NS Simulator for beginners

Lecture notes, 2003-2004
Univ. de Los Andes, Mérida, Venezuela and ESSI,
Sophia-Antipolis, France

Eitan Altman and Tania Jiménez

December 4, 2003

Contents

Chapter 1

Introduction

The ns simulator covers a very large number of applications, of protocols, of network types, of network elements and of traffic models. We call these ”simulated objects”. The goal of our notes is twofold: on one hand to learn how to use ns simulator, and on the other hand, to become acquainted with and to understand the operations of some of the simulated objects using ns simulations. Our notes provide therefore not only some basis and description of ns simulator (especially through a large number of tcl scripts), but also a description of the simulated objects. Finally, we focus on the analysis of the behavior of the simulated objects using ns simulations.

The notes are intended to help students, engineers or researchers who need not have much background in programming or who want to learn through simple examples how to analyse some simulated object using ns. In that purpose, we provide a large number of tcl scripts that can be used by the reader so as to start programming immediately. For readers who are interested to learn from examples, we should mention that a very large number of examples is already available in the software package of the ns simulator[1]. Other tutorials containing many examples are available electronically: Marc Greis's tutorial[2] and the tutorial by Jae Chung and Mark Claypool[3].

For a much deaper study of the ns simulator one should refer to the ns manual which is maintained up-to-date at `http://www.isi.edu/nsnam/ns/`.

We present in this book many simple (but hopefuly useful) scenari for simulations. Simulations may differ from each other in many aspects: the applications, topologies, parameters of network objects (links nodes) and protcoles used etc. We do not aim at being exhaustive; instead we present what we consider to be ”typical” examples. If one needs a more exhaustive description of ns, one may find it very useful to consult the ns manual[4]. An alternative simple way to know about other possibilities for choosing network elements, network protocols or their parameters, application parameters, etc., is to look directly at the library files that define them[5]. For example, the definitions of mobile nodes would be find in the file ns-mobilenode.tcl, those describing queueing disciplines and parameters in the file ns-queue.tcl, etc. Defaults parameters can be found at the file ns-default.tcl. Note: to know which default object is related to which command, one may need to check the file ns-lib.tcl as we shall see in an example in Section 2.2.

[1] it typically appears in the directory ns-2/tcl/ex, where directory ”ns-2” could have other longer names that depend on the ns release, e.g. ”ns-2.1b8a”

[2] http://www.isi.edu/nsnam/ns/tutorial/index.html

[3] http://nile.wpi.edu/NS/

[4] see http://www.isi.edu/nsnam/ns/ns-documentation.html

[5] ns-allinone-2.1b8a/ns-2.1b8a/tcl/lib

1.1 Background on the ns simulator

NS simulator is based on two languages: an object oriented simulator, written in C++, and a OTcl (an object oriented extension of Tcl) interpreter, used to execute user's command scripts.

NS has a rich library of network and protocol objects. There are two class hierarchies: the compiled C++ hierarchy and the interpreted OTcl one, with one to one correspondance between them.

The compiled C++ hierarchy allows us to achieve efficiency in the simulation and faster execution times. This is in particular useful for the detailed definition and operation of protocols. This allows one to reduce packet and event processing time.

Then in the OTcl script provided by the user, we can define a particular network topology, the specific protocols and applications that we wish to simulate (whose behavior is already defined in the compiled hierarchy) and the form of the output that we wish to obtain from the simulator. The OTcl can make use of the objects compiled in C++ through an OTcl linkage (done using tclCL[6]) that creates a matching of OTcl object for each of the C++.

NS is a discrete event simulator, where the advance of time depends on the timing of events which are maintained by a scheduler. An event is an object in the C++ hierarchy with an unique ID, a scheduled time and the pointer to an object that handles the event. The scheduler keeps an ordered data structure (there are four, but by default ns use a simple linked-list) with the events to be executed and fires them one by one, invoking the handler of the event.

1.2 Tcl and Otcl programming

Tcl (Tool Command Language) is used for millions of people in the world. It is a language with a very simple sintaxis and it allows a very easy integration with other languages. Tcl was created by Jhon Ousterhout. The characteristics of these language are:

- It allows a fast development
- It provide a graphique interface
- It is compatible with many plataforms
- It is flexible for integration
- It is easy to use
- It is free

Here are some basics of tcl and Otcl programming.

- Assigning a value to a variable is done through the "`set`" command; for example: "`set b 0`" assigns to b the value of 0. This is equivalent to "`b=0`" in C for example.
- When we wish to use the value assigned to a variable, we should use a $ sign before the variable. For example, if we want to assign to variable `x` the value that variable `a` has, then we should write: "`set x $a`".
- A mathematical operation is done using the expression command. For example, if we wish to assign to a variable `x` the sum of values of some variables `a` and `b`, we should write "`set x [expr $a + $b]`".

[6] TclCL is a Tcl/C++ interface[15]

- In Tcl the variables are not typed, so a variable can be a string or an integer depending on the value you assign to it. For example, assume that we want to print the result of the division 1/60. If we write
  `puts "[expr 1/60]"`, then the result will be 0! To have the correct result, we need to indicate that we do not work with integers, and should thus type
  `puts "[expr 1.0/60.0]"`

- The sign # starts a commented line that is not part of the program, so the tcl interpreter will not execute this line.

- To create a file, one has to give it a name, say "filename", and to assign a pointer to it that will be used within the tcl program in order to relate to it, say "file1". This is done with the command: `set file1 [open filename w]`

- The command `puts` is used for printing an output. Note: each time the "`puts`" command is used, a new line is started. To avoid new line, one has to add `-nonewline` after the "`puts`" command. If we want to print into a file (say the one we defined above) we type `puts $file1 "text"`. Tabulating is done by inserting `\t`. For example, if a variable, say x, has the value 2 and we type `puts $file1 "x  $x"` then this will print a line into the file whose name is "filename" with two elements: "x" and "2" separated by a tabulator space.

- Execution of a unix command: is done by typing "exec" and then the command. For example, we may want ns to initiate the display of a curve whose data are given in a two column file named "data" within the simulation. This can be using the xgraph command and will be written as:

  ```
  exec xgraph data &
  ```

  (note that the "`&`" sign is used to have the command executed in the background).

- The structure of an `if` command is as follows:

  ```
  if { expression } {
     <execute some commands>
     } else {
     <execute some commands>
  }
  ```

  The `if` command can be nested with other "`if`"s and with "`else`"s that can appear in the "`<execute some commands>`" part. Note that when testing equality, we should use "`==`" and not "=". The inequality is written with `!=`.

- Loops have the following form:

  ```
  for { set i 0 } { $i < 5 } { incr i } {
     <execute some commands>
  }
  ```

  In this example the commands in the loop will be executed five times. After the `for` the "{ `set i 0` } declares the variable `i` that will be used as the counter of the loop and initializes it to 0. The second part between { } is the continuation condition of the loop, it says "do the loop while the counter `i` is less than 5. The last part of the statement is for declaring

```
#Create a procedure
proc test {} {
    set a 43
    set b 27
    set c [expr $a + $b]
    set d [expr [expr $a - $b] * $c]
    puts "c =  $c d =  $d"
    for {set k 0} {$k < 10} {incr k} {
if {$k < 5} {
    puts "k < 5, pow = [expr pow($d, $k)]"
} else {
    puts "k >= 5, mod = [expr $d % $k]"
}
    }
}

#calling the procedure
test
```

Table 1.1: Tcl program for doing arithmetic operations

the changing in the counter variable, in this case we increment `i` one by one, but we can also decrement it or use any mathematical expressioni for increment or decrement the counter instead.

- tcl allows to create procedures. They can return some value in which case they contain a ”`return`” command. The general form of a procedure which we name ”blue” is

  ```
  proc blue { par1 par2 ... } {
          global var1 var2
          <commands>
          return $something
  }
  ```

  The procedure receives some parameters that can be objects, files or variables. In our case these are named par1, par2, etc. These parameters will be used within the procedures with these names. The procedure is called by typing `blue x y ...` where the values of `x` and `y` will be used by the procedure for par1 and par2. If `par1` and `par2` are changed within the procedure, this will not affect the values of `x` and `y`. On the other hand, if we wish the procedure to be able to affect directly variables external to it, we have to declare these variables as ”`global`”. In the above example these are `var1` and `var2`.

In Table 1.1 we show un example that present many arithmetic operations in tcl. The pow expresion give the power of variable d to k.

In Table 1.2 there is an example of a tcl program for computing all the prime numbers upto a given limit j. For example, to obtain all the prime numbers up to 11 type simply ”`ns prime.tcl 11`”. The prime numbers example shows how to use an `if` command, loops, and a procedure. The variable `argc` contains the number of parameters passed to the program. The variable `argv` is a vector that have the parameters passed to the program (so, `argc` is the lenght of `argv`), and the command `lindex` allows us to take the case of the vector pointed by the second parameter. So, the line `set j [lindex $argv 0]` assign to the variable `j` the value of the first parameter passed to the program which has been saved on the variable `argv`

```
# Usage: ns prime.tcl NUMBER
#          NUMBER is the number up to which we want to obtain the prime numbers
#
if {$argc  != 1} {
#  Must get a single argument or program fails.
 puts stderr "ERROR! ns called with wrong number of arguments!($argc)"
 exit 1
} else {
 set j [lindex $argv 0]
}

          proc prime {j} {

            # Computes all the prime numbers till j

            for {set a 2} {$a <= $j} {incr a} {
               set b 0
               for {set i 2} {$i < $a} {incr i} {
                  set d [expr fmod($a,$i)]
                  if {$d==0} {
                    set b 1}
               }
               if {$b==1} {
                  puts "$a is not a prime number"
               } else {
                  puts "$a is a prime number"
               }
            }
          }
prime $j
```

Table 1.2: Tcl program for computing prime numbers

We explain briefly througs an example the Object programming paradigm in OTcl. If you don't know another Object Oriented language (C++, Java), we recommend to read [?] to learn how to program using object oriented languages.

The reserved word `Class` followed of the name of the class is used to declare a new class in OTcl. The methods of the classes are declared using the word `instproc` preceded of the name of the class and followed of the name of the method and its parameters. The method `init` is the constructor of the class. The variable `self` is a pointer to the object itself, like `this` in C++ or Java. To declare the instance variable OTcl uses the word `instvar`. The word `-superclass` is used for declaring that a class inherits from another one, in the example the Integer class inherits from the Real class. We recommend you to learn more about OTcl reading: [?]

```
# Usage: ns fact.tcl NUMBER
#          NUMBER is the number we want to obtain the factorial
#
if {$argc  != 1} {
#  Must get a single argument or program fails.
 puts stderr "ERROR! ns called with wrong number of arguments!($argc)"
 exit 1
} else {
 set f [lindex $argv 0]
}

proc Factorial {x} {

for {set result 1} {$x > 1} {set x [expr $x - 1] } {
set result [expr $result * $x]
}
return $result
}
set res [Factorial $f]
puts "Factorial of $f is $res"
```

Table 1.3: Tcl simple program for computing the factorial of a number

```
Class Real

Real instproc init {a} {
        $self instvar value_
        set value_ $a
}

Real instproc sum {x} {
        $self instvar value_
        set op "$value_ + [$x set value_] = \t"
        set value_ [expr $value_ +  [$x set value_]]
        puts "$op $value_"
}

Real instproc multiply {x} {
        $self instvar value_
        set op "$value_ * [$x set value_] = \t"
        set value_ [expr $value_ *  [$x set value_]]
        puts "$op $value_"
}

Real instproc divide {x} {
        $self instvar value_
        set op "$value_ / [$x set value_] = \t"
        set value_ [expr $value_ /  [$x set value_]]
        puts "$op $value_"
}

Class Integer -superclass Real

Integer instproc divide {x} {
        $self instvar value_
        set op "$value_ / [$x set value_] = \t"
        set d [expr $value_ / [$x set value_]]
        set value_ [expr round($d)]
        puts "$op $value_"
        }

         set realA [new Real 12.3]
         set realB [new Real 0.5]

         $realA sum $realB
         $realA multiply $realB
         $realA divide $realB

         set integerA [new Integer 12]
         set integerB [new Integer 5]
         set integerC [new Integer 7]

         $integerA multiply $integerB
         $integerB divide $integerC
```

Table 1.4: Simple Otcl program using real and integer objects

Chapter 2

ns Simulator Preliminaries

In this Chapter we present the first steps that consist of

- Initialization and termination aspects of ns simulator,
- Definition of network nodes, links, queues and topology,
- Definition of agents and of applications,
- The nam visualisation tool,
- Tracing, and
- Random variables.

Some simple examples will be given that will enable us to do the first steps with the ns simulator.

2.1 Initialization and termination

An ns simulation starts with the command

```
set ns [new Simulator]
```

which is thus the first line in the tcl script. This line declares a new variable `ns` using the `set` command, you can call this variable as you wich, but in general people declares it as `ns` because it is an instance of the `Simulator` class, so an object. the code `[new Simulator]` is indeed the instantiation of the class `Simulator` using the reserved word `new`. So, using this new variable `ns` we can used all the methods of the class `Simulator`, that we will see after.

In order to have output files with data on the simulation (trace files) or files used for visualisation (nam files), we need to create the files using the "`open`" command:

```
#Open the Trace file
set tracefile1 [open out.tr w]
$ns trace-all $tracefile1

#Open the NAM trace file
set namfile [open out.nam w]
$ns namtrace-all $namfile
```

The above creates a data trace file called ”out.tr” and a nam visualisation trace file (for the NAM tool) called ”out.nam”. Within the tcl script, these files are not called explicitely by their names, but instead by pointers that are declared above and called ”tracefile1” and ”namfile” respectively. The first and fourth lines in the example are only comments, they are not simulation commands. Remark that they begins with a # symbol. The second line open the file ”out.tr” to be used for writting, declared with the letter ”w”. The third line uses a simulator method called `trace-all` that have as parameter the name of the file where the traces will go. With this simulator command we will trace all the events in a specifique format that we will explain after in this chapter.

The last line tells the simulator to record all simulation traces in NAM input format. It also gives the file name that the trace will be written to later by the command `$ns flush-trace` (see procedure ”`finish`” below). In our case, this will be the file pointed at by the pointer ”`$ namfile`”, i.e. the file ”out.tr”.

Note: the commands trace-all and namtrace-all may result in the creation of huge files. If we wish to save space, other trace commands should be used so as to create trace only a subset of the simulated events which may be directly needed. Such commands are described in Section 2.6.

The termination of the program is done using a ”`finish`” procedure.

```
#Define a 'finish' procedure
proc finish {} {
        global ns tracefile1 namfile
        $ns flush-trace
        close $tracefile1
        close $namfile
        exec nam out.nam &
        exit 0
}
```

The word `proc` declares a procedure in this case called `finish` and without arguments. The word `global` is used to tell that we are using variables declared outside the procedure. The simulator method ”`flush-trace` will dump the traces on the respectives files. The tcl command ”`close` closes the trace files defined before and `exec` executes the nam program for visualisation. Remark that we pass the real name of the file of traces to nam and not the pointer `namfile` because it is un external command. The command `exit` will ends the application and return the number 0 as status to the system. Zero is the dafault for a clean exit. Other values can be used to say that is a exit because something fails.

At the end of the ns program we should call the procedure ”`finish`” and specify at what time the termination should occur. For example,

```
$ns at 125.0 "finish"
```

will be used to call ”`finish`” at time 125 sec. Indeed, the `at` method of the simulator allows us to schedule events explicitly.

The simulation can then begin using the command

```
$ns run
```

2.2 Definition of a network of links and nodes

The way to define a node is

```
set n0 [$ns node]
```

We created a node that is pointed by the variable n0. When we shall refer to that node in the script, we shall thus write $n0.

Once we define several nodes, we can define the links that connect them. An example of a definition of a link is:

```
$ns duplex-link $n0 $n2 10Mb 10ms DropTail
```

which means that nodes $n0 and $n2 are connected using a bi-directional link that has 10ms of propagation delay and a capacity of 10 Mb/sec for each direction.

To define a directional link instead of a bi-directional one, we should replace ”duplex-link” by ”simplex-link”.

In NS, an output queue of a node is implemented as a part of each link whose input is that node. The definition of the link then includes the way to handle overflow at that queue. In our case, if the buffer capacity of the output queue is exceeded then the last packet to arrive is dropped (DropTail option). Many alternative options exist, such as the RED (Random Early Discard) mechanism, the FQ (Fair Queueing), the DRR (Deficit Round Robin), the Stochastic Fair Queueing (SFQ) and the CBQ (which including a priority and a round-robin scheduler); we shall return later to the RED mechanism in more details.

Of course, we should also define the buffer capacity of the queue related to each link. An example would be:

```
#Set Queue Size of link (n0-n2) to 20
$ns queue-limit $n0 $n2 20
```

A simplex link has the form presented in Fig. 2.1. A queue overflow is implemented by sending dropped packets to a Null Agent. The TTL object computes the Time To Live parameter[1] for each received packet. A duplex link is constructed from two parallel simplex links.

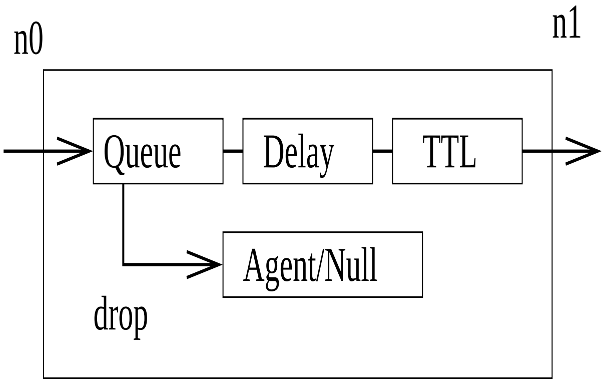

Figure 2.1: A simplex link

As an example of a simple network, consider the one depicted in Fig. 2.2; this network is defined through the script given in Table 2.1.

Note that we defined the buffer capacity corresponding to one link only (between n2 and n3). The queues corresponding to all other links have the default value of 50. This default value can be found at ns-default.tcl[2] in the command

```
Queue set limit_ 50
```

[1] packets have some associated tags which are updated in the network and that indicate how long they can still stay in the network before reaching the destination. When this time expires then the packet is dropped.

[2] in ns-allinone-2.XXX/ns-2.XXX/tcl/lib, where XXX stands for the version number, e.g. 1b9a

```
#Create six nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns simplex-link $n2 $n3 0.3Mb 100ms DropTail
$ns simplex-link $n3 $n2 0.3Mb 100ms DropTail
$ns duplex-link $n3 $n4 0.5Mb 40ms DropTail
$ns duplex-link $n3 $n5 0.5Mb 30ms DropTail

#Set Queue Size of link (n2-n3) to 20
$ns queue-limit $n2 $n3 20
```

Table 2.1: Defining nodes, links and assigning queue size

How could we find this default? by first checking the file ns-lib.tcl where we find the queue-limit procedure

```
Simulator instproc queue-limit { n1 n2 limit } {
        $self instvar link_
        [$link_([$n1 id]:[$n2 id]) queue] set limit_ $limit
}
```

in which we see that the queue limit is indeed a method of the simulator that needs three parameters: the two nodes that define the link and the queue limit. There we see that the limit number is given by the variable `limit_`.

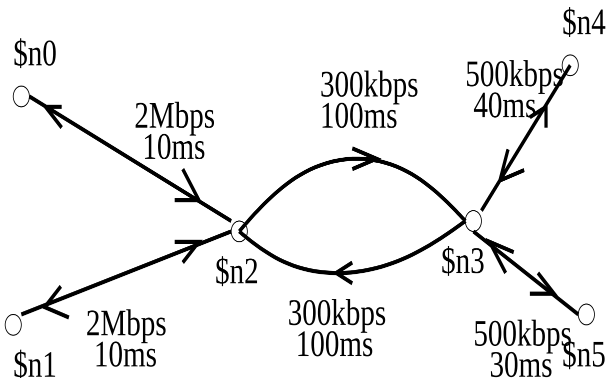

Figure 2.2: Example of a simple network

```
#Setup a TCP connection
set tcp [new Agent/TCP]
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n4 $sink
$ns connect $tcp $sink
$tcp set fid_ 1
$tcp set packetSize_ 552

#Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
```

Table 2.2: The definition of an FTP application using a TCP agent

2.3 Agents and applications

Having defined the topology (nodes and links) we should now make traffic flow through them. To that end, we need to define routing (in particular sources, destinations) the agents (protocoles) and applications that use them.

In the previous example, we may wish to run an FTP (File Transfer Protocol)[?] application between node `$n0` and `$n4`, and a CBR (Constant Bit Rate) application between node `$n1` and `$n5`. The Internet protocol used by FTP is TCP/IP (TCP for Transport Control Protocol/Internet Protocol) and the one used by CBR is UDP (User Datagram Protocol). We should first define in Table 2.2 a TCP agent between the source node `$n0` and the destination node `$n4` and then the FTP application that uses it. We then define in Table 2.3 the UDP agent between the source node `$n1` and the destination node `$n5` and the CBR application that uses it.

2.3.1 FTP over TCP

TCP is a dynamic reliable congestion control protocol which will be explained in details in Chapter 4. It uses acknowledgements created by the destination to know whether packets are well received; lost packets are interpreted as congestion signals, why TCP thus requires bidirectional links in order for the acknowledgements to return to the source.

There are a number variants of the TCP protocol, such as Tahoe, Reno, Newreno, Vegas. The type of agent appears in the first line:

```
 set tcp [new Agent/TCP]
```

This commands also gives a pointer called "`tcp`" here to the TCP agent, which is an object in NS.

The command `$ns attach-agent $n0 $tcp` defines the source node of the TCP connection. The command

```
set sink [new Agent/TCPSink]
```

defines the behavior of the destination node of TCP and assigns to it a pointer called sink. We note that in TCP the destination node has an active role in the protocol of generating acknowledgements in order to guarantee that all packets arrive at the destination.

```
#Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n5 $null
$ns connect $udp $null
$udp set fid_ 2

#Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set packetSize_ 1000
$cbr set rate_ 0.01Mb
$cbr set random_ false
```

Table 2.3: The definition of a CBR application using a UDP agent

The command `$ns attach-agent $n4 $sink` defines the destination node. The command `$ns connect $tcp $sink` finally makes the TCP connection between the source and destination nodes.

TCP has many parameters with initial fixed defaults values that can be changed if mentionned explicitly. For example, the default TCP packet size has a size of 1000 bytes. This can be changed to another value, say 552 bytes, using the command `$tcp set packetSize_ 552`.

When we have several flows, we may wish to distinguish them so that we can identify them with different colors in the visualisation part. This is done by the command `$tcp set fid_ 1` that assigns to the TCP connection a flow identification of "1"; we shall later give the flow identification of "2" to the UDP connection.

Once the TCP connection is defined, the FTP application is defined over it. This is done in the last three lines in Table 2.2.

Note that both the TCP agent as well as the FTP application are given pointers: we called the one for the TCP agent "`tcp`" (but could have used any other name) and the one for FTP we called "`ftp`".

2.3.2 CBR over UDP

Next we define the UDP connection and the CBR application over it, see Table 2.3. A UDP source (Agent/UDP) and destination (Agent/Null) is defined in a similar way as in the case of TCP. For the CBR application that uses UDP, the table 2.3 shows also how to define the transmission rate and packet size.

Instead of defining the rate, in the command `$cbr set rate_ 0.01Mb`, one can define the time interval between transmission of packets using the command

```
$cbr set interval_ 0.005
```

Other characteristics of CBR are `random_` which is a flag indicating whether or not to introduce random "noise" in the scheduled transmission times. It is "off" by default, and can be set to be "on" by typing

```
$cbr set random_ 1
```

The packet size can be set to some value (in bytes) using

```
$cbr set packetSize_ <packet size>
```

2.3.3 UDP with other traffic sources

We may simulate other types of traffic applications that use the UDP protocol: the exponential on-off traffic source, the Pareto on-off source, and a trace driven source. The exponential and Pareto sources are declared, respectively, using

```
set source [new Application/Traffic/Exponential]
set source [new Application/Traffic/Pareto]
```

These sources take as parameters `packetSize_` (in bytes), `burst_time_` which defines the average "on" time, `idle_time_` which defines the average "off" time, and `rate_` which determines the transmission rate during the "on" periods. In the Pareto On/Off source we also define the "shape" paremeter `shape_`. An example of a Pareto On/Off is given by:

```
set source [new Application/Traffic/Pareto]
$source set packetSize_ 500
$source set burst_time_ 200ms
$source set idle_time_ 400ms
$source set rate_ 100k
$source set shape_ 1.5
```

(For a discussion on random variables, see Section 2.7.)

The trace driven application is defined as follows. We first declare the trace file:

```
set tracefile [new Tracefile]
$tracefile filename <file>
```

Then, we define the application to be trace driven and attach it to that file:

```
set src [new Application/Traffic/Trace]
$src attach-tracefile $tracefile
```

The file should be in binary format and contain inter-packet time in msec and packet size in bytes.

2.4 Scheduling events

NS is a discrete event based simulation. The Tcl script defines when event should occur. The initializing command `set ns [new Simulator]` creates an event scheduler, and events are then scheduled using the format:

```
$ns at <time> <event>
```

The scheduler is started when running ns, i.e. through the command `$ns run`.

In our simple example, we should schedule the beginning and end of the FTP and the CBR applications. This can be done through the following commands:

```
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 124.0 "$ftp stop"
$ns at 124.5 "$cbr stop"
```

Thus the FTP will be active during from time 1.0 till 124.0 and the CBR will be active during from time 0.1 till 124.5 (all units are in seconds).

We are now ready to run the whole simulation. If our commands were written in a file called "ex1.tcl" (see table 2.4) we have to simply type "`ns ex1.tcl`".

Note: in Table 2.4 we added at the end another procedure that writes an output file with the instantaneous sizes of the window of TCP at time intervals of 0.1 sec. In the example, the name of the output file is "WinFile". The procedure is a recursive one, after each 0.1 sec it calls itself again. It passes as parameter the TCP source and the file to which we wish to write the output.

```
set ns [new Simulator]

#Define different colors for data flows (for NAM)
$ns color 1 Blue
$ns color 2 Red

#Open the Trace files
set tracefile1 [open out.tr w]
set winfile [open WinFile w]
$ns trace-all $tracefile1

#Open the NAM trace file
set namfile [open out.nam w]
$ns namtrace-all $namfile

#Define a ’finish’ procedure
proc finish {} {
        global ns tracefile1 namfile
        $ns flush-trace
        close $tracefile1
        close $namfile
        exec nam out.nam &
        exit 0
}

#Create six nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns simplex-link $n2 $n3 0.3Mb 100ms DropTail
$ns simplex-link $n3 $n2 0.3Mb 100ms DropTail
$ns duplex-link $n3 $n4 0.5Mb 40ms DropTail
$ns duplex-link $n3 $n5 0.5Mb 30ms DropTail

#Give node position (for NAM)
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns simplex-link-op $n2 $n3 orient right
$ns simplex-link-op $n3 $n2 orient left
$ns duplex-link-op $n3 $n4 orient right-up
$ns duplex-link-op $n3 $n5 orient right-down
```

```
#Set Queue Size of link (n2-n3) to 20
$ns queue-limit $n2 $n3 20

#Setup a TCP connection
set tcp [new Agent/TCP]
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n4 $sink
$ns connect $tcp $sink
$tcp set fid_ 1
$tcp set packetSize_ 552

#Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp

#Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n5 $null
$ns connect $udp $null
$udp set fid_ 2

#Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set packetSize_ 1000
$cbr set rate_ 0.01Mb
$cbr set random_ false

$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 124.0 "$ftp stop"
$ns at 124.5 "$cbr stop"

# Procedure for plotting window size.  Gets as arguments the name
# of the tcp source node (called "tcpSource") and of output file.
proc plotWindow {tcpSource file} {
global ns
set time 0.1
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
puts $file "$now $cwnd"
$ns at [expr $now+$time] "plotWindow $tcpSource $file"
}
$ns at 0.1 "plotWindow $tcp $winfile"

$ns at 125.0 "finish"
$ns run
```

Table 2.4: ex1.tcl script file

2.5 Visualisation: using nam

When we run the example 2.1, the visualisation tool nam will display a 6 nodes network. The location of the nodes could have been chosen at random. In order to reproduce the initial location of the nodes as in Fig. 2.2, we added to the tcl script the following lines:

```
#Give node position (for NAM)
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns simplex-link-op $n2 $n3 orient right
$ns simplex-link-op $n3 $n2 orient left
$ns duplex-link-op $n3 $n4 orient right-up
$ns duplex-link-op $n3 $n5 orient right-down
```

Note: if a random location of nodes is chosen and it is not satisfactory, one can press on the "re-layout" button and then another random location is chosen. One can also edit the location by clicking at the Edit/View button, and then "draggind" each node to its required location (with the help of the mouse).

We note that the nam display shows us with animation the CBR packets (that flow from node 1 to 5) in red, and TCP packets (flowing from node 0 to 4) in blue. TCP ACKs (acknowledgements) that go in the reverse directions are also in blue but are much shorter, since an ACK has a size of 40 bytes whereas the TCP packet is of size 552 bytes. To obtain the colors, we had to define in the beginning of our script ex1.tcl

```
$ns color 1 Blue
$ns color 2 Red
```

Note that if we already have a nam file, we do not have to run ns in order to view it, but instead type directly the command `nam <file name>`.

"Snapshots" from the nam visualisations can be printed (into a printer or into a file) by going into the "File" option in the top menu.

Other things that can be done in NAM:

- Coloring nodes; for example if n0 is to appear in red we write `$n0 color red`.
- Shape of nodes: by default they are round, but can appear differently. For example one can type `$n1 shape box` (or instead of "box" one can use "hexagon" or "circle").
- Coloring links: type for example

  ```
  $ns duplex-link-op $n0 $n2 color "green"
  ```

- Adding and removing marks: We can mark a node at a given time (for example at the same time as we activated some traffic source at that time). For example, we can type:

  ```
  $ns at 2.0 "$n3 add-mark m3 blue box"
  $ns at 30.0 "$n3 delete-mark m3"
  ```

  This results in a blue mark that surrounds the node 3 during the time interval [2,30].

- Adding labels: a label can appear on the screen from a given time onwards, e.g. for giving the label "active node" to a node n3 from time 1.2, type:

  ```
  $ns at 1.2 "$n3 label \"active node\""
  ```

and to give a the label ”TCP input link” to link n0-n2 type

```
$ns duplex-link-op $n0 $n2 label "TCP input link"
```

- Adding text: at the bottom frame of the NAM window one can make text appear at a given time. This can be used to describe some event that is scheduled at that time. An example is

```
$ns at 5 "$ns trace-annotate \"packet drop\""
```

- One may further add in NAM a monitoring of the queue size. For example, to monitor the input queue of the link n2-n3, one type

```
$ns simplex-link-op $n2 $n3 queuePos 0.5
```

(All the examples refer to objects defined in ex1.tcl.)

The graphic interface of NAM is shown in figure 2.3.

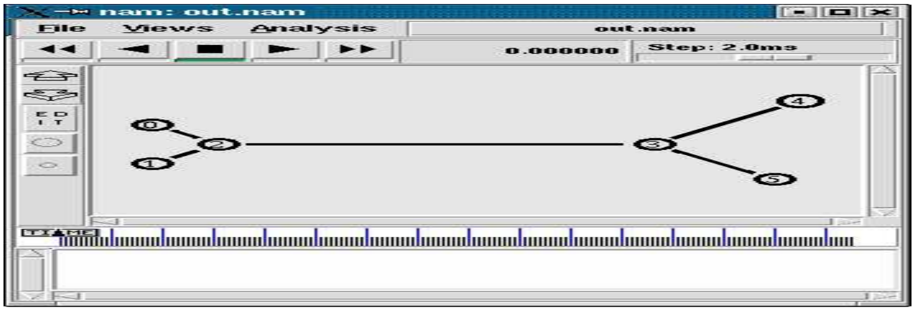

Figure 2.3: NAM graphic interface

2.6 Tracing

2.6.1 Tracing objects

NS simulation can produce both the visualisation trace (for NAM) as well as an ascii file trace corresponding to the events registered at the network.

When we use tracing (as mentionned in Section 2.1), ns inserts four objects in the link: EnqT, DeqT, RecvT and DrpT, as indicated in Fig. 2.4.

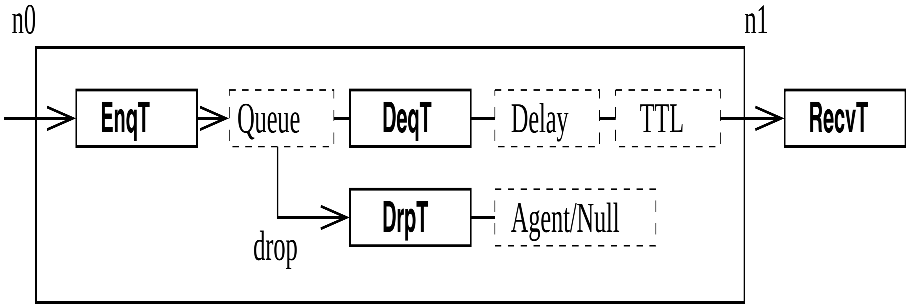

Figure 2.4: Tracing objects in a simplex link

EnqT registers information concerning a packet that arrives and is queued at the input queue of the link. If the packet overflows then information concerning the dropped packet are handled by DrpT. DeqT registers information at the instant the packet is dequed. Finally, RecvT gives us information about packets that have been received at the ouput of the link.

NS allows us to get more information than through the above tracing. One way is by using queue monitoring. This is described at the end of Section 4.3.

2.6.2 Structure of trace files

When tracing into an output ascii file, the trace is organized in 12 fields as follows in Fig. 2.5. The meanings of the fields are:

Event	Time	From node	To node	Pkt type	Pkt size	Flags	Fid	Src addr	Dst addr	Seq num	Pkt id

Figure 2.5: Fields appearing in a trace

1. The first field is the event type. It is given by one of four possible symbols $r, +, -, d$ which correspond respectively to receive (at the output of the link), enqueued, dequeued and dropped.

2. The second field gives the time at which the event occurs.

3. Gives the input node of the link at which the event occurs.

4. Gives the output node of the link at which the event occurs.

5. Gives the packet type (for example, CBR, or TCP. The type corresponds to the name that we gave to those applications. For example, the TCP application in Table 2.2 is called "tcp".

6. Gives the packet size.

7. Some flags (that we shall see later).

8. This is the flow id (fid) of IPv6 that a user can set for each flow at the input OTcl script. One can further use this field for analysis purposes; it is also used when specifying stream color for the NAM display.

9. This is the source address given in the form of "node.port".

10. This is the destination address, given in the same form.

11. This is the network layer protocol's packet sequence number. Even though UDP implementations in a real network do not use sequence number, ns keeps track of UDP packet sequence number for analysis purposes.

12. The last field shows the unique id of the packet.

As an example, consider the first lines of the trace produced by running the script ex1.tcl given in Table 2.4.

```
+ 0.1 1 2 cbr 1000 ------- 2 1.0 5.0 0 0
- 0.1 1 2 cbr 1000 ------- 2 1.0 5.0 0 0
r 0.114 1 2 cbr 1000 ------- 2 1.0 5.0 0 0
+ 0.114 2 3 cbr 1000 ------- 2 1.0 5.0 0 0
- 0.114 2 3 cbr 1000 ------- 2 1.0 5.0 0 0
r 0.240667 2 3 cbr 1000 ------- 2 1.0 5.0 0 0
+ 0.240667 3 5 cbr 1000 ------- 2 1.0 5.0 0 0
- 0.240667 3 5 cbr 1000 ------- 2 1.0 5.0 0 0
r 0.286667 3 5 cbr 1000 ------- 2 1.0 5.0 0 0
+ 0.9 1 2 cbr 1000 ------- 2 1.0 5.0 1 1
- 0.9 1 2 cbr 1000 ------- 2 1.0 5.0 1 1
r 0.914 1 2 cbr 1000 ------- 2 1.0 5.0 1 1
+ 0.914 2 3 cbr 1000 ------- 2 1.0 5.0 1 1
- 0.914 2 3 cbr 1000 ------- 2 1.0 5.0 1 1
+ 1 0 2 tcp 40 ------- 1 0.0 4.0 0 2
- 1 0 2 tcp 40 ------- 1 0.0 4.0 0 2
r 1.01016 0 2 tcp 40 ------- 1 0.0 4.0 0 2
+ 1.01016 2 3 tcp 40 ------- 1 0.0 4.0 0 2
- 1.01016 2 3 tcp 40 ------- 1 0.0 4.0 0 2
r 1.040667 2 3 cbr 1000 ------- 2 1.0 5.0 1 1
+ 1.040667 3 5 cbr 1000 ------- 2 1.0 5.0 1 1
- 1.040667 3 5 cbr 1000 ------- 2 1.0 5.0 1 1
r 1.086667 3 5 cbr 1000 ------- 2 1.0 5.0 1 1
r 1.111227 2 3 tcp 40 ------- 1 0.0 4.0 0 2
+ 1.111227 3 4 tcp 40 ------- 1 0.0 4.0 0 2
- 1.111227 3 4 tcp 40 ------- 1 0.0 4.0 0 2
r 1.151867 3 4 tcp 40 ------- 1 0.0 4.0 0 2
+ 1.251867 4 3 ack 40 ------- 1 4.0 0.0 0 3
- 1.251867 4 3 ack 40 ------- 1 4.0 0.0 0 3
+ 1.251867 4 3 ack 40 ------- 1 4.0 0.0 0 3
- 1.251867 4 3 ack 40 ------- 1 4.0 0.0 0 3
r 1.292507 4 3 ack 40 ------- 1 4.0 0.0 0 3
+ 1.292507 3 2 ack 40 ------- 1 4.0 0.0 0 3
- 1.292507 3 2 ack 40 ------- 1 4.0 0.0 0 3
r 1.393573 3 2 ack 40 ------- 1 4.0 0.0 0 3
+ 1.393573 2 0 ack 40 ------- 1 4.0 0.0 0 3
- 1.393573 2 0 ack 40 ------- 1 4.0 0.0 0 3
r 1.403733 2 0 ack 40 ------- 1 4.0 0.0 0 3
+ 1.403733 0 2 tcp 552 ------- 1 0.0 4.0 1 4
- 1.403733 0 2 tcp 552 ------- 1 0.0 4.0 1 4
+ 1.403733 0 2 tcp 552 ------- 1 0.0 4.0 2 5
- 1.405941 0 2 tcp 552 ------- 1 0.0 4.0 2 5
r 1.415941 0 2 tcp 552 ------- 1 0.0 4.0 1 4
```

Table 2.5: First lines of the trace file "out.tr" produced by ex1.tcl

2.6.3 Tracing a subset of events

In Section 2.1 we already mentioned how to trace all simulated events. We now indicate ways to trace only a subset of these.

The first way to do so is by replacing the command `$ns trace-all <filename>` by the command `$ns trace-queue`. For example, we can type

```
$ns trace-queue $n2 $n3 $file1
```

which will result in an output trace file that contains only events that occured over the link between nodes `n2` and `n3` (these are nodes defined in Table 2.1). (A similar command can be used for the nam trace, using `namtrace-queue` instead of `trace-queue`.) The `trace-queue` line should appear of course after the definition of the links, i.e. after the script part of Table 2.1.

It is also possible to filter events using unix commands within the tcl script. This will be discussed in Section 3.6.

2.7 Random variables

Random variables (RVs) with different distributions can be created in ns. Due to its import role in traffic modeling and in network simulation we briefly recall the definitions and moments of main random variables in Appendix 11. For more background, one can consult e.g. `http://www.xycoon.com/`.

2.7.1 Seeds and generators

In addition to its distribution, there are other aspects that we need to be concerned of when simulating a random variable:

- Do we want to obtain the same value of the random variable when running again the simulation (possibly varying some other parameters of simulations)? this would allow us to compare directly, for a single random set of events, how the simulated results depend on some physical parameters (such as link delays or queue length).
- Often we need random variables to be independent of each other.

The generation of random variables uses a seed (which is some number that we write in the tcl script). The seed value of 0 results in the generation of a new random variable each time we run the simulation, so if we wish to have the same generated random variables for different simulations we would have to save the generated random variables. In contrast, if we use other seeds then each time we run the simulation, the same sequence of random variables that are generated in a simulation will be generated.

In ns, if we use different generators with the same seed and the same distribution, they will create the same values of random variables (unless the seed is zero). We shall see this in an example below.

2.7.2 Creating Random Variables

We first create a new generator and assign to it a seed, say 2, with the command

```
set MyRng1 [new RNG]
$MyRng2 seed 2
```

Then when actually creating a random variable, we have to define its distribution type and its parameters. We give several examples below: we create RVs with Pareto, Constant, Uniform, Exponential and HyperExponential distributions.

1. **Pareto Distribution.** A Pareto distributed RV, say $r1$ is constructed by specifying its expectation and its shape parameter β, the default values are 1.0 and 1.5 respectively.

   ```
   set r1 [new RandomVariable/Pareto]
   $r1 use-rng $MyRng
   $r1 set avg_ 10.0
   $r1 set shape_ 1.2
   ```

2. **Constant.** A degenerated random variable is the constant which equals to its value:

   ```
   set r2 [new RandomVariable/Constant]
   $r2 use-rng $MyRng
   $r2 set val_ 5.0
   ```

3. **Uniform distribution.** It is defined through the smallest and largest point in its support:

   ```
   set r3 [new RandomVariable/Uniform]
   $r3 use-rng $MyRng
   $r3 set min_ 0.0
   $r3 set max_ 10.0
   ```

4. **Exponential distribution.** It's defined through its average value:

   ```
   set r4 [new RandomVariable/Exponential]
   $r4 use-rng $MyRng
   $r4 set avg_ 5
   ```

5. **Hyperexponential distribution.** It is defined as follows:

   ```
   set r5 [new RandomVariable/HyperExponential]
   $r5 use-rng $MyRng
   $r5 set avg_ 1.0
   $r5 set cov_ 4.0
   ```

Next we present a small program (rv1.tcl) that tests Pareto distributed random variables with different seeds and generators but with the same Pareto distribution. It is given in Table 2.6. For each seed (of values 0, 1 and 2) and generator, we create a sequence of three random variables. The "`count`" variable is assigned the number of RVs that we create using the "`test`" for each seed and generator.

When we run this example we can observe that for seed 0, the two generators give different values of variables; we thus obtain 6 different values (three from each generator). For other seeds, a generator creates three different values, but these values do not depend on the generator: the nth value created by generator 1 is the same as the nth created by generator 2.

```
## Simple example demonstrating use of the RandomVariable class from tcl

set count 3

for {set i 0} {$i<3} {incr i} {

puts "===== i = $i "

set MyRng1 [new RNG]
$MyRng1 seed $i

set MyRng2 [new RNG]
$MyRng2 seed $i

set r1 [new RandomVariable/Pareto]
$r1 use-rng $MyRng1
$r1 set avg_ 10.0
$r1 set shape_ 1.2
puts stdout "Testing Pareto Distribution, avg = [$r1 set avg_] shape = [$r1 set shape_]"
$r1 test $count

set r2 [new RandomVariable/Pareto]
$r2 use-rng $MyRng2
$r2 set avg_ 10.0
$r2 set shape_ 1.2
puts stdout "Testing Pareto Distribution, avg = [$r2 set avg_] shape = [$r2 set shape_]"
$r2 test $count
}
```

Table 2.6: Testing Pareto distributed random variables with different seeds

Chapter 3

How to work with trace files

NS simulator can provide a lot of detailed data on events that occur at the network. If we wish to analyze the data we may need to extract relevant information from traces and to manipulate them.

One can of course write programs in any programming language that can handle data files.

Yet several tools that seem particularly adapted for these purposes already exist and are freely available under various operating systems (unix, linux, windows, etc.) All they require is to write short scripts that are interpreted and executed without need for compilation.

3.1 Processing data files with `awk`

The `awk` utility [?] allows us to do simple operations on data files such as averaging the values of a given column, summing or multiplying term by term between several columns, all data-reformatting tasks, etc.

In the two following examples we show how to take the average value of a given column in a file, and then to compute the standard deviation.

```
BEGIN { FS = "\t"} { nl++ } { s=s+$4} END {print "average:" s/nl}
```

Table 3.1: awk script for averaging the values in column 4 of a file

(Note: the `"\t"` should be used if columns are tabulated. If not then one should replace it by `" "`.)

```
BEGIN {FS="\t"}{ln++}{d=$4-t}{s2=s2+d*d} END {print "standev:" sqrt(s2/ln)}
```

Table 3.2: awk script for obtaining the standard deviation of column 4 of a file

To use the first script to compute the average of column four of a file named ”Out.ns” we should type in unix:

```
awk -f Average.awk Out.ns
```

We shall get as a result something like: `average : 29.397` for the average of column 4 (where the first colomn is considered as number 1).

To compute now the standard deviation of that column, we should type

```
awk -v t=29.397 -f StDev.awk Out.ns
```

which will give in response something like `standev : 33.2003` Note that in the above script, we have to copy the average value obtained from the previous script into the command that computes the standard deviation. This example shows how to pass parameters to a awk script.

Note that if we do not divide at the end of the first awk script (Table 3.1) by nl, we shall obtain simply the sum of entries of column 4 instead of their average.

A best way to obtain the average and the standard deviation is using arrays:

```
BEGIN { FS = "\t"} {val[nl]=$4} { nl++ } {s=s+$4} END {
  av=s/nl
  for (i in val) {
    d=val[i]-av
    s2=s2+d*d
  }
  print "average: " av " standev " sqrt(s2/nl)}
```

Table 3.3: Average and Standard Deviation awk script

The next examples takes as input a file with 15 columns (0 to 14). It then creates as output 5 columns, where the first contains column no. 1 of the original file, and columns 2 to 5 are the sum of columbs 3-4, 6-8, 9-11 and 12-14, respectively (12-14 correspond to the three last columns in the original file).

```
BEGIN {FS="\t"}{l1=$3+$4+$5}{l2=$6+$7+$8}{d1=$9+$10+$11} \
{d2=$12+$13+$14}{print $1"\t" l1"\t" l2"\t" d1"\t" d2 } END {}
```

Table 3.4: A cut and paste columns awk script

The use of this script can be as follows:

```
awk -f suma.awk Conn4.tr > outfile
```

The original file here is `Conn4.tr` and the output is written into a file called `oufile`.

3.2 Using grep

The `grep` command in unix allows us to "filter" a file. We can create a new file which consists of only those lines from the original file that contain a given character sequence. For example, output traces in `ns` may contain all types packets that go through all links and we may be interested only in the data concerning tcp packets that went from node 0 to node 2. If lines concerning such events contain the string " `0 2 tcp` " then all we have to do is type

```
grep " 0 2 tcp " tr1.tr > tr2.tr
```

where "tr1.tr" is the original trace and "tr2.tr" is the new file. If we wish to obtain a file containing all lines of tr1.tr that begin with the letter r, we should type

```
grep "^r" tr1.tr > tr2.tr
```

If we wish to make a file of all the lines that begin with "s" and have later "tcp 1020" we should type

```
grep "^s" simple.tr | grep "tcp 1020" > tr3.tr
```

3.3 Processing data files with `perl`

PERL stands for "Practical Extraction and Report Language". Perl[?] allows easy filtering and processing of ASCII data files in unix. This language was created by Larry Wall with the main idea of simplify the task of system administration. Perl has evolved a lot and nowadays is a general purpouse language and one of the most used tools for web an Internet data managing.

Perl is a interpreted language who has many uses, but is mainly addresed to the search, extraction and report. Some advantages of using Perl are:

- Easily implementation of small programs to be used as filters, for extracting information from text files.
- It can be used in many OSs without changing the code.
- Maintaining and debugging of Perl scripts are more simple than programs in other specifique languages.
- Perl is very popular, so there exist a lot of scripts gnu on the web.

We present in this Section some useful perl scripts.

The first example given in Table 3.5 computes dynamically the throughput of TCP connections. The program averages the throughput over periods defined by a parameter called "granularity". As input it takes three arguments: the name of a trace file (e.g. out.tr), the node at which we wish to check the throughput of TCP, and the granularity.

```
# type: perl throughput.pl <trace file> <required node> <granlarity>  >   file

$infile=$ARGV[0];
$tonode=$ARGV[1];
$granularity=$ARGV[2];

#we compute how many bytes were transmitted during time interval specified
#by granularity parameter in seconds
$sum=0;
$clock=0;

      open (DATA,"<$infile")
        || die "Can't open $infile $!";

    while (<DATA>) {
             @x = split(' ');

#column 1 is time
if ($x[1]-$clock <= $granularity)
{
#checking if the event corresponds to a reception
if ($x[0] eq 'r')
{
#checking if the destination corresponds to 1st argument
if ($x[3] eq $tonode)
{
#checking if the packet type is TCP
if ($x[4] eq 'tcp')
{
    $sum=$sum+$x[5];
}
}
}
}
else
{   $throughput=$sum/$granularity;
    print STDOUT "$x[1] $throughput\n";
    $clock=$clock+$granularity;
    $sum=0;
}
}
   $throughput=$sum/$granularity;
    print STDOUT "$x[1] $throughput\n";
    $clock=$clock+$granularity;
    $sum=0;

    close DATA;
exit(0);
```

Table 3.5: Perl program for computing the throughput

3.4 Plotting with gnuplot

Gnuplot is a widely available free software both for unix/linux as well as windows operating systems.

Gnuplot has a help command that can be used to learn details of its operation.

The simplest way to use gnuplot is to type "plot $< fn >$", where the file (whose name we write as fn) has two columns representing the x and y values of points. Points can be joined by a line of different styles by writing commands like:

```
plot 'fn' w lines 1
```

(different numbers can be given instead of "1") that produce. different line styles). Alternatively, one may use different type of points by writing commands of the form

```
plot 'fn' w points 9
```

(again, several types of points can be depicted depending on the number that appears after "points").

Some other features of gnuplot: consider for example the following commands:

```
set size 0.6,0.6
set pointsize 3
set key 100,8
set xrange [90.0:120.0]
plot 'fn1' w lines 1, 'fn2' w lines 8, 'fn3' with points 9
```

- Line 1 will produce a smaller size curve that the default.
- Line 2 will produce points that are larger than the defaults. (In both lines, other numbers can be used).
- Line 4 restricts the range of the x axis to the interval 90-120.
- Line 5 superimposes three curves in a single figure, obtained from three different files: fn1, fn2, fn3.
- Line 3 tells gnuplot where exactly to put the 'key'; the latter is the legend part in the figure describing the plotted objects. In particular, it gives for each plotted object the line type or point type that is used. Instead of an exact poistion, one could use the keywords 'left', 'right', 'top', 'bottom', 'outside' and 'below', e.g.

  ```
  set key below
  ```

  (which sets the key below the graph), or simply "set nokey" which disables the key completely. Note that the default name of each object that appears in the key is simply its corresponding file name. If we wish to give an object a title other than the file name we have to explicit this in the "plot" command, for example:

  ```
  plot 'fn1' t "expectation" w lines 1, 'fn2' t "variance" w lines 2
  ```

  Here, the names "expectation" and "variance" will appear in the key.

If the same sequence of commands are to be used several times, one can write them into a file, say having the name "g1.com", and then simply load the file each time one wishes to use it:

```
load 'g1.com'
```

gnuplot can be used to extract some column from a multicolumn file. This is done as follows

```
plot 'queue.tr' using 1:($4/1000) t "kbytes" w lines 1, \
    'queue.tr' using 1:5 t "packets" w lines 2
```

which means plotting first a curve using column 1 of the file "queue.tr' as the x axis and 4 divided by 1000 as the y axis, and then plotting on the same curve the column 5 for the y axis using the same column 1 for the x axis. Note: this order between "using", "t" and "lines" is important!

3.5 Plotting with xgraph

Xgraph is a plotting utility that is provided by ns. (Sometimes it needs separate compiling using `./configure` and then `make` when at the directory xgraph. Also, sometimes this does not work with the xgraph that arrives with the whole ns single package, and it can then be downloaded and installed separately). Note that it allows to create postscript, Tgif files, and others, by clicking on the botton "Hdcpy". It can be invoced within the tcl command which thus results in an immediate display after the end of the simulation.

As input, the xgraph command expects one or more ascii files containing each $x-y$ data point pair perl line. For example, `xgraph f1 f2` will print on the same figure the files f1 and f2.

Some options in xgraph are:

- Title: use `-t "title"`.
- Size: -geometry `xsize x ysize`.
- Title for axis: `-x "xtitle"` (for the title of the x axis) and `-y "ytitle"` (for the title of the y axis).
- Color of text and grid: with the flag `-v`.

An example of a command would be

```
xgraph f1 f2 -geometry 800x400 -t "Loss rates" -x "time" -y "Lost packets"
```

3.6 Extracting information within a tcl script

It is possible to integrate unix commands such as "grep" and "awk" already into the tcl scripts, so as to start the processing of data while writing the file. For example Another way to limit the tracing files (or in general, to process them online while they are being written) is to use linux commands related to file processing within the tcl command that opens the required file.

For example, we may replace the command `$set file1 [open out.tr w]` (that we had in the beginning of the script ex1.tcl, see Table 2.4) by the command

```
set file1 [open "| grep \"tcp\" > out.tr" w]
```

This will result in filtering the lines written to the file "out.tr" and leaving only those that contain the word "tcp".

Chapter 4

Description and simulation of TCP/IP

TCP (*Transport Control Protocol*) is the transport protocol that is responsible for the transmission of around 90% of the Internet traffic, and understanding TCP is thus crucial for dimensionning the Internet. Although TCP is already largely deployed, it continues to envolve. The IETF [?] (*Internet Engineering Task Force*) is the main standardizing organization that is concerns with TCP. Unlike some other standization organizations (ITU or ATM forum), all standards are free and available on-line.

In the first section we describe the operation of TCP. Then in the subsequent chapters we present several ns scripts that illustrate the analysis of TCP through simulations.

4.1 Description of TCP

4.1.1 Objectives of TCP and window flow control

TCP has several objectives:

- Adapt the transmission rate of packets to the available bandwidth,
- Avoid congestion at the network,
- Create a reliable connection by retransmitting lost packets.

In order to control the transmission rate, the number of packets that have not yet been received (or more precisely, for which the source has not obtained the information of good reception) is bounded by a parameter called a congestion window. We denote it by W, but it is called *cwnd* in the TCP code. This means that the source is obliged to wait and stop transmissions the number of packets that it had transmitted and that have not been "acknowledged" reaches W. In order to acknowledge packets and thus to be able to retransmit lost packets, each transmitted packet has a sequence number.

4.1.2 Acknowledgements

The objectives of Acknowledgements (ACKs) are:

- Regulate the transmission rate of TCP, ensuring that packets can be transmitted only when other have left the network.

- Render the connection reliable by transmitting to the source information it needs so as to retransmit packets that have not reached the destination.

How does the destination know that a packet is missing?

How do we know that a packet is lost?

What information does the ACK carry along?

The ACK tells the source what is the sequence number of the packet it expects. This is illustrated by the following example. Suppose packets 1,2,...,6 have reached the destination (in order). When packet 6 arrives, the destination sends an ACK to say it expects packet number 7. If packet 7 arrives, the destination requests the number 8. Suppose packet 8 is lost and packet 9 arrives well. At that time, the destination sends an ACK called "repeated ACK" as it tells the source that it awaits packet 8. The information carried by the ACK is thus the same as the one carried by the previous ACK.

This method is called "implicit ACK". It is robust under losses of ACKs. Indeed, assume that the ACK saying that the destination waits for packet 5, is lost. When the next ACK arrives, saying it awaits packet 6, the source knows that the destination has received packet 5, so the information sent by the lost ACK is deduced from the next ACK.

A TCP packet is considered lost if

- Three repeated ACKs for the same packets arrive at the source[1], or
- When a packet is transmitted, there is a timer that starts counting. If its ACK does not arrive within a period T_0, there is a "Time-Out" and the packet is considered to be lost.

Retransmitting after three dupplicated ACKS is called "fast retransmit".

How to choose T_0? The source has an estimation of the average round trip time RTT, which is the time necessary for a packet to reach the destination plus the time for its ACK to reach the source. It also has an estimation of the variability of RTT. T_0 is determined as follows:

$$T_0 = \overline{RTT} + 4D$$

Where $\overline{RTT}$ is the current estimation of RTT, and D is the estimation of the variability of RTT. In order to estimate RTT, we measure the difference M between the transmission time of a packet and the time its ACK returns. Then we compute

$$\overline{RTT} \leftarrow a \times \overline{RTT} + (1-a)M,$$

$$D \leftarrow aD + (1-a)|\overline{RTT} - M|.$$

In order to decrease the number of ACKs in the system, TCP frequently uses the "delayed ACK" option where an ACK is transmitted for only every d packets that reach the destination. The standard value of d is 2 (see RFC 1122). However, delaying an ACK till $d > 1$ packets are received could result in a deadlock in case that the window size is one! Theferore, if the first packet (of an expected group of d packets) arrives at the destination, then after some time interval (typically 100ms) if d packets have not yet arrived, then an acknowledgement is generated without further waiting.

[1] One does not consider a single repeated ACK as a loss indication since dupplicated ACKS could be due to resequencing of packets at the Internet

4.1.3 Dynamic congestion window

Since the beginnings of the eighties, during several years, TCP had a fixed congestion window. Networks at that time were unstable, there were many losses, large and severe congestion periods, during which the throughputs decreased substantially, there were many packet retransmissions and large delays. In order to solve this problem, Van Jacobson proposed [25] to use a dynamic congestion window: its size can vary according to the network state. The basic idea is as follows: When the window is small, it can grow rapidly, and when it reaches large values it can only grow slowly. When congestion is detected, the window size decrease drastically. This dynamic mechanism allows to resolve congestion rapidly and yet use efficiently the network's bandwidth.

More precisely, define a threshold W_{th} called "slow start threshold" which represents our estimation of the network capacity. The window starts at a value of one. It thus transmits a single packet. When its ACK returns, we can transmit two packets. For each ACK of these two packets, the window increases by one, so that when the ACKs of these two packets return we transmit four packets. We see that there is an exponential growth of the window. This phase is called "slow start". It is called so because in spight of the rapid growth, it is slower than if we had started directly with a value of $W = W_{th}$.

When $W = W_{th}$, we pass to a second phase called "congestion avoidance", where the window W increases by $\lfloor 1/W \rfloor$ which each ACK that returns. After transmitting W packets, W increases by 1. If we transmit the W packets at t, then at time $t+RTT$ we transmit $W+1$, and at $t+2RTT$ we transmit $W+2$, etc... We see that the window growth is linear.

4.1.4 Losses and a dynamic threshold W_{th}

Not only W is dynamic, W_{th} is too. It is fixed in TCP to half the value of W when there has been a packet loss.

There are several variants of TCP. In the first variant, called "Tahoe", whenever a loss is detected then the window reduces to the value of 1 and a slow-start phase begins. This is a drastic decrease of the window size and thus of the transmission rate.

In the other mostly used variants, called Reno or New-Reno, the window drops to 1 only if the loss is detected through a time-out. When a loss is detected through repeated ACKs then the congestion window drops by half. Slow start is not initiated and we remain in the "congestion avoidance" phase.

4.1.5 Initiating a connection

To initiate a TCP connection, the source sends a "sync" packet of 40 bytes to the destination. The destination then sends an ACK (also 40 packets long, called "sync ACK"). When receiving this ACK, TCP can start sending data. Note: if either of these packets is lost then after a time-out expires (usually 3 or 6 secs) then it is retransmitted. When a retransmitted packet is lost, the time-out duration doubles and the packet is sent again.

4.2 Tracing and analysis of Example ex1.tcl

Let's run the perl program "throughput.pl" (Table 3.5) on the trace file out.tr generated by the ex1.tcl script (see Table 2.4). We have to type:

```
perl throughput.pl out.tr 4 1 > thp
```

We obtain an output file with the averaged received throughput of TCP (in bytes per second) as a function of time, where in our case, each 1 second, a new value of the throughput is obtained. This output file can be displayed using gnuplot by typing:

```
gnuplot
set size 0.4,0.4
set key 60,15000
plot 'thp' w lines 1
```

The result is given in Fig. 4.1.

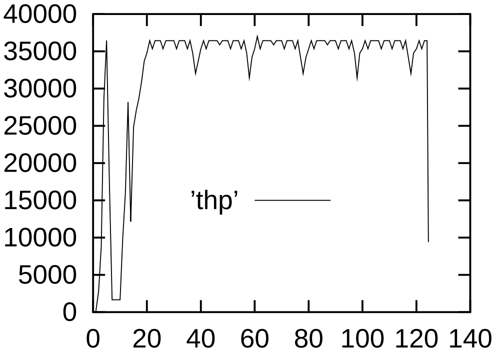

Figure 4.1: Throughput of TCP connection

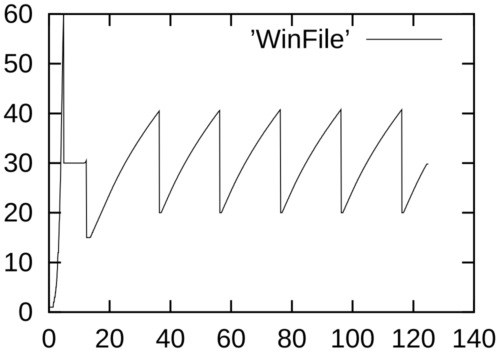

Figure 4.2: Window size of TCP connection

In order to understand better the behavior of the system, we also plot the window size (Fig. 4.2). This is the file "WinFile" created by running ex1.tcl.

We see that from time 20 onwards a steady-state cyclic regime of TCP is attained: TCP is always in congestion avoidance, and its window size increases (almost linearly) until congestion occurs.

Before time 20, we see a transient behavior in which TCP is in the slow-start phase.

At time 4.2 there are losses at the slow start phase. The window halves, whereas the throughput becomes close to zero. How can we explain that? The reason is that at time 4.2 there is a time-out, so although the window is of size 30 (packets), there are no transmissions. At time 11 there are again losses during a slow-start phase.

4.3 TCP over noisy links and queue monitoring

In the previous examples losses were due to congestion. In pratice, losses may also be caused from noisy links. This is especially true in the case of radio links, e.g. in cellular phones or in satellite links. A link may become in fact completely disconnected for some period. We shall see this aspect later, in Section 5.1. Or it may suffer from occasional interferences (due to shadowing, fading etc) that cause packets to contain errors and then to be dropped. In this section we shall show how to introduce the simplest error model: we assume that packets are dropped on the *forward* link according independently with some fixed constant probability.

This link error model, that will be introduced to the link connecting nodes n3 and n2 (in the example in Figure 4.3), is created as follows:

```
#Set error model on link n2 to n3.
set loss_module [new ErrorModel]
$loss_module set rate_ 0.2
```

```
$loss_module ranvar [new RandomVariable/Uniform]
$loss_module drop-target [new Agent/Null]
$ns lossmodel $loss_module $n2 $n3
```

The command `$loss_module set rate_ 0.2` determines a loss rate of 20% of the packets. It uses a generator of a uniformly distributed random variable, which is declared in the next line. The last line determines which link will be affected.

As an example of a TCP connection that shares a noisy bottle link with a UDP connection, we consider the network depicted in Figure 4.3.

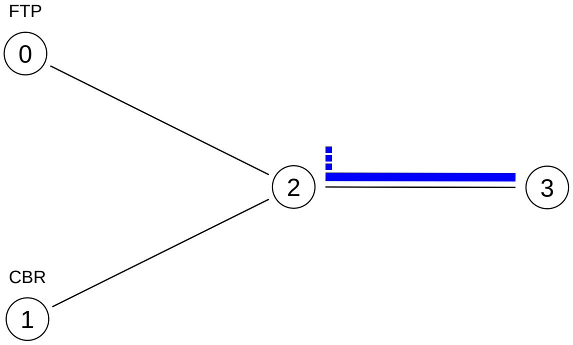

Figure 4.3: Example rdrop.tcl

Queue monitoring An important object of ns is the monitor-queue. It allows to collect much useful information on queue length, on the arrivals, departures and losses. To implement a queue monitor between nodes n2 and n3, we type:

```
set qmon [$ns monitor-queue $n2 $n3 [open qm.out w] 0.1];
[$ns link $n2 $n3] queue-sample-timeout; # [$ns link $n2 $n3] start-tracing
```

The "monitor-queue" object has 4 arguments: the first two defines the link where the queue is located, the third is the output trace file and the last says how frequently we wish to monitor the queue. In our case, the queue at the input of node n2-n3 is monitored every 0.1 sec and the output is printed into the file qm.out. The output file contains the following 11 columns:

- the time,
- the input and output nodes defining the queue,
- the queue size in bytes (corresponds to the attribute `size_` of the monitor-queue object)
- the queue size in packets, (corresponds to the attribute `pkt_`)
- the number of packets that have arrived, (corresponds to the attribute `parrivals_`)
- the number of packets that have departured the link, (corresponds to the attribute `pdepartures_`)
- the number of packets dropped at the queue, (corresponds to the attribute `pdrops_`)
- the number of bytes that have arrived, (corresponds to the attribute `barrivals_`)
- the number of bytes that have departured the link, (corresponds to the attribute `bdepartures_`)
- the number of bytes dropped (corresponds to the attribute `bdrops_`).

An alternative way to work directly with these attributes is described in Section 4.5.

Next example script 4.1 shows the entire script for modelling TCP with noisy drops.

```
# Create the simulator instance
set ns [new Simulator]
$ns color 1 Blue
$ns color 2 Red

#Open the NAM trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Open the Trace file
set tf [open out.tr w]
set windowVsTime2 [open WindowVsTimeNReno w]
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
        close $nf
        close $tf
        exec nam out.nam &
        exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

$ns at 0.1 "$n1 label \"CBR\""
$ns at 1.0 "$n0 label \"FTP\""

#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns simplex-link $n2 $n3 0.07Mb 20ms DropTail
$ns simplex-link $n3 $n2 0.07Mb 20ms DropTail

#Set Queue Size of link (n2-n3) to 10
$ns queue-limit $n2 $n3 10

#Monitor the queue for link (n2-n3). (for NAM)
$ns simplex-link-op $n2 $n3 queuePos 0.5

#Set error model on link n3 to n2.
set loss_module [new ErrorModel]
$loss_module set rate_ 0.2
$loss_module ranvar [new RandomVariable/Uniform]
$loss_module drop-target [new Agent/Null]
$ns lossmodel $loss_module $n2 $n3
```

```
#Setup a TCP connection
set tcp [new Agent/TCP/Newreno]
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $n3 $sink
$ns connect $tcp $sink
$tcp set fid_ 1

#Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP

#Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $null
$ns connect $udp $null
$udp set fid_ 2

#Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packetSize_ 1000
$cbr set rate_ 0.01Mb
$cbr set random_ false

#Schedule events for the CBR and FTP agents
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 624.0 "$ftp stop"
$ns at 624.5 "$cbr stop"

# Printing the window size
proc plotWindow {tcpSource file} {
global ns
set time 0.01
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
puts $file "$now $cwnd"
$ns at [expr $now+$time] "plotWindow $tcpSource $file" }
$ns at 1.1 "plotWindow $tcp $windowVsTime2"

# sample the bottleneck queue every 0.1 sec. store the trace in qm.out
set qmon [$ns monitor-queue $n2 $n3 [open qm.out w] 0.1];
[$ns link $n2 $n3] queue-sample-timeout; # [$ns link $n2 $n3] start-tracing
```

```
#Detach tcp and sink agents (not really necessary)
$ns at 624.5 "$ns detach-agent $n0 $tcp ; $ns detach-agent $n3 $sink"

$ns at 625.0 "finish"
$ns run
```

Table 4.1: tcl script rdrop.tcl for TCP over a noisy channel

In Figure 4.4 we trace (using gnuplot) the file WindowVsTimeNReno created by the simulation. A zoomed version is given in Figure 4.5.

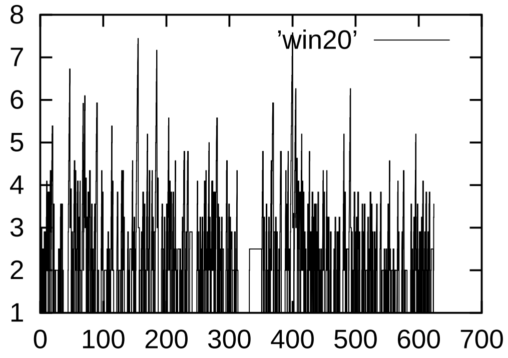

Figure 4.4: Window size of TCP with 20% random losses

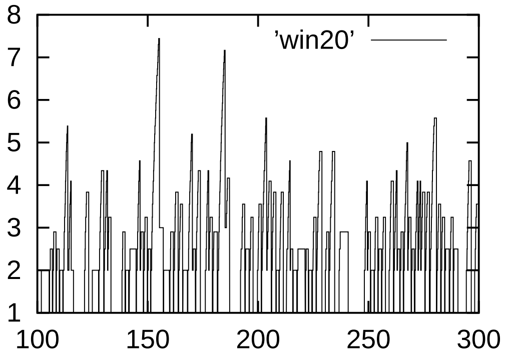

Figure 4.5: Window size of TCP with 20% random losses: a zoom

In several cases we can observe long timeouts, in particular at time 300. To see the huge impact of the random loss on TCP performance, we run again the simulation but with no losses. The result is depicted in Fig. 4.6.

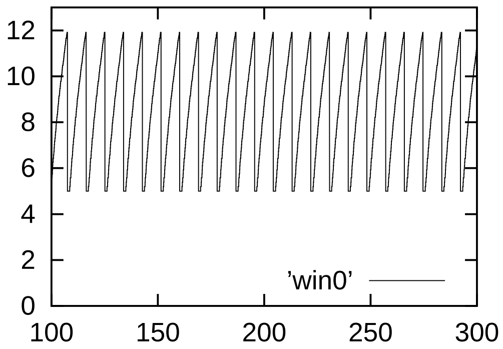

Figure 4.6: TCP window size for 0 loss rate

An important performance measure is the average throughput of TCP. A very simple way to compute it is to search in the trace file `out.tr` the time that a TCP packet was received at the destination (at node 3). In our simulation this is found at time 624.08754 and the corresponding trace line is

```
r 624.082754 2 3 tcp 1000 ------- 1 0.0 3.0 1562 4350
```

The number before the last means that this is the 1562*nd* TCP packet to be well received at the destination. The TCP throughput is thus simply this number divided by the duration of the FTP connection (623 seconds), i.e. 2.507 packets per second, or equivalently, 2.507 Kbytes per second (as a TCP packet contains by default 1000 bytes) or 20058 bps.

Note: if we look at the first lines of the `out.tr` file, we shall see that there are other TCP packets (of size 40 packets each) which we have not counted in the total number 1562. Their serial number is zero. We do not count them because they correspond to *signalling* packets that are involved in the opening of the TCP connection.

Note that we used the delayed Ack version of TCP by using the command

```
set sink [new Agent/TCPSink/DelAck]
```

instead of simply `set sink [new Agent/TCPSink]`.

4.4 Creating many connections with random features

In order to create many connections, it is useful instead of defining each node, link, connection or application individually, to define them as vectors (or arrays) in tcl (within loop statements).

Furthermore, it becomes of interest to choose connections' parameters (such as time of beginning or end of activity, link delays, etc) at a random way. We treat both issues in this Section, and then provide an example. Note that we have already considered other aspects of randomness in Section 4.3.

An example Consider the network at Figure 4.7. The tcl script is given in Table 4.2.

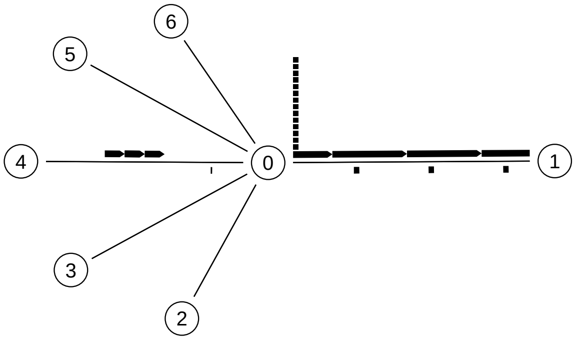

Figure 4.7: Example of a network with several TCP connections

We create 5 FTP connections that start at random: the starting time is uniformly distributed between 0 and 7 sec. The whole simulation duration is 10 seconds. We create links with delay that is chosen at random, uniformly distributed between 1ms and 5ms.

In addition to the standard trace outputs, we also create a file named ”win” that will contain the evolution of the window size of all connection at a granularity of 0.03sec. This is done in the procedure plotWindow. Note that the file ”win” is addressed using the pointer ”windowVsTimes”. The procedure is called recursively for each of the 5 connections.

```
#Create the simulator instance
set ns [new Simulator]

#Opening the trace files
set nf [open out.nam w]
$ns namtrace-all $nf

set tf [open out.tr w]
set windowVsTime [open win w]
set param [open parameters w]
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
        close $nf
        close $tf
        exec nam out.nam &
        exit 0
}

#Create bottleneck and dest nodes and link between them
set n2 [$ns node]
set n3 [$ns node]
$ns duplex-link $n2 $n3 0.7Mb 20ms DropTail

set NumbSrc 5
set Duration 10

#Source nodes
for {set j 1} {$j<=$NumbSrc} { incr j } {
set S($j) [$ns node]
}

# Create a random generator for starting the ftp and for bottleneck link delays
set rng [new RNG]
$rng seed 0

# parameters for random variables for delays
set RVdly [new RandomVariable/Uniform]
$RVdly set min_ 1
$RVdly set max_ 5
$RVdly use-rng $rng

# parameters for random variables for beginning of ftp connections
set RVstart [new RandomVariable/Uniform]
$RVstart set min_ 0
$RVstart set max_ 7
$RVstart use-rng $rng
}
```

```
#We define two random parameters for each connection
for {set i 1} {$i<=$NumbSrc} { incr i } {
set startT($i)  [expr [$RVstart value]]
set dly($i) [expr [$RVdly value]]
puts $param "dly($i) $dly($i) ms"
puts $param "startT($i)  $startT($i) sec" }

#Links between source and bottleneck
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns duplex-link $S($j) $n2 10Mb $dly($j)ms DropTail
$ns queue-limit $S($j) $n2 100
}

#Monitor the queue for link (n2-n3). (for NAM)
$ns duplex-link-op $n2 $n3 queuePos 0.5

#Set Queue Size of link (n2-n3) to 10
$ns queue-limit $n2 $n3 10

#TCP Sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
set tcp_src($j) [new Agent/TCP/Reno]
}

#TCP Destinations
for {set j 1} {$j<=$NumbSrc} { incr j } {
set tcp_snk($j) [new Agent/TCPSink]
}

#Connections
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns attach-agent $S($j) $tcp_src($j)
$ns attach-agent $n3 $tcp_snk($j)
$ns connect $tcp_src($j) $tcp_snk($j)
}

#FTP sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
set ftp($j) [$tcp_src($j) attach-source FTP]
}

#Parametrisation of TCP sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
$tcp_src($j) set packetSize_ 552
}

#Schedule events for the FTP agents:
for {set i 1} {$i<=$NumbSrc} { incr i } {
$ns at $startT($i) "$ftp($i) start"
$ns at $Duration "$ftp($i) stop"
}
```

```
proc plotWindow {tcpSource file k} {
global ns

set time 0.03
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
puts $file "$now $cwnd"
$ns at [expr $now+$time] "plotWindow $tcpSource $file $k" }

# The procedure will now be called for all tcp sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns at 0.1 "plotWindow $tcp_src($j) $windowVsTime $j"
}

$ns at [expr $Duration] "finish"
$ns run
```

Table 4.2: Tcl script ex3.tcl for several competing TCP connections

4.5 Short TCP connections

The majority of the traffic over the Internet constitutes of file transfers. The average transfered file is around 10Kbytes. This means that an "average" file has no more than 10 TCP packets taking the typical TCP packet size to be 1Kbyte [10, 36]. This means that most of the file transfers end in slow start phase. These files are frequently called "mice". Surprisingly, however, most traffic in the Internet is transmitted by very long files. These are called "elephants". A typical distribution that describes the file size is the Pareto [10], with shape parameter of between 1 and 2 [10] (and average of 10KB). The median of the file size is around 2.5Kbytes ([36] and references therein). Note that a Pareto distribution with mean 10Kbytes and a median size of 2.5Kbytes defines a Pareto distribution with shape parameter $\beta = 1.16$ and with a minimum size of 1.37Kbytes. The distribution of interarrival times of new connections is frequently taken to be exponential.

In this Section we shall present ways to simulate short sessions, and to measure the distribution of the transmission duration, of the number of ongoing connections and the throughput.

We shall consider a network with the same topology as the one in Figure 4.2: several sources sharing a common bottleneck node and a common destination. The number of sources is given by the parameter "NodeNb" (in our example it is 6). TCP sources are parameterized now by two parameters: the source node and the session number from that node. For each TCP agent we define a new FTP application. New TCP connections arrive according to a Poisson process. We shall therefore generate the beginning of new TCP connection using exponentially distributed random variables.

The bottleneck link is assumed to be of 2Mbps, to have a delay of 1ms and to have a queue of size 3000. All other input links that join this link have bandwidth of 100 Mbps and a delay of 1ms. We use the New Reno version with a maximum window size of 2000.

The average time between the arrivals of new TCP sessions at each node is in our example 45 msec. This means that on the average, 22.22 new sessions arrive at each node so that the global arrival rate of sessions is 22.22 times NodeNb, which gives in our case 133.33 sessions/sec. We generate sessions with random size with a mean of 10Kbytes, with Pareto distribution with shape

1.5. The global rate of generation of bits is thus

$$133.33 \times 10^4 \times 8 = 10.67 Mbps.$$

We see that the rate of generation of bits is larger than the bottleneck capacity, so we shall expect a congestion phenomenon to appear. However, TCP has the capacity to avoid congestion in the network (at the bottleneck queue). Congestion will therefore appear in other forms that we shall see.

Monitoring the number of sessions In the context of short TCP sessions we are interested not only in packet statistics but also in session statistics. In the ns program we shall define a recursive procedure, called "Test", that checks for each session whether it has ended. The procedure calls itself each 0.1 sec (this is set in the variable "time"). If a connection has ended then we print in an output file

- the connection identifiers i and j (where (i, j) stand for the jth connection from node i),
- the start and end time of that connection,
- the throughput of that connection,
- the size of that transfer in bytes.

The procedure then defines another beginning of transfer after a random time. In the script that follows, the output file will be Out.ns. To check whether a session has ended, we use the command

```
if {[$tcpsrc($i,$j) set ack_]==[$tcpsrc($i,$j) set maxseq_]} {
```

Another recursive procedure called "countFlows" is used to update the number of active connections from each node (stored in a vector "Cnts" whose jth element corresponds to the number of ongoing connections from node j.[2] The procedure has two parameters: "ind" and "sign". The "ind" indicates which source node is concerns. The "sign" indicates to the procedure what to do: it is 0 when a call ends and 1 when it begins. These parameters are used when calling the procedure at the beginning or end of a connection. The procedure also calls itself periodically watch 0.2 and it then prints the number of active calls into a file (Conn.tr). To do that, the "sign" parameter that is passed should be neither 1 nor 0 (we set it as 3).

Monitoring the queue In the tcl program to come, we present an alternative way to do queue monitoring, more sophisticated than the method we saw in Section 4.3. We use again the commands

```
set qfile [$ns monitor-queue $N $D  [open queue.tr w] 0.05]
[$ns link $N $D] queue-sample-timeout;
```

We could however delete the second line. Instead of restricting to that command, we work directly with the attributes of the "monitor-queue" which have been described in Section 4.3. This is done in a procedure called "record" that is recursively called every 0.05 sec. For example, we print the used bandwidth of the queue (in Kbytes per second) into a file by dividing the number of departures in a time epoch by the epoch duration. Note that the monitor-queue keeps track of the total number of arrived bytes in the attribute `bdepartures_`. In order to count only the number of departures in a time epoch (and not during the entire simulation duration), we have to reset the value of `bdepartures_` at the end of each new computation of the bandwidth.

[2] The interest in having different counters at different nodes lies in the fact that we can also use the program for the case of asymetric input links, in which case we shall be able to study the dependence of the performance on the link's delay and bandwidth.

```
set ns [new Simulator]

# There are several sources of TCP sharing a bottleneck link
# and a single destination. Their number is given by the paramter NodeNb

#       S(1)        ----
#       .               |
#       .           ---- ---- N -------- D(1)...D(NodeNb)
#       .               |
#       S(NodeNb)   ----

# Next file will contain the transfer time of different connections
set Out [open Out.ns w]
# Next file will contain the number of connections
set Conn [open Conn.tr w]
#Open the Trace file
set tf [open out.tr w]
$ns trace-all $tf

# We define three files that will be used to trace the queue size,
# the bandwidth and losses at the bottleneck.
set qsize [open queuesize.tr w]
set qbw   [open queuebw.tr   w]
set qlost [open queuelost.tr w]

# defining the topology
set N [$ns node]
set D [$ns node]
$ns duplex-link $N $D 2Mb 1ms DropTail
$ns queue-limit $N $D 3000

# Number of sources
set NodeNb 6
# Number of flows per source node
set NumberFlows 530

#Nodes and links
for {set j 1} {$j<=$NodeNb} { incr j } {
set S($j) [$ns node]
$ns duplex-link $S($j) $N 100Mb 1ms DropTail
$ns queue-limit $S($j) $N 1000
}

#TCP Sources and Destinations
for {set i 1} {$i<=$NodeNb} { incr i } {
for {set j 1} {$j<=$NumberFlows} { incr j } {
set tcpsrc($i,$j) [new Agent/TCP/Newreno]
set tcp_snk($i,$j) [new Agent/TCPSink]
$tcpsrc($i,$j) set window_ 2000
  }
}
```

```
#Connections
for {set i 1} {$i<=$NodeNb} { incr i } {
for {set j 1} {$j<=$NumberFlows} { incr j } {
$ns attach-agent $S($i) $tcpsrc($i,$j)
$ns attach-agent $D $tcp_snk($i,$j)
$ns connect $tcpsrc($i,$j) $tcp_snk($i,$j)
  }
}

#FTP sources
for {set i 1} {$i<=$NodeNb} { incr i } {
for {set j 1} {$j<=$NumberFlows} { incr j } {
set ftp($i,$j) [$tcpsrc($i,$j) attach-source FTP]
  }
}

# Generators for random size of files.
set rng1 [new RNG]
$rng1 seed 0
set rng2 [new RNG]
$rng2 seed 0

# Random interarrival time of TCP transfers at each source i
set RV [new RandomVariable/Exponential]
$RV set avg_ 0.045
$RV use-rng $rng1

# Random size of files to transmit
set RVSize [new RandomVariable/Pareto]
$RVSize set avg_ 10000
$RVSize set shape_ 1.5
$RVSize use-rng $rng2

# We now define the beginning times of transfers and the transfer sizes
# Arrivals of sessions follow a Poisson process.
#
for {set i 1} {$i<=$NodeNb} { incr i } {
     set t [$ns now]
     for {set j 1} {$j<=$NumberFlows} { incr j } {
  # set the beginning time of next transfer from source i
  set t [expr $t + [$RV value]]
  set Conct($i,$j) $t
  # set the size of next transfer from source i
  set Size($i,$j) [expr [$RVSize value]]
  $ns at $Conct($i,$j) "$ftp($i,$j) send $Size($i,$j)"

  # update the number of flows
   $ns at $Conct($i,$j) "countFlows $i 1"
} }

# Next is a recursive procedure that checks for each session whether
# it has ended. The procedure calls itself each 0.1 sec (this is
# set in the variable "time").
# If a connection has ended then we print in the file $Out
#    * the connection identifiers i and j,
#    * the start and end time of the connection,
#    * the throughput of the session,
#    * the size of the transfer in bytes
# and we further define another beginning of transfer after a random time.

proc Test {} {
global Conct tcpsrc Size NodeNb NumberFlows ns RV ftp Out tcp_snk RVSize
set time 0.1
for {set i 1} {$i<=$NodeNb} { incr i } {
for {set j 1} {$j<=$NumberFlows} { incr j } {

# We now check if the transfer is over
 if {[$tcpsrc($i,$j) set ack_]==[$tcpsrc($i,$j) set maxseq_]} {
       if {[$tcpsrc($i,$j) set ack_]>=0} {
                # If the transfer is over, we print relevant information in $Out
    puts $Out "$i,$j\t$Conct($i,$j)\t[expr [$ns now]]\t\
  [expr ($Size($i,$j))/(1000*([expr [$ns now]] - $Conct($i,$j)))]\t$Size($i,$j)"
    countFlows $i 0
    $tcpsrc($i,$j)  reset
    $tcp_snk($i,$j) reset
} } } }
$ns at [expr [$ns now]+$time] "Test"
}

for {set j 1} {$j<=$NodeNb} { incr j } {
set Cnts($j) 0
}

# The following recursive procedure updates the number of connections
# as a function of time. Each 0.2 it prints them into $Conn. This
# is done by calling the procedure with the "sign" parameter equal
# 3 (in which case the "ind" parameter does not play a role). The
# procedure is also called by the Test procedure whenever a connection
# from source i ends by assigning the "sign" parameter 0, or when
# it begins, by assigning it 1 (i is passed through the "ind" variable).

proc countFlows { ind sign } {
global Cnts Conn NodeNb
set ns [Simulator instance]
```

```
       if { $sign==0 } { set Cnts($ind) [expr $Cnts($ind) - 1]
} elseif { $sign==1 } { set Cnts($ind) [expr $Cnts($ind) + 1]
} else {
    puts -nonewline $Conn "[$ns now] \t"
     set sum 0
     for {set j 1} {$j<=$NodeNb} { incr j } {
       puts -nonewline $Conn "$Cnts($j) \t"
       set sum [expr $sum + $Cnts($j)]
   }
   puts $Conn "$sum"
    $ns at [expr [$ns now] + 0.2] "countFlows 1 3"
}}

#Define a 'finish' procedure
proc finish {} {
        global ns tf qsize qbw qlost
        $ns flush-trace
close $qsize
close $qbw
close $qlost
# Execute xgraph to display the queue size, queue bandwidth and loss rate
exec xgraph queuesize.tr -geometry 800x400 -t "Queue size" -x "secs" -y "# packets" &
exec xgraph queuebw.tr -geometry 800x400 -t "bandwidth" -x "secs" -y "Kbps" -fg white &
exec xgraph queuelost.tr -geometry 800x400 -t "# Packets lost" -x "secs" -y "packets" &
        exit 0
}

# QUEUE MONiTORiNG
set qfile [$ns monitor-queue $N $D  [open queue.tr w] 0.05]
[$ns link $N $D] queue-sample-timeout;

# The following procedure records queue size, bandwidth and loss rate
proc record {} {
global ns qfile qsize qbw qlost N D
set time 0.05
set now [$ns now]

# print the current queue size in $qsize, the current used
# bandwidth in $qbw, and the loss rate in $qloss
$qfile instvar parrivals_ pdepartures_ bdrops_ bdepartures_ pdrops_
puts $qsize "$now [expr $parrivals_-$pdepartures_-$pdrops_]"
puts $qbw   "$now [expr $bdepartures_*8/1024/$time]"
set bdepartures_ 0
puts $qlost "$now [expr $pdrops_/$time]"
$ns at [expr $now+$time] "record"
}

$ns at 0.0 "record"
$ns at 0.01 "Test"
$ns at 0.5 "countFlows 1 3"
$ns at 20 "finish"
$ns run
```

Table 4.3: Tcl script shortTcp.tcl for short TCP connections

The number of sessions generated (530 per source) insured that arrivals from all nodes continued till the end of the simulations.

When running the script we obtain the queue size in Kbytes and in packets as depicted in Fig. 4.8.

We also ran later the simulation with a reduced number of 130 sessions per node, and the queue size in Kbytes and in packets as depicted in Fig. 4.9.

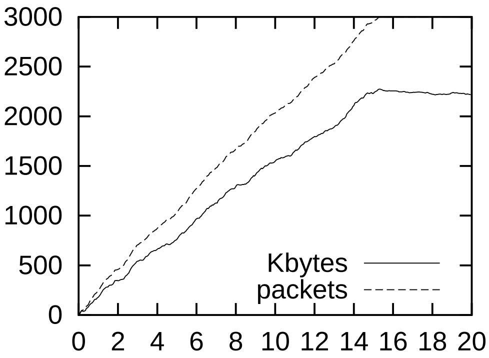

Figure 4.8: Queue size for the example in shortTcp.tcl

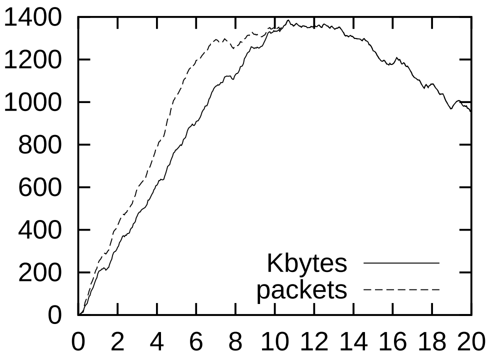

Figure 4.9: Queue size for the example in shortTcp.tcl where we limit the number of sessions

Here are some observations:

1. In both figures, the number of packets at the queue is larger than the number of Kbytes queued. This may seem strange, since a TCP packet has a size of 1Kbyte! The reason is that a very large number of sessions are very small (3 packets or less). Therefore the number of over head packets of size 40 bytes (that are sent at the beginning of each TCP connection) is considerable (around one out of three!). Taking into account these short packets as well, there are more packets than Kbytes.

2. Observe that in Figure 4.8 the queue size stabilizes at 3000, this is the maximum queue size that is reached. From this moment on there will be losses at the queue.

3. Whereas the number of packets is always larger than the number of Kbytes queued in Fig. 4.8, we see that in Fig. 4.9 after some time, the number of packets agrees with the number of Kbytes. At this point all packets at the queue are TCP data packets and there are no packets of 40 bytes corresponding to beginning of sessions. This is due to the fact that we limitted the number of sessions per node to 130.

4. If we subtract the output rate of the bottleneck link from the generatioin rate of data, we shall obtain much more than the amount of data queued at the bottleneck queue. The reason is that the data is also buffered at the senders's buffer.

Next we observe the evolution of the number of ongoing connections at the system, as given in Fig. 4.10 and the used bandwidth at the bottleneck link, see Fig. 4.11.

4.6 Advanced monitoring tools

In Section 4.5 we checked the termination of each TCP session periodically by comparing the current ack sequence number with the maximum sequence number of connection. This probing approach is quite costly. We mention two alternative monitoring approaches:

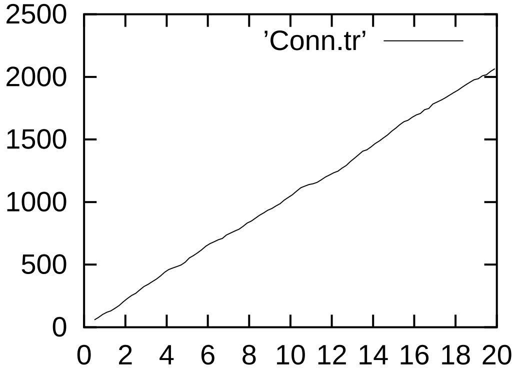

Figure 4.10: Number of Connections

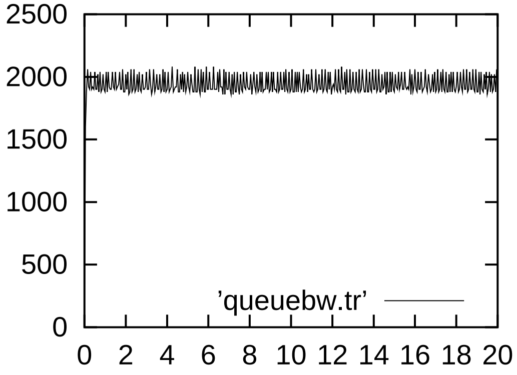

Figure 4.11: Used bandwidth at the bottleneck

1. The first is to define the actions to be taken upon termination within a procedure called "done" that is automatically invoked when a connection is ended. The id of the connection that has ended as well as other properties of the connection (such as its start time) can be used by the procedure if defined as states of the connection. The approach is presented in the tcl script shortTcp2.tcl in Table 4.4.
2. One can use a per-flow monitor. It can give statistics on each flow with information such as the amount of transfered packets, tranfered bytes, losses, etc. We delay the discussion on this approach to Section 6.4.

The "state" definitions of the TCP connections in the script are done in the same way we define the maximum window size of TCP, the slow-start initial threshold, etc. In our script we define the beginning time of the session, the session and node identity and the transfer size as such states:

```
$tcpsrc($i,$j) set starts $t
$tcpsrc($i,$j) set sess $j
$tcpsrc($i,$j) set node $i
$tcpsrc($i,$j) set size [expr [$RVSize value]]
```

The procedure "done" is defined as follows (it replaces the "Test" procedure in the previous approach of the script shortTcp.tcl in Table 4.3):

```
Agent/TCP instproc done {} {
global tcpsrc NodeNb NumberFlows ns RV ftp Out tcp_snk RVSize
# print in $Out: node, session, start time,  end time, duration,
# trans-pkts, transm-bytes, retrans-bytes, throughput
set duration [expr [$ns now] - [$self set starts] ]
puts $Out "[$self set node] \t [$self set sess] \t [$self set starts] \t\
      [$ns now] \t $duration \t [$self set ndatapack_] \t\
      [$self set ndatabytes_] \t [$self set  nrexmitbytes_] \t\
      [expr [$self set ndatabytes_]/$duration ]"
        countFlows [$self set node] 0
}
```

Note that we use other states of TCP connections:

- ndatapack_ is the number of packets transmitted by the connection (if a packet is retransmitted several times, it is counted here only once).

- `ndatabytes_` is the number of data bytes transmitted by the connection,
- `nrexmitpackets_` is the number of packets retransmitted by the connection.
- `nrexmitbytes_` is the number of bytes retransmitted by the connection.

```
set ns [new Simulator]

# There are several sources each generating many TCP sessions sharing a bottleneck
# link and a single destination. Their number is given by the paramter NodeNb

#       S(1)        ----
#       .               |
#       .           ---- ---- N -------- D(1)...D(NodeNb)
#       .               |
#      S(NodeNb)    ----

# Next file will contain the transfer time of different connections
set Out [open Out.ns w]
# Next file will contain the number of connections
set Conn [open Conn.tr w]
#Open the Trace file
set tf [open out.tr w]
$ns trace-all $tf

# defining the topology
set N [$ns node]
set D [$ns node]
$ns duplex-link $N $D 2Mb 1ms DropTail
$ns queue-limit $N $D 3000

# Number of Nodes
set NodeNb 6

# Number of flows per source node
set NumberFlows 530

#Nodes and links
for {set j 1} {$j<=$NodeNb} { incr j } {
set S($j) [$ns node]
$ns duplex-link $S($j) $N 100Mb 1ms DropTail
$ns queue-limit $S($j) $N 1000 }

#TCP Sources, destinations, connections
for {set i 1} {$i<=$NodeNb} { incr i } {
for {set j 1} {$j<=$NumberFlows} { incr j } {
set tcpsrc($i,$j) [new Agent/TCP/Newreno]
set tcp_snk($i,$j) [new Agent/TCPSink]
$tcpsrc($i,$j) set window_ 2000
$ns attach-agent $S($i) $tcpsrc($i,$j)
$ns attach-agent $D $tcp_snk($i,$j)
$ns connect $tcpsrc($i,$j) $tcp_snk($i,$j)
set ftp($i,$j) [$tcpsrc($i,$j) attach-source FTP]
} }
```

```
# Generators for random size of files.
set rng1 [new RNG]
$rng1 seed 0
set rng2 [new RNG]
$rng2 seed 0

# Random inter-arrival times of TCP transfer at each source i
set RV [new RandomVariable/Exponential]
$RV set avg_ 0.045
$RV use-rng $rng1

# Random size of files to transmit
set RVSize [new RandomVariable/Pareto]
$RVSize set avg_ 10000
$RVSize set shape_ 1.5
$RVSize use-rng $rng2

# We now define the beginning times of transfers and the transfer sizes
# Arrivals of sessions follow a Poisson process.
#
for {set i 1} {$i<=$NodeNb} { incr i } {
     set t [$ns now]
     for {set j 1} {$j<=$NumberFlows} { incr j } {
 # set the beginning time of next transfer from source and attributes
 set t [expr $t + [$RV value]]
 $tcpsrc($i,$j) set starts $t
 $tcpsrc($i,$j) set sess $j
 $tcpsrc($i,$j) set node $i
         $tcpsrc($i,$j) set size [expr [$RVSize value]]

  $ns at [$tcpsrc($i,$j) set starts] "$ftp($i,$j) send [$tcpsrc($i,$j) set size]"

  # update the number of flows
   $ns at [$tcpsrc($i,$j) set starts] "countFlows $i 1"
}}

for {set j 1} {$j<=$NodeNb} { incr j } {
set Cnts($j) 0
}

# The following procedure is called whenever a connection ends
Agent/TCP instproc done {} {
global tcpsrc NodeNb NumberFlows ns RV ftp Out tcp_snk RVSize
# print in $Out: node, session, start time,  end time, duration,
# trans-pkts, transm-bytes, retrans-bytes, throughput
set duration [expr [$ns now] - [$self set starts] ]
puts $Out "[$self set node] \t [$self set sess] \t [$self set starts] \t\
        [$ns now] \t $duration \t [$self set ndatapack_] \t\
        [$self set ndatabytes_] \t [$self set  nrexmitbytes_] \t\
        [expr [$self set ndatabytes_]/$duration ]"
        countFlows [$self set node] 0
}
```

```
# The following recursive procedure updates the number of connections
# as a function of time. Each 0.2 sec it prints them into $Conn. This
# is done by calling the procedure with the "sign" parameter equal
# 3 (in which case the "ind" parameter does not play a role). The
# procedure is also called by the "done" procedure whenever a connection
# from source i ends by assigning the "sign" parameter 0, or when
# it begins, by assigning it 1 (i is passed through the "ind" variable).
#
proc countFlows { ind sign } {
global Cnts Conn NodeNb
set ns [Simulator instance]
       if { $sign==0 } { set Cnts($ind) [expr $Cnts($ind) - 1]
} elseif { $sign==1 } { set Cnts($ind) [expr $Cnts($ind) + 1]
} else {
   puts -nonewline $Conn "[$ns now] \t"
   set sum 0
  for {set j 1} {$j<=$NodeNb} { incr j } {
     puts -nonewline $Conn "$Cnts($j) \t"
     set sum [expr $sum + $Cnts($j)]
   }
   puts $Conn "$sum"
   $ns at [expr [$ns now] + 0.2] "countFlows 1 3"
} }

#Define a 'finish' procedure
proc finish {} {
        global ns tf
close $tf
        $ns flush-trace
        exit 0
}

$ns at 0.5 "countFlows 1 3"
$ns at 20 "finish"

$ns run
```

Table 4.4: Tcl script shortTcp2.tcl for short TCP connections

4.7 Exercises

Exercise 4.7.1 *Explain why the window size oscillates much more than the throughput in Figures 4.1 and 4.2.*

Exercise 4.7.2 *What is the average throughput and loss rate of the TCP connection for Example ex1.tcl?*

Exercise 4.7.3 *What is the average queue size for Example ex1.tcl?*

Exercise 4.7.4 *Study the effect of the packet loss probability in the noisy model of rdrop.tcl on TCP throughput for loss probability ranging between 0 and 40 percent.*

Exercise 4.7.5 *Modify the script rdrop.tcl in order to study the effect of the loss probability of packets (Acknowledgements) on the reverse link n3-n2. Plot the throughput as a function of the loss probabilities for loss rates ranging between 0 and 40 percent. Is TCP more sensitive to forward random losses of packets or to backword random losses of Acknowledgements?*

Exercise 4.7.6 *Simulate two symetric competing TCP connections sharing a common bottleneck link. Which fraction of the bandwidth does each one take if*
(i) only one connection uses the delayed ACK option and both connections are NewReno
(ii) both connections have the simple ACK option, the first connection uses the Tahoe version and the second the NewReno version

Exercise 4.7.7 *In the procedure plotWindow at the end of the script ex3.tcl in Table 4.2, we passed the connection number as an argument of the procedure. What would happen if we passed it as a global variable (i.e. if we wrote "global ns j")?*

Exercise 4.7.8 *Analyze the loss processes obtained in ex3.tcl (see Table 4.2). What should the queue size at link n2-n3 be so as to avoid losses?*

Exercise 4.7.9 *Add to the script shortTcp.tcl (Table 4.3) random losses (i) at the forward link (ii) at the backward link $N - D$. Vary the packet loss rate between 0% to 40%. Analyze the average time to transfer a file and the standard deviation of this time as a function of the loss rate. Explain the results! Note: in a context of many users, one may expect that if some sessions have low throughput due to losses, there will be more available throughput to other sessions, so that short TCP sessions are less sensitive than long ones to losses. Do the simulations confirm this or not? If not, explain what happens.*

Chapter 5

Routing and network dynamics

We shall review in this chapter both unicast as well as multicast routing. Routing protocols that fix a permanent route (static routing) will be compared to dynamic routing. The influence of dynamic connectivity on the routing will be examined. A good reference for routing over the Internet is [24].

5.1 Unicast routing

There are several routing possibilities over the Internet. The simplest one is the static routing in which the shortest route (in terms of number of hops) is chosen throughout the connection.

ns can simulate noisy links (as we saw in Section 4.3) or even links that become disconnected. To simulate a disconnection of a link between nodes $n1 and $n4 from time 1 to 4.5, for example, we should type

```
$ns rtmodel-at 1.0 down $n1 $n4
$ns rtmodel-at 4.5 up $n1 $n4
```

We now consider the network depicted in Figure 5.1 which has two alternative routes between the source node 0 and the destination node 5. The default static routing, used by ns, will choose the route 0-1-4-5 for setting connections.

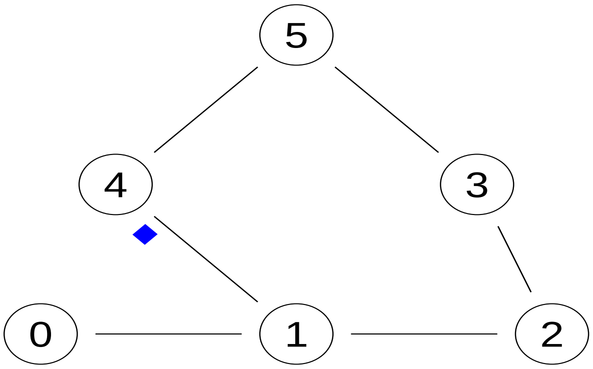

Figure 5.1: A routing example

```
set ns [new Simulator]

#Define different colors for data flows (for NAM)
$ns color 1 Blue
$ns color 2 Red

#Open the Trace file
set file1 [open out.tr w]
$ns trace-all $file1

#Open the NAM trace file
set file2 [open out.nam w]
$ns namtrace-all $file2

#Define a 'finish' procedure
proc finish {} {
        global ns file1 file2
        $ns flush-trace
        close $file1
        close $file2
        exec nam out.nam &
        exit 0
}

# Next line should be commented out to have the static routing
$ns rtproto DV

#Create six nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n1 0.3Mb 10ms DropTail
$ns duplex-link $n1 $n2 0.3Mb 10ms DropTail
$ns duplex-link $n2 $n3 0.3Mb 10ms DropTail
$ns duplex-link $n1 $n4 0.3Mb 10ms DropTail
$ns duplex-link $n3 $n5 0.5Mb 10ms DropTail
$ns duplex-link $n4 $n5 0.5Mb 10ms DropTail
```

```
#Give node position (for NAM)
$ns duplex-link-op  $n0 $n1 orient right
$ns duplex-link-op  $n1 $n2 orient right
$ns duplex-link-op $n2 $n3 orient up
$ns duplex-link-op $n1 $n4 orient up-left
$ns duplex-link-op  $n3 $n5 orient left-up
$ns duplex-link-op  $n4 $n5 orient right-up

#Setup a TCP connection
set tcp [new Agent/TCP/Newreno]
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $n5 $sink
$ns connect $tcp $sink
$tcp set fid_ 1

#Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP

$ns rtmodel-at 1.0 down $n1 $n4
$ns rtmodel-at 4.5 up $n1 $n4

$ns at 0.1 "$ftp start"

$ns at 12.0 "finish"

$ns run
```

Table 5.1: tcl script for static and dynamic routing (ex2.tcl)

In contrast to the static routing, the Internet can find an alternative route once it discovers that a route is disconnected. This option is used in ns by adding the command (see Table 5.1)

```
$ns rtproto DV
```

In the Example ex2.tcl given in Table 5.1, the link link 1-4 is down during the time interval $[1, 4.5]$. In man, we can see this link becoming red at this time. A TCP connection is set between node 0 and 5. When running the script, with the static routing (commenting out the command `$ns rtproto DV`) we see that even though the connection is resumed at time 4.5, the TCP connection resumes only at time 8 approximately. The reason is that timeouts had occurred in the absence of ACKs returning to node 0, and their duration doubles with each new timeout.

In the nam trace we can see in the dynamic routing case, the signaling packets that are used to determine the path, not only at the beginning, but also at connectivity changes.

5.2 Network dynamics

We saw in the last Section that we can determine link states explicitely: the link can go down and up at preselected times. There are however other possibilities change dynamically the connectivity: according to an Exponential On-Off process, or a deterministic On-Off process, or according to some given trace file.

The deterministic and exponential models have four parameters: start time (0.5sec from the beginning of the simulation by default), up interval (10sec by default), down interval (1sec by default) and finish time (end of the simulation by default). In the exponential case, the up and down parameters correspond to the expected durations. For example, the syntax for the deterministic model applied to link n1-n2 is

```
$ns rtmodel Deterministic {0.8 1.0 1.0} $n1 $n2
```

(the finish time is the default). In a command of the form

```
$ns rtmodel Deterministic {0.8 1.0} $n1 $n2
```

the start and end times are the defaults, and in a command of the form

```
$ns rtmodel Deterministic {- 1.0} $n1 $n2
```

the only non default parameter is the down interval. The exponential connectivity is obtained above by replacing "Deterministic" by "Exponential".

The command that corresponds to connectivity based on a trace file is

```
$ns rtmodel Trace <config_file> $n0 $n1
```

Finally, one can also generate a sequence of routing states in ns, and use it as an input (see [14]).

Node failures There is a possibility of a node going down and up. This is done exactly as we saw for the case of links, except that only one node appears as argument at the end.

5.3 Multicast protocols

In multicast, there may be several multicast groups of members; the groups may overlap. In IP multicast, receiver must request membership in multicast group whereas a sender can send without first joing a group. Senders do not receive feedback from the network about the receivers in IP multicast routing. Not all network nodes may be able to handle multicast; in ns one can declare which nodes indeed have multicast capabilities.

A routing protocol defines the mechanism by which the multicast tree is computed in the simulation. There are two main classes of routing classes:
1. a "dense mode" type which is appropriate for the case of a large number of multicast users; in that case multicast trees are constructed for any pair of source and its multicast group. The construction of the trees require broadcasting to all nodes in the network.
2. a "sparse mode" in which there is a small number of nodes. Therefore the routing can be handled using a single shared tree.

Four multicast routing protocols are available in ns: the Dense Mode (DM), the Centralised (CtrMcast), the Shared Tree Mode (ST) and the Bi-directional Sahred Tree mode (BST). Unfortumately, the way ns simulates the protocols does not include much of the signaling, especially in the initialization. The DM protocol is the only one that has a dynamic version in ns, called dynamicDM.

5.3.1 The Dense mode

The DM protocol has two modes which are quite similar: the protocol pimDM (Protocol Independant Multicast - Dense Mode) and the `dvmrp` (Distance Vector Multicast Routing Protocol) mode [40], pimDM being somewhat simpler. They are based on an initial flooding of the network (using the RFP approach) and then on the computation of the shortest reverse path. We suppose that point-to-point routing tables are availble. This is done as follows.

- If a router receives a multipoint packet from a source S to a group G, it checks first (using point-to-point routing tables) that the input reception interface corresponds to packet S: this means that this router is in the shortest from the source (this is thus called a "shortest reverse path" approach). If the result is negative then it sends a message "delete(S,G)", i.e. a message to the source requesting to stop sending to it packets from S to G.

- If the result is positive then the router sends a copy of the message to the set T of all the interfaces through which it has not yet received a request "delete(S,G)". If T is empty, then it destroys the packet and sends a message "delete(S,G)" to the interfece through which it received the message.

5.3.2 Routinng based on a RV point

The centralized mcast (CrtMcast) is similar to the so called PIM-SM (the Sparse Mode of PIM [11]). There is a Rendezvous Point (RP). A shared tree is built for a multicast group rooted at this RP. A centralized computation agent is used to compute the forwarding trees and set up multicast forwarding state, S, G (the state S corresponds to the source of a packet and G to the address of the multicast group that it is destinated to) at the relevant nodes as new receivers join a group. Data packets from the senders to a group are unicast to the RP. The multicasting from the RP to the group is done according to a shortest path tree.

The ST mode is a simplified version of the above sparse mode routing protocol. This protocol has a bidirectional version in ns called BST, which is used in the standard version CBT [4] and in the BGMP protocol for inter-domain multicast [38].

In protocols based on a RP point, all multicast traffic traverses the RV point, which is thus a bottleneck. A failure in that node is crtical for the whole group. Another problem with this approach is that traffic travels on non-optimal paths. The advantages of this approach are 1. the simplicity in the state information: only one entry per-source per-group, 2. signalling does not invlolve the whole network.

Note that in PIM-SM, there is a possibility of switching to optimized source-based trees (S,G) instead of routingn through the RV point. This occurs if the source data rate exceeds some threshold. Thus the RV point can cease to be a bottleneck if traffic rate is large. ST mode does not simulate this feature.

5.4 Simulating multicast routing

Multicast requires enhancements to the nodes and links of the network. ns has therefore specific requirements from the Simulator class before creating the topology. We thus begin by the special command

```
set ns [new Simulator]
$ns multicast
```

In the tcl script we define group addresses using the command `set group1 [Node allocaddr]`. We then define an application and a transport protocol agent attached on one hand to a given source node and on the other hand to a group destination.

We consider below the DM protocol. When a source S sending to a group G becomes active it begins flooding the network along the attached tree corresponding to group G. When a leaf that has not joined the multicast group receives a packet to that group, it sends a mesage to the incoming interface to delete it from the tree (S,G) (a "prune" packet). This then propagate backwords to the source: a node that receives a message from all its output links within the tree of (S,G) requesting to delete these links, sends back to its incoming interface a message to delete it from the tree (S,G).

A source will stop completely sending packets if there are no connected receivers in that group; it will resume sending packet when a receiver connects.

An example of a multicast configuration with a six node network is depicted in Figure 5.2:

We now consider the network depicted in Figure 5.1

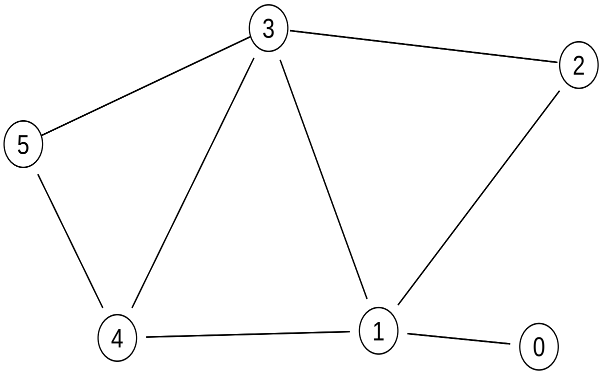

Figure 5.2: A multicast routing example

```
set ns [new Simulator]
$ns multicast

set f [open out.tr w]
$ns trace-all $f
$ns namtrace-all [open out.nam w]

$ns color 1 red
# the nam colors for the prune packets
$ns color 30 purple
# the nam colors for the graft packets
$ns color 31 green

# allocate a multicast address;
set group [Node allocaddr]

# nod is the number of nodes
set nod 6

# create multicast capable nodes;
for {set i 1} {$i <= $nod} {incr i} {
   set n($i) [$ns node]
}

#Create links between the nodes
$ns duplex-link $n(1) $n(2) 0.3Mb 10ms DropTail
$ns duplex-link $n(2) $n(3) 0.3Mb 10ms DropTail
$ns duplex-link $n(2) $n(4) 0.5Mb 10ms DropTail
$ns duplex-link $n(2) $n(5) 0.3Mb 10ms DropTail
$ns duplex-link $n(3) $n(4) 0.3Mb 10ms DropTail
$ns duplex-link $n(4) $n(5) 0.5Mb 10ms DropTail
$ns duplex-link $n(4) $n(6) 0.5Mb 10ms DropTail
$ns duplex-link $n(5) $n(6) 0.5Mb 10ms DropTail

# configure multicast protocol;
set mproto DM

# all nodes will contain multicast protocol agents;
set mrthandle [$ns mrtproto $mproto]

set udp1 [new Agent/UDP]
set udp2 [new Agent/UDP]

$ns attach-agent $n(1) $udp1
$ns attach-agent $n(2) $udp2
```

```
set src1 [new Application/Traffic/CBR]
$src1 attach-agent $udp1
$udp1 set dst_addr_ $group
$udp1 set dst_port_ 0
$src1 set random_ false

set src2 [new Application/Traffic/CBR]
$src2 attach-agent $udp2
$udp2 set dst_addr_ $group
$udp2 set dst_port_ 1
$src2 set random_ false

# create receiver agents
set rcvr [new Agent/LossMonitor]

# joining and leaving the group;
$ns at 0.6 "$n(3) join-group $rcvr $group"
$ns at 1.3 "$n(4) join-group $rcvr $group"
$ns at 1.6 "$n(5) join-group $rcvr $group"
$ns at 1.9 "$n(4) leave-group $rcvr $group"
$ns at 2.3 "$n(6) join-group $rcvr $group"
$ns at 3.5 "$n(3) leave-group $rcvr $group"

$ns at 0.4 "$src1 start"
$ns at 2.0 "$src2 start"

$ns at 4.0 "finish"

proc finish {} {
        global ns
        $ns flush-trace
        exec nam out.nam &
        exit 0
}

$ns run
```

Table 5.2: Example for multicast with DM model: pimdm.tcl

The Loss Monitor Agent We used here the LossMonitor Agent, which is a packet sink agent that maintains statistics about the received traffic, such as the amount of received as well as lost information. In particular, we can access the following state variables: `nlost_` (number of lost packets), `npkts_` (number of received packets), `bytes_` (number of received bytes), `lastPktTime_` (time at which the last packet was received) and `expected_` (the expected sequence number of the next packet). One can use instead of the LossMonitor agent, the Null agent, as we did before, i.e. type `set rcvr [new Agent/Null]` instead of `set rcvr [new Agent/LossMonitor]`.

5.4.1 DM mode

The command `set mproto DM` indictes that we use the Dense Mode protocol. By default, the pimDM is used. In order to use the dvmrp mode, one has to add the line

```
DM set CacheMissMode dvmrp
```

just before the line `set mproto DM`.

In the DM mode, flooding occurs periodically so as to detect the nodes that are connected to the group. The timer value for the period is given in a variable called **PruneTimeout**. It's default value is 0.5sec; If another value is required, say 0.8 sec, then one should add to the tcl script the command

```
DM set PruneTimeout 0.8
```

just before the line `set mproto DM`.

5.4.2 Routing with a centralized RV point

For the centralized mode one needs:

```
# configure multicast protocol;
set mproto CtrMcast
# all nodes will contain multicast protocol agents;
set mrthandle [$ns mrtproto $mproto]
# set RV and bootstrap points
$mrthandle set_c_rp $n(2)
```

Here we chose n(2) to be the RP point.

In both the centralized as well as in the ST mode, the signalling (prune packets) are not simulated.

We present in Table 5.3 the same example as in pimdm.tcl (Table 5.2) but with the BST routing protocol.

```
set ns [new Simulator -multicast on]

set f [open out.tr w]
$ns trace-all $f
$ns namtrace-all [open out.nam w]

$ns color 1 red
# the nam colors for the prune packets
$ns color 30 purple
# the nam colors for the graft packets
$ns color 31 green

# allocate a multicast address;
set group [Node allocaddr]

# nod is the number of nodes
set nod 6

# create multicast capable nodes;
for {set i 1} {$i <= $nod} {incr i} {
   set n($i) [$ns node]
}

#Create links between the nodes
$ns duplex-link $n(1) $n(2) 0.3Mb 10ms DropTail
$ns duplex-link $n(2) $n(3) 0.3Mb 10ms DropTail
$ns duplex-link $n(2) $n(4) 0.5Mb 10ms DropTail
$ns duplex-link $n(2) $n(5) 0.3Mb 10ms DropTail
$ns duplex-link $n(3) $n(4) 0.3Mb 10ms DropTail
$ns duplex-link $n(4) $n(5) 0.5Mb 10ms DropTail
$ns duplex-link $n(4) $n(6) 0.5Mb 10ms DropTail
$ns duplex-link $n(5) $n(6) 0.5Mb 10ms DropTail

# configure multicast protocol;
BST set RP_($group) $n(2)
$ns mrtproto BST

set udp1 [new Agent/UDP]
set udp2 [new Agent/UDP]

$ns attach-agent $n(1) $udp1
$ns attach-agent $n(2) $udp2
```

```
set src1 [new Application/Traffic/CBR]
$src1 attach-agent $udp1
$udp1 set dst_addr_ $group
$udp1 set dst_port_ 0
$src1 set random_ false

set src2 [new Application/Traffic/CBR]
$src2 attach-agent $udp2
$udp2 set dst_addr_ $group
$udp2 set dst_port_ 1
$src2 set random_ false

# create receiver agents
set rcvr [new Agent/LossMonitor]

# joining and leaving the group;
$ns at 0.6 "$n(3) join-group $rcvr $group"
$ns at 1.3 "$n(4) join-group $rcvr $group"
$ns at 1.6 "$n(5) join-group $rcvr $group"
$ns at 1.9 "$n(4) leave-group $rcvr $group"
$ns at 2.3 "$n(6) join-group $rcvr $group"
$ns at 3.5 "$n(3) leave-group $rcvr $group"

$ns at 0.4 "$src1 start"
$ns at 2.0 "$src2 start"

$ns at 4.0 "finish"

proc finish {} {
        global ns
        $ns flush-trace
        exec nam out.nam &
        exit 0
}

$ns run
```

Table 5.3: Example for multicast with RV point: bst.tcl

5.5 Observations on the simulation of pimdm.tcl

Dense mode: pimdm and dvmrp If we run the simulation and observe the trace, we shall see that in addition to the CBR packets, there are two other types of packets: the "prune" packet, and the "graft" packet. The role of the prune packet sent by a node N is to signal to the node that had sent a previous packet to N to stop sending packets to N. The "graft" packet is a signal originating from a node that wishes to join the group (after it had been disconnected). In the NAM display of our simulation, the graft packets are light green, and the prune are purple.

We can see that at time 0.4, node 0 starts sending CBR packets that flood the network. But there is no receivers at the multicast group, so eventually, prune packets return to the source and transmission is stopped (time 0.579). At time 0.6, a graft packet is sent from node 2 (who wishes to join the group) to node 1, and then from node 1 to node 0. Node 0 then restarts transmission. At time 0.9978, there is again an attempt to check whether there are connected receiver nodes in the group other than 2 and the network is again flooded; prune packets return to stop the trahsmission to nodes 3, 4, 5.

The centralized mode We see in the trace encapsulated packets that are sent from a source to the RV point of size 230 byte. The header is then removed by the RV point which then forwards the packet (size 210 bytes) to the members of the group.

5.6 Exercises

Exercise 5.6.1 Run the program ex2.tcl (see Table 5.1) commenting out the command "$ns rtproto DV" and explain what happens.

Exercise 5.6.2 Run the program ex2.tcl (see Table 5.1) with the command "$ns rtproto DV" and explain the differences with the previous static routing.

Exercise 5.6.3 Change and run simulation ex2.tcl (see Table 5.1) for a duration of 20sec with static routing but with a dynamic exponential ON-OFF connectivity, with ON average time of 3 sec and OFF average time of 0.5 sec. Analyze the behavior of the TCP connection and the time-out behavior.

Exercise 5.6.4 *Run the pimdm.tcl script (see Table 5.2). How many CBR packets have been transmitted from each source, and how many have been lost? How many CBR packets have been received at nodes that did not need them (more precisely, how many prune packets have been generated)?*

Exercise 5.6.5 *Consider the trace obtained from the pimdm.tcl script. At time 1.8375 we start to have losses at node 0. At time 2.481 packets start getting lost also at node 1. Explain these losses!*

Exercise 5.6.6 *Run the program pimdm.tcl with the dvmrp mode of DM. What are the differences that you observe between dvmrp and the pimDM version?*

Exercise 5.6.7 *Run the centralised version of the multicast. Explain what happens when the RP is changed to node n(5) (in the NAM it will correspond to node 4, since NAM counts from 0). Explain why this is less efficient than choosing the RP node to be n(2). How can we measure efficiency?*

Chapter 6

RED: Random Early Discard

6.1 Description of RED

The RED buffer management scheme has been introduced on 1993 by Floyd and Jacobson [17], and is further described in the RFC 2309 [7]. Many important references on RED can be found at http://www.icir.org/floyd/red.html. The basic idea is that one should not wait till the buffer is full in order detect detection (drop packets), but start detecting congestion before the buffer overflows. Congestion signals could still be through packet dropping, but could now also be through marking of packets without the need to actually drop them.

Some of the goals of the RED buffer management are:

1. Accomodate short bursts that might be delay sensitive, but not to allow the average queue size increase too much. Using some low past filtering of the queue size, the aim is to detect congestion that lasts long enough.

2. Drop tail and random drop gateways have a bias against bursty traffic. Indeed in such buffers, the more a traffic of a connection is bursty, the more likely it is that the queue will overflow during the arrival time of packets of that connection.

3. Avoid synchronization: in drop tail buffer, many connections may receive a congestion signal at the same time leading to undesirable oscillations in the throughputs. Such oscillation may cause lower average throughputs and high jitter. To avoid synchronization, congestions signals are chosen using radomization.

4. Control the average queue size. Note that this also means controlling the average queueing delay.

To achieve these objectives, RED monitors the average queue size avg, and checks whether it lies between some minimum threshold min_{th} and a maximum threshold max_{th}. If it does, then an arriving packet is dropped or marked with probability $p = p(avg)$ which is an increasing function of the average queue size. All arriving packets that arrive when avg exceeds max_{th} are marked/dropped.

The probability $p(avg)$ is chosen as follows. As the average queue size varies between min_{th} and max_{th}, a probability p_b varies linearly between 0 and some value max_p, i.e.

$$p_b(avg) = max_p \frac{avg - min_{th}}{max_{th} - min_{th}}.$$

This probability is used as $p(avg)$ if at the arrival of the previous packet, $avg \geq min_{th}$. Otherwize $p(avg)$ is set to the value $p(avg)/(1 + p(avg))$.

The average queue size is monitored as follows. The avg paremater is intially set to zero. Then with each arriving packet, the new value avg is assigned the value

$$(1 - w_q)avg + w_q q$$

where q is the actual queue size and w_q is some small constant. If the queue becomes empty some other formula is used to update its size, which takes into account the time since it became empty and an estimate on the number of packets that could have been sent during this idle time, see [17]. For estimating the latter, we shall need in ns to give as parameter a rough estimation of the mean packet size.

Examples of RED parameters studied in [17] are $w_q = 0.002$, $min_{th} = 5$packets, $max_{th} = 15$packets, $max_p = 1/50$ and the queue size is 100. More generally they also investigate min_{th} ranging between 3 to 50, and keep $max_{th} = 3min_{th}$.

The implementation of red in ns can be found in ns-allinone-2.XXX/ns-2.XXX/queue/red.cc (XXX stands for the version, e.g. 1b9a).

6.2 Setting RED parameters in ns

The parameters of RED in ns are provided in the following objects:

1. `bytes_`: takes either the value "true" if we work in the "byte mode" or "false" in the packet mode (the default value). In the "byte mode", the size of an arriving packet affects the likelihood of marking it.

2. `queue-in-bytes_`: the average queue size will be measured in bytes if this is set to "true". In that case, also `thresh_` and `maxthres_` are scaled by the estimated average packet size parameter `mean_pktsize_`. It is "false" by default.

3. `thres_`: is the minimum queue size threshold min_{th}.

4. `maxthres_`: is the maximum queue size threshold max_{th}.

5. `mean_pktsize_`: is the estimate of the average packet size in bytes. The default value is 500.

6. `q_weight_`: the weight factor w_q in computing the averaged queue length.

7. `wait_`: This is a parametter that allows to maintain an interval between dropped packets when set to "true" (the default value).

8. `linterm_`: This is the resiprocal of max_p. Its default value is 10.

9. `setbit_`: is "false" in the case that RED is used to actually drop packets, and is "true" if RED marks the packet with a congestion bit, instead. (The ECN version of TCP reacts to these congestion bits).

10. `drop-tail_`: This is a parameter that allows, when setting its value to "true" (the default value), to use the drop-tail policy when queue overflows or when the average queue size exceeds max_{th}.

The default values of `q_weight_`, `maxthresh_` and `thres_` have been 0.002, 15 and 5 respectively till end 2001. In the more recent releases they a configured automatically.

RED has other paramters and variants that are implemented in ns. In particular, S. Floyd recommends in http://www.icir.org/floyd/red/gentle.html for the best behavior of RED (in simulations and in implementations), to use the `gentle_` parameter set to "true" (this is the default since April, 2001). In the `gentle_` modification to RED in ns, the packet-dropping probability varies from max_p to 1 as the average queue size varies from `maxthresh_` to twice `maxthresh_`. This option makes RED much more robust to the setting of the parameters `maxthresh` and `max_p`.

Another version is the adaptive RED that adapts the choice of parameters to the network traffic, as described in [18].

In order to monitor a given red buffer, say one between nodes `$n2` and `$n3`, one can type

```
set redq [[$ns link $n2 $n3] queue]
set traceq [open red-queue.tr w]
$redq trace curq_
$redq trace ave_
$redq attach $traceq
```

Here `curq_` is the current queue value and `ave_` is the averaged value. This gives an output file (in our case "red-queue.tr") with three columns. The first indicates whether it is a value of the current queue size (by using the flag "Q") or the averaged queue size (using the flag "a"). Then comes the current time and finally the monitored value.

6.3 Simulation examples

We consider the following network, depicted at Figure 6.1: We shall compare the behavior of several queue management schemes.

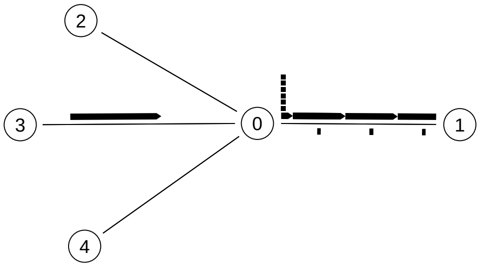

Figure 6.1: Network setting for the study of RED

6.3.1 Drop tail buffer

The first buffer management scheme is a simple drop tail mechanism. We consider three input links with delay 1msec each and bandwidth of 10Mbps each. The common bottleneck link has 20msec of delay and bandwidth of 700 kbps. We consider three FTP connections using TCP and set the maximum window sizes to 8000. The bottleneck queue size is 100. The three connections start at random times, uniformly distributed between 0 and 7 sec. There delay till the bottleneck is 1msec. We chose a TCP packet size of 552 bytes. Note: in version 2.1b9a of ns, when we type the command

```
$tcp_src($j) set packetSize_ 552
```

then the actual packet size created in the simulation is 592, since an extra 40 bytes of header are added. The whole simulation lasts 50 sec.

Using the monitor-queue option that we saw already in Section 4.3, we create a file called queue.tr whose first column is the time and the fifth column is the queue size in packets. We shall also use a procedure, called plotWindow, to monitor the window sizes: it creates a file where the first column is time, and the other three columns correspond to the window sizes of the three connections.

```
set ns [new Simulator]

set nf [open out.nam w]
$ns namtrace-all $nf

set tf [open out.tr w]
set windowVsTime [open win w]
set param [open parameters w]
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
        close $nf
        close $tf
        exec nam out.nam &
        exit 0
}

#Create bottleneck and dest nodes
set n2 [$ns node]
set n3 [$ns node]

#Create links between these nodes
$ns duplex-link $n2 $n3 0.7Mb 20ms DropTail

set NumbSrc 3
set Duration 50

#Source nodes
for {set j 1} {$j<=$NumbSrc} { incr j } {
set S($j) [$ns node]
}

# Create a random generator for starting the ftp and for bottleneck link delays
set rng [new RNG]
$rng seed 2

# parameters for random variables for begenning of ftp connections
set RVstart [new RandomVariable/Uniform]
$RVstart set min_ 0
$RVstart set max_ 7
$RVstart use-rng $rng

#We define random starting times for each connection
for {set i 1} {$i<=$NumbSrc} { incr i } {
set startT($i)  [expr [$RVstart value]]
set dly($i) 1
puts $param "startT($i)  $startT($i) sec"
}
```

```
#Links between source and bottleneck
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns duplex-link $S($j) $n2 10Mb $dly($j)ms DropTail
$ns queue-limit $S($j) $n2 20
}

#Set Queue Size of link (n2-n3) to 100
$ns queue-limit $n2 $n3 100

#TCP Sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
set tcp_src($j) [new Agent/TCP/Reno]
$tcp_src($j) set window_ 8000
}

#TCP Destinations
for {set j 1} {$j<=$NumbSrc} { incr j } {
set tcp_snk($j) [new Agent/TCPSink]
}

#Connections
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns attach-agent $S($j) $tcp_src($j)
$ns attach-agent $n3 $tcp_snk($j)
$ns connect $tcp_src($j) $tcp_snk($j)
}

#FTP sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
set ftp($j) [$tcp_src($j) attach-source FTP]
}

#Parametrisation of TCP sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
$tcp_src($j) set packetSize_ 552
}

#Schedule events for the FTP agents:
for {set i 1} {$i<=$NumbSrc} { incr i } {
$ns at $startT($i) "$ftp($i) start"
$ns at $Duration "$ftp($i) stop"
}
```

```
proc plotWindow {tcpSource file k} {
global ns NumbSrc
set time 0.03
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
if {$k == 1} {
   puts -nonewline $file "$now \t $cwnd \t"
  } else {
   if {$k < $NumbSrc } {
   puts -nonewline $file "$cwnd \t" }
}
if { $k == $NumbSrc } {
   puts -nonewline $file "$cwnd \n" }
$ns at [expr $now+$time] "plotWindow $tcpSource $file $k" }

# The procedure will now be called for all tcp sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns at 0.1 "plotWindow $tcp_src($j) $windowVsTime $j"
}

set qfile [$ns monitor-queue $n2 $n3  [open queue.tr w] 0.05]
[$ns link $n2 $n3] queue-sample-timeout;

$ns at [expr $Duration] "finish"
$ns run
```

Table 6.1: tcl script drprail.tcl

During the 50 sec of simulation time, the source received 7211 TCP packets. Next we plot the queue size (Fig. 6.2) and the window size (Fig. 6.3).

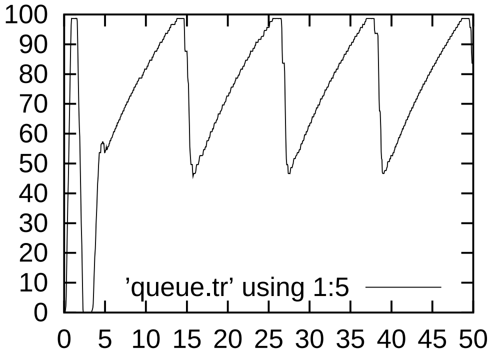

Figure 6.2: Queue size evolution

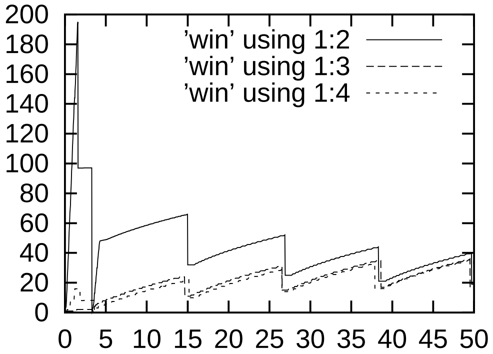

Figure 6.3: Window size of all TCP connections

We see from the figures that there is a high level of synchronization between the window sizes: they all lose packets at the same time. Moreover, we have high oscillationos of the queue sizes that correspond to those of the windows, and the average queue size is around 75 packets. This means that there is an additional average queueing delay which equals

$$D_q = \frac{75 \times 592 \times 8}{700 \times 10^3} = 507.42 msec$$

Remark 6.3.1 The drop-tail queue can be simulated using a RED buffer with $min_{th} = max_{th}$ set on the maximum queue size and max_p set to a value close to zero. This allows us to use the monitoring tools for the instantaneous and average queue length of RED. Of course, the `drop-tail_` parameter has have the value "true".

6.3.2 RED buffer with automatic parameter configuration

We run a second simulation with the same parmaeters. Note that we choose for the ranndom delay a seed 2 in all the simulations since unlike the seed 0, it will gurantee the same random parameters are used in all simulations.

During the 50 sec of simulation time, the source received 7310 TCP packets, slightly more than with the drop tail case (where we had 7211 packets). Next we plot the queue size (Fig. 6.4 and 6.5) and the window size (Fig.6.6).

We see from the figures that there is no synchronization between the window sizes, and that the average queue size is much lower than in the drop tail case: it is around 10 (instead of 75 in the drop tail case). Thus the average delay of the connections thus also smaller, it is

$$D_q = \frac{10 \times 592 \times 8}{700 \times 10^3} = 67.66 msec.$$

We observe that instead of the large oscillations of the queue size and the window sizes, we now get much faster and smaller variationsn in both window size as well as queue size. We finally notice that during the simulation, the queue never overflowed, unlike the case of drop tail. Yet RED did allow the queue to grow very much during the transient spike at the beginning of the connection, which shows that short bursts are indeed not penalized with RED.

We provide in Table 6.2 the tcl script we used.

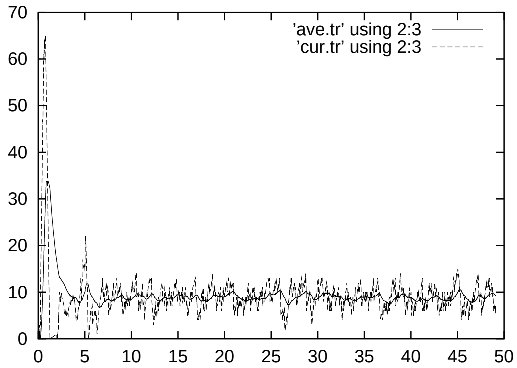

Figure 6.4: Current and Average queue size evolution

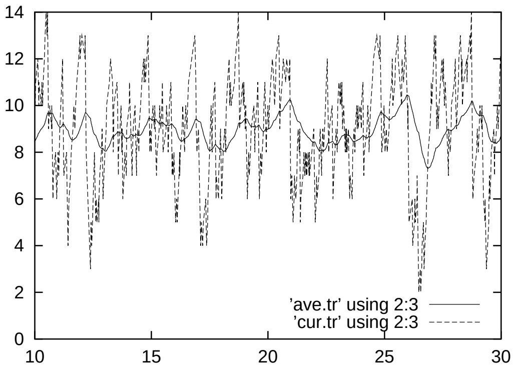

Figure 6.5: Current and Average queue size evolution: a zoom

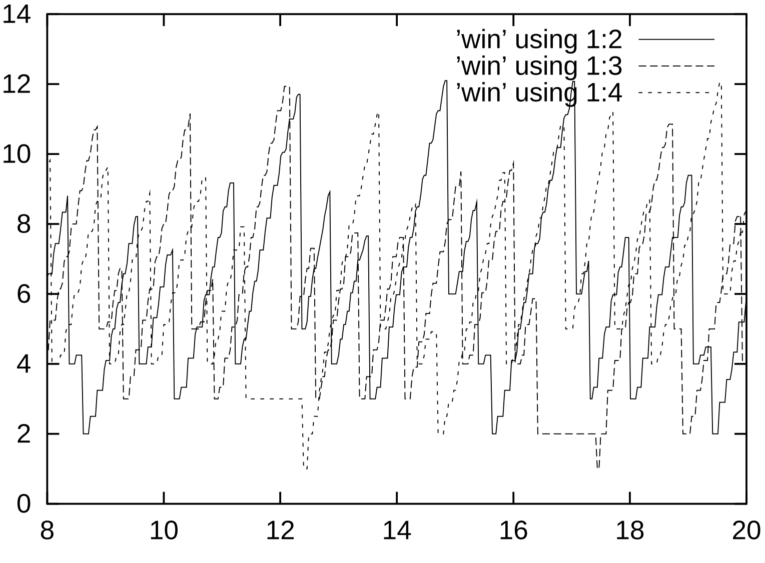

Figure 6.6: Window size of all TCP connections for Red buffer with automatic parameter configuration

```
set ns [new Simulator]

set nf [open out.nam w]
$ns namtrace-all $nf

set tf [open out.tr w]
set windowVsTime [open win w]
set param [open parameters w]
$ns trace-all $tf


#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
        close $nf
        close $tf
        exec nam out.nam &
exec grep  "a" red-queue.tr > ave.tr
exec grep  "Q" red-queue.tr > cur.tr
        exit 0
}

#Create bottleneck and dest nodes
set n2 [$ns node]
set n3 [$ns node]

#Create links between these nodes
$ns duplex-link $n2 $n3 0.7Mb 20ms RED

set NumbSrc 3
set Duration 50

#Source nodes
for {set j 1} {$j<=$NumbSrc} { incr j } {
set S($j) [$ns node]
}

# Create a random generator for starting the ftp and for bottleneck link delays
set rng [new RNG]
$rng seed 2

# parameters for random variables for begenning of ftp connections
set RVstart [new RandomVariable/Uniform]
$RVstart set min_ 0
$RVstart set max_ 7
$RVstart use-rng $rng
```

```
#We define random starting times for each connection
for {set i 1} {$i<=$NumbSrc} { incr i } {
set startT($i)  [expr [$RVstart value]]
set dly($i) 1
puts $param "startT($i)  $startT($i) sec"
}

#Links between source and bottleneck
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns duplex-link $S($j) $n2 10Mb $dly($j)ms DropTail
$ns queue-limit $S($j) $n2 20
}

#Set Queue Size of link (n2-n3) to 100
$ns queue-limit $n2 $n3 100

set redq [[$ns link $n2 $n3] queue]
set traceq [open red-queue.tr w]
$redq trace curq_
$redq trace ave_
$redq attach $traceq

#TCP Sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
set tcp_src($j) [new Agent/TCP/Reno]
$tcp_src($j) set window_ 8000
}

#TCP Destinations
for {set j 1} {$j<=$NumbSrc} { incr j } {
set tcp_snk($j) [new Agent/TCPSink]
}

#Connections
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns attach-agent $S($j) $tcp_src($j)
$ns attach-agent $n3 $tcp_snk($j)
$ns connect $tcp_src($j) $tcp_snk($j)
}

#FTP sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
set ftp($j) [$tcp_src($j) attach-source FTP]
}

#Parametrisation of TCP sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
$tcp_src($j) set packetSize_ 552
}
```

```
#Schedule events for the FTP agents:
for {set i 1} {$i<=$NumbSrc} { incr i } {
$ns at $startT($i) "$ftp($i) start"
$ns at $Duration "$ftp($i) stop"
}

proc plotWindow {tcpSource file k} {
global ns NumbSrc
set time 0.03
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
if {$k == 1} {
   puts -nonewline $file "$now \t $cwnd \t"
  } else {
   if {$k < $NumbSrc } {
   puts -nonewline $file "$cwnd \t" }
}
if { $k == $NumbSrc } {
   puts -nonewline $file "$cwnd \n" }
$ns at [expr $now+$time] "plotWindow $tcpSource $file $k" }

# The procedure will now be called for all tcp sources
for {set j 1} {$j<=$NumbSrc} { incr j } {
$ns at 0.1 "plotWindow $tcp_src($j) $windowVsTime $j"
}

$ns at [expr $Duration] "finish"
$ns run
```

Table 6.2: tcl script red.tcl

6.3.3 RED buffer with other parmaters

Suppose we wish to define our own parameters for RED rather than use the default ones. For example, assume we wish to have in our previous example `max_{th}=60`, `min_{th}=40` and `q_weight_=0.02`. Then we should add the commands

```
Queue/RED set thresh_ 60
Queue/RED set maxthresh_ 80
Queue/RED set q_weight_ 0.002
```

Important note: these commands should be put at the beginning, before the links are defined!

The resulting window and queue size processes are given in Figures 6.8 and 6.7 respectively.

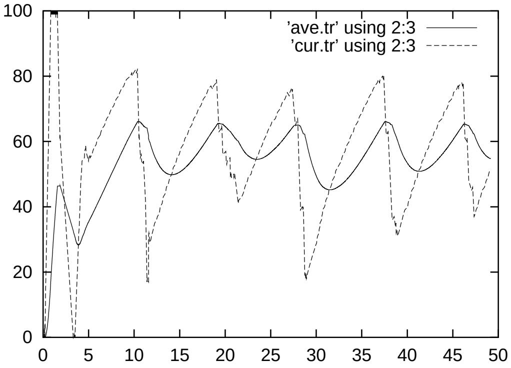

Figure 6.7: Current and Average queue size evolution

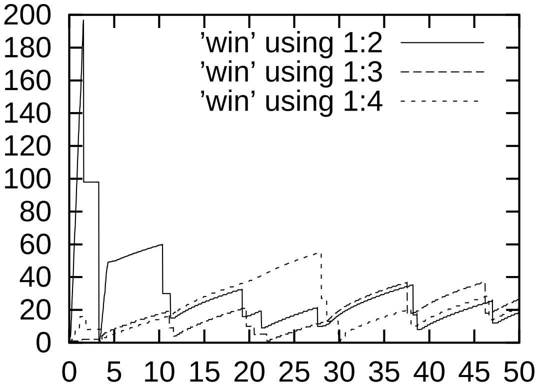

Figure 6.8: Window size of all TCP connections for Red buffer

Note that with the parameters that we chose, the queue lengths are kept aroud an average of 50. The number of TCP packets received during the simulation was 7212.

6.4 Monitoring flows

We introduce in this section the flow monitor, which is an efficient way to monitor per-flow quantities such as losses and amount of transmitted traffic. We shall modify the ns script of shortTcp2.tcl (Table 4.4) to include a RED buffer with monitoring.

A flow monitors a simplex link, so we first define the link we wish to monitor: be defined, e.g.

```
set flink [$ns simplex-link $N $D 2Mb 1ms RED]
```

and then the flow-monitor is defined as follows with respect to this link:

```
set monfile [open mon.tr w]
set fmon [$ns makeflowmon Fid]
$ns attach-fmon $flink $fmon
$fmon attach $monfile
```

When we activate the monitoring, we get the statistics up to the activiation time in a file. This is done as follows:

```
$ns at $time "$fmon dump"
```

We next present in Table 6.3 the full script shortRed.tcl that allows us to study short TCP sessions interacting with a RED buffer.

```
set ns [new Simulator]

# There are several sources each generating many TCP sessions sharing a bottleneck
# link and a single destination. Their number is given by the paramter NodeNb

#     S(1)          ----
#       .               |
#       .           ----N --------------- D
#       .               |
#    S(NodeNb)      ----

set Out [open Out.ns w];   # file containing transfer times of different connections
set Conn [open Conn.tr w]; # file containing the number of connections
set tf [open out.tr w];    #Open the Trace file
$ns trace-all $tf

set NodeNb      6;   # Number od source nodes
set NumberFlows 253; # Number of flows per source node
set sduration    50;  # Duration of simulation

# When the following parameters are commented, the RED is
# configured automatically.
# Queue/RED set thresh_ 5
# Queue/RED set maxthresh_ 15
# Queue/RED set q_weight_ 0.002

# defining the topology
set N [$ns node]
set D [$ns node]
set flink [$ns simplex-link $N $D 2Mb 1ms RED]
$ns simplex-link $D $N 1Mb 1ms DropTail
$ns queue-limit $N $D 50

# queue monitoring, RED
set redq [[$ns link $N $D] queue]
set traceq [open red-queue.tr w]
$redq trace curq_
$redq trace ave_
$redq attach $traceq

#Nodes and links
for {set j 1} {$j<=$NodeNb} { incr j } {
set S($j) [$ns node]
$ns duplex-link $S($j) $N 100Mb 1ms DropTail
$ns queue-limit $S($j) $N 100
}
```

```
# set flow monitor
set monfile [open mon.tr w]
set fmon [$ns makeflowmon Fid]
$ns attach-fmon $flink $fmon
$fmon attach $monfile

#TCP Sources, destinations, connections
for {set i 1} {$i<=$NodeNb} { incr i } {
for {set j 1} {$j<=$NumberFlows} { incr j } {
set tcpsrc($i,$j) [new Agent/TCP/Newreno]
set tcp_snk($i,$j) [new Agent/TCPSink]
set k [expr $i*1000 +$j];
$tcpsrc($i,$j) set fid_ $k
$tcpsrc($i,$j) set window_ 2000
$ns attach-agent $S($i) $tcpsrc($i,$j)
$ns attach-agent $D $tcp_snk($i,$j)
$ns connect $tcpsrc($i,$j) $tcp_snk($i,$j)
set ftp($i,$j) [$tcpsrc($i,$j) attach-source FTP]
} }

# Generators for random size of files.
set rng1 [new RNG]
$rng1 seed 0
set rng2 [new RNG]
$rng2 seed 0

# Random inter-arrival times of TCP transfer at each source i
set RV [new RandomVariable/Exponential]
$RV set avg_ 0.3
$RV use-rng $rng1

# Random size of files to transmit
set RVSize [new RandomVariable/Pareto]
$RVSize set avg_ 10000
$RVSize set shape_ 1.5
$RVSize use-rng $rng2

# We now define the beginning times of transfers and the transfer sizes
# Arrivals of sessions follow a Poisson process.
#
for {set i 1} {$i<=$NodeNb} { incr i } {
     set t [$ns now]
     for {set j 1} {$j<=$NumberFlows} { incr j } {
 # set the beginning time of next transfer from source and attributes
 set t [expr $t + [$RV value]]
 $tcpsrc($i,$j) set starts $t
 $tcpsrc($i,$j) set sess $j
 $tcpsrc($i,$j) set node $i
         $tcpsrc($i,$j) set size [expr [$RVSize value]]
  $ns at [$tcpsrc($i,$j) set starts] "$ftp($i,$j) send [$tcpsrc($i,$j) set size]"
  # update the number of flows
   $ns at [$tcpsrc($i,$j) set starts] "countFlows $i 1"
}}
```

```
for {set j 1} {$j<=$NodeNb} { incr j } {
set Cnts($j) 0
}

# The following procedure is called whenever a connection ends
Agent/TCP instproc done {} {
global tcpsrc NodeNb NumberFlows ns RV ftp Out tcp_snk RVSize
# print in $Out: node, session, start time,  end time, duration,
# trans-pkts, transm-bytes, retrans-bytes, throughput
  set duration [expr [$ns now] - [$self set starts] ]
  puts $Out "[$self set node] \t [$self set sess] \t [$self set starts] \t\
      [$ns now] \t $duration \t [$self set ndatapack_] \t\
      [$self set ndatabytes_] \t [$self set  nrexmitbytes_] \t\
      [expr [$self set ndatabytes_]/$duration ]"
  countFlows [$self set node] 0
}
# The following recursive procedure updates the number of connections
# as a function of time. Each 0.2 it prints them into $Conn. This
# is done by calling the procedure with the "sign" parameter equal
# 3 (in which case the "ind" parameter does not play a role). The
# procedure is also called by the "done" procedure whenever a connection
# from source i ends by assigning the "sign" parameter 0, or when
# it begins, by assigning it 1 (i is passed through the "ind" variable).
#
proc countFlows { ind sign } {
global Cnts Conn NodeNb
set ns [Simulator instance]
      if { $sign==0 } { set Cnts($ind) [expr $Cnts($ind) - 1]
} elseif { $sign==1 } { set Cnts($ind) [expr $Cnts($ind) + 1]
} else {
  puts -nonewline $Conn "[$ns now] \t"
  set sum 0
  for {set j 1} {$j<=$NodeNb} { incr j } {
    puts -nonewline $Conn "$Cnts($j) \t"
    set sum [expr $sum + $Cnts($j)]
  }
  puts $Conn "$sum"
  $ns at [expr [$ns now] + 0.2] "countFlows 1 3"
} }

proc finish {} {
        global ns tf
        $ns flush-trace
close $tf
        exec grep  "a" red-queue.tr > ave.tr
        exec grep  "Q" red-queue.tr > cur.tr
        exit 0
}
$ns at 0.5 "countFlows 1 3"
$ns at [expr $sduration - 0.01] "$fmon dump"
$ns at $sduration "finish"
$ns run
```

Table 6.3: tcl script shortRed.tcl

The flow monitor file includes more detailed information on the drop type. It allows to distinguish between Early drops (ED) due to early discard of packets, and actual drops due to buffer overvlow. The file has the following format:

1. Column 1: the time at which "dump" was performed.

2. Columns 2 and 5: both give the flow id.

3. Column 3: null (a zero entry).

4. Column 4: flow type.

5. Columns 6 and 7: source and destination of the flow.

6. Columns 8 and 9: total number of arrivals of the flow in packets and in bytes, respectively.

7. Columns 10 and 11: amount of early drops of the flow in packets and in bytes, respectively.

8. Columns 12 and 13: total number of arrivals of all flows in packets and in bytes, respectively.

9. Columns 14 and 15: amount of early drops of all flows in packets and in bytes, respectively.

10. Columns 16 and 17: total amount of drops of all flows in packets and in bytes, respectively.

11. Columns 18 and 19: total amount of drops of the particular flow in packets and in bytes, respectively.

Note: in order to apply the flow monitor, each TCP connection that we wish to monitor should have a flow id. In our case, we initially identnify a flow by its number and its source node (e.g. the third TCP connection that starts at node 4). We transform this into a one dimensional vector as follows:

```
set k [expr $i*1000 +$j];
$tcpsrc($i,$j) set fid_ $k
```

The simulation produced the following output files:

1. cur.tr and ave.tr that monitor the evolution of the queue size and its averaged version;

2. Conn.tr for monitoring the number of active connections from each of the six sources (the number six is given as parameter in the script to the variable NumberFlows) as well as the sum of active sessions, as a function of time

3. Out.ns for monitoring for each session (identified with the source node and the session number originating from that node), start time, end time and duration of the connection, the number of transmitted packets, transmitted bytes and retransmitted bytes, and the throughput experienced by the session.

4. mon.tr is the trace produced by the flow monitor, that contains number of transmitted packets and bytes and number of losses per connection.

5. out.tr is the global trace of all events.

We used in the above script with the RED version withi automatic configuration. We plot the queue size and its averaged dynamics in Fig. 6.9-6.10. We see that the queue length process is much more bursty and variable than in the case of persistant TCP connections (which we saw in Fig. 6.4 and 6.5). The number of active connection is given in Fig. 6.11.

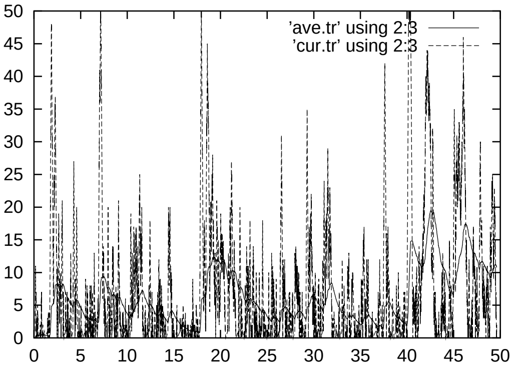

Figure 6.9: Evolution of the queue size and of its average

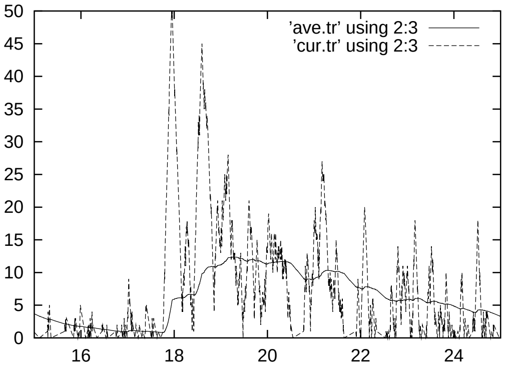

Figure 6.10: Queue size and averaged size evolution: a zoom

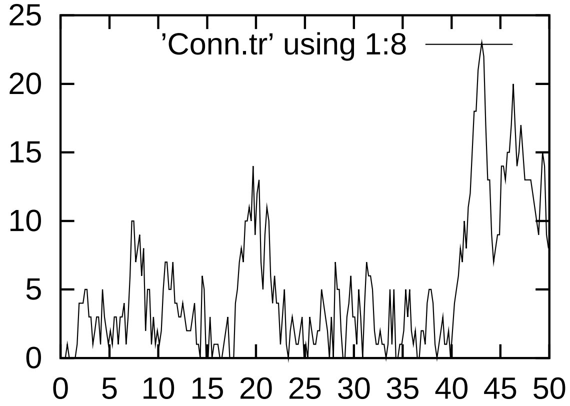

Figure 6.11: Number of active connections as a function of time

We included in the above script various ways of monitoring. The direct way of monitoring the number of retransmissions and arrivals of packets through the procedure ”done” has the is global: it gives all the data related to the connection. The flow monitor gives on the other hand local information on losses at a particular link. If the connection traverses several bottleneck links, the first method is thus more advantageous. The second method has the advantage of giving more detayled local information which can be useful to understand the contribution of each of several nodes to congestion suffered by a session.

6.5 Exercises

Exercise 6.5.1 *Consider the script shortRed.tcl (Table 6.3) and modify the program to have manual adjustment of the parameters* `thresh_, maxthresh_, q_weight_`. *How should these parameters, as well as the queue size on link N-D be chosen so as to maximize the throughput? Study this by simulation and explain the tradeoffs.*

Exercise 6.5.2 *One of the objectives of RED is to allow more fairness to short bursts. Analyze the throughput and the loss probability of a connection as a function of its size with RED and compare it to Drop-Tail. Use the script shortRed.tcl (Table 6.3). Try various parameters for RED to get better fairness. The exercise is based on [2].*

Chapter 7

Differentiated Services

In traditional Internet, all connections get the same treatment in the network. This is in contrast with other networking concepts, such as the ATM (Asynchronous Transfer Mode), that can offer quality of service requirements to connections at the price of much higher signalling and processing related to the acceptance of new connections and maintaining the guarantees of ongoing connections. Moreover, since network resources are limitted, offering guarantees on performance measures requires to reject new connections if resources are not available. This is in contrast with the best effort nature of todays the Internet where no admission control is performed.

Yet, it has been recognized it is important to differentiate between connection classes and to be able to allocate resources to connections according to their class. Thus a subscriber that is willing to pay more could benefit of smaller delays and larger throughputs. This is in particular of interest for real time applications over the Internet (voice, video).

For that reason, the Diffserv has been introduced. It is based on marking packets at the edge of the network according to the performance level that the network wishes to provide them; then according to the marks, the packets are treated differently at the network's nodes. A common way to differentiate packets is by using RED buffers using different parameters for different packets.

The ns module that handles diffserv has been developed in Nortel Networks, and this Chapter is based in large part on the excellent Nortel Report [33].

7.1 Description of assured forwarding Diffserv

The Diffserv implemented in ns follows the "Assured forwarding" approach standardized in [20]. A packet belonging to a flow may receive three possible priority levels within the flow. These are called sometime "drop precedences". This can be used, for example to provide a lower loss probability to sync packets in a TCP connection, since unlike other packets, the losses of sync packets result in very long time-outs [28]. In addition to differentiation within each flow, all flows are classified to several classes (at most four), and different treatment can be given to the different classes.

Moreover, it is possible to differentiate between flows. Four classes of flows are defined, and packets of a given class are queued in a class-dependent queue. In order to differentiate between packets belonging to the same class, three virtual queues are implemented in each of the four queues. To each of the 12 combinations of the four flow class and the three internal priority levels within a flow correspond a code point that a packet is given when entering the network. In practive not all queues and all priority groups need to be implemented.

Diffserv architecture has three components:

1. Policy and resource manager: it creates policies and distributes them to diffserv routers. A

policy determines which level of services in the network are assigned to which packets. This assignment may depend on the behavior of the source of the flow (e.g. its average rate and its burstiness) and special network elements are therefore added at the edge of the network so as to measure the source behavior. In ns simulation, the policy is fully determined in the tcl script.

2. Edge routers: are responsible to assign the code points to the packets according to the policy specified by the network administrator. To do so they measure parameters of the input traffic of each flow.

3. Core routers: the basic approach of diffserv is to keep the intelligence in the edge of the network; routers within the network have simply to assign the appropriate priority to packets according to their code mark. The priority translates to parameters of the scheduling and of the dropping decisions in the core routers.

7.2 MRED routers

7.2.1 General description

The fact that there are three virtual RED buffers (called MRED - Multi RED) in each physical queue allows to enhance its behavior and to create dependence between their operation. One way to do that is through the RIO C (Rio Coupled) version of MRED, in which the probability of dropping low priority packets (called ”out-of-profile packets”) is based on the weighted average lengths of all virtual queues, whereas the probability of dropping a high priority (”in-profile”) packet is based only on the weighted average length of its own virtual queue.

In contrast, in RIO-D (RIO De-coupled) the probability of dropping each packet is based on the size of its virtual queue. Another version is the WRED (Weighted Red) in which all probabilities are based on a single queue length [8]. It is possible to use also the dropTail queue.

7.2.2 Configuration of MRED in ns

To determine the number of physical queues, we use the command

```
$dsredq set numQueues_ $m
```

where m can take values between 1 and 4.

Configuring queue 0 to be a RIO-C is done with the command

```
$dsredq setMREDMode RIO-C 0
```

If the last argument is not given then all queues are set to be RIO-C. Similarly, types other than RIO-C can be defined. To specify the number $n of virtual queues, we use the command:

```
$dsredq setNumPrec $n
```

Red parameters are then congidured using the command

```
$dsredq configQ $queueNum $virtualQueueNum $minTh $maxTh $maxP
```

It thus has 5 parameters: the queue number, virtual queue number, min_{th}, max_{th} and max_p. The parameter q_w can also be given (as the 6th parmaeter) and if it is not stated then it is taken to be 0.002 by default.

The droptail queue can also be used with the command

```
$dsredq setMREDMode DROP
```

The configuation then is given as before with only the first three parameters:

```
$dsredq configQ $queueNum $virtualQueueNum $minTh
```

All arriving packets are dropped when the min_{th} value is reached.

As we saw in the chapter on RED, for computing the drop probability we need an estimate of the packet size. For a packet of size 1000 bytes this command is given by the command

```
$dsredq meanPktSize 1000
```

Scheduling Particular scheduling regimes can be defined. for example the weighted round robin with queue weights 5 and 1 respectively will be defined through

```
$dsredq setSchedularMode WRR
$dsredq addQueueWeights 1 5
```

Other possible scheduling are Weighted Interleaved Round Robin (WIRR), Round Robin (RR) which is the default scheduling, and the strict priorities (PRI).

PHB table The set of four queues along with the virtual queues is supplemented with a PHB (Per Hop Behavior) table. Its entries are defined by (i) the code point (ii) the class (physical queue) and (iii) the "precedence" (virtual queue). An entry is assigned with the command of the form

```
$dsredq addPHBEntry 11 0  1
```

which means that code point 11 is mapped to the virtual queue 1 of the physical queue 0.

7.2.3 TCL querying

The following three commands result in pringing respectively (i) the PHB table, (ii) the number of physical and virtual queues and (iii) the RED weighted average size of the specified physical queues (0 in our case):

```
$dsredq printPHBTable
$dsredq printStats
$dsredq getAverage 0
```

7.3 Defining policies

7.3.1 Description

All flows having the same source and destination are subject to a common policy. A policy defines a policer type, a target rate, and other policy specific parameters. It specifies at least two code points. The choice between them depends on the comparison between the flow's target and its current sending rate, and possibly on the policy-dependent parmaeters (such as burstiness). The policy specifies meter types that are used for measuring the relevant input traffic parameters. A packet arriving at the edge device causes the meter to update the state variables corresponding to the flow, and the packet is then marked according to the policy. The packet has an initial code point corresponding to the required service level; the marking can result in downgrading the service level with respect to the initial required one.

A policy table is used in ns to store the policy type of each flow. Not all entries are actually used. The entries are

1. Source node ID
2. Destination node ID
3. Policer type
4. Meter type
5. Initial code point
6. CIR (committed information rate)
7. CBS (committed burst size)
8. C bucket (current size of the committed bucket)
9. EBS (excess burst size)
10. E bucket (current size of the excess bucket)
11. PIR (peak information rate)
12. PBS (peak burst size)
13. P bucket (current size of the peak bucket)
14. Arrival time of last packet
15. Average sending rate
16. TSW window length (TSW is a policer based on average transmission rates and the averaging is performed over the window length, in seconds, of data). The default value is 1sec.

The following are the possible policer types:

1. **TSW2CM (TSW2CMPolicer):** uses a CIR and two drop precedences. The lower one is used probabilistically when the CIR is exceeded.
2. **TSW3CM (TSW3CMPolicer)** [16]: uses a CIR, a PIR and three drop precedences. The medium priority level is used probabilistically when the CIR is exceeded, and the lowest one is used probabilistically when the PIR is exceeded.
3. **Token Bucket (TokenBucketPolicer)**: uses CIR and a CBS, and two drop precedences.
4. **Single Rate Three Color Marker (srTCMPolicer)** [22]: uses CIR, CBS and EBS to choose from three drop precedences.
5. **Two Rate Three Color Marker (trTCMPolicer)** [22]: uses CIR, CBS, EBS and PBS to choose from three drop precedences.

Each of the above policer type defines the meter it uses. A policer table defines for each policy type the initial code point as well as one or two downgraded code points. The initial code point is often called "green code" and the lowest downgraded code is "red". If there is another code point inbetween, it is called "yellow".

7.3.2 Configuration

To update the policy table, the "addPolicyEntry" command is used which contains the edge queue variable denoting the edge queue, the source and destination nodes of the flow, the policer type, its initial code point, and then the values of the parameters that it uses; these are some or all of CIR, CBS, PIR and PBS as stated above. CIR and PIR are given in bps, and CBS, EBS and PBS in bytes. An example is:

```
$edgeQueue addPolicerEntry [$n1 id] [$n8 id] trTCM 10 200000 1000 300000 1000
```

Here we added a policy for the flow that originates in $n1 and ends at $n8. If the TSW policers are used, one can add at the end the TSW window length. If not added, it is taken to be 1sec by default.

Then another "addPolicyEntry" command specific to the policy and to the initial code point (and not to a particular flow) defines the downgraded code points which are common to all flows that use the policy with the same initial code point. An example is:

```
$edgeQueue addPolicerEntry srTCM 10 11 12
```

7.3.3 TCL querying

The following three commands result in pringing respectively (i) the entire policy table, (ii) the entire policer table and (iii) the current size in bytes of the C buckets:

```
$edgeQueue printPolicyTable
$edgeQueue printPolicerTable
$edgeQueue getBucket
```

7.4 Simulation of diffserv: protection of vulnerable packets

In TCP connections, the loss of some segments has more impact than others on the performance of the connection. These segments are (i) the connection establishment segments, (ii) the segments sent when the connection has a small window, and (iii) the segments sent after a timeout or a fast retransmit. We call these "vulnerable" segments, or packets. In a recent paper [28], the authors show that by marking these segments with a higher priority and implementing the priority using a diffserv architecture, the performance of the TCP connection considerably improves. This marking requires, however, that network layer elements be aware of transport layer information, i.e. of the state of the TCP connection. The goal of the simulation example we introduce is to show that one can achieve prioritization of sensitive segments without any use of transport layer information, thus simplifying the implementation of diffserv marking of TCP packets. This part is based on [1].

7.4.1 The simulated scenario

Perliminaries on the service differentiation Two priority levels are defined. The higher "In packets" or "green packets" and the lower "Out packets" or "red packets". We focus on the simplest policer available in ns: the time-sliding window (TSW2CM). A CIR is defined for each edge router. As long as the connection's rate is below CIR, all packets are marked as high priority. When the rate exceeds CIR, packets are marked probabilistically such that at the average, the rate of packets marked with high priority corresponds to the CIR. The transmitted rate is computed as the rate averaged over the "TSW window"; in our simulation its duration is 20msec.

In our experimentations we vary the CIR level at the source edge nodes and study its impact on performance.

The topology We consnider the simple network topology with a signle bottleneck, depicted in Fig. 7.1.

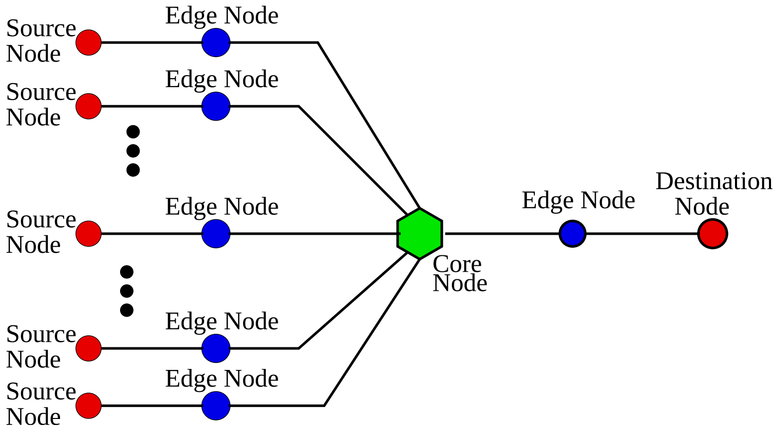

Figure 7.1: Network topology

Each source node is connected to a corresponding edge node where the traffic is marked according to parameters that will be specified. The edge routers are connected to a bottleneck corerouter, and then through another edge router, to a destination node.

There are 20 source nodes, and each one of them generates TCP connections.

We experiment with a high-speed local area type network (short propagation delays) with completely symetric links:

- Links between the edge nodes and the corresponding source nodes have delays of 10 μsec and 6Mbps bandwidth.
- Links between the edge node and the corresponding destination node has 10 μsec delay and 10Mbps bandwidth.
- Links between the core node and edge nodes that are attached to the sources have 0.1msec delay and 6Mbps bandwidth.
- The link between the core node and edge node attached to the destination has 1msec delay and 10Mbps bandwidth.

The traffic model A transfered file has a Pareto distribution with shape parameter 1.25 and an average size of 10kbytes (see [6, 36] for similar parameters).

Files to be transmitted arrive at each source node according to a Poisson process with an average rate of 5 files per second. Several sessions from the same source node can be active simultaneously.

Queueing management parameters Queue can build up only at the bottleneck router, i.e. at the link between the core node and the edge node that connects to the destination. We chose its size to be of 100 packets. Thus the qeueue management parameters at other nodes did not have an influence on the results. In the bottleneck queue at the core node, a multi-RED queue management is used with a RIO-D version; we choose the same parameters for both priority levels (more details will be given below). Our aim in choosing the parameters was not to obtain necessarily an optimal performance but rather to create conditions that allow us to study the effect of diffserv on diminishing the loss probabilities of vulnerable segments, and the impact of this action on TCP performance (delay, throughput). For that reason, we choose the same parameters for the two priority levels (this will be explained below).

For each color of packets (red, green), the averaged queue sized is monitored (this is done using the standard exponential averaging with parameter $w_q = 0.01$). Packets of a given color start to be dropped when the averaged number of queued packets of this color exceeds min_w; we choose $min_w = 15$; this drop probability increases linearly with the averaged queue size until it reaches the value $max_w = 45$, where the drop probability is taken to be $max_p = 0.5$. When this value is exceeded, the drop probability is 1.

Note that often the differentiation between the priorities is done using different sets of parameters: drops are performed at a larger queue size for green packets (e.g. [35]). We prefer not to use this approach, since with rejection at a larger window size we also get larger delays, which in some experimented parameters result in a lower throughput for green packets than for red packets and in global degradation of performance! By giving the same parameters to both priorities, we can learn about the direct effect of protecting vulnerable packets on the TCP performance. The differentiation is then done by using the RIO-D approach, in which the rejection probability of each type of color depends on the average number of packets of that type. Thus to have green packets dropped less than red ones, we simply choose their throughput (and consequently also the corresponding averge queue size) to be lower; this is done by the proper choice of the CIR value which determines the fraction of packets that will be marked green.

Simulations are 80sec long. The rate of arrival of bits to the bottleneck is

$$\frac{20 \times 1.04 \times 10^4 \times 8}{0.22} = 7.563 Mbps$$

This is obtained as follows: An average packet size is 1040 bytes of which 1000 are data and 40 bytes are an extra header. An average ftp file is assumed to contain 10^4 10^4 bytes of data, which means that its total average size (including the extra headers) is approximately $1.04 \times 10^4 \times 8$ bits. The result is obtained by multiplying by the number of source nodes and dividing by the average time between arrivals of files at a node.

The ns script is given in Table 7.1.

```
set ns [new Simulator]

# There are several sources each generating many TCP sessions sharing a bottleneck
# link and a single destination. Their number is given by the paramter NodeNb

#     S(1) ----- E(1) ----
#     .                   |
#     .     ---- E(i) ---Core---- Ed -------- D
#     .                   |
#     S(NodeNb)- E(NodeNb)-

set cir0       100000; # policing parameter
set cir1       100000; # policing parameter
set pktSize    1000
set NodeNb       20; # Number of source nodes
set NumberFlows 360 ; # Number of flows per source node
set sduration   60  ; # Duration of simulation

#Define different colors for data flows (for NAM)
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green
$ns color 4 Brown
$ns color 5 Yellow
$ns color 6 Black

set Out [open Out.ns w];   # file containing transfer
                           # times of different connections
set Conn [open Conn.tr w]; # file containing the number of connections

set tf   [open out.tr w];  # Open the Trace file
$ns trace-all $tf

#Open the NAM trace file
set file2 [open out.nam w]
# $ns namtrace-all $file2

# defining the topology
set D    [$ns node]
set Ed   [$ns node]
set Core [$ns node]
set flink [$ns simplex-link $Core $Ed 10Mb 1ms dsRED/core]
$ns queue-limit  $Core $Ed  100
$ns simplex-link $Ed $Core 10Mb 1ms dsRED/edge
$ns duplex-link  $Ed   $D  10Mb   0.01ms DropTail

for {set j 1} {$j<=$NodeNb} { incr j } {
 set S($j) [$ns node]
 set E($j) [$ns node]
 $ns duplex-link  $S($j) $E($j)  6Mb   0.01ms DropTail
 $ns simplex-link $E($j) $Core   6Mb   0.1ms dsRED/edge
 $ns simplex-link $Core  $E($j)  6Mb   0.1ms dsRED/core
 $ns queue-limit $S($j) $E($j) 100
}

#Config Diffserv
set qEdC    [[$ns link $Ed $Core] queue]
$qEdC       meanPktSize 40
$qEdC   set numQueues_   1
$qEdC    setNumPrec      2
for {set j 1} {$j<=$NodeNb} { incr j } {
 $qEdC addPolicyEntry [$D id] [$S($j) id] TSW2CM 10 $cir0 0.02
}
$qEdC addPolicerEntry TSW2CM 10 11
$qEdC addPHBEntry  10 0 0
$qEdC addPHBEntry  11 0 1
$qEdC configQ 0 0 10 30 0.1
$qEdC configQ 0 1 10 30 0.1

$qEdC printPolicyTable
$qEdC printPolicerTable

set qCEd    [[$ns link $Core $Ed] queue]
# set qCEd    [$flink queue]
$qCEd     meanPktSize $pktSize
$qCEd set numQueues_   1
$qCEd set NumPrec       2
$qCEd addPHBEntry  10 0 0
$qCEd addPHBEntry  11 0 1
$qCEd setMREDMode RIO-D
$qCEd configQ 0 0 15 45  0.5 0.01
$qCEd configQ 0 1 15 45  0.5 0.01
```

```
for {set j 1} {$j<=$NodeNb} { incr j } {
 set qEC($j) [[$ns link $E($j) $Core] queue]
 $qEC($j) meanPktSize $pktSize
 $qEC($j) set numQueues_   1
 $qEC($j) setNumPrec      2
 $qEC($j) addPolicyEntry [$S($j) id] [$D id] TSW2CM 10 $cir1 0.02
 $qEC($j) addPolicerEntry TSW2CM 10 11
 $qEC($j) addPHBEntry  10 0 0
 $qEC($j) addPHBEntry  11 0 1
# $qEC($j) configQ 0 0 20 40 0.02
 $qEC($j) configQ 0 0 10 20 0.1
 $qEC($j) configQ 0 1 10 20 0.1

$qEC($j) printPolicyTable
$qEC($j) printPolicerTable

 set qCE($j) [[$ns link $Core $E($j)] queue]
 $qCE($j) meanPktSize      40
 $qCE($j) set numQueues_   1
 $qCE($j) setNumPrec      2
 $qCE($j) addPHBEntry  10 0 0
 $qCE($j) addPHBEntry  11 0 1
# $qCE($j) configQ 0 0 20 40 0.02
 $qCE($j) configQ 0 0 10 20 0.1
 $qCE($j) configQ 0 1 10 20 0.1
}

# set flow monitor
set monfile [open mon.tr w]
set fmon [$ns makeflowmon Fid]
$ns attach-fmon $flink $fmon
$fmon attach $monfile

#TCP Sources, destinations, connections
for {set i 1} {$i<=$NodeNb} { incr i } {
for {set j 1} {$j<=$NumberFlows} { incr j } {
set tcpsrc($i,$j) [new Agent/TCP/Newreno]
set tcp_snk($i,$j) [new Agent/TCPSink]
set k [expr $i*1000 +$j];
$tcpsrc($i,$j) set fid_ $k
$tcpsrc($i,$j) set window_ 2000
$ns attach-agent $S($i) $tcpsrc($i,$j)
$ns attach-agent $D $tcp_snk($i,$j)
$ns connect $tcpsrc($i,$j) $tcp_snk($i,$j)
set ftp($i,$j) [$tcpsrc($i,$j) attach-source FTP]
} }
```

```
# Generators for random size of files.
set rng1 [new RNG]
$rng1 seed 22

# Random inter-arrival times of TCP transfer at each source i
set RV [new RandomVariable/Exponential]
$RV set avg_ 0.22
$RV use-rng $rng1

# Random size of files to transmit
set RVSize [new RandomVariable/Pareto]
$RVSize set avg_ 10000
$RVSize set shape_ 1.25
$RVSize use-rng $rng1

# dummy command
set t [$RVSize value]

# We now define the beginning times of transfers and the transfer sizes
# Arrivals of sessions follow a Poisson process.
#
for {set i 1} {$i<=$NodeNb} { incr i } {
     set t [$ns now]
     for {set j 1} {$j<=$NumberFlows} { incr j } {
 # set the beginning time of next transfer from source and attributes
 $tcpsrc($i,$j) set sess $j
 $tcpsrc($i,$j) set node $i
 set t [expr $t + [$RV value]]
 $tcpsrc($i,$j) set starts $t
         $tcpsrc($i,$j) set size [expr [$RVSize value]]
  $ns at [$tcpsrc($i,$j) set starts] "$ftp($i,$j) send [$tcpsrc($i,$j) set size]"
  $ns at [$tcpsrc($i,$j) set starts ] "countFlows $i 1"
}}

for {set j 1} {$j<=$NodeNb} { incr j } {
set Cnts($j) 0
}

# The following procedure is called whenever a connection ends
Agent/TCP instproc done {} {
global tcpsrc NodeNb NumberFlows ns RV ftp Out tcp_snk RVSize
# print in $Out: node, session, start time,  end time, duration,
# trans-pkts, transm-bytes, retrans-bytes, throughput
  set duration [expr [$ns now] - [$self set starts] ]
  set i [$self set node]
  set j [$self set sess]
  set time [$ns now]
  puts $Out "$i \t $j \t $time \t\
      $time \t $duration \t [$self set ndatapack_] \t\
      [$self set ndatabytes_] \t [$self set  nrexmitbytes_] \t\
      [expr [$self set ndatabytes_]/$duration ]"
  # update the number of flows
      countFlows [$self set node] 0
}

# The following recursive procedure updates the number of connections
# as a function of time. Each 0.2 it prints them into $Conn. This
# is done by calling the procedure with the "sign" parameter equal
# 3 (in which case the "ind" parameter does not play a role). The
# procedure is also called by the "done" procedure whenever a connection
# from source i ends by assigning the "sign" parameter 0, or when
# it begins, by assigning it 1 (i is passed through the "ind" variable).
#
proc countFlows { ind sign } {
global Cnts Conn NodeNb
set ns [Simulator instance]
      if { $sign==0 } { set Cnts($ind) [expr $Cnts($ind) - 1]
} elseif { $sign==1 } { set Cnts($ind) [expr $Cnts($ind) + 1]
} else {
  puts -nonewline $Conn "[$ns now] \t"
  set sum 0
  for {set j 1} {$j<=$NodeNb} { incr j } {
    puts -nonewline $Conn "$Cnts($j) \t"
    set sum [expr $sum + $Cnts($j)]
  }
  puts $Conn "$sum"
  $ns at [expr [$ns now] + 0.2] "countFlows 1 3"
} }

#Define a 'finish' procedure
proc finish {} {
        global ns tf qsize qbw qlost file2
        $ns flush-trace
        close $file2
        exit 0
}

$ns at 0.5 "countFlows 1 3"
$ns at [expr $sduration - 0.01] "$fmon dump"
$ns at [expr $sduration - 0.001] "$qCEd printStats"
$ns at $sduration "finish"
$ns run
```

Table 7.1: diffs.tcl Script

CIR	10kbps	30kbps	100kbps	200kbps	300kbps	1Mbps	10Mbps
lost SYN packets	120	95	53	45	17	78	114
First packets lost	125	119	90	56	37	73	115
Total losses	1699	1612	1476	1286	1088	1290	1577

Table 7.2: Protection of vulnerable packets as a function of CIR

7.5 Simulation results

Losses We check the influence of the CIR marking rate on the loss probabilities of the SYN packets and of the first data packet in a connection, as before.

We see that we manage to decrease the losses of SYN packets by a factor of seven, and the losses of the first data packet of a connection by a factor of around 3.4, both obtained at CIR of 300kbps.

Throughput and Goodput The number of data packets that were successfully transmitted during the simulations was quite independent on the CIR: it was in the average 58285, with a standard deviation of 395 packets. This is due to the fact that arrival rate of sessions does not depend on the CIR. In view of the low loss probabilities, the throughput too, is almost constant as a function of CIR.

The number of sessions The total number of sessions as a function of time is given in Fig. 7.2 for the CIR of 200kbps (the optimal) and for the case of no prioritization (CIR of 10Mbps). We see from the simulation result that our marking scheme with CIR of 300kbps gives a better performance: less number of active sessions are present under this marking. This is related to the fact that the session duration in our marking is shorter, as we show in the next section. Indeed, since the arrival rate of sessions is the same independently of CIR the average number of session should be proportional to the average duration of a session (the proportionality factor being the arrival rate of sessions).

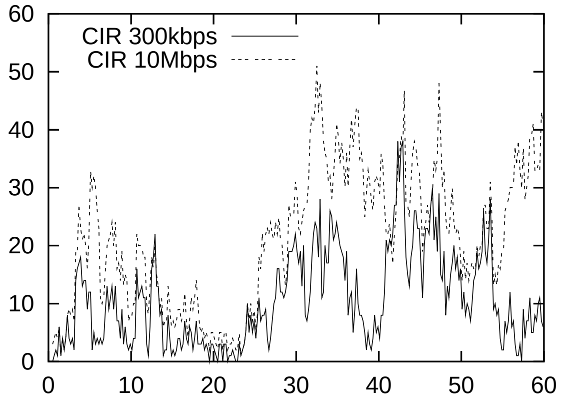

Figure 7.2: Evolution of total number of sessions

Session duration In Table 7.3 we present the average duration of a session as a function the CIR. We see that in the range of 100kbps till 300kbps the average duration decreases by a factor between 1/3 (for 100kbps) and 1/2 (for 300kbps) with respect to the case of no prioritization.

CIR	10kbps	30kbps	100kbps	200kbps	300kbps	1Mbps	10Mbps
sess. duration	0.252616	0.231077	0.167948	0.149478	0.119509	0.203202	0.225075

Table 7.3: Average duration of a session as a function of CIR

7.6 Discussions and conclusions

There a few limitation of the marking approach. The significant improvement that we obtain would not be obtained in any scenario, and we propose a few guidelines, which we validated through further simulations, to describe its limitations.

1. Vulnerable packets deteriorate considerably performance since they cause long time-outs. This is especially the case for the loss of a syn that results in a timeout of 3sec or of 6sec. In high speed networks the duration of a file transfer is short (often the whole transfer is much shorter than timeout), so we can expect to gain much by eliminating these long timeouts. In low speed networks, this is no more the case so the gains in our approach become marginal.

2. In our simulation, an average file size is 10kbytes, which is the avereaged measured file size in the Internet [36]. This means that around 10% of the packets is a SYN packet and further, another 10% of the packets are first in a transfer. Thus in the absence of our approach, around 20% of lost packets would correspond to these types of vulnerable packets, so elliminating these losses can result in a considerable improvement in performance. If we were to use our approach to much longer files, the fraction of vullnerable packets would be much smaller, so that the added value of our approach would be smaller.

7.7 Exercises

1. Our simulations have restricted to FTP type traffic. In this exercise we shall consider **HTTP type traffic:** The time between the end of transmission of a file till the beginning of the next transmission is exponentially distributed with a mean of 0.1 sec. This is called a "thinking time". Thus from each source node there can be only one file transmitted at the same time (at most one active session). Write a tcl program for this traffic model and check its performance as a function of the CIR. Compare with FTP type traffic

Chapter 8

Local area networks

8.1 Background

The objective of local area networks is to share resources that are either costly or that are not monopolized by a single user. Examples are memory hard disks and printers. Local area networks simplifies message exchanges, sharing of files and distributed computing. They allow to share instalation and operational costs.

Four main topologies of local area networks exist: the star, the meshed, the ring and the bus.

In a star topology, terminals are connected to a central hub through point-to-point links. This allows to control in a centralized way conflicts and allows to handle easily broadcast. The drawback is the reliability of its operation that is limitted due to the dependence on the central hub. Furthermore, the capacity of the central hub limits the number of stations that can be connected.

The meshed topology allows for several possible routes between two users. An example is the telephone system.

The ring network consists of repeaters linked through point-to-point unidirectional links so as to form a closed loop. Examples are the IBM token ring, IEEE 802.5 and the FDDI. It is simple, and allows to transmit on one link while receiving from the other. In order to avoid collisions only one station can transmit at a time, which means inefficient use of the resources.

indexEthernet The bus topology consists of a single segment of cable to which stations are connected; the connection is of the multipoint type: all the stations conneted to the bus here the signal, and if two stations transmit at the same time there is a collision. Examples are the token bus (IEEE 802.4) and the Ethernet (IEEE 802.3, IEEE802.9, IEEE802.12 and IEEE802.14). The bus itself is passive (it does not regenerate nor amplify the signal), which results in a limitted range. In order to extend the range one needs to use a repeaters which can connect several buses to a cable.

In topologies that are other than stars, distributed MAC (Multiple ACcess) protocols to the channel need to be implemented. The access protocols are divided into three categories: static sharing, random access, and access by demand

In **static sharing protocols**, some resource is shared in a fixed way between users. The main protocols are

1. TDMA (Time Devision Multiple Access) in which time is slotted and different slots are allocated to different users.

2. FDMA (Frequency Devision Multiple Access) in which several users can transmit simultanuously using different frequencies each. The access to satellites is frequently based on a

combination of TDMA and FDMA.

3. CDMA (Code Devision Multiple Access) several users can transmit at the same time and using the same frequencies, but each using another code. Codes are often orthogonal which decreases interference. Used in the third generation mobile networks such as the Universal Mobiles Telecommunications System (UMTS).

In **random access protocols**, access is attempted at random which may cause collisions and a need of retransmissions. Examples are the Aloha used in satellite communications and the Ethernet. In order to reduce collisions, one can first listen if there is no other signal being transmitted on the common channel, and if there is, transmission is delayed. This is called "Carrier Sense Multiple Access" (CSMA). Moreover, one can avoid transmission a long tram if during the transmission, a collision is detected. This is called "Collision Detection" (CD). This version of CSMA is denoted CSMA/CD and CSMA is implemented in ns, and in particular the CSMA/CD.

The Ethernet. A MAC protocol based on CSMA/CD is the Ethernet, and various versions of it are standarised, e.g. IEEE802.3 (10Mbps), IEEE802.9 (multimedia), IEEE802.11 (wireless Ethernet), IEEE802.12 ("AnyLan" high-speed 100Mbps, compatible with different types of LANs) and IEEE802.14 (high speed). The Ethernet includes a backoff algorithms for the retransmissions. A number M is chosen randomly, $0 \leq M < 2^k$ where $k = \min(n, 10)$, and where n is the total number of collisions already experienced by the packet. The time before retransmission is taken to be M times the collision window (which is twice the maximum propagation time of the signal in the local area network). When $n = 16$, the transmission is abandoned.

The IEE802.3 is implemented in ns. The collision window is bounded by 51.2 μsec and the local area network is limitted to 5km. It has several versions, e.g. 10base5, 10base2, 10broad36 nd others; The first number stands for the throughput in Mbps, the second for the modulation (with "broad" meaning no modulation). The silence period between transmitted frames in 10base5 is of 9.6μsec. When using a faster Ethernet, this silence time as well as the maximum propagation time reduce, and so does the maximum physical range of the network.

There exists another Ethernet technology based on switching where larger maximum throughputs can be obtained and where there are less collisions.

8.2 Simulating LANs with ns

ns simulator simulates three the levels related to a local area network: link layer protocols (such as the ARQ (Automatic Repeat reQuest protocol), the MAC protocol (e.g. Ethernet or token ring) and the physical channel.

The basic way to define a LAN through which a group of nodes is connected is by the command

```
set lan [$ns newLan <arguments>]
```

There are seven arguments:

1. A group of nodes, e.g. `"$n3 $n4 $n5"`,

2. the delay,

3. the bandwidth

4. a link layer type (e.g. "LL"),

5. the interference queue type, e.g. "QueueDrop Tail",

6. the MAC type (e.g. "`Mac/Csma/Cd`" or "`Mac/802_3`")

7. the Channel type (e.g. ”Channel”)

As an example, consider the network in Fig. 8.1 which is a modeification of the network studied in Chapter 2 (Fig. 2.2) in which nodes n3, n4 and n5 are put on a common LAN. This means that TCP packets destinated to node n4 arrive also at node n5 (and area dropped there), TCP acknowledgements sent by node n4 to node n0 also arrive at node n5 (and are dropped there) and UDP packets destinated to node n5 also arrive at node n4 (and are dropped there).

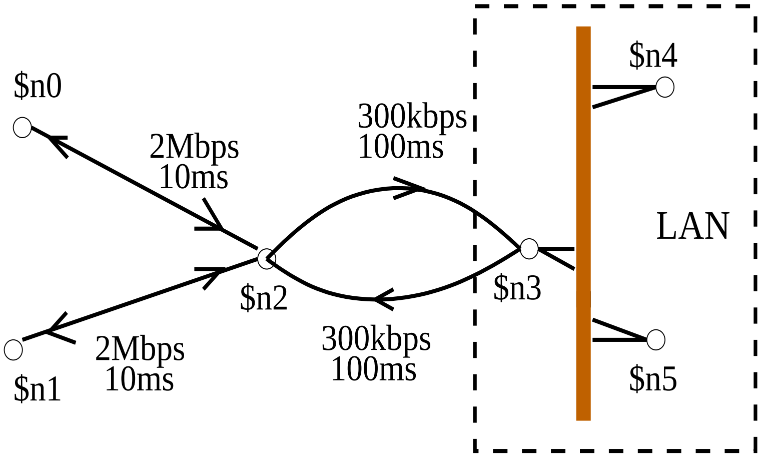

Figure 8.1: A LAN example

The script file is then the same as ex1.tcl (Table 2.4) but where we replace the commands

```
$ns duplex-link $n3 $n4 0.5Mb 40ms DropTail
$ns duplex-link $n3 $n5 0.5Mb 30ms DropTail
```

by the command

```
set lan [$ns newLan "$n3 $n4 $n5" 0.5Mb 40ms LL Queue/DropTail MAC/Csma/Cd Channel]
```

Chapter 9

Mobile networks

There are two approaches for wireless communication between two hosts. The first is the centralized cellular network in which each mobile is connected to one or more fixed base stations (each base station is responsible for another cell), so that a communication between two mobile stations require the to involve one or more base stations. A second decentralized approach consists based of an ad-hoc network between users that wish to communicate between each other. due to the more limited range of a mobile terminal (with respect to a fixed base station), this approach requires mobile nodes not only to be sources or destination of packets but also to forward packets between other mobiles. Cellular station has a much larger range than ad-hoc networks. However, ad-hoc networks have the advantage of being quickly deployable as they do not require an existing infrastructure.

In cellular networks, the wireless part is restricted only to the access to a network, and within the network classical routing protocols can be used. Ad-hoc network in contrast rely on special routing protocols that have to be adapted to frequent topology changes.

To model well cellular networks, often sophisticated simulation tools of the physical radio channel are needed, as well as the simulation of power control mechanisms. ns does not have an advanced physical layer module (although it contains some simple modeling features of radio channels).

In ad-hoc networks, in contrast, the routing protocols are central. ns allows to simulate the main existing routing as well as transport and applications that use them. Moreover, it allows to take into account the MAC and link layer, the mobility, and some basic features of the physical layer.

The current routing protocols implemented by ns are

- DSDV - Destination Sequenced Distance Vector [32],
- DSR - Dynamic Source Routing [26],
- TORA/IMPE - Temporally Ordered Routing Algorithm / Internet MANET Encapsulation Protocol [9, 29, 30],
- AODV - Ad-hoc On Demand Distance Vector [31].

9.1 The routing algorithms

There are several approaches in conventional routing algorithms in traditional wireline networks, and some ideas from these are also used in ad-hoc networks. Among the traditional approaches we shall mention the following:

1. **Link State.** Each node maintains a view of the complete topology with a cost per each link. Each node periodically broadcasts the link costs of its outgoing links to all other nodes using flooding. Each node updates its view of the network and applies a shortest path algorithm for choosing the next-hop for each destination.
2. **Distance Vector.** Each node only monitors the cost of its outgoing links. Instead of broadcasting the information to all nodes, it periodically broadcasts to each of its neighbors an estimate of the shortest distance to every other node in the network. The receiving nodes use this information to recaluclate routing tables using a shortest path algorithm. This method is more computation efficient, easier to implement and requires less storage space than link state routing.
3. **Source routing.** Routing decisions are taken at the source, and packets carry along the complete path they should take.
4. **Flooding.** The source sends the information to all neighbors who continue to sending it to their neighbors etc. By using sequence numbers for the packets, a node is able to relay a packet only once.

Next we describe the Ad-hoc routing protocols implemented in ns.

9.1.1 Destination Sequenced Distance Vector - DSDV

DSDV is a distance vector routing protocol. Each node has a routing table that indicates for each destination, which is the next hop and number of hops to the destination. Each node periodically broadcasts routing updates. A sequence number is used to tag each route. It shows the freshness of the route: a route with higher sequence number is more favorable. In addition, among two routes with the same sequence number, the one with less hops is more favorable. If a node detects that a route to a destination has broken, then its hop number is set to infinity and its sequence number updated (increased) but assigned an odd number: even numbers correspond to sequence numbers of connected paths.

9.1.2 Ad-hoc On Demand Distance Vector - AODV

AODV is a distance vector type routing. It does not require nodes to maintain routes to destinations that are not actively used. As long as the endpoints of a communication connection have valid routes te each other, AODV does not play a role.

The protocol uses different messages to discover and maintain links: Route Requests (RREQs), Route Replies (RREPs), and Route Errors (RERRs). These message types are received via UDP, and normal IP header processing applies.

AODV uses a destination sequence number for each route entry. The destination sequence number is created by the destination for any route information it sends to requesting nodes. Using destination sequence numbers ensures loop freedom and allows to know which of several routes is more "fresh". Given the choice between two routes to a destination, a requesting node always selects the one with the greatest sequence number.

When a node wants to find a route to another one, it broadcasts a RREQ to all the network till either the destination is reached or another node is found with a "fresh enough" route to the destination (a "fresh enough" route is a valid route entry for the destination whose associated sequence number is at least as great as that contained in the RREQ). Then a RREP is sent back to the source and the discovered route is made available.

Nodes that are part of an active route may offer connectivity information by broadcasting periodically loacl Hello messages (special RREP messages) to its immediate neighbours. If Hello messages stop arriving from a neighbor beyond some given time threshold, the connection is assumed to be lost.

When a node detects that a route to a neighbor node is not valid it removes the routing entry and sends a RERR message to neighbors that are active and use the route; this is possible by maintaining active neighbour lists. This procedure is repeated at nodes that receive RERR messages. A source that receives an RERR can reinitiate a RREQ message.

AODV does not allow to handle unidirectional links.

9.1.3 Dynamic Source Routing - DSR

Designed for mobile ad hoc networks with up to around two hundred nodes with possibly high mobility rate. The protocol works "on demand", i.e. without any periodic updates.

Packets carry along the complete path they should take. This reduces overheads for large routing updates at the network. The nodes store in their cache all known routes. The protocol is composed of route discovery and route maintenance.

At **route discovery**, a source requesting to send a packet to a destination broadcasts a Route Request (RREQ) packet. Nodes receiving RREQ search in their Route Cache for a route to the destination. If a route is not found then the RREQ is further transmitted and the node adds its own address to the recorded hop sequence. This continues till the destination or a node with a route to the destination are reached. The route back can be retreived by the reverse hop record. As routes need not be symetric, DSR checks the Route Cache of the replying node and if a route is found, it is used instead. Alternatively, one can piggyback the reply on a RREQ targeted at the originator. Hence unidirectional links can be handled.

Route maintenance: When originating or forwarding a packet using a source route, each node transmitting the packet is responsible for confirming that data can flow over the link from that node to the next hop. An acknowledgment can provide confirmation that a link is capable of carrying data. Acknowledgments are often already part of the MAC protocol in use (such as the link-layer acknowledgment frame defined by IEEE 802.11), or are "passive acknowledgment", i.e. a node knows that its packet is received by an intermediate node since it can hear that the intermediate node further forwards it. If such acknowledgements are not available then a node can request an acknowledgement (which can be sent directly to the source using another route). Acknowledgements may be requested several times (till some given bound), and in the persistent absence of acknowledgement, the route is removed from the Route Cache and return a "Route Error" to each node that has sent a packet routed over that link since an acknowledgement was last received. Nodes overhearing or forwarding packets should make use all carried routing information to update its own Route Packet.

9.1.4 Temporally Ordered Routing Algorithm - TORA

This protocol is of the family of link reversal protocols. It may provide several routes between a source and a destination. TORA contains three parts: creating, maintaining and erasing routes. At each node, a separate copy of TORA is is run per each destination. TORA builds a directed acyclic graph rooted at the destination. It associates a height with each node in the network (with respect to a common destination). Messages flow from nodes with heigher height to those with lower heights. Routes are discovered using Query (QRY) and Update (UPD) packets.

When a node with no downstream links needs a route to a destination, it broadcasts a QRY packet that propagates till it either finds a node with a route to the destination or the destination itself. That node will respond by broadcasting a UPD packet containing the node's height. A node receiving the UPD packet updates its height accordingly and broadcasts another UPD. This may result in a number of directed paths from the source to the destination.

If a node discovers a particular destination to be unreachable, it sets the corresponding local height to a maximum value. In case the node cannot find any neighbour with finite height w.r.t. this destination, it attempts to find a new route. In case there is no route to a destination (i.e. of a network partition), the node broadcasts a Clear (CLR) message resetting all routing states and removing invalid routes from its part of the network.

TORA operates on top of IMEP (Internet MANET Encapsulation Protocol) that provides reliable delivery of route-messages and that informs the routing protocol of changes of the links to its neighbours. IMEP tries to aggregate IMEP and TORA messages to a single packet (called block) so as to reduce overhead. To get information on the status of neighborinig links, IMEP periodically sends BEACON messages answered by HELLO response messages.

9.2 Simulating mobile networks

9.2.1 Simulation scenario

We start by presenting simple script that runs a single TCP connection over a 3-nodes network over an area of a size of 500m over 400m depicted in Fig 9.1. The location process is as follows.

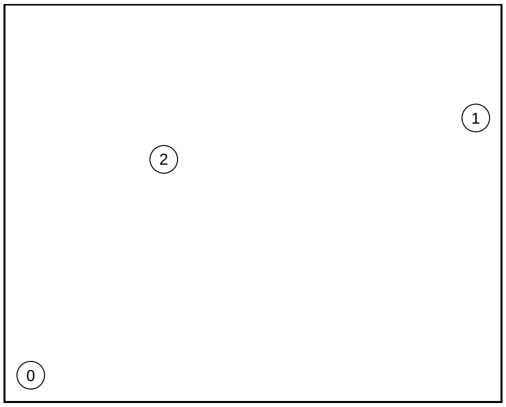

Figure 9.1: Example of a three node ad-hoc network

- The initial locations of nodes 0, 1, and 2 are respectively (5,5), (490,285) and (150,240) (the z coordinate is assumed throughout to be 0).
- At time 10, node 0 statrts moving towords point (250,250) at a speed of 3m/sec.
  At time 15, node 1 statrts moving towords point (45,285) at a speed of 5m/sec.
  At time 10, node 0 statrts moving towords point (480,300) at a speed of 5m/sec.
  Node 2 is still throughout the simulation.

The simulation lasts 150sec. At time 10, a TCP connection is initiated between node 0 and node 1.

We shall use below the DSDV ad-hoc routing protocol and the IEEE802.11 MAC protocol.

9.2.2 Writing the tcl script

We begin by specifying some basic parameters for the simulations, providing information for the different layers. This is done as follows:

```
set val(chan)           Channel/WirelessChannel    ;# channel type
set val(prop)           Propagation/TwoRayGround   ;# radio-propagation model
set val(netif)          Phy/WirelessPhy            ;# network interface type
set val(mac)            Mac/802_11                 ;# MAC type
set val(ifq)            Queue/DropTail/PriQueue    ;# interface queue type
set val(ll)             LL                         ;# link layer type
set val(ant)            Antenna/OmniAntenna        ;# antenna model
set val(ifqlen)         50                         ;# max packet in ifq
set val(nn)             3                          ;# number of mobilenodes
set val(rp)             DSDV                       ;# routing protocol
set val(x)              500                        ;# X dimension of topography
set val(y)              400                        ;# Y dimension of topography
set val(stop)           150                        ;# time of simulation end
```

These parameters are used in the configuring of the nodes, whidh is done with the help of the following command

```
      $ns node-config -adhocRouting $val(rp) \
                        -llType $val(ll) \
                        -macType $val(mac) \
                        -ifqType $val(ifq) \
                        -ifqLen $val(ifqlen) \
                        -antType $val(ant) \
                        -propType $val(prop) \
                        -phyType $val(netif) \
                        -channelType $val(chan) \
                        -topoInstance $topo \
                        -agentTrace ON \
                        -routerTrace ON \
                        -macTrace OFF \
                        -movementTrace ON

        for {set i 0} {$i < $val(nn) } { incr i } {
                set node_($i) [$ns node]
        }
```

The four last options in the node-config can each be given a value of ON or OFF. The agentTrace will give in our case the trace of TCP, routerTrace provides tracing of packets involved in the routing, macTrace is related to tracing MAC protocol packets, and movementTrace is used to allow tracing the motion of nodes (for nam).

The initial location of node 0 is given as follows:

```
$node_(0) set X_ 5.0
$node_(0) set Y_ 5.0
$node_(0) set Z_ 0.0
```

and similarly we provide the initial locaation of other nodes.

A linear momvement of a node is generated by specifying the time in which it starts, the x and y values of the target point and the speed. For eaxmple, node's 1 movement will be written as

```
$ns at 15.0 "$node_(1) setdest 45.0 285.0 5.0"
```

We need to create the initial node position for nam using

```
for {set i 0} {$i < $val(nn)} { incr i } {
# 30 defines the node size for nam
$ns initial_node_pos $node_($i) 30
}
```

We tell nodes when the simulation ends with

```
for {set i 0} {$i < $val(nn) } { incr i } {
    $ns at $val(stop) "$node_($i) reset";
}
```

We then create the TCP connection and the ftp application over it as usual, see e.g. Chapter 4. Ending the simulation is also as usual, except for an additional command for ending nam:

```
$ns at $val(stop) "$ns nam-end-wireless $val(stop)"
```

The complete trace of our program is given in Table 9.1.

9.3 Trace format

An example of a line in the output trace is

```
r 40.639943289 _1_ AGT  --- 1569 tcp 1032 [a2 1 2 800] -------
   [0:0 1:0 32 1] [35 0] 2 0
```

- The first field is a letter that can have the values r,s,f,D for "received", "sent", "forwarded" and "dropped", respectively. It can also be M for giving a location or a momvement indication, this is described later.
- The second field is the time.
- The third field is the node number.
- The fourth field is MAC to indicate if the packet concerns a MAC layer, it is AGT to indicate a the transport layer (e.g. tcp) packet, or RTR if it concerns the routed packet. It can also be IFQ to indicate events related to the interference priority queue (like drop of packets).
- After the dahses come the global sequence number of the packet (this is not the tcp sequence number).
- At the next field comes more information on the packet type (e.g. tcp, ack or udp).
- Then comes the packet size in bytes.
- The 4 numbers in the first square barkets concern mac layer information. The first hexadecimal number, a2 (which equals 162 in decimal) specifies the expected time in seconds to send this data packet over the wireless channel. The second number, 1, stands for the MAC-id of the sending node, and the third, 2, is that of the receiveing node. The fourth number, 800, specifies that the MAC type is `ETHERTYPE_IP`.
- The next numbers in the second square brackets concern the IP source and destination addresses, then the ttl (Time To Live) of the packet (in our case 32),
- The third brackets concern the tcp information: its sequence number and the acknowledgement number.

There are other formats related to other routing mechanisms and/or packet types.

A movement command has the form:

```
M 10.00000 0 (5.00, 5.00, 0.00), (250.00, 250.00), 3.00
```

where the first number is the time, the second is the node number, then comes the origin and destination locations, and finally is given the speed.

```
# A 3-node example for ad-hoc simulation with DSDV

# Define options
set val(chan)           Channel/WirelessChannel    ;# channel type
set val(prop)           Propagation/TwoRayGround   ;# radio-propagation model
set val(netif)          Phy/WirelessPhy            ;# network interface type
set val(mac)            Mac/802_11                 ;# MAC type
set val(ifq)            Queue/DropTail/PriQueue    ;# interface queue type
set val(ll)             LL                         ;# link layer type
set val(ant)            Antenna/OmniAntenna        ;# antenna model
set val(ifqlen)         50                         ;# max packet in ifq
set val(nn)             3                          ;# number of mobilenodes
set val(rp)             DSDV                       ;# routing protocol
set val(x)              500                        ;# X dimension of topography
set val(y)              400                        ;# Y dimension of topography
set val(stop) 150                        ;# time of simulation end

set ns   [new Simulator]
set tracefd       [open simple.tr w]
set windowVsTime2 [open win.tr w]
set namtrace      [open simwrls.nam w]

$ns trace-all $tracefd
$ns namtrace-all-wireless $namtrace $val(x) $val(y)

# set up topography object
set topo       [new Topography]

$topo load_flatgrid $val(x) $val(y)

create-god $val(nn)

#
#  Create nn mobilenodes [$val(nn)] and attach them to the channel.
#
```

```
# configure the nodes
        $ns node-config -adhocRouting $val(rp) \
                         -llType $val(ll) \
                         -macType $val(mac) \
                         -ifqType $val(ifq) \
                         -ifqLen $val(ifqlen) \
                         -antType $val(ant) \
                         -propType $val(prop) \
                         -phyType $val(netif) \
                         -channelType $val(chan) \
                         -topoInstance $topo \
                         -agentTrace ON \
                         -routerTrace ON \
                         -macTrace OFF \
                         -movementTrace ON

for {set i 0} {$i < $val(nn) } { incr i } {
set node_($i) [$ns node]
}

# Provide initial location of mobilenodes
$node_(0) set X_ 5.0
$node_(0) set Y_ 5.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 490.0
$node_(1) set Y_ 285.0
$node_(1) set Z_ 0.0

$node_(2) set X_ 150.0
$node_(2) set Y_ 240.0
$node_(2) set Z_ 0.0

# Generation of movements
$ns at 10.0 "$node_(0) setdest 250.0 250.0 3.0"
$ns at 15.0 "$node_(1) setdest 45.0 285.0 5.0"
$ns at 110.0 "$node_(0) setdest 480.0 300.0 5.0"

# Set a TCP connection between node_(0) and node_(1)
set tcp [new Agent/TCP/Newreno]
$tcp set class_ 2
set sink [new Agent/TCPSink]
$ns attach-agent $node_(0) $tcp
$ns attach-agent $node_(1) $sink
$ns connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns at 10.0 "$ftp start"

# Printing the window size
proc plotWindow {tcpSource file} {
global ns
set time 0.01
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
puts $file "$now $cwnd"
$ns at [expr $now+$time] "plotWindow $tcpSource $file" }
$ns at 10.1 "plotWindow $tcp $windowVsTime2"

# Define node initial position in nam
for {set i 0} {$i < $val(nn)} { incr i } {
# 30 defines the node size for nam
$ns initial_node_pos $node_($i) 30
}

# Telling nodes when the simulation ends
for {set i 0} {$i < $val(nn) } { incr i } {
    $ns at $val(stop) "$node_($i) reset";
}

# ending nam and the simulation
$ns at $val(stop) "$ns nam-end-wireless $val(stop)"
$ns at $val(stop) "stop"
$ns at 150.01 "puts \"end simulation\" ; $ns halt"
proc stop {} {
    global ns tracefd namtrace
    $ns flush-trace
    close $tracefd
    close $namtrace
}

$ns run
```

Table 9.1: tcl script wrls-dsdv.tcl for TCP over an ad-hoc network

9.4 Analysis of simulation results

At the beginning the nodes are too far away and a connection cannot be set. The first TCP signaling packet is transmitted at time 10 but the connection cannot be opened. Meanwhile nodes 0 and nodes 1 start moving towards node 2. After 6 second (timeout) a second reatempt occurs but still the connection cannot be established and the timeout value is doubled to 12sec. At time 28 another transmission attempt occurs. The timeout value is doubled again to 24 sec and again to 48 sec. Thus only at time 100 sec the connection has been established. The nodes 1 and 0 are close to each other so there is a direct connection established. The mobiles get further apart till the direct link brakes. The routing protocol is too slow to react and to create an alternative route. The window evolution is given in Fig. 9.2 and a snap-shot of nam at time 124.15 sec is given in Fig. 9.4.

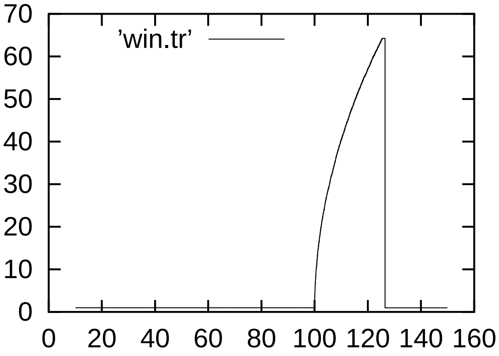

Figure 9.2: TCP window size in a three node scenario with DSDV routing protocol

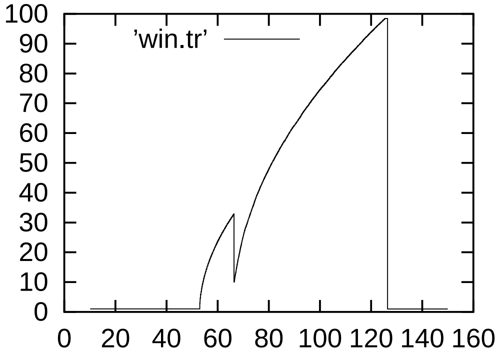

Figure 9.3: TCP window size in a three node scenario with DSDV routing protocol with both two and a single hop path

Next we slightly change the parameters of the simulation. The only change is in fact that the ftp transfer will start now at time 12 instead of at time 10. This will cause both nodes 0 as well as node 1 to be within the radio of node 2 when the timeout at around 53 sec expires so that when tcp connection is reattempted at that time a two hop path is established between node 0 and node 1. This is illustarted in Fig. 9.5. At time 66 there noeds 0 and 1 are sufficiently close so a direct connection is established. The window size evolution is given in Fig 9.3. At the moment of the path change there is a single TCP packet loss that causes the window to decrease.

At time 125.5 nodes 0 and 1 are too far apart for the connection to be maintained and the connection brakes.

9.5 Comparison with other ad-hoc routing

9.5.1 TCP over DSR

We first change the routing protocol to DSR by changing in wrls-dsdv.tcl the corresponding line to

```
set val(rp)  DSR  ;# routing protocol
```

When performing the simulation, we observe five phases of operation. In the first and last, the nodes are too far away and there is no connectivity. During phase 2 and 4, connection between

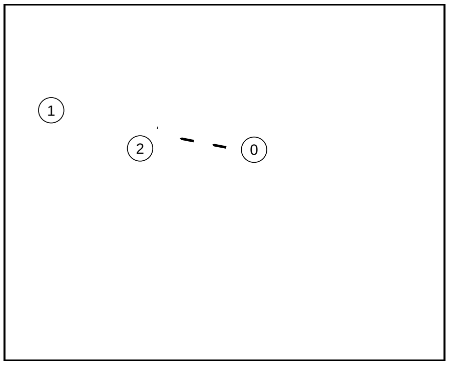

Figure 9.4: TCP in a three node scenario with DSDV routing protocol, time 124.14 sec, a single hop path

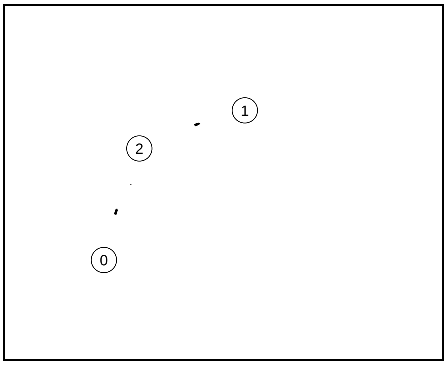

Figure 9.5: TCP in a three node scenario with DSDV routing protocol, time 58 sec: a 2 hop path

nodes 0 and 1 use node 2 as a relay, whereas in the 3rd phase, there is a direct path between node 0 and 1.

Phase 2 starts at around time 40. Phase 3 starts at around 60 sec. At time 125.50 the fourth phase starts and at time 149 sec it ends, which ends the whole connection. This is described in Figure 9.6.

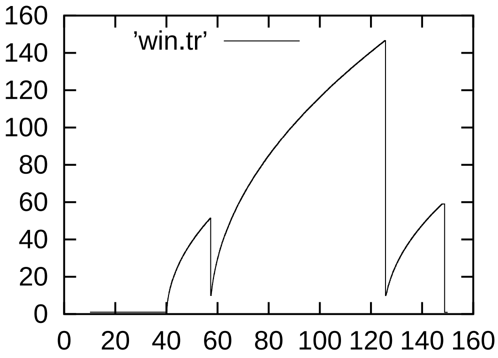

Figure 9.6: Window size evolution of the TCP connection for DSR

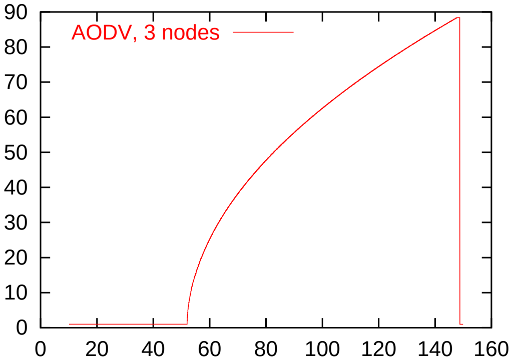

Figure 9.7: Window size evolution of the TCP connection for AODV

Here are some further observations:

- We note that in the DSDV, the system was not able to provide the 4th phase, so the connection was ended much earlier.
- The total number of TCP packets transfered using DSR is much larger than in DSDV. In DSR, 6770 TCP (data) packets have been received during the simulation, whereas in DSDV with the same parameters (corresponding to the script wrls-dsdv.tcl) it is 2079. (We can obtain this information by typing

  ```
  grep "^r" simple.tr | grep "tcp" | grep "_1_ AGT" > tcp.tr
  ```

  and then counting the number of lines. Or we can be more precise and look at the sequence number of the last received tcp packet.)

If we follow the trace of a TCP packet, say the one with sequence number 6, we see that it appears various times:

```
s 40.298003207 _0_ AGT  --- 1507 tcp 1040 [0 0 0 0] ...
r 40.298003207 _0_ RTR  --- 1507 tcp 1040 [0 0 0 0] ...
s 40.298003207 _0_ RTR  --- 1507 tcp 1060 [0 0 0 0] ...
f 40.310503613 _2_ RTR  --- 1507 tcp 1060 [13a 2 0 800] ...
r 40.310528613 _2_ RTR  --- 1507 tcp 1060 [13a 2 0 800] ...
f 40.310528613 _2_ RTR  --- 1507 tcp 1068 [13a 2 0 800] ...
r 40.348863637 _1_ RTR  --- 1507 tcp 1068 [13a 1 2 800] ...
r 40.348863637 _1_ AGT  --- 1507 tcp 1040 [13a 1 2 800] ...
```

It is first sent by the TCP agent at node 0, then received by the routing protocol of the same node and sent from there with an additional header. It is then received and forwarded by node 2, till finally it is received at node 1 at the routing level and then by the TCP agent. The above trace was obtained by enabeling the tracing of agentTrace and routerTrace. 4 other lines concerning the same packet will appear if we enable also the tracing of macTrace.

9.5.2 TCP over AODV

The simulations with the same parameters as before is repeated with AODV. The window size is given in Figure 9.7. The connection transfered altogether 3924 TCP data packets. It had throughout a long single phase in which the same two hop path was used, in which node 2 relayed the packets.

Due to the fact that changes in paths were avoided, there were no losses so the window remained high. However, we see that it reaches values less than DSR. This is due to the fact that the round trip time (needed to increase the window by one unit) is longer since a direct path is not used here. This explains the fact that it transfers less data during the simulation than DSR. We thus see that finding a shorter path results in a better TCP performance.

9.5.3 TCP over TORA

With the same parameters as the previous simulations, i.e. wrls-dsdv.tcl, TORA gave no packet transfers at all! To increase connectivity, we added another fixed node at point (250,240) which only serves to relay packets. The window size evolution is given in Fig 9.8.

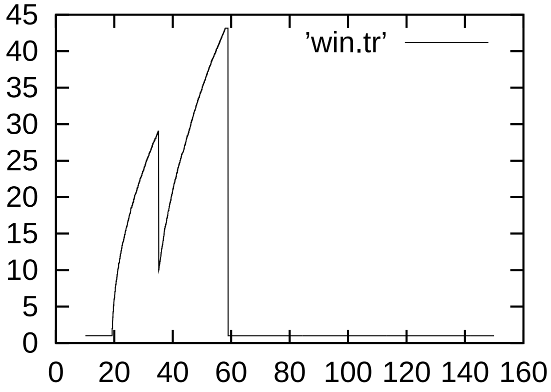

Figure 9.8: Window size of TCP over Tora with 4 nodes

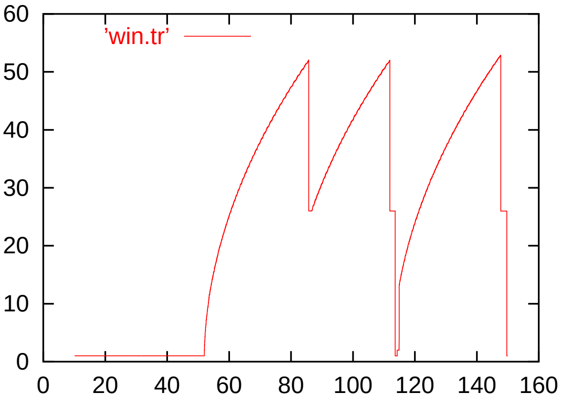

Figure 9.9: TCP over AODV with large value of maximum window

We from the nam animation (or from the output trace) the following evolution. At the beginning there is no connectivity. When connectivity starts, a path is established using all nodes: 0-2-3-1 (see Fig. 9.10 that describes the situation at time 33). At time 34.5sec a shorter forward path is established: 0-2-1, but the path of ACKs remains unchanged. Then at time 44 the ACK path changes to 1-3-0 (e.g. Fig. 9.11).

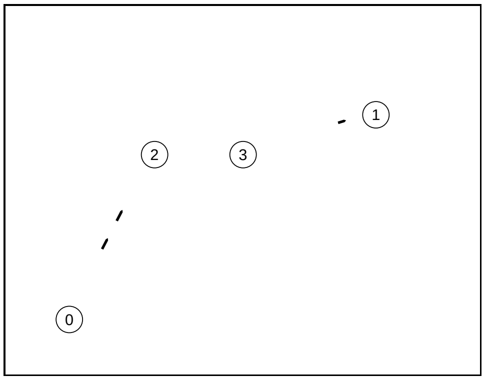

Figure 9.10: TCP over Tora with 4 nodes, time 33

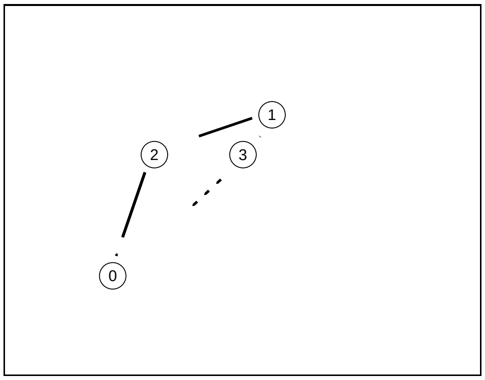

Figure 9.11: TCP over Tora with 4 nodes, time 56

9.5.4 Some comments

In the examples that we considered, losses occurred either when the geographical range was too far for reception or when there was a route change, and there were no losses due to buffer overflow. This is due to the fact that we used the default value of the maximum window size of TCP of 20. Thus the actual window that is used is the minimum between the congestion window and 20. In Fig. 9.9 we present the window size ovultion of TCP using AODV under the same conditions as those that were used to obtain Fig. 9.7 but with a maximum window size of 2000. We see that we obtain also losses due to overflow.

9.6 The interaction of TCP with the MAC protocol

9.6.1 Background

In the previous sectoins we considered a small number of mobiles, and saw how mobility phenomena influenced the performance of TCP. When there are a large number of terminals, new particular phenomena due to the MAC and physical layers may have a critical influence on TCP performance. To understand this interaction we first describe some aspects of the operation of the IEEE802.11 MAC layer and of the physical layer.

Each transmission of a DATA packet at the MAC level is part of a four-way handshake protocol. The mobile that wishes to send a packet, which we call M1, first sends an RTS (Request to Send) packet. If the destination mobile, which we call M2, can receive the packet, it sends a CTS (Clear to Send) packet. If M1 receives the CTS it can then send the DATA packet (e.g. TCP data or ACK packet). Finally, M2 sends a (MAC layer) ACK so that M1 knows that the data packet has been well received.

This handshake protocol is intended to reduce the collision probability. Collisions may occur since a mobile, say M3, may wish to send a packet to M2 at the same time as M1 does; M3 may be out of range to sense the transmission from M1, so a collision of M1's and M3's packets may occur at M2. This phenomenon is called the "hidden terminal phenomenon". With the handshake protocol, M3 will not attempt to send any packet when it hears the CTS packet sent by M2 to M1.

If a sender M1 does not receive a CTS packet then it differs its transmission and makes later attempts of sending an RTS. A sender drops the DATA packet if it has resent the RTS message seven times and has not heard a CTS reply from the receiver. A DATA packet is also dropped after four retransmissions without receveing a (MAC layer) ACK.

Although the handshake reduces the probability of "hidden terminal" collisions, it does not elliminate them. To understand how such collisoins may occur, we should take into account the geographical range of interference and reception. Current hardware specifies that transmission range is aobut 250m and the carrier sensing range as well as the interference range are about 550m. Consider the chain topology in Figure 9.12, where the distance between noes is 200m. Although nodes that are two hops apart are not hidden from each other, nodes that are three hops apart are, and may create collisions. Indeed, if node M4 wishes to send a packet to M5 during a transmission from node M1 to M2, it cannot hear the CTS from node M2 since it is out of the 250m for good reception. It cannot hear M1's RTS or DATA packet since it is more than 550m away from M1. Therefore M4 may initiate transmission to M5 that will collide at node M2 with transmissions from M1. We shall study in this Section the impact of this types of collisions on TCP performance using ns simulations, restricting to the chain topology. We shall not consider mobility aspects here. We refer to [3, 19, 37] for more details.

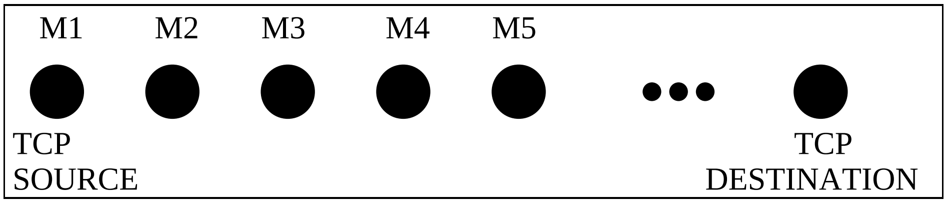

Figure 9.12: The chain topology

The phenomenon that we just described limits the number of packets that can be simultaneously transmitted in an ad-hoc network without collisions. This spatial constraint turns out to be the main factor limitting the performance of TCP in such environment and not buffer overflow. It is shown in [19] that for our chain topology, it is benefitial to limit the maximum window size of TCP to around $n/4$; further increase in the maximum window size causes more collisions and a deterioration of the throughput. In this section we shall check this assersion by simulations. Moreover, since the number of simultaneous packets that can be transmitted is limited, we shall try to improve TCP throughput by decreasing the ACK flows, using delayed ACK. ns allows us to simulate delayed ACKs with $d = 2$. We shall further show how to handle the case of $d > 2$ by making change in ns simulator.

9.6.2 The simulated scenario

We use the standard two-ray ground propagation model, the IEEE802.11 MAC, and an omnidirectional antenna model of ns. We use the AODV routing algorithm, an interface queue length of 50 at each node We tested the NewReno version of TCP, which is the most deployed one. We tested four scenarios: 3,9,20 and 30 nodes. The cases of 3 and 9 nodes required 150 sec per simulation (to obtain stationary behavior). The other cases required 1500 sec per simulation. A TCP data packet is taken to be of size 1040 bytes (including the header). The script for the case of delayed ACK (with $d = 2$) is given in Table 9.2. Below, when configuring the nodes we shall use the option "macTrace ON" in order to have detailed tracing of MAC protocols packets. This will allow us to analyse the reason of each TCP packet loss that occurs.

```
# Define options
set val(chan)           Channel/WirelessChannel    ;# channel type
set val(prop)           Propagation/TwoRayGround   ;# radio-propagation model
set val(netif)          Phy/WirelessPhy            ;# network interface type
set val(mac)            Mac/802_11                 ;# MAC type
set val(ifq)            Queue/DropTail/PriQueue    ;# interface queue type
set val(ll)             LL                         ;# link layer type
set val(ant)            Antenna/OmniAntenna        ;# antenna model
set val(ifqlen)         50                         ;# max packet in ifq
set val(nn)             9                          ;# number of mobilenodes
set val(rp)             AODV                       ;# routing protocol
set val(x)              2200       ;# X dimension of topography
set val(y)               500       ;# Y dimension of topography
set val(stop) 150    ;# time of simulation end

set ns   [new Simulator]
set tracefd       [open simple.tr w]
set windowVsTime2 [open win.tr w]

$ns trace-all $tracefd

# set up topography object
set topo       [new Topography]

$topo load_flatgrid $val(x) $val(y)

create-god $val(nn)

#
#  Create nn mobilenodes [$val(nn)] and attach them to the channel.
#

# configure the nodes
        $ns node-config -adhocRouting $val(rp) \
 -llType $val(ll) \
 -macType $val(mac) \
 -ifqType $val(ifq) \
 -ifqLen $val(ifqlen) \
 -antType $val(ant) \
 -propType $val(prop) \
 -phyType $val(netif) \
 -channelType $val(chan) \
 -topoInstance $topo \
 -agentTrace ON \
 -routerTrace ON \
 -macTrace ON \
 -movementTrace OFF

for {set i 0} {$i < $val(nn) } { incr i } {
set node_($i) [$ns node]
}
```

```
# Provide initial location of mobilenodes


for {set i 0} {$i < $val(nn)} { incr i } {
$node_($i) set X_ [expr ($i+1)*200.0]
$node_($i) set Y_ 250.0
$node_($i) set Z_ 0.0
}

# Set a TCP connection between node_(0) and node_(8)
set tcp [new Agent/TCP/Newreno]
$tcp set class_ 2
$tcp set  window_ 2000
Agent/TCPSink/DelAck set interval_ 100ms
set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $node_(0) $tcp
$ns attach-agent $node_(8) $sink
$ns connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns at 1.0 "$ftp start"

# Printing the window size
proc plotWindow {tcpSource file} {
global ns
set time 0.1
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
puts $file "$now $cwnd"
$ns at [expr $now+$time] "plotWindow $tcpSource $file" }
$ns at 1.1 "plotWindow $tcp $windowVsTime2"


# Telling nodes when the simulation ends
for {set i 0} {$i < $val(nn) } { incr i } {
    $ns at $val(stop) "$node_($i) reset";
}

$ns at $val(stop) "stop"
$ns at [expr $val(stop)+0.1]  "puts \"end simulation\" ; $ns halt"
proc stop {} {
    global ns tracefd
    $ns flush-trace
    close $tracefd
}

$ns run
```

Table 9.2: tcl script tcpwD.tcl for TCP over a static ad-hoc network with a chain topology

9.6.3 Simulation results

Our simulation results for $n = 9, 20$ and 30 nodes are summarized in Tables 9.13-9.15, respectively.

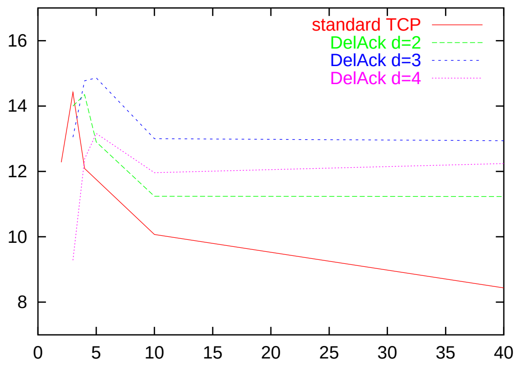

Figure 9.13: Throughput in pkt/sec for $n = 9$ as a function of the maximum window size

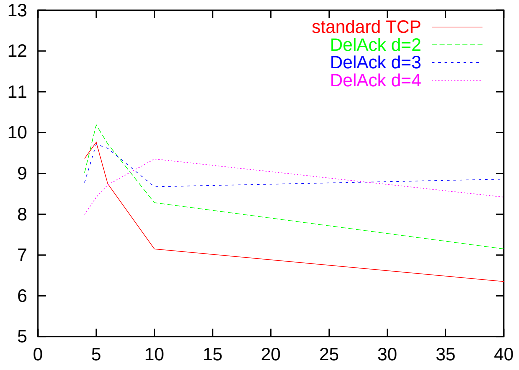

Figure 9.14: Throughput in pkt/sec for $n = 20$ as a function of the maximum window size

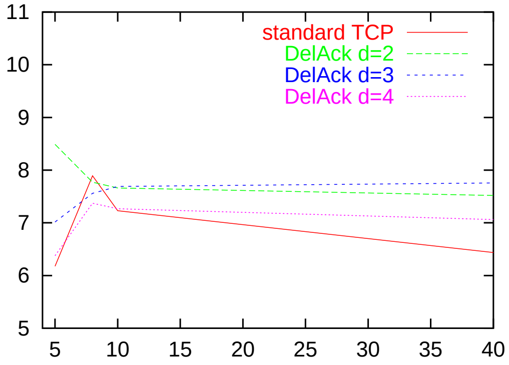

Figure 9.15: Throughput in pkt/sec for $n = 30$ as a function of the maximum window size

We see that the standard Delayed Ack option $(d = 2)$ slightly outperforms the standard TCP (yet with another value of maximum window size) for $n = 9$, and largely outperforms (more than 10%) the standard TCP for $n = 30$. A further improvement is obtained by the Delayed Ack with $d = 3$ (for both $n = 9$ as well as $n = 20$). But the most important improvement that we see is that all delayed ACK versions are better than the standard TCP for maximum window sizes of more than 10, with the options of $d = 3$ or $d = 4$ outperforming the standard delayed ACK option. For $n = 9$, the Delayed ACK version with $d = 3$ is seen to yield between 30% to 40% of improvement over standard TCP for any maximum window sizes larger than 10; in that range it also outperforms standard TCP by 20%-30% for $n = 20$ and by $6\% - 20\%$ for $n = 30$. The version $d = 4$ performs even better for $n = 20$ for maximum windows between 10 to 25. An even better performance of delayed ACK can be obtained by optimizing over the timer duration of the Delayed Ack options, as we shall see later.

Yet the most important conclusion from the curves is the robustness of the Delayed Ack options. In practice, when we do not know the number of nodes, there is no reason to limit the maximum window size to a small value, since this could deteriorate the throughput considerably. When choosing large maximum window, the delayed ACK versions considerably outperform standard

TCP. They achieve almost the optimal value that the standard TCP could achieve if it knew the number of nodes and could choose accordingly the maximum window.

For a fixed small size of maximum window size, the Delayed Ack option does not outperform the standard TCP version since most of the time, the window size limits the number of transmitted TCP packets to less than d, which means that the delayed ACK option has to wait until the timer (of 100ms by default) expires before generating an ACK; during that time the source cannot transmit packets.

Next, we plot the window size evolution for $n = 9$ for standard TCP and for TCP with delayed ACK option with $d = 3$. The window size is sampled every 0.1 sec. We see that although the maximum window size is 2000, the actual congestion window does not exceed the value of 13. We see that from the figures that in standard TCP, losses are more frequent and more severe (resulting in timeouts) whereas the $d = 3$ version of delayed ACK does not give rise to timeouts.

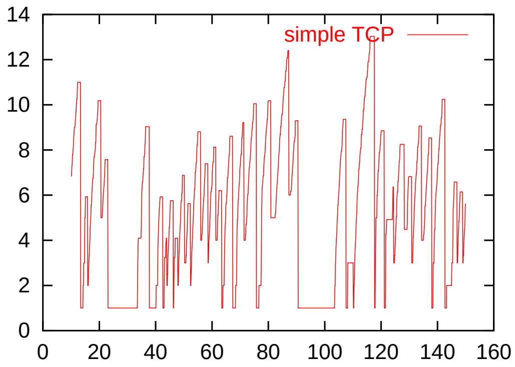

Figure 9.16: Window size evolution for standard TCP with maximum window of 2000

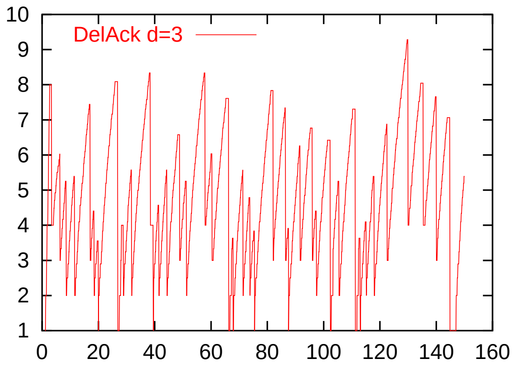

Figure 9.17: Window size evolution for DelAck TCP with $d = 3$, with maximum window of 2000

In Figure 9.18 we present the evolution of the congestion window size for standard TCP with maximum window size of 3 for the case of 9 nodes. We know from [19] that a maximum size of between 2 and 3 should indeed give optimal performance (and this is confirmed in Figure 9.13). We see in Figure 9.18 that there are almost no losses. Note that the actual window size is the minimum between the congestion window (depicted in the Figure) and the maximum window size (whose value here is 3).

In the previous Figures, all versions using delayed Acks had the default interval of 100msec (as explained in the Introduction). Next, we vary the interval length and check its impact on throughput, see Fig. 9.19. We consider the delayed ACK version with $d = 3$. We see that the default value performs quite well, although for small maximum windows, shorter intervals perform slightly better, whereas with large maximum window, a larger interval (130ms) is slightly better. We tried to further increase the time interval beyond 130ms but then the throughput decreased.

Finally, we consider the case of $n = 3$ nodes. In that case the hidden terminal phenomenon does not occur anymore, so we do not observe TCP losses for any value of window size. Even then, delayed ACKs can be used to improve considerably the performance. This is illustrated in Table 9.3 that gives the number of TCP packets successfully received within 149 sec for $n = 3$. Since there are no losses, then as long as d is greater than the max window, we expect to improve the performance as d gets larger, since TCP packets compete with less ACKs. This is indeed confirmed in Table 9.3. The improvement that increases from 10% to 15% as d grows from 2 to 4, does not depend on the maximum window (as long as it is greater than d). However for $d = 4$ we see, as can be expected, that we get a bad performance for a maximum window of 3, since the

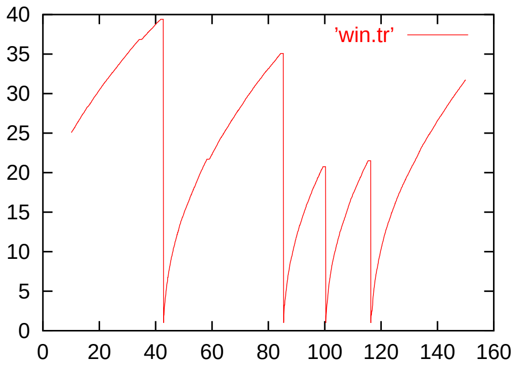

Figure 9.18: Window size evolution for standard TCP (delayed ACK disabled) with 9 nodes and maximum window size of 3

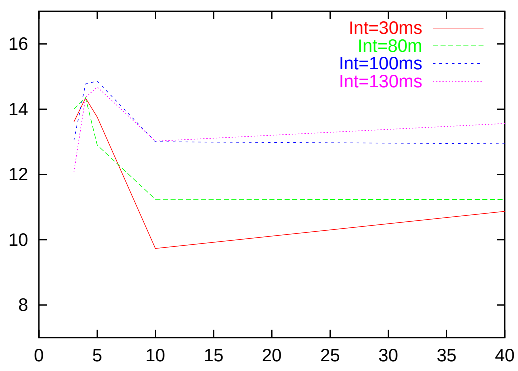

Figure 9.19: The influence of Delayed Ack interval on TCP throughput, as a function of the maximum window size. $d = 3$

	Standard TCP	*Delayed Ack Versions*		
WinMax	Standard	$d = 2$	$d = 3$	$d = 4$
3	6068	6602	6763	2699
2000	6094	6565	6779	6888

Table 9.3: Number of transmitter packets during 149sec for $n = 3$ as a function of the maximum window size

destination always needs to wait till the 100ms interval of the Delayed Ack option expires in order to send an ACK (since the windows allows for sending only 3 data packets).

9.6.4 Modification of ns for the case $d > 2$

Chapter 10

Classical queueing models

ns simulator can be used to simulate classical queueing models. In the simplest classical models, the time between packets arrival is random and has some general probability distribution, and the time it takes to transmit a packet is random as well distributed according to some other distribution. The fact that the transmission time varies may reflect a situation of a constant transmission rate but a varying size of a packet. The mathematical analysis of queueing example we present here as well as many others can be found [27].

10.1 Simulating an M/M/1, M/D/1 and D/M/1 queues

The queueing example which is the simplest for mathematical analysis is the $M/M/1$ queue: interarrival times are exponentially distributed with some parameter, say λ, and the transmission duration of a packet has an exponential distributioin with another parameter, say μ. One packet can be transmitted at a time, and the buffer size is infinite. If we denote $\rho = \lambda/\mu$, the time average number of packets in the system is given by

$$E[Q] = \frac{\rho}{1-\rho}. \tag{10.1}$$

In Table 10.1 we present a simulation of this queue. The simulation produces a trace file out.tr with all events, and also a monitor-queue trace called qm.out, as discussed in Section 4.3. By plotting columns 5 (queue size in packets) against column 1 (time) we obtain (see Fig. 10.1) the queue length evolution. The average simulated queue size over 1000sec is 9.69117, a good approximation of the value 10 obtained by (10.1).

Note that we use a simpler way to declare and manipulate random variables than the one described in Section 2.7: we do not declare generators and seeds.

It is quite interesting to analyze the simulation results and try to find the possible reasons for the difference. Once we do so we may find several reasons for the simulation's impressisions (and use the conclusions to improve the simulations):

```
set ns [new Simulator]

set tf [open out.tr w]
$ns trace-all $tf

set lambda 30.0
set mu     33.0

set n1 [$ns node]
set n2 [$ns node]
# Since packet sizes will be rounded to an integer
# number of bytes, we should have large packets and
# to have small rounding errors, and so we take large bandwidth
set link [$ns simplex-link $n1 $n2 100kb 0ms DropTail]
$ns queue-limit $n1 $n2 100000

# generate random interarrival times and packet sizes
set InterArrivalTime [new RandomVariable/Exponential]
$InterArrivalTime set avg_ [expr 1/$lambda]
set pktSize [new RandomVariable/Exponential]
$pktSize set avg_ [expr 100000.0/(8*$mu)]

set src [new Agent/UDP]
$ns attach-agent $n1 $src

# queue monitoring
set qmon [$ns monitor-queue $n1 $n2 [open qm.out w] 0.1]
$link queue-sample-timeout

proc finish {} {
    global ns tf
    $ns flush-trace
    close $tf
    exit 0
}

proc sendpacket {} {
    global ns src InterArrivalTime pktSize
    set time [$ns now]
    $ns at [expr $time + [$InterArrivalTime value]] "sendpacket"
    set bytes [expr round ([$pktSize value])]
    $src send $bytes
}

set sink [new Agent/Null]
$ns attach-agent $n2 $sink
$ns connect $src $sink
$ns at 0.0001 "sendpacket"
$ns at 1000.0 "finish"

$ns run
```

Table 10.1: tcl script mm1.tcl for simulating an MM1 queue

- the formula (10.1) counts the whole packet that is being transmitted, where as the simulation counts only the fraction of the transmitted packet that is still in the queue. This difference should make the simulated result lower than the exact one by about 0.5 a packet.
- On the other hand, the simulated packets turn out to be truncated at the value of 1kbyte, which is the default size of a UDP packet. Thus transmission times are a little shorter than we intended them to be. To correct that, one should change the default maximum packet size, for example to 100000. This is done by adding the line

  ```
  $src set packetSize_ 100000
  ```

  after the command `set src [new Agent/UDP]`.
- The simulation time is not sufficiently long. With a duration of 20000, we get a much more precise value.

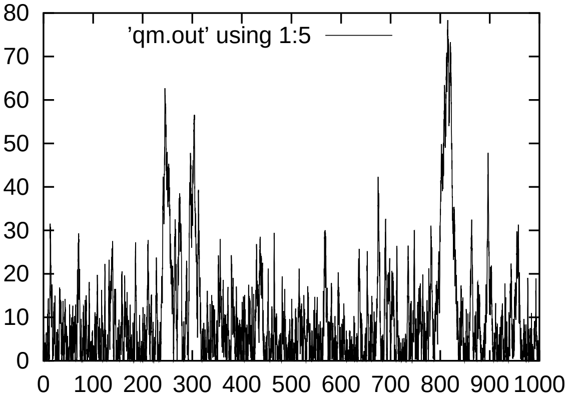

Figure 10.1: Evolution of an M/M/1 queue size

The M/D/1 queue is is one where inter-arrival times are exponentially distgributed but transmission times of packets are constant. To simulate it, simply replace the random variable pktSize by its average. Similarly, a D/M/1 queue is one where transmission duration has an exponential distribution and interarrival times are constant. To simulate that, we should replace the InterarrivalTime random variable by its average.

10.2 Finite queue

In the above simulation, we used very large buffers to avoid losses. One can use smaller buffers and observe losses. For M/M/1/K queue with K buffers, the loss probability is given by

$$P(loss) = \frac{\rho^K}{\sum_{i=0}^{K} \rho^i}.$$

The way to compute the loss probability from the simulation is simply to divide the total number of losses by the total number of arrivals, both given in the last line of the monitor-queue file.

```
set ns [new Simulator]

set tf [open out.tr w]
$ns trace-all $tf

set lambda   30.0
set mu       33.0
set qsize     2
set duration  2000

set n1 [$ns node]
set n2 [$ns node]
set link [$ns simplex-link $n1 $n2 100kb 0ms DropTail]
$ns queue-limit $n1 $n2 $qsize

# generate random interarrival times and packet sizes
set InterArrivalTime [new RandomVariable/Exponential]
$InterArrivalTime set avg_ [expr 1/$lambda]
set pktSize [new RandomVariable/Exponential]
$pktSize set avg_ [expr 100000.0/(8*$mu)]

set src [new Agent/UDP]
$src set packetSize_ 100000
$ns attach-agent $n1 $src

# queue monitoring
set qmon [$ns monitor-queue $n1 $n2 [open qm.out w] 0.1]
$link queue-sample-timeout

proc finish {} {
    global ns tf
    $ns flush-trace
    close $tf
    exit 0
}

proc sendpacket {} {
    global ns src InterArrivalTime pktSize
    set time [$ns now]
    $ns at [expr $time + [$InterArrivalTime value]] "sendpacket"
    set bytes [expr round ([$pktSize value])]
    $src send $bytes
}

set sink [new Agent/Null]
$ns attach-agent $n2 $sink
$ns connect $src $sink
$ns at 0.0001 "sendpacket"
$ns at $duration "finish"

$ns run
```

Table 10.2: tcl script mm1k.tcl for simulating an MM1 queue

Adding the command `$src set packetSize_ 100000` as mentioned in the previous Section, we get very good agreement between the simulation and the formula. For example, for $K = 2$ we get $P(loss) = 0.298$ by simulation of duration of 2000sec and $P(loss) = 0.3025$ through the above formula. For $K = 5$ we obtain 0.131 and 0.128 by simulation and through the formula, respectively. The script is given in Table 10.2.

Remark: For $K = 1$, the simulation does not work well; in that case all arriving packets are lost!

Chapter 11

Appendix I: Random variables: background

Random variables with different distributions can be created in ns. Due to its import role in traffic modeling and in network simulation we briefly recall the definitions and moments of main random variables in Appendix 11. For more background, one can conslut e.g. `http://www.xycoon.com/`.

For a random variable (RV) X, we denote $F_x(s) = P(X \leq s)$, $\overline{F}_x(s) = P(X > s)$ bnd y $f_x(s)$ we denote its density. (We often omit the subscript x.)

1. **Pareto distribution.** A Pareto RV is defined through

$$\overline{F}(s) = (k/s)^{\beta},$$

where k is the minimum size and $\beta > 0$ is the so called "shape parameter". It is defined on the range $X \geq k$. The density is given by

$$f(s) = \frac{\beta k^{\beta}}{s^{\beta+1}}.$$

The expectation and other moments are

$$\begin{aligned} E[X] &= \frac{\beta k}{\beta - 1}, \qquad 1 < \beta \\ E[X^n] &= \frac{\beta k^n}{\beta - n}, \qquad n < \beta \end{aligned}$$

The nth moment is infinite if $n \geq \beta$.

The size of files transferred over the Internet is often characterized with a Pareto distribution with $1 < \beta \leq 2$, see [36, 10]. A typical value is $\beta = 1.2$ [6]. A typical value for the expected size of a file in Internet transfers is 10Kbits. In the context of WEB transfers, typical values are $\beta = 1.1$ and $k = 81.5$Kbytes (see [13, p.34-35].)

2. **The exponential Random variable.** An exponentially distributed RV with parameter α is defined through

$$\overline{F}(s) = \exp(-\alpha s), \quad f(s) = \alpha \exp(-\alpha s).$$

All its moments exist and are given by

$$E[X^n] = \frac{n!}{\alpha^n}.$$

In a WEB transfer, Pareto distributed transfers are typically separated with Exponentially distributed silence times ("thinking times") with average duriation of $\alpha^{-1} = 5sec$ [23].

3. **Normal distribution.** It is characterized by two parameters (μ, σ^2). Its probability density is given by

$$f(s) = \frac{1}{2\pi} \exp\left[-\frac{1}{2}\left(\frac{s-\mu}{\sigma}\right)^2\right]$$

and its first moments by

$$E[X] = \mu, \quad E[X^2] = \mu^2 + \sigma^2.$$

This distribution is mostly used to discribe thermal noise that should be taken into account when computing the signal to noise ratio in radio links.

4. **Lognormal distribution.** It is characterized by two parameters (μ, σ^2). Its density function is given by

$$f(s) = \frac{\exp\left[-\frac{1}{2}\left(\frac{ln(s)-\mu}{\sigma}\right)^2\right]}{\sqrt{2\pi\sigma^2 s^2}}.$$

and its moments are given by

$$E[X^n] = \exp\left[j\mu + \frac{1}{2}(j\sigma)^2\right].$$

X is lognormally distributed with parameters (μ, σ^2) if and only if $\ln(X)$ is normally distributed with the same parameters. It can thus be written as $X = \exp(Y)$ where $Y \sim N(\mu, \sigma^2)$. In CDMA wireless communications, the received power from power controlled sources with fading channels have lognormal distribution where σ is typically between 0.3 and 3dB [39].

5. **Gamma distribution** A Gamma distributed RV with parameters (α, r) has a probability density of

$$f(s) = \frac{\alpha^r}{\Gamma[r]} s^{r-1} e^{-\alpha r}$$

where Γ is the Γ-function which satisfies $\Gamma(r) = (r-1)!$ for r integers. The moments are

$$E[X] = \frac{r}{\alpha}, \qquad E[X^n] = \alpha^{-n} \prod_{i=0}^{n-1} (r+i), n > 1.$$

The distribution is defined on the range $0 \leq s \leq \infty$, and its parameters are defined for $\alpha > 0$ and $r > 0$. In the special case where r is an integer, this distribution is called the Erlang distribution.

Chapter 12

Appendix II: Confidence intervals

In this Appendix we briefly recall the notion of confidence intervals that addresses the question of how to estimate the correctness of a simulated result.

A standard way to obtain a better precision of performance measures obtained from simulations is to take the average of several "independent" runs (independence can be obtained by using different seeds and generators). Indeed, by the strong law of large number, the average $\overline{X}$ of n independent and identically distributed values X_i, $i = 1, ..., n$ approaches the expectation $E[X]$ which we may wish to estimate.

Our goal is to check how accurate $\overline{X}$ is as an estimator of $E[X]$. In particular, we wish to establish some constant d such that the probability that $\overline{X} \in \Big[E[X] - d, E[X] + d\Big]$ be at least 1-α, where α is some small error probability (say 5%).

The variance of $\overline{X}$ is given by

$$Var(\overline{X}) = \frac{\sigma^2}{n}$$

Let σ^2 be the variance of X_i. If we knew σ, we could estimate the accuracy of $\overline{X}$ as a prediction of $E[X]$ by using the central limit theorem, which implies that

$$\sqrt{n}\frac{\overline{X} - E[X]}{\sigma} \sim N(0,1).$$

If $\Psi(x)$ is the probability that a standard Gaussian RV is not greater than x, then this suggests that

$$P\Big(\overline{X} \in \Big[E[X] - d, E[X] + d\Big]\Big) = \Psi\left(\frac{d\sqrt{n}}{\sigma}\right) - \Psi\left(-\frac{d\sqrt{n}}{\sigma}\right). \tag{12.1}$$

For example, if $\alpha = 5\%$ then the constant d that gurantees that $P\Big(\overline{X} \in \Big[E[X]-d, E[X]+d\Big]\Big) \geq 1 - \alpha = 0.95$ is given by $d = 1.96\sigma/\sqrt{n}$.

In practice, σ is typically unknown and has to be estimated together with $E[X]$. One could use $\sum_{i=1}^{n}(X_i - \overline{X}_i)^2/n$ as an estimator for σ^2, but this would give a biased estimator, i.e. an estimator whose expected value differs from σ^2. Instead, the estimator

$$S^2 = \frac{\sum_{i=1}^{n}(X_i - \overline{X}_i)^2}{n-1}$$

turns out to be unbiased, i.e. $E[S^2] = \sigma^2$, see [34, p. 111]. It is called the *sample variance.*

One then uses (12.1) with S replacing σ as an approximation of the probability that $\overline{X}$ is within the confidence interval.

The next script in awk can be used to compute the sample average of an output file, where we average over the numbers appearing in the 3rd column:

```
BEGIN { FS = "\t"} { nl++ } { s = s + $3 } END {print "del : " s/nl}
```

If this script is written in a file called "thpR.awk" and the values of X_i's are given in the third column of a file called "a40n" then one should type

```
awk -f thpR.awk a40n
```

in order to get $\overline{X}$.

The following then computes the confidence itnerval related to $\alpha = 5\%$:

```
BEGIN { FS = "\t"} {ln++}{ d = $3 - t } { s2 = s2 + d*d } END \
{s=sqrt(s2/(ln-1)); print "sample variance: " s " \
Conf. Int. 95%: " t "+/-" 1.96*s/sqrt(ln)}
```

If "ConfInt.awk" is the name of the file containig this script, type

```
awk -v t=XXX -f ConfInt.awk a40n
```

where instead of XXX one should put the value of $\overline{X}$. This will give both the sample variance as well as the required confidence interval.

Bibliography

[1] E. Altman, "A stateless approach for improving TCP performance using Diffserv", submitted.

[2] E. Altman and T. Jiménez, “Simulation analysis of RED with short lived TCP connections”, submitted.

[3] E. Altman and T. Jiménez, “Improving TCP over multihop networks using delayed ACK”, submitted.

[4] A. Ballardie, "Core Based Trees (CBT) Multicast Routing Architecture", Internet RFC 2201 (Experimental), September 1997.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, “An architecture for differentiated services”, RFC 2475, Dec. 1998.

[6] T. Bonald and J. Roberts, "Performance modeling of elastic trafic in overload", ACM Sigmetrics, pp. 342-343, 2001.

[7] B. Braden et al., “Recommendations on Queue Management and Congestion Avoidance in the Internet”, April 1998. Available as RFC 2309 at ftp://ftp.isi.edu/in-notes/rfc2309.txt.

[8] D. D. Clark and W. Fang. Explicit Allocation of Best-E ort Packet Delivery Service. *IEEE/ACM Trans on Networking*, 6(4), 362-373, August 1998.

[9] M. S. Corson, S. Papademetriou, P. Papadopolous, V. D. Park and A. Qayyum, “An Internet Manet Encapsulation Protocol (IMEP) Specification”, Internet draft, draft-ietf-manet-imep-spec01.txt, August 1998.

[10] M. Crovella and A. Bestravos, "Self-similarity in World Wide Web traffic: Evidence and possible cause", *ACM Sigmetrics*, 1996.

[11] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, Ching-Gung Liu, and L. Wei. An architecture for wise-area multicast routing. Technical Report USC-SC-94-565, Computer Science Department, University of Southern California, Los Angeles, CA 90089., 1994.

[12] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint communication:a survey of protocols, functions and mechanisms. IEEE Journal on Selected Areas in Communications, 15(3), Apr 1997.

[13] ETSI, *Universal Mobile Telecommunication System (UMTS); Selection procedures for the choice of radio transmission technologies of the UMTS*, UMTS 30.03 Version 3.2.0, 1998-04. Available (public domain) at
http://www.etsi.org/getastandard/home.htm

[14] K. Fall and K. Varadhan, *The ns Manual*, available at `http://www.isi.edu/nsnam/ns/`.

[15] TclCL web page. http://otcl-tclcl.sourceforge.net/tclcl/

[16] W. Fang, N. Seddigh and B. Nandy, A Time Sliding Window Three Colour Marker (TSWTCM), RFC 2859, http://rfc.sunsite.dk/rfc/rfc2859.html, June 2000.

[17] Floyd, S., and V. Jacobson, "Random Early Detection gateways for Congestion Avoidance" V.1 N.4, August 1993, p. 397-413. Available at http://www.icir.org/floyd/papers/red/red.html

[18] S. Floyd, R. Gummadi and S. Shenker, "Adaptive Red: an algorithm for increasing the robustness of RED's active queue management", submitted.

[19] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, M. Gerla, "The impact of multihop wireless channel on TCP throughput and loss", *Proc. of IEEE INFOCOM*, 2003. Available at www.cs.ucla.edu/wing/publication/publication.html

[20] J. Heinanen, F. Baker, W. Weiss, J. Wroclawski, "Assured forwarding PHB group", RFC 2597, June 1999.

[21] J. Heinanen and R. Guerin, "A single rate three color marker", RFC 2697, http://rfc.sunsite.dk/rfc/rfc2697.html Sept. 1999.

[22] J. Heinanen and R. Guerin, "A two rate three color marker", RFC 2698, http://rfc.sunsite.dk/rfc/rfc2698.html Sept. 1999.

[23] D. P. Heyman, T. V. Lakshman and A. L. Neidhardt, "A new method for analysing feedback-based protocols with applications to engineering web traffic over the Internet", *Proc. ACM Sigmetrics*, Seattle, 1997.

[24] Christian Huitema. Routing in the Internet, Prentice Hall, April 1995. 2nd Edition: 1999.

[25] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM 88*, pages 273–288, 1988.

[26] D. B. Johnson, D. A. Maltz, Y.-C. Hu, J. G. Jetcheva, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)", IETF MANET Working Group INTERNET-DRAFT, available at http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-07.txt 21 February, 2002.

[27] L. Kleinrock. *Queueing systems.* John Wiley, New York, 1976.

[28] W. Noureddine and F. Tobagi, "Improving the Performance of Interactive TCP Applications using Service Differentiation", *Proceedings of IEEE Infocom*, New-York, USA, June 2002.

[29] V. D. Park and M. S. Corson, "Temporally-Ordered Routing Algorithm (TORA) version 1: functional specifications", Internet draft, draft-ietf-manet-tora-spec-01.txt, August 1998.

[30] V. D. Park and M. S. Corson, "A performance conparison of the temporally-ordered routing algorithm and ideal link-state routing", *Proceedings of IEEE Symposium on Computers and Communication*, June 1998.

[31] C. E. Perkins and S. R. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing", IETF MANET Working Group INTERNET-DRAFT, available on http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-11.txt, 19 June, 2002.

[32] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers", *Proceedings of SIGCOMM*, pp. 234-244, 1994.

[33] P. Pieda, J. Ethridge, M. Baines and F. Shallwani, "A network simulator differentiated services implementation", Open IP, Nortel Networks, July 26, 2000.

[34] S. M. Ross, *Simulation*, 3rd Edition, Academic Press, San-Diego, London, Boston, New York, Sydney, Tokyo, Toronto, 2002.

[35] S. Sahu, P. Nain, C. Diot, V. Firoiu and D. F. Towsley", "On achievable service differentiation with token bucket marking for TCP", Proc. ACM SIGMETRICS'00, Santa Clara, CA, June 2000.

[36] B. Sikdar, S. Kalyanaraman and K. S. Vastola, "An Integrated Model for the Latency and Steady-State Throughput of TCP Connections", *Performance Evaluation*, v.46, no.2-3, pp.139-154, September 2001.

[37] P. Sinha, *Routing and transport layer protocols for wireless networks*, Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, Computer Science, 2001.

[38] D. Thaler, D. Estrin, D. Meyer (Editors), "Border Gateway Multicast Protocol (BGMP): Protocol Specification", Internet Draft ¡draft-ietf-bgmp-spec-02¿, work in progress, November 2000.

[39] A. J. Viterbi, A. M. Viterbi and E. Zehavi. Performance of power-controlled wideband terrestrial digital communication. *IEEE Transactions on Communications*, 41(4):559–69, April 1993.

[40] D. Waitzman, C. Partridge, and S.E. Deering. Distance Vector Multicast Routing Protocol, RFC 1075 edition, 1988.

Index