18      Enter "y"
                                    to continue ...y
Fahrenheit / Celsius : 50 / 10      Enter "y"
                                    to continue ...y
Fahrenheit / Celsius : 100 / 37.778 Enter "y"
                                    to continue ...y
```

Alternatively, since NS2 is written in Tcl, the following invocation would lead to the same result.

```
>>ns convert.tcl
```

For now, it is not important to understand every line of the above program. We shall repeatedly visit this example, as we explain the syntax of Tcl part by part.

Lines 1–3 show examples of value assignment using "`set`," where `$lower`, `$upper`, `$step` are set to 0, 140, and 50, respectively. Line 4 shows an example use of the substitution character "$," where the value stored in the variable `$lower` is stored in the variable `$fahr`.

Line 7 shows an example use of the output *command* "`puts`" which prints a string on the screen. Here, `$fahr` is enclosed within the quotation marks (i.e., `""`). Since the quotation marks allow value substitution, the output shows the value stored in `$fahr`. Note that the option "`-nonewline`" is used to suppress a new line. The "`puts`" statement on Line 9 concatenates another string to that generated by line 7. Line 9 also shows two example uses of the symbol "\." The first one "`\t`" represents a tab character. The second one, "`\"`" is treated as a regular character "`"`" string, rather than a special character.

Finally Line 13 shows example use of the input *command* "`gets`." Here, Tcl stops and waits for an input from the standard input (i.e., keyboard). The value supplied by the user will be stored in the variable "`cont`." □

Example A.2. Insert the following two lines into the end of the program in Example A.1.

```
16 unset lower
17 puts $lower
```

After executing the Tcl script "`convert.tcl`," the following result should appear on the screen:

```
>>tclsh convert.tcl
Fahrenheit / Celsius : 0 / -18      Enter "y"
                                    to continue ...y
Fahrenheit / Celsius : 50 / 10      Enter "y"
                                    to continue ...y
Fahrenheit / Celsius : 100 / 37.778 Enter "y"
                                    to continue ...y
can't read "lower": no such variable
 while executing
"puts lower"
 (file "convert.tcl" line 17)
```

After being "`unset`," the variable `$lower` becomes unknown to Tcl. Printing this variable would result in a runtime error. □

A.1.4 Logical and Mathematical Operations

The main logical and mathematical operations include

Logical statement: `A <ops> B`

Return `true` or `false` value of the statement, where the list of logical operators `<ops>` is given below.

```
<  (less than)                  =  (equal)
<= (less than or equal)         != (Not Equal)
>  (greater than)               || (OR)
>= (greater than or equal)      && (AND)
!  (negation)
```

Increment: `incr <varName> [<incrVal>]`

Increment the variable `<varName>` by `<incrVal>`, where `<incrVal>` can be zero, positive, or negative. The default value of `<incrVal>` is 1.

Arithmetic operation: `expr A <ops> B`

Interpret "`A <ops> B`" as an arithmetic expression, where the list of *basic* arithmetic operators `<ops>` are given below:

```
+       (addition)          -    (subtraction)
*       (multiplication)    /    (division)
%       (modulo)
```

The advanced arithmetic operators `<ops>` are shown below:

```
&       (Bitwise AND)       |     (Bitwise OR)
<<      (Left shift)        >>    (Right shift)
x?y:z   (If x is nonzero, then y. Otherwise, z)
```

Mathematical function: `expr <fn>([<args>])`

Interpret `<fn>(<args>)` as a mathematical function, where the mathematical functions `<fn>` are defined below:

```
abs(x)      cosh(x)     log(x)      sqrt(x)
acos(x)     double(x)   log10(x)    srand(x)
asin(x)     exp(x)      pow(x,y)    tan(x)
atan(x)     floor(x)    rand(x)     tanh(x)
atan2(x)    fmod(x)     round(x)    wide(x)
ceil(x)     hypot(x,y)  sin(x)
cos(x)      int(x)      sinh(x)
```

Example A.3. Insert the following lines at the end of the program in Example A.1.

```
16 puts "1+2 = [expr 1+2]"
17 puts "Log(10) = [expr log10(10)]"
18 puts "Absolute value of -10 = [expr abs(-10)]"
19 puts -nonewline {{}: }
20 puts {expr $lower}
21 puts -nonewline {"": }
22 puts "expr $lower"
23 puts -nonewline {[]: }
24 puts [expr $lower]
```

After executing the Tcl script "`convert.tcl`," the following result should appear on the screen:

```
>>tclsh convert.tcl
Fahrenheit / Celsius : 0 / -18      Enter "y"
                                    to continue ...y
Fahrenheit / Celsius : 50 / 10      Enter "y"
                                    to continue ...y
Fahrenheit / Celsius : 100 / 37.778 Enter "y"
                                    to continue ...y
Log(10) = 1.0
Absolute value of -10 = 10
1+2 = 3
{}: expr $fahr + $step
"": expr 150 + 50
[]: 200
```

Lines 16–18 show example use of arithmetic operations and mathematical functions. Lines 19–24 show example uses of various brackets in Tcl. Curly braces group words as a string. Quotation marks group words, and substitute variables with their values. Finally, square brackets group words as a Tcl statement. □

A.1.5 Control Structure

Tcl control structure defines how the program proceeds, using the *commands* `if/else/elseif`, `switch`, `for`, `while`, `foreach`, and `break/continue`.

Selection Structure

Selection provides a program with choices, and let the programs decide conditions under which certain choices are taken. Tcl provides two key selection control structures.

While an `if/else/elseif` structure provides a program with a selective choice, the `switch` structure is a convenient replacement a long series of a `if/else/elseif` structure. Their syntaxes are shown below:

```
if {<condition1>} { <actionSet1> }
elseif {<condition2>} { <actionSet2> }
    ...
else { <actionSetN> }
```

```
switch <variable> {
    <pattern_1> { <actionSet1> }
    <pattern_2> { <actionSet2> }
    ...
    default { <actionSetN> }
}
```

A.1.5.1 Repetition Structure

The *commands* "`while`," "`for`," and "`foreach`" are used when actions need to be repeated for several times.

```
while {<condition>} { <actions> }
for {<init>} {<condition>} {<mod>} { <actions> }
foreach {<var>} {<list>} { <actions> }
```

The "`while`" structure repeats the `<actions>` as long as the `<condition>` is `true`.[3] The "`for`" structure begins with an initialization statement `<init>`. After taking `<actions>`, it executes the Tcl statement `<mod>` and checks whether the `<condition>` is `true`. It will repeat the `<actions>` if so, and terminate otherwise. The structure "`foreach`" repeats `<actions>` for every item in the list variable `<list>`. In each repetition, the item is stored in the variable `<var>` and can be used inside the loop.

A.1.5.2 Jumping Structure

The structure "`break`" and "`continue`" are used to stop the iterative flow of the above "`while`," "`for`," and "`foreach`" repetitive structures. Their key difference is that while the structure "`break`" immediately exits the loop, the structure "`continue`" restarts the loop.

[3] See examples of the usage of "`while`" on Lines 5 and 12 in Example A.1.

A.1.6 Modularization

Modularization breaks down a large program into small manageable pieces. Each small portion of a program can be stored in a file or a procedure.

A.1.6.1 Storing a Program Portion into a File and File Sourcing

File Sourcing is an act of loading a Tcl file whose name is `<filename>` into an active Tcl program. The syntax of sourcing is shown below:

```
source <filename>
```

A.1.6.2 Storing a Program Portion into a Procedure

Procedure Declaration

```
proc <name> {<argList>} {
     <actions>
     [return <returned_value>]
}
```

The declaration of a procedure begins with a command word "`proc`," following by the name (e.g., `<name>`), the list of input arguments (e.g., `<argList>`), and the body, respectively. Within the body, the procedure may optionally return a value (e.g., `<returned_value>`) using a "`return`" statement.

The list of input arguments (i.e., `<argList>`) may consist of several input arguments. Each input argument is separated by a white space. Also, Tcl allows programmers to specify a default value for each input argument. The syntax for declaring input arguments is as follows:

```
{ {<n1> <d1>} {<n2> <d2>} ...}
```

where `<n1>` and `<n2>` are names of the first and second input arguments whose default values are `<d1>` and `<d2>`, respectively.

Procedure Invocation

After declaration, a procedure `<name>` can be invoked using the following syntax:

```
<name> <valList>
```

where `<valList>` is the list of the input argument values. If `<valList>` is missing, the default value will be used as input values.

Example A.4. The program in Example A.1 can be modified into a procedure (Lines 1–14) as follows:

```
# convert_proc.tcl
1  proc convert_proc {{lower 0} {upper 140}
   {step 50}} {
2      set fahr $lower
3      while {$fahr < $upper} {
4         set celsius [expr 5*($fahr - 32)/9]
5         puts -nonewline "Fahrenheit / Celsius :
                                    $fahr / $celsius"
6         set fahr [expr $fahr + $step]
7         puts "\t Enter \"y\" to continue ..."
8         flush stdout
9         set cont 0
10        while { $cont != y } {
11            gets stdin cont
12        }
13     }
14 }
```

Here, the keyword "`proc`" marks the beginning of the procedure. It is followed by the procedure name (i.e., `convert_proc`) and three input arguments (i.e., `$lower`, `$upper`, and `$step`) accompanied by their default values (i.e., 0, 140, and 50).

The file "`convert_run.tcl`" (Lines 15 and 16) contains the main program.

```
# convert_run.tcl
15 source convert_proc.tcl
16 convert_proc 0 140 70
```

At the command prompt, we can execute the program as follows:

```
>> tclsh convert_run.tcl
Fahrenheit / Celsius : 0 / -18    Enter "y"
                                  to continue ...y
Fahrenheit / Celsius : 70 / 21    Enter "y"
                                  to continue ...y
```

Line 15 sources the Tcl file "`convert_proc.tcl`," which contains the procedure "`convert_proc`," into the active program. Line 16 executes the procedure "`convert_proc`" feeding 0, 140, and 70 as the input arguments. These three values are stored in the variables `$lower`, `$upper`, and `$step`, respectively. □

A.1.6.3 Global and Local Variables

Like in other programming languages, variables in Tcl are local by default. That is, they are understood within a certain boundary. For example, the procedure in Example A.4 understands variables `$lower`, `$upper`, `$step`, and `$fahr`, but it does not understand the variables defined outside the procedure. The syntax to make a variable `$varname` global is as follows:

```
global <varname>
```

Example A.5. Modify Example A.4 as shown below:

```
# convert_proc.tcl
proc convert_proc { { lower 0 } { upper 140 }
  { step 50 } } {
    global ext_var
    ...
    puts "The variable ext_var is $ext_var"
}
```

Here, we add two lines to the procedure "`convert_proc`." The first one is to tell procedure that the variable `$ext_var` is defined in the global scope (i.e., in the main program). The second one shows the value of the variable `$ext_var` on the screen.

```
# convert_run.tcl
source convert_proc.tcl
set ext_var 999
convert_proc 0 140 70
```

In the main program, the variable `$ext_var` is set to 999 before the invocation of the procedure. Therefore, the result on the screen would be as shown below:

```
>> tclsh convert_run.tcl
Fahrenheit / Celsius : 0 / -18    Enter "y"
                                  to continue ...y
Fahrenheit / Celsius : 70 / 21    Enter "y"
                                  to continue ...y
The variable ext_var is 999
```

where the last line is shown in addition to the result from Example A.4. □

A.1.7 Advanced Input/Output: Files and Channels

Tcl uses a so-called Tcl channel to receive an input using a *command* "`gets`" and to send an output using a *command* "`puts`."

A.1.7.1 Tcl Channels

A Tcl channel refers to an interface which interacts with the outside world. Two main types of Tcl channels are standard reading/writing channels and file channels. The former are classified into "`stdin`" for reading, "`stdout`" for writing, and "`stderr`" for error reporting, while the latter needs to be attached to a file before it is usable.

There are three commands related to Tcl channels:

Open the channel: `open <fileName> [<opt>]`
(e.g., `open "in.txt" "r"`)
Create and connect a channel to a file whose name is `<fileName>`, where the option `<opt>` can be `"r"` (reading), `"w"` (writing), or `"a"` (appending). This command returns a handle (i.e., a reference) to the created channel.

Close the channel: `close $<filech>` (e.g., `close $ch`)
Close the channel `$<ch>`.

Flush the buffer: `flush $<filech>` (e.g., `flush $ch`)
Flush the internal buffer associated with the channel `$<filech>`.

The Tcl output process is rather subtle. Tcl does not immediately writes to the channel. Instead, it holds the output received from the *command* "`puts`" in its buffer, and releases the output under one of the three following conditions: the buffer is full, the channel is closed, or an explicit flushing command is executed.

Flushing buffer forces Tcl to release content in its buffer without closing the channel. Example use of the *command* "`flush`" is shown in Line 10 of Example A.1, where we force the buffer to releases its string before the statement "`gets stdin cont`."

Example A.6. Consider the following Tcl program, which copies the input file "`input.txt`" to the output file "`output.txt`" line by line. In addition, the line number is prefixed at the beginning of each line.

```
# file.tcl
puts "Press any key to begin prefixing file..."
gets stdin
set ch_in [open "input.txt" "r"]
set ch_out [open "output.txt" "a"]
set line_no 0
while {[gets $ch_in line] >= 0} {
      puts $ch_out "[incr line_no] $line"
}
close $ch_in
close $ch_out
```

From Sects. A.1.3 and A.1.5, the statement "`while {[gets $ch_in line] >= 0}`" reads a line from the file until reaching the end of the file where "−1" is returned. Readers are encouraged to provide their own input file, run the program, and observe the result. □

A.1.8 Data Types

Tcl allows variable usage without declaration. Therefore, its data type is limited to string, list, and associative array.

A.1.8.1 String

By default, everything in Tcl is string. The basic string manipulation was shown in Sect. A.1.3. Additionally, Tcl provides three following string manipulation commands:

`string <cmd> [<args>]`

Main manipulation command, where `<cmd>` is a sub-command, and `<args>` are the input arguments of the sub-command.

- `string length <str>`: Return the length of the string `<str>`.
- `string first <txt> <str>`: Return the first location within the string `<str>` which contains `<txt>`, or "-1" if the string does not contain `<txt>`.
- `string last <txt> <str>`: Similar to the "`string first`" *command*, but look for the last occurrence.
- `string match [-nocase] <pattern> <str>`: Determine whether the string `<str>` contains `<patterm>`. Return 1 if so and 0 otherwise. See the details of pattern matching in Sect. A.3.4.
- `string range <str> <fst> <lst>`: Return the `<fst>`th character to the `<lst>`th character of the string `<str>`.

`format <fmt> [<d1> <d2> ...]` (e.g., `format %5.3f [expr 2.0/3]`)

Create a string using the C++ `printf` format. The format is defined in `<fmt>`, and the values are defined in `<d1>`, `<d2>`, and so on.

`scan <str> <fmt> [<v1> <v2> ...]` (e.g., `scan "My NS2" {%s %s} v1 v2`)

This command is the reverse of the *command* "`format`." It scans the input string `<str>` according to the given format `<fmt>`, and extracts and stores the matched values in the corresponding variables `<v1>`, `<v2>`, and so on.

A.1.8.2 List

A list contains a series of strings or lists. It has the following properties.

- Items are group into a list using curly braces (i.e., {. . .}).
- Each item is separated by a white space.
- Each item is indexed with 0, 1, 2, and so on. The last index can be referred to by a keyword "`end`."
- An item can be a list.
- A list can be manipulated using the following commands:

Creation (method 1): {`<i1> [<i2> ...]`}
Creation (method 2): {`list <i1> [<i2> ...]`}
(e.g., `set mylist {1 2 3}`, `set mylist [list 1 2 3]`)
Construct a list whose items are `<i1>`, `<i2>`, and so on.

Appending: {`lappend <l> <i1> [<i2> ...]`}
(e.g., `lappend mylist 4 5`)
Append the items `<i1>`, `<i2>`, and so on to the end of the list `<l>`.

String to list: {`split <str> <char>`} (e.g., `set mylist [split "1&2&3" &]`)
Separate a string `<str>` using a delimiting character `<char>`. Then, return a list whose items store the extracted phrases.

List to string: {`join <l> <char>`} (e.g., `join $mylist &`)
Convert a list `<l>` to a string, placing a character `<char>` between each pair of items.

Length: {`llength <l> `} (e.g., `llength $mylist`)
Return the number of items in the list `<l>`.

Get an item: {`lindex <l> <index> `} (e.g., `lindex $mylist 2`)
Return the `<index>`th item of the list.
Get a sublist: {`lrange <l> <fi> <li>`}
(e.g., `lrange $mylist 0 end`)
Return a list containing all items between the `<fi>`th item and the `<li>`th item of the list.

A.1.8.3 Associative Array

An associative array is a two-dimensional data structure. The first dimension is the index, while the second dimension is the associated value. In Tcl, the first dimension

is a string, while the second one can be a string, a list, or an array. An array can be manipulated as follows:

Set the value: `set $<aName>(<index>) <value>`
(e.g., `set $myarray(Apple) 2`)
Create an entry for an array whose name is `<aName>`. The index and value for this entry are `<index>` and `<value>`, respectively.

List to array: `array set <aName> $<listname>`
(e.g., `array set myarray {Apple 2 Banana 1}`)
Convert a list variable `$<listname>` to an array whose name is `<aName>`. The items in the list are used alternately as indexes and values, respectively, of the array.

Array to list: `array get <aName>` (e.g., `array get myarray`)
Return a list variable containing concatenated pairs of indexes and values of the array whose name is `<aName>`.

Get index list: `array name <aName> *` (e.g., `array name myarray *`)
Return a list variable whose items contain the indexes of the array.

Note that when creating an array, a list can be replaced with all its items embraced within curly braces.

There are two key differences between lists and associative arrays. First, a list is a collection of *ordered* items. The indexes of a list are 0, 1, and so on. Entries in an associated array have no order. There is no such thing as the *next entry* in an array.[4] Each entry can be accessed directly.

Another difference is that an array cannot be directly outputted using "`puts`." It must first be converted into a list, which can be outputted using "`puts`."

A.2 Objected-Oriented Tcl Programming

OTcl is an object-oriented version of Tcl, just like C++ is an object-oriented version of C [31]. All the basic architecture and syntax of Tcl are carried over to OTcl. By incorporating the Object Oriented Programming (OOP) concept, OTcl has additional benefits such as scalability, modularization, and protection from unintentional access.

[4]The above example does not imply that the entry `$myarray(Banana)` is the entry next to `$myarray(Apple)`.

A.2.1 OTcl Language Structure

In OOP, all variables and methods must be associated with classes. They cannot be invoked as standalone variables or procedures. OTcl statements always follow one of the following three patterns:

1. Class declaration: `Class <className> [<options>]`
2. Class-associated method invocation:
   `<className> <instproc> [<options>]`
3. Object-associated method invocation:
   `$<objName> <instproc> [<options>]`

Like C++, OTcl does not allow direction access to class variables. OTcl class variables must be accessed via the instproc "`set`."

The first statement is the class declaration statement. The latter two are the ones which invoke class methods `<instproc>`, which can be associated with either class `<className>` or object `$<obj_name>`.

Like in Tcl, OTcl is a positional language. The first position can be either a keyword "`Class`," a class name `<className>`, or a variable which hold the object `$<obj_name>`. The second position is either a class name (i.e., `<className>`) or the name of class method (i.e., `<instproc>`). Finally, the last position is the optional input arguments.

A.2.2 Classes and Inheritance

A class is a representation of a group of objects which share similar properties including behavior(s) or attribute(s). These properties can be passed down to child classes.[5] In this respect, the donor and the receiver of the properties are called a superclass (base or parent class) and a subclass (derived or child class), respectively. Apart from inheriting properties from a superclass, a class defines its own properties to make itself more specific.

In OTcl the syntaxes for class declaration are shown below:

Class declaration: `Class <className> [-superclass <superCN>]`
(e.g., `class Mobile -superclass Node`)

Declare a class whose name is `<className>`, and optionally specify the class `<superCN>` as the superclass of the class `<className>`.

[5] In OTcl, the top-level class is class `Object`, which provides basic variables and functions, from which every user-defined class inherits.

Specifying the superclass: `<className> superclass <superCN>`
(e.g., `Mobile superclass Node`)
Specify the class `<superCN>` as the superclass of class `<className>`.

A.2.3 Objects and Object Construction Process

The key distinction between classes and objects is as follows. A class is a passive definition of similar objects defined before the runtime. An object, on the other hand, is an active entity created from a template defined in a class at runtime. Its values change as the program runs.

A.2.3.1 Object Construction Methods

An OTcl object can be created using the following two methods:

Object construction (Method 1) `<className> <name> [<args>]`
Object construction (Method 2) `<className> create $<name> [<args>]`
(e.g., `Node n1`, `Node create $n1`)
Construct (i.e., instantiate) an object from class `<className>` and store the object in the variable whose name is `<name>`, optionally feeding `<args>` as input arguments to the constructor.

The first method is the standard object instantiation method.[6] However, the second method 2 is preferable since it allocates space and invokes the constructor of all the classes along the inheritance tree.

A.2.3.2 Object Construction Process

When constructing an object using Method 2, the instproc "`create`" invokes the following two instprocs:

1. Instproc "`alloc`" allocates memory space to store the object.
2. Instproc "`init`" initializes the object (e.g., its class variable).

[6] When using Method 2, the variable `$<name>` must exist in the list of known variable before the construction.

In most cases, the instproc "alloc" is internal to OTcl, while the instproc "init" is provided by the programmer. Throughout this book, we shall refer to the instproc "init" as the OTcl constructor.

In general, OOP languages construct an object by invoking all the constructors along the entire inheritance tree. The invocation is not automatically carried out in OTcl. A programmer needs to explicitly invoke a constructor of all classes along the inheritance tree via the instproc "next."

Defined in the OTcl top-level class "Object," the instproc "next" executes the same instproc defined in the superclass. It is usually seen in the form

```
$<object> next [<args>]
```

Suppose the above statement is located in an instproc <instprocName> of class <className>. At the execution, this statement would invoke the instproc <instprocName> of the superclass of class <className>.

By inserting the following statement into the constructor (i.e., the instproc "init")

```
$self next [<args>]
```

the constructor of the superclass will be executed.

A.2.4 Member Variables and Functions

A class contains two main types of members: variables and methods. A variable defines an attribute and/or holds the current status. A method, on the other hand, defines the actions that an object can do. By convention, OTcl calls a member variable an "instance variable" or an "instvar," and it calls a member method an "instance procedure" or an "instproc."

A.2.4.1 Instance Variables

Instance variables can be manipulated as follows:

Declaration (tied to class): <className> instvar <n1> [<n2> ...]
Declaration (tied to object): $self instvar <n1> [<n2> ...]
(e.g., Node instvar state, $self instvar state)

Declare variables whose names are <n1>, <n2>, and so on, instvars of the class <className> or of the object $self – a special variable which holds the object itself (i.e., $self corresponds to the pointer "this" in C++).

Access (tied to class): `<className> set <instvarName> [<value>]`
Access (tied to object): `$<varName> set <instvarName> [<value>]`
(e.g., `Node set state 1`, `$n set state 1`)

If `<value>` exists, it will be stored in the instvar `$<instvarName>` of class `<className>` or of the object `$<varName>`. Otherwise, the above statements will return the value stored in the instvar `$<instvarName>`.

As an interpreted language, Tcl requires no formal declaration of member variables. The only requirements are that the variables must be declared before its use, and that the declaration statement must be within an instance procedure.

If an instvar is tied to a class, all objects instantiated from this class will automatically contain the instvar. On the other hand, if the instvar is tied to an object, it will be a member of that object only. Other objects from the same class will not contain this member variable. Therefore, from within every instance procedure, it is advisable to declare all necessary instvars as the first OTcl statement.

A.2.4.2 Instance Procedures

OTcl declares and invokes instance procedures as follows:

Declaration: `<className> instproc <instprocName> {[<args>]} {<body>}`
(e.g., `MyNode instproc recv {pkt} {...}`)

Declare `<instprocName>` as an instproc of class `<className>`. The optional input arguments and the body of the instproc are specified in `<args>` and in `<body>` respectively.

Invocation: `$<objName> <instprocName> [<values>]`
(e.g., `$n recv p`)

Execute the instproc `<instprocName>` which is associated with the class of the object `$<objName>`, where `<values>` are optionally fed as input arguments.

At the invocation, OTcl proceeds as follows:

1. Look for the instproc `<instprocName>` in the class corresponding to `$<objName>`. Execute and terminate if found. Otherwise, go the next step.
2. Look for the instproc "`unknown`" in the same class. Execute and terminate if found. Otherwise, go the next step.
3. Move to the superclass and go back to step 1. If the current class does not have any superclass, report the error.

A.2.4.3 Non-Inheritable Members

Both instvars and instprocs are inheritable. Once defined for a class, these members would be available for all objects instantiated from that class and its subclasses.

OTcl also has another type of members specific to an object only. Once defined for a certain object, these members would not appear in other objects instantiated from the same class or its subclasses. These members are just plain (i.e., local) *variables* and *procedures*.

A variable is declared using the above method to access variable, i.e., using "`set`." From the above example, the OTcl statement "`$n set state 1`" declares a variable "`state`" to be a member variable of the object `$n`. Other variables instantiated from the same class as `$n` would not contain the variable "`state`."

A procedure is declared using the following syntax:

```
<objName> proc <procName> {[<args>]} {<body>}
```

which is similar to the instproc declaration. Once declared, procedures can be invoked as if they are instprocs.

A.2.5 A List of Useful Instance Procedures

Here is a list of useful instprocs defined in the top-level class `Object`. Again, these instprocs are available to all classes derived from class `Object`.

`create $<name> [<args>]`:
: Create and store an object in the variable `$<name>`.

`destroy $<name>`:
: Destroy the object stored in the variable `$<name>`.

`superclass <name>`:
: Specify class `<name>` as its superclass.

`instvar $<name>`:
: Declare `$<name>` as its instvar.

`instproc <name> [<args>]`:
: Declare `<name>` as its instproc.

`alloc $<name>`:
: Allocate memory for an object `$<name>`.

`init [<args>]` (i.e., the constructor):
: Initialize the created object.

`next [<args>]`:
: Invoke the instproc with the same name, defined in the superclass.

`info <option>`:
: Return related information as specified in `<option>`. Note that the options tied to a class are different from those tied to an object (see Tables A.1 and A.2).

Table A.1 Options of the instproc "info" associated with classes

Options	Functions
superclass	Return the superclass of the current class.
subclass	Return the list of all the subclasses.
heritage	Return the list of the inheritance tree.
instances	Return the list of instances of the class.
instprocs	Return the list of instprocs defined on the class.
instcommands	Return the list of instprocs and OTcl commands defined on the class.
instargs <proc>	Return the list of arguments of the instproc <proc> defined on the class.
instbody <proc>	Return the body of the instproc <proc> defined on the class.
instdefault <proc> <arg> <var>	Return 1 if the default value of the argument <arg> of the instproc <proc> is <var>, and return 0 otherwise.

Table A.2 Options of the "info" instproc associated with objects

Options	Functions
class	Return the class of the object.
procs	Return the list of all associated procedures.
commands	Return the list of associated procedures and OTcl commands.
vars	Return the list of variables defined on the object.
args <proc>	Return the list of arguments of the procedure <proc> defined on the object.
body <proc>	Return the body of the procedure <proc> defined on the object.
default <proc> <arg> <var>	Return 1 if the default value of the argument <arg> of the procedure <proc> is <var>, and returns 0 otherwise.

Example A.7. Consider a general network node. When equipped with mobility, this node becomes a mobile node. Declarations of a class "Node" and its subclass "Mobile" are shown in Lines 1 and 2 below. This declaration allows class "Mobile" to inherit capabilities of class "Node" (e.g., receiving packets) and to have more capabilities (e.g., moving) of its own.

```
# node.tcl
1 Class Node
2 Class Mobile -superclass Node
3  Node instproc recv {pkt} {
4       $self instvar state
5       set state 1
6       $self process-pkt $pkt
7  }
8  Mobile instproc move {x y} {
9       $self instvar location
10      set location[0] $x
11      set location[1] $y
```

```
12 }

13 Node instproc init {} {
14     $self instvar state
15     set state 0
16 }
17 Mobile instproc init {} {
18     $self next
19     $self instvar location
20     set location[0] 0
21     set location[1] 0
22 }

23 set n _1
24 Node create $n
25 puts "The instance of class Node is
                                    [Node info instances]"
26 puts "The class of $n is [$n info class]"
```

By executing the file "`node.tcl`," the following result should appear on the screen.

```
>>ns node.tcl
The instance of class Node is _1
The class of _1 is Node
```

The key points of this example are the constructor and the use of the instproc "`info`." Lines 13–22 show examples of constructor. At the construction, class "`Node`" sets its instvar "`state`" to 0 (i.e., inactive). Class "`Mobile`" first invokes the constructor of class "`Node`" in Line 18 using the instproc "`next`." Then, Lines 20 and 21 set the location of the mobile node to `(0,0)`.

Line 23 shows an example which instantiates an object from class "`Node`" and stores the object in the variable `$n`. Lines 24 and 25 show two example uses of the instproc "`info`" which is tied to the class "`Node`" and the object `$n`, respectively. □

A.3 AWK Programming

AWK is a programming language designed to process text files [32]. AWK refers to each line in a file as a *record*. Each record consists of *field*s, each of which is separated by a field separator.[7] Generally, AWK reads data from a file consisting of fields of records, processes those fields, and outputs the results to a file as a formatted report.

[7]The default field separator is a white space.

To process an input file, AWK follows an instruction specified in an *AWK script*. An AWK script can be specified at the command prompt or in a file. While the strength of the former is the simplicity (in invocation), that of the latter is the functionality. In the latter, the programming functionalities such as variables, loop, and conditions can be included into an AWK script to perform the desired actions. In what follows we give a brief introduction to the AWK language. The details of AWK programming can be found in [33].

A.3.1 Program Invocation

AWK can be invoked from a command prompt in two ways based on the following syntax:

```
>>awk [ -F<ch> ] {<pgm>} [ <vars> ] [ <data_file> ]
>>awk [ -F<ch> ] { -f <pgm_file> } [ <vars> ]
      [ <data_file> ]
```

where <ch> is a field separator, <pgm> is an AWK script, <pgm_file> is a file containing an AWK script (i.e., an AWK file), <vars> is a list of variables used in an AWK script, and <data_file> is an input text file.

By default, AWK uses a white space (i.e., one or more spaces or tabs) as a field separator. However, if the option "-F" is present, AWK will use <ch> as a field separator.[8] The upper invocation takes an AWK script <pgm> as an input argument, while the lower one takes an AWK file <pgm_file> as an input argument. In both cases, variables <vars> and input text file <data_file> can be optionally provided. If an input text file is not provided, AWK will wait for input arguments from the standard input (e.g., keyboard) line by line.

Example A.8. Let an input text file "infile.txt" be defined below. We shall use this input file repeatedly in this section.

```
#infile.txt
Rcv 0.162 FromNode 2 ToNode 3 cbr PktSize= 500  UID= 3
EnQ 0.164 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 8
DeQ 0.164 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 8
Rcv 0.170 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 7
EnQ 0.170 FromNode 2 ToNode 3 cbr PktSize= 1000 UID= 7
DeQ 0.170 FromNode 2 ToNode 3 cbr PktSize= 1000 UID= 7
Rcv 0.171 FromNode 2 ToNode 3 cbr PktSize= 1000 UID= 4
EnQ 0.172 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 9
DeQ 0.172 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 9
Rcv 0.178 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 8
```

[8] For example, "awk -F:" uses a colon ":" as a field separator.

```
EnQ 0.178 FromNode 2 ToNode 3 cbr PktSize= 1000 UID= 8
DeQ 0.178 FromNode 2 ToNode 3 cbr PktSize= 1000 UID= 8
```

Note that in AWK, "#" marks the beginning of a comment line.

At the command prompt, we may run an AWK script to show the lines which contain "EnQ" as follows:

```
>>awk /EnQ/ infile.txt
EnQ 0.164 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 8
EnQ 0.170 FromNode 2 ToNode 3 cbr PktSize= 1000 UID= 7
EnQ 0.172 FromNode 1 ToNode 2 cbr PktSize= 1000 UID= 9
EnQ 0.178 FromNode 2 ToNode 3 cbr PktSize= 1000 UID= 8
```

Here, the <pgm> is specified as /EnQ/ and the <data_file> is specifies as infile.txt. An AWK script /EnQ/ looks for lines which contain a string "EnQ" and display the lines on the screen. □

A.3.2 An AWK Script

An AWK script contains an instruction for what AWK will perform. It asks AWK to look for a pattern in a record, and performs actions on a matched pattern. The syntax of an AWK script is as follows:

```
<pattern> {<actions>}
```

A <pattern> could be a logical expression or a regular expression.[9] The <actions> specifies actions for the matched pattern. Each action in the curly braces is separated by a semi-colon (";").

A.3.3 AWK Programming Structure

The general form of an AWK program is shown below:

```
BEGIN {<initialization>}
<pattern1> {<actionSet1>}
<pattern2> {<actionSet2>}
.
.
.
END {<final finalActionSet>}
```

[9]While a logical expression is usually implemented by an if statement, a regular expression returns true when finding a matched pattern. The formal definition of a regular expression can be found in [34].

Table A.3 Special characters used in regular expressions

Character	Description
`/.../`	Contain a regular expression (e.g., `/text/`, `//` matches every line.)
`^`	Match the beginning of a record only (e.g., `/^text/`)
`$`	Match the end of a record only (e.g., `/text$/`)
`[]`	Match any character inside (e.g., `[text]`)
`[^<str>]`	Match any character except `<str>` (e.g., `[^t^e^x^t]`)
`[<a>-<b>]`	Match any character between `<a>` and `<b>`
`.`	Match any character (e.g., `/tex./`)
`*`	Match zero or more character in front of it (e.g., `/tex*/`)
`.*`	Match any string of characters
`?`	Match zero or one regular expression in front of it (e.g., `/[a-z]?/`)
`+`	Match one or more regular expression in front of it (e.g., `/[a-z]+/`)
`\w`	A word character (i.e., an alphanumeric or a "_") which is equivalent to `[A-Za-z0-9_]`
`\s,\S,\t,\n`	A white space, a non-white-space, a tab character, and a newline character

which consists of three main steps:

1. Before file processing, run `<initialization>`.
2. For each line of the input file, run the action sets which match with the associated patterns.
3. After all the lines are processed, run `<finalActionSet>`.

A.3.4 Pattern Matching

The first part of an AWK script is a pattern as specified in `<pattern>`. The pattern can be a logical or a regular expression. If this part evaluates to "`true`," the corresponding `<actions>` will be taken.

The syntax for logical expression is the same as that used in Tcl (see Sect. A.1.4 for logical operators). Although slightly more complicated, regular expression provides a more concise and flexible means to represent a text of interest. Syntactically, a regular expression is enclosed within a pair of forward slashes ("/," e.g., `/EnQ/`). Table A.3 shows frequently used regular expression symbols.

A.3.5 Basic Actions

The second part of an AWK script is to take `<actions>` for a matching `<pattern>`. This section explains the basic actions in AWK.

Table A.4 Built-in variables

Variables	Descriptions
`$0`	The current record
`$1,$2,...`	The 1st, 2nd,... field of the record
`FILENAME`	Name of the input text file
`FS`	(Input) field separator (a white space by default)
`RS`	(Input) record separator (a newline by default)
`NF`	Number of fields in a current record
`NR`	Total number of records
`OFMT`	Format for numeric output (`%6g` be default)
`OFS`	Output field separator (a space by default)
`ORS`	Output record separator (a newline by default)

A.3.5.1 Arithmetic Operation

Basic arithmetic operations include addition, subtraction, multiplication, division, and modulus. The arithmetic operators are the same as *basic* Tcl operators (See Sect. A.1.4).

A.3.5.2 Variables

As an interpreter, AWK does not need to declare data type for variables. It can simply assign a value to a variable using an assignment operator ("="). To avoid ambiguity, AWK differentiates a variable from a string by quotation marks (`""`). For example, "`var`" is a variable while "`"var"`" is a string (see Example A.9).[10]

AWK also supports one-dimensional arrays. Identified by a square bracket (`[]`), indexes of an array can be both numeric (i.e., a regular array) or string (i.e., an associative array). Example of arrays are `node[1]`, `node[2]`, and `link["1:2"]`.

Apart from the above user-defined variables, AWK also provides several useful built-in variables as shown in Table A.4.

A.3.5.3 Outputs

AWK outputs a variable or a string to a screen using either "`print`" or "`printf`," whose syntax are as follows:

```
print <item1> <item2> ...
printf(<format>,<item1>,<item2>,...)
```

[10]Unlike Tcl, AWK retrieves the value stored in a variable without a prefix (not like "$" in Tcl).

where <item1>, <item2>, and so on can be either variables or strings, and <format> is the format of the output. Using "print," a string needs to be enclosed within a quotation mark (""), while a variable could be indicated as it is.

Example A.9. Define an AWK file "myscript.awk" as shown below.

```
# myscript.awk
BEGIN{}
/EnQ/ {var = 10; print "No Quotation: " var;}
/DeQ/ {var = 10; print "In Quotation: " "var";}
END{}
```

Run this script for the input text file "infile.txt" defined in Example A.8. The following result should appear on the screen.

```
>>awk -f myscript.awk infile.txt
No Quotation: 10
In Quotation: var
No Quotation: 10
In Quotation: var
No Quotation: 10
In Quotation: var
No Quotation: 10
In Quotation: var
```

The above AWK script prints out two versions of the variable "var." The upper line prints out the value (i.e., 10) stored in the variable "var." In the lower line, the variable "var" is enclosed within a quotation mark. Therefore, the string "var" will be printed instead. □

The *command* "printf" provides more printing functionality. It is very similar to function "printf" in C++. In particular, it specifies the printing format as the first input argument. The subsequent arguments simply provide the values for the place-holders in the first input argument. The readers are encouraged to find the details of the printing format in any C++ book (e.g., [16]) or in [33].

A.3.6 Redirection and Output to Files

AWK does not have a direct command to export the results to files. To do so, the following UNIX redirection[11] *command*s can be used:

[11]Examples of Unix output redirection are "awk /EnQ/ infile.txt > output.txt," "ls > output.txt," and "pwd | ls."

`>`	Redirect the standard output
`>>`	Append the standard output
`|`	Redirect the standard output to a command

Note that while "`>`" redirects the output to a new file, "`>>`" appends the output to an existing file. If the file exists, "`>`" will delete and recreate the file. The *command* "`>`," on the other hand, appends the output to the file without destroying the existing file.

A.3.7 Control Structure

AWK supports three major types of selection and repetition control structures: "`if/else`," "`while`," and "`for`." The syntaxes of these control structures are as follows:

```
if(<condition>) <actionSet1> [else <actionSet2>]
while(<condition>) <actions>
for(<initialization>;<condition>;<end-of-loop
    -actions>) <actions>
```

Again, when the actions contain more than one statement, each statement must be terminated by a semi-colon (i.e., "`;`"), and all of the statements must be embraced within curly braces.

AWK also contains four jumping control structures:

break	(Exit the loop)
continue	(Restart the loop)
next	(Process the next record)
exit	(Exit the program)

A.4 Exercises

1. Repeat Example A.1 but write all the output to file "`convert.out`" using the Tcl file channel.

2. From Example A.5, do the following items one by one. Then run the program, and observe and discuss the results.

   a. Remove the variable `ext_var`,
   b. Make the variable `ext_var` a local variable of the procedure "`convert_proc`,"
   c. Make the variable `ext_var` a local variable of the main program, and
   d. Remove the statement `flush stdout`.

3. Modify Example A.6 to take input arguments as input and output files. If no input arguments are given, use "`input.txt`" and "`output.txt`" as input and output files, respectively.

4. From Example A.7, declare `$bufferSize` as

   a. An instvar of class "`Node`"
   b. A variable of the object `$n`

   For each case, write an OTcl program to verify that `$bufferSize` is declared properly. Hint: Use the instproc "`info`."

5. Write OTcl codes which make use of the above options for instproc `info` in Tables A.1 and A.2. Create your own examples.

6. Write an input string which matches with *each* of the following regular expressions. The input string should not match with other regular expressions.

```
/^Node          /Node$          /[Nn]ode
/Node.          /Node*          /Nod[Ee]?       /Nod[Ee]+
```

7. Based on the input file in Example A.8, develop an AWK script to show

   a. Total number of "`EnQ`" events,
   b. The number of packets that Node 3 receives, and
   c. Total number of bytes that Node 3 receives.

8. Repeat Example A.9, but print the result in a file "`outfile .txt`." Show the difference when using "`>`" and "`>>`."

B
A Review of the Polymorphism Concept in OOP

B.1 Fundamentals of Polymorphism

As one of the main OOP concepts, polymorphism refers to the ability to invoke the same function with different implementation under different context. This concept should be simple to understand, since it occurs in our daily life.

Example B.1. Consider receptionists and how they greet customers. Friendly, moody, and rude receptionists greet customers by saying "Good morning. How can I help you today?", "What do you want?", and "What do you want? I'm busy. Come back later!!", respectively. We design a class hierarchy for receptionists as shown in Fig. B.1. The base class of the hierarchy is class `Receptionist`. Based on the personality, we derive classes `FriendlyReceptionist` and `MoodyReceptionist` directly from class `Receptionist`. Also, we derive another class `RudeReceptionist` from class `Moody Receptionist`. The C++ code which represents these four classes is given below:

```
    //receptionist.cc
1   #include "iostream.h"
2   class Receptionist {
3       public:
4       void greet() {cout<<"Say:\n";};
5   };

6   class FriendlyReceptionist : public Receptionist {
7       public:
8       void greet(){
9         cout<<"Say: Good morning. How can I help
          you today?\n"
10      }
11  };
```

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_17, © Springer Science+Business Media, LLC 2012

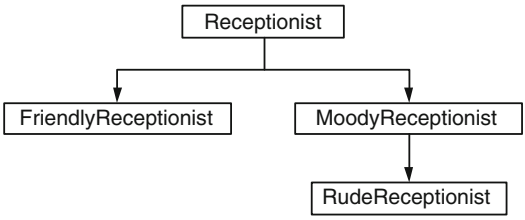

Fig. B.1 A polymorphism example: Receptionist class hierarchy

```
12 class MoodyReceptionist : public Receptionist {
13     public:
14     void greet() { cout<<"Say: What do you
       want?\n"; };
15 };

16 class RudeReceptionist : public MoodyReceptionist {
17     public:
18     void greet(){
19         MoodyReceptionist::greet();
20         cout<<"Say: I'm busy. Come back later.\n";
21     };
22 };

23 main() {
24     FriendlyReceptionist f_obj;
25     MoodyReceptionist m_obj;
26     RudeReceptionist r_obj;
27     cout<<"\n------------ Friendly Receptionist
       ---\n";
28     f_obj.greet();
29    cout<<"\n------------ Moody Receptionist
      -------\n";
30     m_obj.greet();
31     cout<<"\n------------ Rude Receptionist
       -------\n";
32     r_obj.greet();
33     cout<<"-----------------------------------
       ----\n";
34 }
```

Function `main()` instantiates three receptionist objects. Objects `f_obj`, `m_obj`, and `r_obj` are of classes `FriendlyReceptionist`, `Moody Receptionist`, and `RudeReceptionist`, respectively (Lines 24–26). They greet a customer in Lines 28, 30, and 32 by invoking function `greet()` in

Lines 8–10, 14, and 18–21, respectively.[1] By running `receptionist`, the following results should appear on the screen.

```
>>./receptionist
------------ Friendly Receptionist ---------
Say: Good morning. How can I help you today?

------------ Moody Receptionist ---------
Say: What do you want?

------------ Rude Receptionist ---------
Say: What do you want?
Say: I'm busy. Come back later!!
-----------------------------------------
```

Example B.2. Remove Line 14 in Example B.1 and run "`./receptionist`" again. The following result should appear on the screen:

```
>>./receptionist
------------ Friendly Receptionist ---------
Say: Good morning. How can I help you today?

------------ Moody Receptionist ---------
Say:

------------ Rude Receptionist ---------
Say:
Say: I'm busy. Come back later!!
-----------------------------------------
```

Since class `MoodyReceptionist` does not define function `greet` (Line 14 is removed), it uses the function `greet()` inherited from class `Receptionist` (i.e., printing "`Say:`" on the screen).

Examples B.1 and B.2 demonstrate the concepts of polymorphism through receptionists and how they greet customers. When invoking the same function (e.g., `greet()`), three objects of different classes act differently (e.g., by saying differently). Example B.1 shows a basic polymorphism mechanism, where each class has its own implementation. Example B.2 shows that it is also possible not to override function `greet()`.[2]

[1] Note that in Line 19 function `greet()` of class `MoodyReceptionist` is invoked in the scope of class `RudeReceptionist` using "`::`."

[2] For example, class `MoodyReceptionist` inherits function `greet()` from class `Receptionist`.

B.2 Type Casting and Function Ambiguity

In most cases, polymorphism is fairly straightforward. A derived class may inherit or override functions from the base class. When polymorphism involves type casting, the mechanism in Examples B.1 and B.2 may lead to different results. To see how, consider the following examples.

Example B.3. Replace function "`main`" in Example B.1 with the following:

```
1 main() {
2 FriendlyReceptionist *f_pt;
3 MoodyReceptionist *m_pt;
4 RudeReceptionist *r_pt;
5 f_pt = new FriendlyReceptionist();
6 m_pt = new MoodyReceptionist();
7 r_pt = new RudeReceptionist();

8 cout<<"\n------------ Friendly Receptionist ----\n";
9 f_pt->greet();
10 cout<<"\n------------ Moody Receptionist ----\n";
11 m_pt->greet();
12 cout<<"\n------------ Rude Receptionist ----\n";
13 r_pt->greet();
14 cout<<"-----------------------------------
   -----\n";
15 }
```

With the above code, the result for running `./receptionist` would be the same as that in Example B.1. The major difference in the above `main()` function is the use of pointers (Lines 2–4), instead of regular objects (in Example B.1). □

Example B.4. In Example B.3, replace Lines 3 and 4 with the following:

```
MoodyReceptionist *m_pt,*r_pt;
```

This is an example of ambiguity caused by type casting. The pointer `r_pt` is declared as a pointer to a `MoodyReceptionist` object; however, the statement "`new RudeReceptionist()`" creates an object of type `Rude Receptionist`. When invoking a function (e.g., `greet()`), the key question is which class should function `greet()` be associated with: `MoodyReceptionist` (i.e., the declaration class) or `RudeReceptionist` (i.e., the construction class)? To answer this question, we can simply run "`./receptionist`," and obtain the following results:

```
>>./receptionist
------------ Friendly Receptionist ---------
```

```
Say: Good morning. How can I help you today?

------------ Moody Receptionist ---------
Say: What do you want?

------------ Rude Receptionist -------------
Say: What do you want?
-----------------------------------------
```

From the above result, the answer is the former one: `MoodyReceptionist`.

Consider the statement "`r_pt = new RudeReceptionist`." The latter part, "`new RudeReceptionist`," allocates memory space to an object of class `RudeReceptionist`, and returns a pointer to the created object. The former part "`r_pt = `" assigns the returned pointer to `r_pt`. Since `r_pt` is a pointer to a `MoodyReceptionist` object, this statement implicitly casts the created `RudeReceptionist` object to a `MoodyReceptionist`, before the pointer assignment process. It is now clear that the type of `r_pt` before and after the casting is `MoodyReceptionist*`. Therefore, function `r_pt->greet()` is associated with class `MoodyReceptionist`.

Unlike a regular object, a pointer needs two memory spaces: one for itself and another for the object that it points to. The former space is created at the pointer declaration, while the latter is created using "`new`." Function ambiguity occurs when the pointer is declared to point to an object of one type, but the pointed object is created to store an object of another type. By default, the pointer and the object will be associated with the declaration type, not the construction type.

B.3 Virtual Functions

The result in Example B.3 is different from that in Example B.1. When creating a pointer by executing "`new RudeReceptionist`," we expect the rude receptionist to say "What do you want? I'm busy. Come back later!!", not just "What do you want?" as in Example B.4. To do so, a `RudeReceptionist` object needs to be associated with the construction type not the declaration type. In C++, such an association is carried out through *virtual function*s.

Unlike regular functions, virtual functions always belong to the construction type, regardless of type casting. C++ declares a virtual function by putting a keyword "`virtual`" in front of the function declaration. Note that, the virtuality property is inheritable. We only need to declare the virtual function once in the base class. The same function in the derived class automatically inherits the virtuality property.

Example B.5. In Example B.4, replace Line 4 from Example B.1 with the following line:

```
virtual void greet() {cout<<"Say:\n";};
```

which declares the function `greet()` of class `Receptionist` as virtual.

Since `r_pt` is created using "`new RudeReceptionist`," virtual function `r_pt->greet()` belongs to class `RudeReceptionist`. At the declaration "`MoodyReceptionist *r_pt`," the pointer `r_pt` is created by its default constructor. However, the space where `r_pt` points to (i.e., `*r_pt`) is created by the statement "`new RudeReceptionist`." Since a virtual function sticks to the construction type, the statement `r_pt->greet()` invokes function `greet()` of class `RudeReceptionist`. After running `./receptionist`, we will obtain the same result as that in Examples B.1 and B.3.

B.4 Abstract Classes and Pure Virtual Functions

An *abstract class* provides a general concept from which more specific classes derive. Conforming to the polymorphism concept, it specifies "what to do" in special functions called *pure virtual function*s, and forces its derived classes to define their own "how to do" by overriding the pure virtual functions. Containing at least one pure virtual function, an abstract class is said to be *incomplete* since it does not have a "how to do" part. Consequently, no object can be initiated from an abstract class. By not implementing *all* virtual functions, the derived class would still be an abstract class (i.e., incomplete), and cannot initiate any object.

C++ declares a pure virtual function by putting "`virtual`" and "`=0`" at the beginning and the end of function declaration, respectively.

Example B.6. Consider again the example on receptionists and how they greet customers. We keep the class hierarchy in Fig. B.1 unchanged. To make class `Receptionist` an abstract class, we modify Example B.5 by removing Lines 4 in Example B.1 replacing the declaration of class `Receptionist` in Example B.1 with the following codes:

```
1 class Receptionist {
2     public:
3     virtual void greet()=0;
4 };
```

After running "`./receptionist`," we should obtain the same results as in Example B.1. In this example, three main components are related to the use of an abstract class.

- *A pure virtual function*: Function `greet()` is declared in class `Receptionist` as a pure virtual function (Line 3 in Example B.6).
- *An abstract class*: Containing a pure virtual function `greet()`, class `Receptionist` is an abstract class. No object can be instantiated from class `Receptionist`. Class `Receptionist` therefore acts as a template class for classes `FriendlyReceptionist`, `MoodyReceptionist`, and `Rude Receptionist`.

Table B.1 Declaration with no implementation, declaration with no action, and invalid declaration

Declaration	Example
Pure virtual declaration	`virtual void greet()=0;`
Declaration with no action	`virtual void greet() {};`
Invalid declaration	`virtual void greet();`

- *An instantiable class*: Classes `FriendlyReceptionist`, `MoodyReceptionist`, and `RudeReceptionist` must provide implementation for function `greet()` (see Example B.1). Unlike Example B.2, removing the implementation (e.g., Line 16 in Example B.1) leaves the derived classes (e.g., `MoodyReceptionist`) an abstract class, and the instantiation (e.g., `m_pt = new MoodyReceptionist`) would cause a compilation error. □

There are three related declarations for a virtual function (see Table B.1). First, a pure virtual function is declared as explained above (e.g., `virtual void greet() = 0;`). Secondly, a (non-pure) virtual function of a derived instantiable class must contain implementation but may have no action. For example, "`virtual void greet() {};`" contains no action inside its curly braces. This function overrides the pure virtual function of its parent class, making the class non-abstract and instantiable. Finally, consider a class whose parent class is an abstract class. By opting out "{}" (i.e, "`virtual void greet();`"), the pure virtual function is left unimplemented and the class would still be an abstract class. Again, any object instantiation would lead to a compilation error.[3] *An important note for NS2 users: You cannot opt out both "`=0`" and "{}." If you do not want to provide an implementation, leave the curly braces with no action after the declaration. Otherwise, NS2 will show an error at the compilation.*

B.5 Class Composition: An Application of Type Casting Polymorphism

Upto this point, the readers may raise few questions. That is, why do we need to cast an object to a different type and use the keyword `virtual`? Wouldn't it be easier to declare and construct an object with the same type? For example, can't we use Example B.3 instead of Example B.4? Doesn't it remove function ambiguity? The answer is "yes"; nevertheless, type casting makes the programming more scalable, elegant, and interesting. For this reason, programming with type casting is a common practice in NS2.

[3]Here, we assume that declaration and implementation are in one file. When declaration and implementation are separated in two files, you can opt out "{}" in a "`.h`" file and provide the implementation in another "`.cc`" file.

B.6 Programming Polymorphism with No Type Casting: An Example

Example B.7 below shows a scenario, which needs no virtual function. However, we will see later that Example B.7 leads to programming inconvenience as the program becomes larger.

Example B.7. Consider a company and how it serves a customer. The main functionality of the company is to serve customers. As a courtesy, the company greets every customer before serving. Assume that the company has one receptionist to greet the customer. The receptionist can be friendly, moody, or rude as specified in Example B.1. The following C++ code represents the company with the above description:

```
    //company.cc
1   class Company {
2       public:
3           void serve() {
4               greet();
5               cout<<"\nServing the customer ... \n";
6           };
7           void greet () {};
8   };

9   class MoodyCompany : public Company {
10      public:
11          MoodyCompany(){employee_ = new Moody
            Receptionist;};
12          void greet(){employee_->greet();};
13      private:
14          MoodyReceptionist* employee_;
15  };

16  int main() {
17      MoodyCompany my_company;
18      my_company.serve();
19      return 0;
20  }
```

where class `MoodyReceptionist` is defined in Example B.1.

Class `Company` (Lines 1–8) has two functions. Function `serve()` in Lines 3–6 greets the customers by invoking function `greet()`. Then, it serves the customer by showing the message "`Serving the customer ...`" on the screen. The function `greet()` in Line 7 has no action in class `Company`, and is implemented by child classes of class `Company`.

Class `MoodyCompany` (Lines 9–15) derives from class `Company`. It has one moody receptionist stored in the variable `employee_` (Line 14). Class `MoodyCompany` implements function `greet()` by having `employee_->greet()` in Line 12.

In the function `main()`, an object `my_company` of class `MoodyCompany` is instantiated in Line 17. Line 18 invokes function `serve()` associated with the object `my_company`. By running the executable file `company`, the following result will appear on the screen:

```
>>./company
Say: What do you want?
Serving the customer ...
```

which is quite expected from the code. Clearly, we do not need virtual functions in this example.

B.7 A Scalability Problem Caused by Non-Type Casting Polymorphism

The main problem of polymorphism with non-type casting is the scalability. As the inheritance tree becomes more complicated, we may need to develop a large number of classes. For example, suppose we would like to change the receptionist in the company to be a friendly receptionist. We will have to define another class as follows:

```
class FriendlyCompany : public Company {
    public:
        FriendlyCompany() { employee_ =
                          new FriendlyReceptionist}
        void greet() {employee_->greet();};
    private:
        FriendlyReceptionist* employee_;
};
```

Also, replace Line 17 in Example B.7 with

```
FriendlyCompany my_company;
```

By running "`./company`," the following result should appear on the screen:

```
>>./company
Say: Good morning. How can I help you today?
Serving the customer ...
```

The problem is that a new `Company` class (e.g., `FriendlyCompany`) is required for every new `Receptionist` class (e.g., `FriendlyReceptionist`).

Furthermore, the company may have other types of employee such as technicians, managers, etc. If there are ten classes for receptionists and ten classes for technicians, we need to define 100 classes to cover all combination of employee types. In the next section, we will show how this scalability problem can be avoided using class composition.

B.8 The Class Composition Programming Concept

Type casting acts as a tool which helps avoid the scalability problem. Instead of deriving all class combination (e.g., 100 classes of combinations of ten receptionists and ten technicians), we may declare an *abstract user class* object (e.g., `Receptionist`), and cast the abstract user class object to a more specific object (e.g., `FriendlyReceptionist`).

Example B.8. Consider a company and how it serves a customer in Example B.7. By allowing type casting, the code representing the company is given below:

```
    //company.cc
1   class Company {
2       public:
3           void hire(Receptionist* r) {
4               employee_ = (Receptionist*)r;
5            };
6           void serve() {
7               employee_->greet();
8               cout<<"\nServing the customer ... \n";
9           };
10      private:
11          Receptionist* employee_;
12  };

13  int main() {
14      MoodyReceptionist *m_pt= new MoodyReceptionist
          ();
15      Company my_company;
16      my_company.hire(m_pt);
17      my_company.greet();
18      return 0;
19  }
```

Also, to bind function `greet()` to the construction type, we need to declare function `greet` of class `Receptionist` as virtual. Here, we replace Line 4 in Example B.1 with "`virtual void greet();`" or "`virtual void greet() = 0;`."

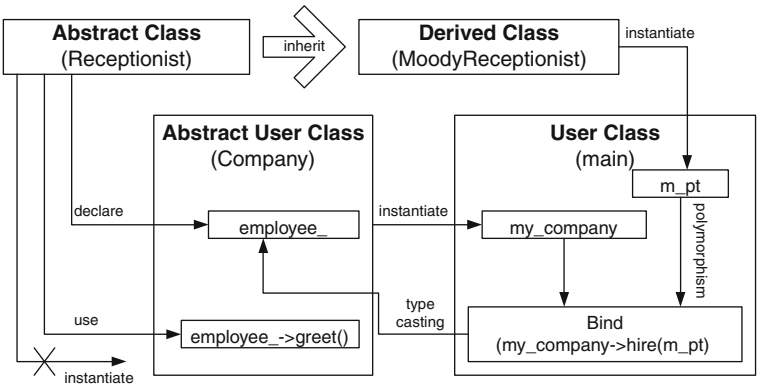

Fig. B.2 A diagram of the class composition concept with type casting polymorphism

Class `Company` declares a variable `employee_` as a `Receptionist` pointer in Line 11. The company hires an employee by invoking function `hire(r)` in Lines 3–5. Taking a `Receptionist*` object, `r`, as an input argument, function `hire(r)` assigns an input `Recectionist` pointer to its private variable `employee_`. In Lines 6–9, the company `serves` the customers as it does in Example B.7.

In function "`main()`," an object of class `Company`, `my_company`, is created in Line 15. In Line 16, `my_company` hires an employee `m_pt` which is a pointer to a `MoodyReceptionist` object. From Lines 3–5, function `hire(m_pt)` casts the pointer `m_pt` to a `Receptionist` pointer. Since the function `greet()` of class `MoodyReceptionist` is virtual, `employee_->greet()` is associated with the construction type in Line 14 (i.e., class `MoodyReceptionist`). By running "`./company`," we will obtain the following result:

```
>>./company
Say: What do you want?
Serving the customer ...
```

which is the same as that in Example B.7. □

As shown in Fig. B.2, the class composition programming concept with type-casting polymorphism consists of four main class types.

- An *abstract class* (e.g., `Receptionist`) is a template class.
- A *derived class* (e.g., classes `MoodyReceptionist`) derives from the above abstract class.
- An *abstract user class* (e.g., class `Company`) declares objects of the *abstract class* (e.g., `Receptionist`). It uses the functions of the abstract class without the need to know the detailed implementation of the abstract class. In Example B.8, class `Company` does not need to know what type of `Receptionist` the `employee_` is, nor how the `employee_` greets the customers.

- A *user class* (e.g., `main`) declares objects of the *derived class* (e.g., `Moody Receptionist`). It makes the abstract class more specific by binding (e.g., using function `hire(r)`) the abstract variable (e.g., `*employee_`) belonging to the abstract user object (e.g., `my_company`) to the derived object (e.g., `m_pt`).

The concept of class composition is to have an abstract user class (e.g., `Company`) declare its variable from an abstract class (e.g., `Receptionist`) and later cast the declared object (e.g., `employee_`) to a more specific type (e.g., `MoodyReceptionist`). In particular, the mechanism consists of four following steps:

1. Declare an abstract class (e.g., `Receptionist`).
2. From within an abstract user class (e.g., `Company`), declare (e.g., Receptionist* `employee_`) and use (e.g., `employee_->greet()`) objects of the above abstract class.[4]
3. In a user class (e.g., `main()`),
   a. Instantiate an object (e.g., `my_company`) of the abstract user class (e.g., `Company`).
   b. Instantiate an object (e.g., `*m_pt`) of the derived class (e.g., `Moody Receptionist`).
4. Bind (e.g., using `hire(r)`) the abstract class object (e.g., `*employee_`) in the abstract user class (e.g., `Company`) to the object initiated from within the user class (e.g., `*m_pt`). Since the latter object class derives from the former one, the type casting is fairly straightforward.

To change the company's receptionist to be a friendly receptionist, we only need to change function `main` as follows, without having to modify other parts of the codes:

```
int main() {
    FriendlyReceptionist *f_pt;
    f_pt = new FriendlyReceptionist();

    Company my_company();
    my_company.hire (f_pt);
    my_company.serve();
    return 0;
}
```

[4]Again, declaration of a too specific (e.g., `MoodyReceptionist` as opposed to `Receptionist`) class in non-type-casting polymorphism leads to the scalability problem. As the program becomes larger, we need to redefine classes for every new class, hence substantially increasing the total number of classes. To avoid the scalability problem, we need to declare classes to be as general as possible. This general class can later be cast as a more specific class.

To see how type casting helps avoid scalability, consider the above example where a company may have one of ten possible receptionist classes and one of ten possible technician classes. Without type casting, we need to define 100 classes to cover all the combination of receptionists and technicians in addition to one base class `Company`. By allowing type casting, we can declare two variables (of abstract classes `Receptionist` and `Technician`) in a company. In the main program, we can instantiate a receptionist and a technician from any of these `Receptionist` and `Technician` classes. After instantiating receptionist and technician objects from the derived class, we can cast the instantiated objects back to classes `Receptionist` and `Technician` and assign them to the company. Under the same scenario, the class composition concept requires only 20 classes for receptionists and technicians, and therefore, greatly alleviates the scalability problem.

C
BSD Link List and Bit Level Functions

C.1 BSD Link List

Similar to an array, a link list is a data structure which can contain a collection of data items [16]. Link lists are implemented using pointers. Therefore, programmers do not need to specify memory requirement for a link list. The memory is allocated to the link list at runtime.

NS2 defines macros for link lists in the file *~ns*/lib/bsd-list.h as follows:

`LIST_HEAD(<name>,<type>)`:
: Declare a `struct` data type which models a head pointer to the link list. The name of the data type is `<name>`, and the data type of each item in the link list is `<type>`.

`LIST_ENTRY(<type>)`:
: Declare a `struct` data type which models an entry in a link list. The data type of each item in the link list is `<type>`.

`LIST_INSERT_AFTER(head,elm,field)`:
: Insert `*<elm>` as the first element of the link list whose head pointer is `<head>`, where `<field>` is a `struct` variable containing pointers which make up the link list.

C.2 Bit Level Functions

NS2 employs several C++ bit level operation functions.

`bcopy(p1,p2,n)`
: Copy `n` bytes from `*p1` to `*p2`.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_18, © Springer Science+Business Media, LLC 2012

`bzero(p,n)`

Set first `n` bytes of `*p` to be zero.

`bsearch(keypt,arrpt,num,bytes,cmp_fn)`

Look for `*keypt` within `arrpt` which contains `num` elements, each with size `bytes`. It returns a pointer to the matching element, and zero if no matching key is found. For each element in the array `arrpt`, this function feeds `*keypt` and the array element as the input arguments of the function `cmp_fn`. The function `cmp_fn` must return negative, zero, and positive values if `*keypt` is less than, equal to, and greater than array element, respectively.

References

1. A. S. Tanenbaum, *Computer Networks*, 3rd ed. Prentice Hall, 1996.
2. R. E. Shannon, "Introduction to the art and science of simulation," in *Proc. of the 30th conference on Winter simulation (WSC'98)*, 1989.
3. W. T. Kasch, J. R. Ward, and J. Andrusenko "Wireless network modeling and simulation tools for designers and developers," IEEE Communication Magazine, pp. 120–127, March, 2009. *Proc. of the 30th conference on Winter simulation (WSC'98)*, 1989.
4. R. G. Ingalls, "Introduction to simulation: Introduction to simulation," in *WSC '02: Proceedings of the 34th conference on Winter simulation*. Winter Simulation Conference, 2002, pp. 7–16.
5. W. H. Tranter, et al., *Principles of Communication Systems Simulation*. Prentice Hall, 2004.
6. A. Papoulis and S. U. Pillai, *Probability, Random Variables and Stochastic Processes*, 2nd ed. McGrawHill, 2002.
7. W. H. Press, et al., *Numerical Recipes in C*, 2nd ed. Cambridge University Press, 1997.
8. R. M. Goldberg, *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc., 2000.
9. J. Banks andI. J. S. Carson, *Discrete-Event Systems Simulation*. Prentice-Hall, Inc., 1984.
10. The Network Simulator Wiki. [Online]. Available: http://nsnam.isi.edu/nsnam/index.php/
11. The Network Simulator Wiki–NS-2Trace Formats. [Online]. Available: http://nsnam.isi.edu/nsnam//index.php/NS-2_Trace_Formats#Old_Wireless_Trace_Formats
12. The Network Simulator – ns-2. [Online]. Available: http://www.isi.edu/nsnam/ns/
13. M. Greis. Tutorial for the Network Simulator NS2. [Online]. Available: http://www.isi.edu/nsnam/ns/tutorial/
14. J. Chung and M. Claypool. Ns by example. [Online]. Available: http://nile.wpi.edu/NS/
15. The Network Simulator Wiki–Contributed Code. [Online]. Available: http://nsnam.isi.edu/nsnam/index.php/Contributed_Code
16. P. Deitel and H. Deitel, *C++ How to Program, 7th Edition*, Pearson, 2010. 4th ed. McGraw-Hill/Osborne Media, 2002.
17. K. Fall and K. Varadhan. (2007, Aug.) The ns manual (formerly known as ns notes and documentation). [Online]. Available: http://www.isi.edu/nsnam/ns/ns-documentation.html
18. Réseaux et Performances. NS 2.26 source original: Hierarchical index. [Online]. Available: http://www-rp.lip6.fr/ns-doc/ns226-doc/html/hierarchy.htm
19. T. H. Cormen, et al., *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001.
20. M. Mathis, et al., *TCP selective acknowledgement options*, RFC 768 Std., 1996.
21. J. Kurose. The TCP/IP course website. [Online]. Available: http://www.networksorcery.com/enp/protocol/udp.htm
22. J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. Pearson Addison-Wesley, 2008.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3, © Springer Science+Business Media, LLC 2012

23. V. Paxson and M. Allman, *Computing TCP's Retransmission Timer*, RFC 2988 Std., November 2000.
24. C. E. Perkins, *Ad Hoc Networking*, Addison-Wesley, 2001.
25. P. L'Ecuyer, "Good parameters and implementations for combined multiple recursive random number generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.
26. D. Libes, "A debugger for Tcl applications," in *Tcl/Tk Workshop*, June 1993. [Online]. Available: http://expect.nist.gov/tcl-debug/tcl-debug.ps.Z
27. The GDB Developers. GDB: The GNU project debugger. [Online]. Available: http://www.gnu.org/software/gdb/
28. T. Nandagopal, S. Lu, and V. Bhargharvan "A unified architecture for the design and evaluation of wireless fair queuing algorithms," *MOBICOM99*, pp. 132–142, 1999.
29. S. Sanfilippo. An introduction to the Tcl programming language. [Online]. Available: http://www.invece.org/tclwise
30. How can I do math in Tcl, Welcome to the Tclers Wiki!, http://wiki.tcl.tk/528
31. Berkeley Continuous Media Toolkit. OTcl tutorial. [Online]. Available: http://bmrc.berkeley.edu/research/cmt/cmtdoc/otcl/
32. A. Robbins and D. Gilly, *Unix in a Nutshell: System V Edition*. O'Reilly & Associates, Inc., 1999.
33. An AWK primer. [Online]. Available: http://www.vectorsite.net/tsawk.html
34. Wikipedia. Regular expression. [Online]. Available: http://en.wikipedia.org/wiki/Regular_expression
35. V. Paxson and M. Allman, *RFC 2988: Computing TCP's Retransmission Timer*,. [Online]. Available: http://www.ietf.org/rfc/rfc2988.txt
36. C. Flynt, *Tcl/Tk: A Developer's Guide*, Second Edition, Morgan Kaufmann, 2003.
37. C. E. Perkins and E. M. Royer "Ad-hoc On-Demand Distance Vector Routing," *IEEE Workshop on Mobile Computing Systems and Applications*, 1999, LA.
38. M. Schwartz, *Mobile Wireless Communications*, Cambridge University Press, 2005.
39. *IEEE Standard for Wireless LAN Medium Access Control and Physical Layer Specification, P802.11*, IEEE, 1999.
40. T. Issariyakul, E. Hossain, and D. I. Kim, "Medium access control protocols for wireless mobile ad hoc networks: Issues and approaches," *Wiley Interscience Wireless Communications and Mobile Computing*, vol.3, no. 8, Dec., 2003.

General Index

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3, © Springer Science+Business Media, LLC 2012

Code Index

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3, © Springer Science+Business Media, LLC 2012