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 which NS is based on. Since 1995 the Defense Advanced Research Projects Agency (DARPA) supported development of NS through the Virtual InterNetwork Testbed (VINT) project [9].[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 [10] provides NS2 source code as well as detailed installation instruction. The web pages in [11] and [12] are among highly recommended ones which provide tutorial and

[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, E. Hossain, *Introduction to Network Simulator NS2*,
DOI: 10.1007/978-0-387-71760-9_2,

examples for setting up basic NS2 simulation. A comprehensive list of NS2 codes contributed by researchers can be found in [13]. 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, Section 2.2 presents the basic architecture of NS2. The information on NS2 installation is given in Section 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 Section 2.6. Section 2.7 describes how to include C++ modules in NS2. Finally, Section 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 on input argument, the name of a Tcl simulation scripting file. Users are feeding the name of a Tcl simulation script (which sets up a simulation) as an input argument of an NS2 executable command `ns`. 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 objects, 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 (e.g., `n` as a `Node` 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 defines its own procedures and variables to facilitate the interaction. Note that the member procedures and variables in the OTcl domain are called instance procedures

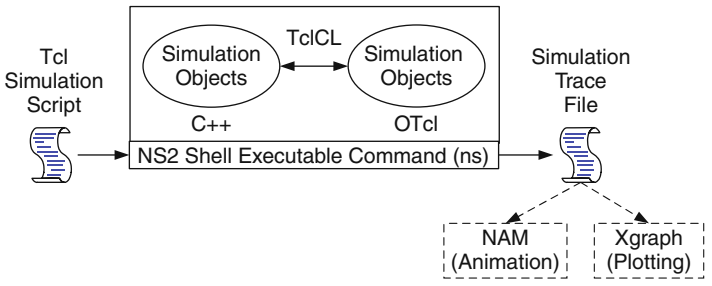

Fig. 2.1. Basic architecture of NS.

(instprocs) and instance variables (instvars), respectively. Before proceeding further, the readers are encouraged to learn C++ and OTcl languages. We refer the readers to [14] 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++ objects. It is advisable to use these C++ objects to set up a simulation using a Tcl simulation script. However, advance users may find these objects insufficient. They need to develop their own C++ objects, and use a OTcl configuration interface to put together these objects.

After simulation, NS2 outputs either text-based or animation-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 [9]. 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. Unless otherwise specified, 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 2.30,
- Tcl/Tk release 8.4.13,
- OTcl release 1.12, and
- TclCL release 1.18.

and the following are the optional components:

- NAM release 1.12: 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 12.1: This is a data plotter with interactive buttons for panning, zooming, printing, and selecting display options.

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 commands show how the all-in-one NS2 suite can be installed and validated, respectively:

```
shell>./install
shell>./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 [9], where the information on post-installation troubles can also be found.

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

Table 2.1. Additional Cygwin packages required to run NS2.

Category	Packages
Development	`gcc`, `gcc-objc`, `gcc-g++`, `make`
Utils	`patch`
X11	`xorg-x11-base`, `xorg-x11-devel`

[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-2.30`. Figure 2.2 shows the directory structure under directory `nsallinone-2.30`. Here, directory `nsallinone-2.30` is on the Level 1. On the Level 2, directory `tclcl-1.18` contains classes in TclCL (e.g., `Tcl, TclObject, TclClass`). All NS2 simulation modules are in directory `ns-2.30` on the Level 2. Hereafter, we will refer to directories `ns-2.30` and `tclcl-1.18` as ~*ns*/ and ~*tclcl*/, respectively.

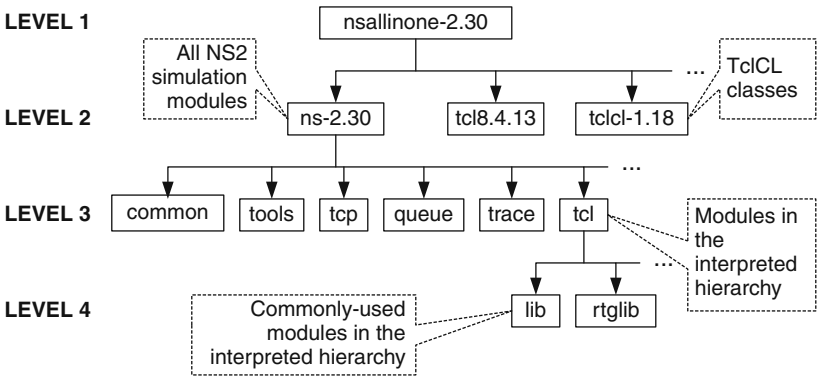

Fig. 2.2. Directory structure of NS2 [12].

On Level 3, the modules in the interpreted hierarchy are under directory `tcl`. Among these modules, the frequently-used ones (e.g., `ns-lib.tcl, ns-node.tcl, ns-link.tcl`) are stored under directory `lib` on Level 4. Simulation modules in the compiled hierarchy are classified in directories on Level 2. For example, directory `tools` contains 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 which are used in NS2 and in this book hereafter are shown below:

Terminology

- An NS2 simulation script (e.g., `myfirst_ns.tcl`) is referred to as a *Tcl simulation script* .
- 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 Section 3.4.4, *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 commands, while a C++ object is associated with variables and functions.
- Despite their minor differences, the terms "OTcl" and "interpreted" are used interchangeably throughout the book. Likewise, "C++" and "compiled" are used interchangeably. These terms can be used as adjectives to indicate the domain under consideration. For example, both OTcl variables and interpreted variables refer to variables in the interpreted hierarchy. Similarly, both C++ functions and compiled functions refer to functions in the compiled hierarchy. Also, we will refer to the C++ compiler and the OTcl interpreter simply as the compiler and the interpreter, respectively.
- 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 right is referred to as an *upstream object*. In a general case, an object can have more than one target. However, a packet must be forwarded to one of these targets. From the perspective of an upstream object, a downstream object which receive the packet is also referred to as a *forwarding object*.

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.
- 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 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 and a slash character ("`/`") as a prefix, while that in the compiled hierarchy is 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.

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 (2^{nd} level)	`Agent/Tcp/Reno`	`RenoTcpAgent`
Class functions	`installNext`	`install_next`
Class variables	`windowOption_`	`wnd_option_`

Exercise 2.1. Design C++ and OTcl classes (e.g., Class `My TCP`). Derive this class from the TCP Reno classes shown in Table 2.2. Use the convention defined above to name the class names, variables/instvars, and functions/instprocs in both the domain.

2.5 Running NS2 Simulation

2.5.1 NS2 Program Invocation

After the installation and/or recompilation (see Section 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 interpreted 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>`, 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

The followings show the three key step guideline in defining a simulation scenario in a NS2:

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 and 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 reached 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: Post Simulation 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 Chapter 13).

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 2.3 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

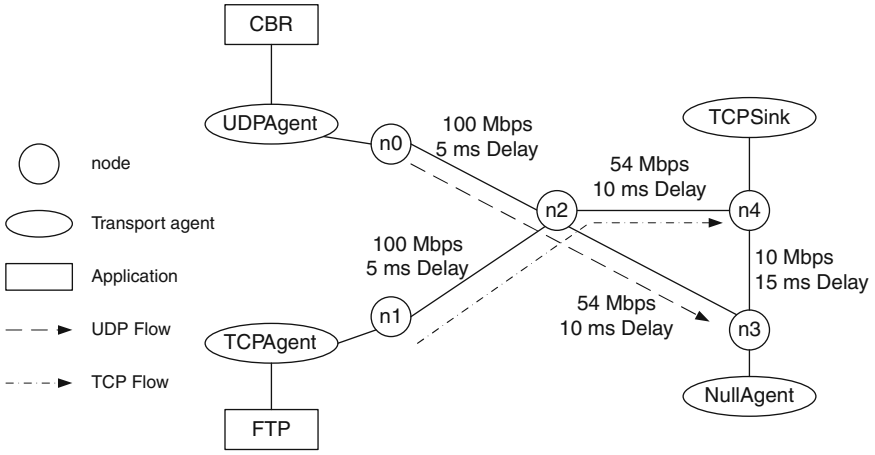

Fig. 2.3. A sample network topology.

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.

Step 2: Configuring and Running Simulation

Programs 2.1–2.2 show two portions of a Tcl simulation script which implements the scenario in Fig. 2.3.

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{}` is 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 9–10 close the file associated with handles `myTrace` and `myNAM`. Line 11 executes the statement "`nam out.nam &`" from the shell environment. Finally, Line 12 tells NS2 to exit with code 0.

Lines 14–18 creates Nodes using the instproc `node` of the Simulator whose handle is `ns`. Lines 19–23 connects 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 an 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 the instproc `duplex-link{...}`, Line 23 create a uni-directional link using an instproc `simplex-link{...}` of class `Simulator`. Finally, Line 24 sets the queue 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 schedules 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

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
```

argument, an instproc `attach-agent{...}` of class `Simulator` in Line 26 attaches a UDP 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.

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
```

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 1000 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 61*st* second. Note that the FTP and CBR traffic source can be started and stopped by invoking its 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
...
```

Figure 2.4 shows the format of each trace line, which consists of 12 columns.

The general format of each trace line is shown in Fig. 2.4, 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

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

Fig. 2.4. Format of each line in a normal trace file.

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 to 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 these languages are described in Appendix A.

Text-based packet tracing is activated by executing "`$ns trace-all $file`", where `ns` is the Simulator handle and `file` is a handle associated with the file which stores the tracing text. 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 prior to object creation. We shall discuss the detail of text-based packet tracing later in Chapter 13.

Network AniMation (NAM) Trace

NAM trace is records simulation detail in a text file, and uses the text file the 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 2.2):

```
>>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 the 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 what specified in the default descriptor file `Makefile`. If the descriptor file `mydescriptor` is specified, the utility is use this file in place of the default file `Makefile`.

2.7.2 A Make Descriptor File

A descriptor file contains an instructor of how the codes should be recompiled and relinked. Again, the default descriptor file is the file named "`Makefile`". A descriptor file contains the names of the 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> ...] : [<dependency1> ...]
    [<command>]
```

A target with a colon sign is mandatory. Everything else 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 line `executes` to regenerate the target file.

Example 2.2 (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 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 *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 2.2 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.

[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`.

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 2.2 becomes a bit more handy as shown in Example 2.3.

Example 2.3 (Example of makefile/Makefile with Macros.).

```
# 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}
```

2.7.3 NS2 Descriptor File

The NS2 descriptor file is defined in a file `Makefile` located in the home directiory of NS2. It contains the 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 should be added here.
- `NS_TCL_LIB =` : The items bind this 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:

(i) Include a string "`-I./myfiles`" into the Line beginning with `INCLUDES =` in the `Makefile`.
(ii) Include a string "`myfile/myc.o`" into the Line beginning with `OBJ_CC =` or `OBJ_STL =` in the `Makefile`.
(iii) Include a string "`myfile/mytcl.tcl`" into the Line beginning with `NS_TCL_LIB =` in the `Makefile`.
(iv) Run `make` from the shell.

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

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, utilizes 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. Secondly, configuring and running simulation implements 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 employs 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 re-run `make`.