S-38.3148
Ns2 simulation exercise

Fall 2006

Table of contents

1. Introduction

In this exercise you will have to simulate the transfer times of files, transported on top of TCP, in two test networks and compare the analytical results with the simulated results. The purpose is to make you familiar with TCP's congestion control mechanisms and performance as well as the ns2 simulator. Notice that you are supposed to carry out this exercise independently, group work is not allowed.

2. Theoretical background

2.1. Overview of TCP's congestion control

TCP implements a window based flow control mechanism, as explained in [APS99]. Roughly speaking, a window based protocol means that the so called current window size defines a strict upper bound on the amount of unacknowledged data that can be in transit between a given sender-receiver pair. Originally TCP's flow control was governed simply by the maximum allowed window size advertised by the receiver and the policy that allowed the sender to send new packets only after receiving the acknowledgement for the previous packet.

After the occurrence of the so called congestion collapse in the Internet in the late 80's it was realised, however, that special congestion control algorithms would be required to prevent the TCP senders from overrunning the resources of the network. In 1988, Tahoe TCP was released including three congestion control algorithms: slow start, congestion avoidance and fast retransmit. In 1990 Reno TCP, providing one more algorithm called fast recovery, was released.

Besides the receiver's advertised window, *awnd*, TCP's congestion control introduced two new variables for the connection: the congestion window, *cwnd*, and the slowstart threshold, *ssthresh*. The window size of the sender, *w*, was defined to be

$w = \min(cwnd, awnd)$,

instead of being equal to *awnd*. The congestion window can be thought of as being a counterpart to advertised window. Whereas *awnd* is used to prevent the sender from overrunning the resources of the receiver, the purpose of *cwnd* is to prevent the sender from sending more data than the network can accommodate in the current load conditions.

The idea is to modify *cwnd* adaptively to reflect the current load of the network. In practice, this is done through detection of lost packets. A packet loss can basically be detected either via a time-out mechanism or via duplicate ACKs.

Timeouts:

Associated with each packet is a timer. If it expires, timeout occurs, and the packet is retransmitted. The value of the timer, denoted by RTO, should ideally be of the order of an RTT. However, as the value of RTT is not known in practice, it is measured by the TCP connection by using, e.g, the so called Karn algorithm.

Duplicate ACKs:

If a packet has been lost, the receiver keeps sending acknowledgements but does not modify the sequence number field in the ACK packets. When the sender observes several ACKs acknowledging the same packet, it concludes that a packet has been lost.

2.1.1. Slow start and congestion avoidance

In slow start, when a connection is established, the value of *cwnd* is first set to 1 and after each received ACK the value is updated to

$cwnd = cwnd + 1$

implying doubling of *cwnd* for each RTT.

The exponential growth of *cwnd* continues until a packet loss is observed, causing the value of *ssthresh* to be updated to

$ssthresh = cwnd/2.$

After the packet loss, the connection starts from slow start again with $cwnd = 1$, and the window is increased exponentially until it equals *ssthresh*, the estimate for the available bandwidth in the network. At this point, the connection goes to congestion avoidance phase where the value of *cwnd* is increased less aggressively with the pattern

cwnd = cwnd + 1/cwnd,

implying linear instead of exponential growth. This linear increase will continue until a packet loss is detected.

2.1.2. Fast Retransmit

Duplicate ACKs that were mentioned to be one way of detecting lost packets, can also be caused by reordered packets. When receiving one duplicate ACK the sender can not yet know whether the packet has been lost or just gotten out of order but after receiving several duplicate ACKs it is reasonable to assume that a packet loss has occurred. The purpose of fast retransmit mechanism is to speed up the retransmission process by allowing the sender to retransmit a packet as soon as it has enough evidence that a packet has been lost. This means that instead of waiting for the retransmit timer to expire, the sender can retransmit a packet immediately after receiving three duplicate ACKs.

2.1.3. Fast Recovery

In Tahoe TCP the connection always goes to slow start after a packet loss. However, if the window size is large and packet losses are rare, it would be better for the connection to continue from the congestion avoidance phase, since it will take a while to increase the window size from 1 to *ssthresh*. The purpose of the fast recovery algorithm in Reno TCP is to achieve this behaviour.

In a connection with fast retransmit, the source can use the flow of duplicate ACKs to clock the transmission of packets. When a possibly lost packet is retransmitted, the values of *ssthresh* and *cwnd* will be set to

*ssthresh = cwnd/*2

and

cwnd = ssthresh

meaning that the connection will continue from the congestion avoidance phase and increases its window size linearly.

## 2.2. Modelling TCP's performance

The traditional methods for examining the performance of TCP have been simulation, implementations and measurements. However, efforts have also been made to analytically characterize the throughput of TCP as a function of parameters such as packet drop rate and round trip time. Also TCP flow transfer times have been analyzed. For the analysis, both packet level [FM80] and flow level models have been developed. Some efforts for integrating the packet and flow level models have also been made [LBMK03].

2.2.1. Simple packet-level model for the steady-state TCP throughput

The simple model presented in [F99] provides an upper bound on TCP's average sending rate that applies to any conformant tcp. A conformant TCP is defined in [F99] as a TCP connection where the TCP sender adheres to the two essential components of TCP's congestion control: First, whenever a packet drop occurs in a window of data, the TCP sender interpretes this as a signal of congestion and responds by cutting the congestion window at least in half. Second, in the congestion avoidance phase where there is currently no congestion, the TCP sender increases the congestion window by at most one packet per window of data. Thus, this behaviour corresponds to TCP Reno in the presence of only triple duplicate loss indications.

In [F99] a steady-state model is assumed. It is also assumed for the purpose of the analysis that a packet is dropped from a TCP connection if and only if the congestion window has increased to W packets. Because of the steady-state model the average packet drop rate, p, is assumed to be nonbursty.

The TCP sender follows the two components of TCP's congestion control as mentioned above. When a packet is dropped, the congestion window is halved. After the drop, the TCP sender increases linearly its congestion window until the congestion window has reached its old value W and another packet drop occurs. The development of TCP's congestion window under these assumptions is depicted in Figure 1.
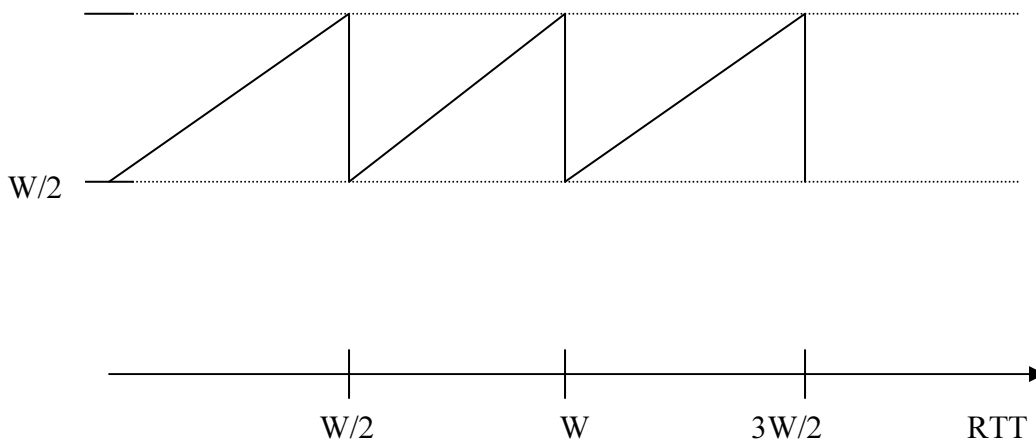
W

W/2   W   3W/2   RTT
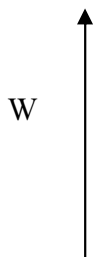
Figure 1 Development of TCP's congestion window

If a TCP sender with packets of B bytes and a reasonably constant roundtrip time of R seconds is considered, it is clear that with the assumptions of the model the TCP sender transmits at least

(1) $\qquad \dfrac{W}{2} + \left(\dfrac{W}{2} + 1 \right) + \ldots + W \approx \dfrac{3}{8} W^2$

packets per a dropped packet. Thus the packet drop rate p is bounded by

(2) $\qquad p \le \dfrac{8}{3W^2}.$

From (2), the upper bound for W is:

(3) $\qquad W \le \sqrt{\dfrac{8}{3p}}.$

In the steady-state model the average congestion window is $0.75W$ over a single cycle. Thus the maximum sending rate for the TCP connection over this cycle in bytes is

(4) $\qquad T \le \dfrac{0.75 * W * B}{RTT}.$

Substituting the upper limit for W, we get

(5) $\qquad T \le \dfrac{1.5\sqrt{2/3} * B}{RTT * \sqrt{p}} \approx \dfrac{1.22 * B}{RTT * \sqrt{p}},$

where B is the packet size, RTT is the round trip delay and p is the steady-state packet drop rate.

This model should give reasonably reliable results with small packet losses ($< 2\%$), but with higher loss rates it can considerably overestimate TCP's throughput. Also, the equations derived do not take into account the effect of retransmit timers. Basically, TCP can detect packet loss either by receiving "triple-duplicate" acknowledgements (four ACKs having the same sequence number), or

via time-outs. In this model it is assumed that packet loss is observed solely by triple duplicate ACKs.

2.2.2. Complex packet-level model for the steady-state TCP throughput

A good model for predicting TCP throughput should capture both the time-out and "triple-duplicate" ACK loss indications and provide fairly accurate estimates also with higher packet losses. A more complex model presented in [PFTK98] takes time-outs into account and is applicable for broader range of loss rates. In [PFTK98] the following approximation of TCP's throughput, $T(p)$:

$$(6) \qquad T(p) \approx \min\left(\frac{W_{max}}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0 \min\left(1,3\sqrt{\frac{3bp}{8}}\right)p\left(1+32p^2\right)} \right) \; ,$$

where W_{max} is the receiver's advertised window and thus the upper bound for the congestion window, RTT is the round trip time, p the loss indication rate, T_0 TCP's average retransmission time-out value and b the number of packets that are acknowledged by each ACK. In the denominator, the first term is due to triple-duplicate acks, and the second term models the timeouts. With larger loss rates the second term dominates.

2.2.3. A flow-level processor sharing model for the flow transfer times

In [FM80] the transfer times of jobs in a processor-sharing system with M job classes is analyzed. The processor sharing system divides the processor resources among the job classes so that a class k job receives a fraction

$$(7) \qquad g_k / (\sum_{j=1}^{M} g_j N_j)$$

of the resources, where N_j denotes the number of jobs in class j and g_j denotes the weighting factor of class j. Thus, the resource is not shared evenly between jobs of different classes, and hence this processor sharing model is sometimes referred to as the Discriminatory Processor Sharing model (DPS). Assuming that class-k jobs arrive according to a Poisson process with intensity λ_k, the unconditional average response times satisfy the following equations:

$$(8) \qquad W_k\left[1 - \sum_{j=1}^{M} \frac{\lambda_j g_j}{\mu_j g_j + \mu_k g_k}\right] - \sum_{j=1}^{M} \frac{\lambda_j g_j W_j}{\mu_j g_j + \mu_k g_k} = \frac{1}{\mu_k}, \quad k = 1,2,...,M,$$

where $1/\mu_k$ is the average required service time for the job, assuming that the service times are exponentially distributed. The equations in (8) can be alternatively expressed as

$$(9) \quad W_k \left[1 - \sum_{j=1}^{M} \frac{\lambda_j}{\mu_j + \mu_k \dfrac{g_k}{g_j}} \right] - \sum_{j=1}^{M} \frac{\lambda_j W_j}{\mu_j + \mu_k \dfrac{g_k}{g_j}} = \frac{1}{\mu_k}, \quad k = 1,2,\ldots,M .$$

Now, it is not necessary to know the exact weighing factors g_j but only their ratios.

Application to TCP: Assuming that the throughput of a TCP flow in class k can be approximated by $c / \left(RTT_k * \sqrt{p} \right)$, where c is a constant, the ratio g_i/g_j becomes

$$(10) \quad \frac{g_i}{g_j} = \frac{RTT_j}{RTT_i} .$$

The equations in (9) can be used to calculate the average transfer times for TCP flows, if the weights are interpreted according to (10). Observe that for a given TCP sender, the RTTs are random due to queuing delays at the bottleneck. The simplest approximation for class-k RTT is to assume it consists of the constant parts, i.e., RTT_k equals the sum of propagation delays + packet transmission delays at access link and bottleneck link (remember that RTT means by definition the total delay in both directions). This is more accurate the less the random queuing delays impact the RTT. A more detailed approximation would take into account also the mean queuing delay at the bottleneck. Furthermore, the parameter λ_k is the intensity at which flows arrive and the parameter $1/\mu_k$ equals B/C, where B is the mean file size (file sizes are assumed to be exponentially distributed) and C is the bottleneck bandwidth. Finally, the throughput of a class-k flow, denoted by T_k, is by definition the mean file size divided by the average class-k transfer time, i.e.,

$$(11) \quad T_k = \frac{B}{W_k} .$$

3. Exercise

Note that, a primer on OTcl and ns2 is given in Appendix A.

In this exercise, the idea is to perform flow level simulations of TCP performance, i.e., the system to be simulated consists of randomly arriving files that will be transported via TCP across a bottleneck link. Two tasks are given, where in the first one, traffic is generated directly from TCP sources to the bottleneck and packet losses are introduced artificially through a so called loss module. In the second task, TCP traffic is concentrated through a tree-type access network and the packet losses occur naturally due to finite buffers.

3.1. Skeleton code

A skeleton code is provided which contains functions for generating randomly arriving flows with random files sizes. Additionally, the routine to be performed upon completion of a flow is given, which implements the basic statistics collection.

The skeleton code can be downloaded from the course web page.

3.2. Using ns2

Ns2 is installed in the Linux machines located in the class rooms Maari A and Maari C. For a listing of the machines in these rooms, see:

- http://www.hut.fi/atk/luokat/Maari-A.html
- http://www.hut.fi/atk/luokat/Maari-C.html

In order to be able to use ns2, you first have to source the file /p/edu/s-38.180/usens2.csh, i.e., enter in your shell the command:

```
source /p/edu/s-38.180/usens2.csh
```

The file usens2.csh contains the required settings for environmental variables. Give this command each time you start an ns2 session in a shell.

Then, to run your simulation script "myscript.tcl", just write:

```
ns myscript.tcl
```

3.3. General simulation parameters

You should use the following general parameters in the configuration for both tasks:

- Queue management – use DropTail both in the bottleneck link and in the access links.
- TCP type – Reno TCP
- TCP's maximum window size – 1000 (so that it's big enough!)
    - o   Name of variable in TCL: `window_`
- TCP's packet size – 1460 (in bytes)
    - o   Name of variable in TCL: `packetSize_`
- Starting times of the TCP sources – the starting times of the sources should follow the flow interarrival time distribution (flow arrivals are from a Poisson process)
- File sizes obey an exponential distribution with mean 500 packets

3.4. Task1

In the first task you will compare the analytical average flow transfer times provided by the equations in (9) to the corresponding simulated transfer times. In order to fulfil the assumption of (9) of the equal loss-rates for the different *RTT* classes, you should use a separate loss-module in the bottleneck link. The artificial loss-module in ns2 generates losses randomly in the bottleneck link queue so that each packet entering the queue is dropped with an equal probability p (an example of introducing a loss module can be found in section A.5, in the appendix)

Use four different *RTT* classes and set the inter-arrival times of the TCP-flows in each class so that the theoretical offered load in the bottleneck link at the flow level equals 0.8. This ensures that the network will not be in a pathological congestion situation. The task is then to compare the results of the theoretical model to simulated results with different values of the packet loss probability p. In more detail measure and compare the following:

- Using data from the simulation, give estimates for the mean file transfer times and throughputs of each class based on applying Eq. (11).
- Include also confidence intervals in your results.
- Plot your results in a graph as a function of p.
    1. Loss rate Vs Mean file transfer times
    2. Loss rate Vs Throughput
- Assume that RTTs for each class consists of only the propagation delays when applying the DPS model to TCP. Compare the simulated mean delays and throughputs to the theoretical mean delays and throughputs given by Eq. (9). What can you observe? Can you give reasons why the DPS model and measured TCP performance differ somewhat?
- Compare also the ratios of the performance, i.e., compute W_k/W_1, $k = 1,\ldots,4$, for simulations and for the theoretical model. What can you observe as p is varied? When is the DPS model result most accurate when compared with simulations?
- Repeat the above simulations also for loads 0.7 and 0.9. Do the results change?

Extra questions: Can you explain the behaviour in the mean delay ratios between the simulations and model? Hint: think about the impact of queuing. If you want to measure the mean queuing delay, consider the code given already in redtcpmain.tcl and redtcpsub.tcl (examples from 2nd ns2 lecture) and think how you could configure RED such that RED mechanism is turned "off".

3.3.1. Simulation parameters for Task1

In task1, you should use the following topology parameters:
- Access-link bandwidths – 100 Mbps
- Bottleneck-link bandwidth – 10 Mbps
- Access-link delays – The access link delays of the four delay classes should range from 10 ms to 100 ms with 30 ms spacing.
- Bottleneck-link delay – 10 ms
- Queue limits: Use 1000 packets for the bottleneck queue and 1000 packets for the access queues. Thus packet losses are caused mainly by the loss module, not by small queues.
- Flow-level load: 0.8 (you can vary this also if you want to do extra!)
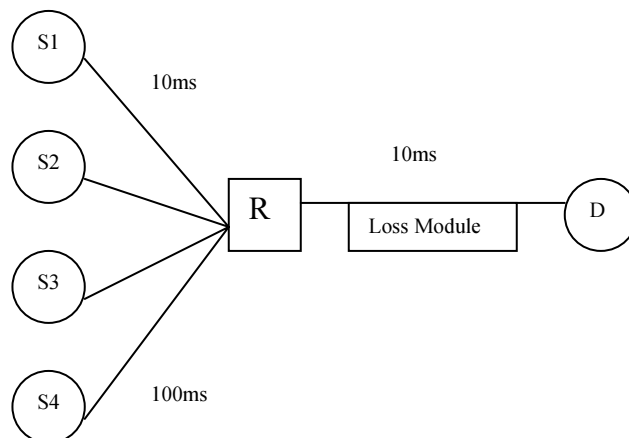- Packet loss in the loss module: Use six different values for p: [0.1, 0.5, 1, 2, 4, 5] %

Figure 2 The topology for task 1

3.4 Task2

In this task you will perform the same simulations as in task1 but in a larger topology. Now, you will not add any artificial losses but assign a finite limit for all the buffers, allowing the losses to occur naturally. You should consider how the larger topology and "real" queuing behaviour changes the flow transfer times of different *RTT* classes compared with the more controlled setup in task1. You should modify the load level in the bottleneck link and observe what happens to the flow transfer times and flow throughputs of different *RTT* classes and to their relations. In more detail, you are expected to:

- Using data from the simulation, give estimates for the mean file transfer times and throughputs of each class based on applying Eq. (11).
- Include also confidence intervals in your results.
- Plot your results in a graph as a function of the load.
- Comment the behaviour of the system as a function of the load. Compare also the ratios of the performance, i.e., compute W_k/W_1, $k = 1,...,4$, for different values of the offered load. What can you conclude from these, i.e., how does the load affect the results?
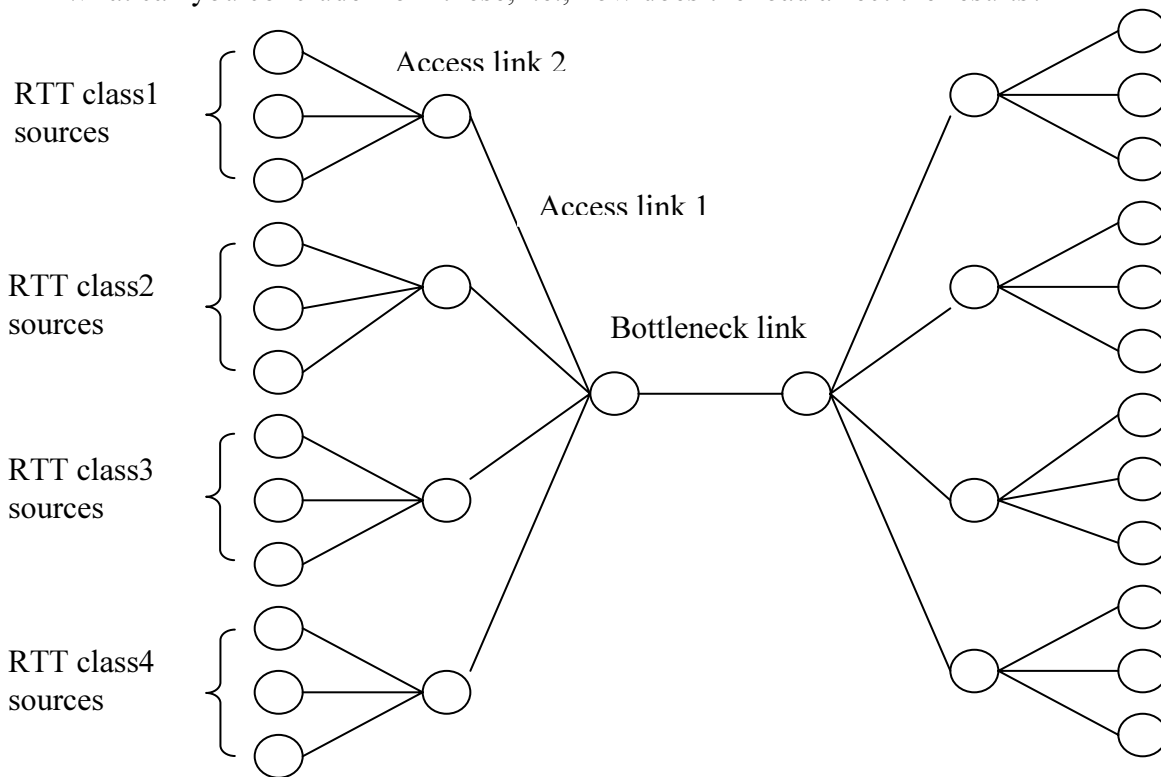

Figure 3 Topology for task 2

3.4.1. Simulation parameters for Task2

In task2, use the following topology parameters:

- Access-link-1 bandwidths – 10 Mbps
- Access-link-2 bandwidth – 100 Mbps
- Bottleneck-link bandwidth – 10 Mbps
- Access-link-1 delays -10 ms
- Access-link-2 delays – The access link delays of the four delay classes should range from 5 ms to 50 ms with 15 ms spacing.

- Bottleneck-link delay – 30 ms
- Queue limits: Use 100 packets for the access and bottleneck queues.
- Use five different load levels: [0.5 0.6 0.7, 0.8, 0.9]

3.5. Handout requirements

The deadline of the exercise is on:

- Monday, November 20, at 12 o'clock.

Return the exercise to the following e-mail address: jegadish@netlab.hut.fi

The following elements are required in the handout:

- Listing of simulation scripts
- Listing of post processing scripts
- Graphs of the results
- Analysis and comments of the results

3.6. Getting help

Two consultation sessions will be arranged for the students:

- Session I: Wed, Nov 8, at 10-12, place: Linux class room Maari-C (at Maarintalo)
- Session II: Wed, Nov 15, at 10-12, place: Linux class room Maari-C (at Maarintalo)

In addition, if you have some questions concerning the exercise, send e-mail to: jegadish@netlab.hut.fi

References

[F99] S.Floyd, "Promoting the Use of End-toEnd Congestion Control in the Internet", IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 7, NO.4, AUGUST 1999.

[PFTK98]   J.Padhye, V.Firoiu, D.Towsley, J.Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation", ACM SIGCOMM 1998.

[APS99]   M.Allman, V.Paxson, W.R.Stevens, "TCP Congestion Control", STD1, RFC 2581, April 1999.

[FM80] G.Fayolle, I.Mitrani, "Sharing a Processor Among Many Job Classes", Journal of the Association for Computing Machinery, Vol.27, No.3, July 1980.

[LBMK03] P.Lassila, H. van den Berg, M. Mandjes, R. Kooij, "An integrated packet/flow model for TCP performance analysis"

A. Appendix

A.1. Installing and using ns2

Ns2 can be built and run both under Unix and Windows. Instructions on how to install ns2 on Windows can be found at: http://www.isi.edu/nsnam/ns/ns-win32-build.html. However, the installation may be smoother under Unix.

Modifying ns2

In ns2 it is also possible to modify the existing C++ code or create completely new C++ code. After the changes have been made, the code has to be recompiled by running make in the ns-2.xx directory. However, in this exercise you do not have to (and you are not allowed to!) make any changes at the C++ level, only at the tcl-level.

A.2. General description

Ns2 is an event driven, object oriented network simulator enabling the simulation of a variety of local and wide area networks. It implements different network protocols (TCP, UDP), traffic sources (FTP, web, CBR, Exponential on/off), queue management mechanisms (RED, DropTail), routing protocols (Dijkstra) etc. Ns2 is written in C++ and Otcl to separate the control and data path implementations. The simulator supports a class hierarchy in C++ (the compiled hierarchy) and a corresponding hierarchy within the Otcl interpreter (interpreted hierarchy).

The reason why ns2 uses two languages is that different tasks have different requirements: For example simulation of protocols requires efficient manipulation of bytes and packet headers making the run-time speed very important. On the other hand, in network studies where the aim is to vary some parameters and to quickly examine a number of scenarios the time to change the model and run it again is more important.

In ns2, C++ is used for detailed protocol implementation and in general for such cases where every packet of a flow has to be processed. For instance, if you want to implement a new queuing discipline, then C++ is the language of choice. Otcl, on the other hand, is suitable for configuration and setup. Otcl runs quite slowly, but it can be changed very quickly making the construction of simulations easier. In ns2, the compiled C++ objects can be made available to the Otcl interpreter. In this way, the ready-made C++ objects can be controlled from the OTcl level.

There are quite many understandable tutorials available for new ns-users. By going through, for example, the following tutorials should give you a rather good view of how to create simple simulation scenarios with ns2:

> http://nile.wpi.edu/NS/
> http://www.isi.edu/nsnam/ns/tutorial/index.html

The next chapters will summarise and explain the key features of tcl and ns2, but in case you need more detailed information, the ns-manual provides the required information.

> http://www.isi.edu/nsnam/ns/ns-documentation.html

Other useful ns2 related links, such as archives of ns2 mailing lists, can be found from ns2 homepage:

http://www.isi.edu/nsnam/ns/index.html

A.3. Otcl basics

This chapter introduces the syntax and the basic commands of the Otcl language used by ns2. It is important that you understand how Otcl works before moving to the chapters handling the creation of the actual simulation scenario.

A.3.1. Assigning values to variables

In tcl, values can be stored to variables and these values can be further used in commands:

set a 5
set b [expr $a/5]

In the first line, the variable 'a' is assigned the value "5". In the second line, the result of the command [expr $a/5], which equals 1, is then used as an argument to another command, which in turn assigns a value to the variable b. The "$" sign is used to obtain a value contained in a variable and square brackets are an indication of a command substitution.

A.3.2. Procedures

You can define new procedures with the proc command. The first argument to proc is the name of the procedure and the second argument contains the list of the argument names to that procedure. For instance a procedure that calculates the sum of two numbers can be defined as follows:

proc sum {a b} {
**   expr $a + $b**
}

The next procedure calculates the factorial of a number:

proc factorial a {
**   if {$a <= 1} {**
**   return 1**
}
#here the procedure is called again
expr $x * [factorial [expr $x-1]]}

It is also possible to give an empty string as an argument list. However, in this case the variables that are used by the procedure have to be defined as global. For instance:

proc sum {} {
**   global a b**
**   expr $a + $b**

}

A.3.3.   Files and lists

In tcl, a file can be opened for reading with the command:

set testfile [open test.dat r]

The first line of the file can be stored to a list with a command:

gets $testfile list

Now it is possible to obtain the elements of the list with commands (numbering of elements starts from 0) :

set first [lindex $list 0]
set second [lindex $list 1]

Similarly, a file can be written with a puts command:

set testfile [open test.dat w]
puts $testfile "testi"

A.3.4.   Calling subprocesses

The command exec creates a subprocess and waits for it to complete. The use of exec is similar to giving a command line to a shell program. For instance, to remove a file:

exec rm $testfile

The exec command is particularly useful when one wants to call a tcl-script from within another tcl-script. For instance, in order to run the tcl-script example.tcl multiple times with the value of the parameter "test" ranging from 1 to 10, one can type the following lines to another tcl-script:

for {set ind 1} {$ind <= 10} {incr ind} {

    set test $ind
    exec ns example.tcl test
}

A.4. Creating the topology

To be able to run a simulation scenario, a network topology must first be created. In ns2, the topology consists of a collection of nodes and links.

Before the topology can be set up, a new simulator object must be created at the beginning of the script with the command:

set ns [new Simulator]

The simulator object has member functions that enable creating the nodes and the links, connecting agents etc. All these basic functions can be found from the class Simulator. When using functions belonging to this class, the command begins with "$ns", since ns was defined to be a handle to the Simulator object.

A.4.1.   Nodes

New node objects can be created with the command

set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

The member function of the Simulator class, called "node" creates four nodes and assigns them to the handles n0, n1, n2 and n3. These handles can later be used when referring to the nodes. If the node is not a router but an end system, traffic agents (TCP, UDP etc.) and traffic sources (FTP, CBR etc.) must be set up, i.e, sources need to be attached to the agents and the agents to the nodes, respectively.

A.4.2.   Agents, applications and traffic sources

The most common agents used in ns2 are UDP and TCP agents. In case of a TCP agent, several types are available. The most common agent types are:

- Agent/TCP – a Tahoe TCP sender
- Agent/TCP/Reno – a Reno TCP sender
- Agent/TCP/Sack1 – TCP with selective acknowledgement

The most common applications and traffic sources provided by ns2 are:

- Application/FTP – produces bulk data that TCP will send
- Application/Traffic/CBR – generates packets with a constant bit rate
- Application/Traffic/Exponential – during off-periods, no traffic is sent. During on-periods, packets are generated with a constant rate. The length of both on and off-periods is exponentially distributed.
- Application/Traffic/Trace – Traffic is generated from a trace file, where the sizes and interarrival times of the packets are defined.

In addition to these ready-made applications, it is possible to generate traffic by using the methods provided by the class Agent. For example, if one wants to send data over UDP, the method

send(int nbytes)

can be used at the tcl-level provided that the udp-agent is first configured and attached to some node.

Below is a complete example of how to create a CBR traffic source using UDP as transport protocol and attach it to node n0:

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packet_size_ 1000
$cbr0 set rate_ 1000000

An FTP application using TCP as a transport protocol can be created and attached to node n1 in much the same way:

set tcp1 [new Agent/TCP]
$ns attach-agent $n1 $tcp1
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$tcp1 set packet_size_ 1000

The UDP and TCP classes are both child-classes of the class Agent. With the expressions [new Agent/TCP] and [new Agent/UDP] the properties of these classes can be combined to the new objects udp0 and tcp1. These objects are then attached to nodes n0 and n1. Next, the application is defined and attached to the transport protocol. Finally, the configuration parameters of the traffic source are set. In case of CBR, the traffic can be defined by parameters rate_ (or equivalently interval_, determining the interarrival time of the packets), packetSize_ and random_ . With the random_ parameter it is possible to add some randomness in the interarrival times of the packets. The default value is 0, meaning that no randomness is added.

A.4.3.   Traffic Sinks

If the information flows are to be terminated without processing, the udp and tcp sources have to be connected with traffic sinks. A TCP sink is defined in the class Agent/TCPSink and an UDP sink is defined in the class Agent/Null.

A UDP sink can be attached to n2 and connected with udp0 in the following way:

set null [new Agent/Null]
$ns attach-agent $n2 $null
$ns connect $udp0 $null

A standard TCP sink that creates one acknowledgement per a received packet can be attached to n3 and connected with tcp1 with the commands:

set sink [new Agent/Sink]
$ns attach-agent $n3 $sink

$ns connect $tcp1 $sink

There is also a shorter way to define connections between a source and the destination with the command:

$ns create-connection <srctype> <src> <dsttype> <dst> <pktclass>

For example, to create a standard TCP connection between n1 and n3 with a class ID of 1:

$ns create-connection TCP $n1 TCPSink $n3 1

One can very easily create several tcp-connections by using this command inside a for-loop.

A.4.4.   Links

Links are required to complete the topology. In ns2, the output queue of a node is implemented as part of the link, so when creating links the user also has to define the queue-type.
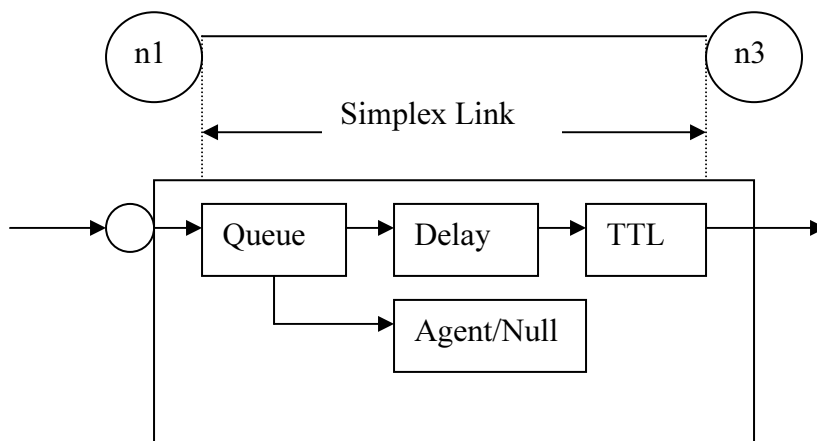

Figure 4 Link in ns2

Figure 4 shows the construction of a simplex link in ns2. If a duplex-link is created, two simplex-links will be created, one for each direction. In the link, packet is first enqueued at the queue. After this, it is either dropped, passed to the Null Agent and freed there, or dequeued and passed to the Delay object which simulates the link delay. Finally, the TTL (time to live) value is calculated and updated.

Links can be created with the following command:

$ns duplex/simplex-link endpoint1 endpoint2 bandwidth delay queue-type

For example, to create a duplex-link with DropTail queue management between n0 and n2:

$ns duplex-link $n0 $n2 15Mb 10ms DropTail

Creating a simplex-link with RED queue management between n1 and n3:

$ns simplex-link $n1 $n3 10Mb 5ms RED

The values for bandwidth can be given as a pure number or by using qualifiers k (kilo), M (mega), b (bit) and B (byte). The delay can also be expressed in the same manner, by using m (milli) and u (mikro) as qualifiers.

There are several queue management algorithms implemented in ns2, but in this exercise only DropTail and RED will be needed.

A.5. Adding loss modules

You can create packet losses artificially by adding a separate loss module to a link. The loss module can be created and configured as follows:

 # create a random variable that follows the uniform distribution
set loss_random_variable [new RandomVariable/Uniform]
$loss_random_variable set min_ 0 # the range of the random variable;
$loss_random_variable set max_ 100

set loss_module [new ErrorModel] # create the error model;
$loss_module drop-target [new Agent/Null] #a null agent where the dropped packets go to
$loss_module set rate_ 10 # error rate will then be (0.1 = 10 / (100 - 0));
$loss_module ranvar $loss_random_variable # attach the random variable to loss module;

You can attach the loss module to a certain link between two nodes, n1 and n2, with the command:

$ns lossmodel $loss_module $n1 $n2

A.6. Tracing and monitoring

In order to be able to calculate the results from the simulations, the data has to be collected somehow. Ns2 supports two primary monitoring capabilities: traces and monitors. The traces enable recording of packets whenever an event such as packet drop or arrival occurs in a queue or a link. The monitors provide a means for collecting quantities, such as number of packet drops or number of arrived packets in the queue. The monitor can be used to collect these quantities for all packets or just for a specified flow (a flow monitor)

A.6.1.   Traces

All events from the simulation can be recorded to a file with the following commands:

set trace_all [open all.dat w]
$ns trace-all $trace_all
$ns flush-trace
close $trace_all

First, the output file is opened and a handle is attached to it. Then the events are recorded to the file specified by the handle. Finally, at the end of the simulation the trace buffer has to be flushed and the file has to be closed. This is usually done with a separate finish procedure.

If links are created after these commands, additional objects for tracing (EnqT, DeqT, DrpT and RecvT) will be inserted into them.
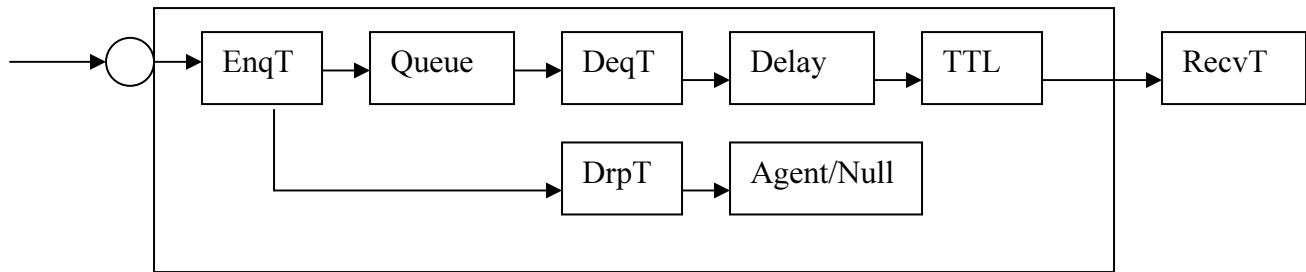

Figure 5 Link in ns2 when tracing is enabled

These new objects will then write to a trace file whenever they receive a packet. The format of the trace file is following:

```
+ 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
- 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
r 1.84471 2 1 cbr 210 ------- 1 3.0 1.0 195 600
r 1.84566 2 0 ack 40 ------- 2 3.2 0.1 82 602
+ 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
- 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
```

+ : enqueue
- : dequeue
d : drop
r : receive

The fields in the trace file are: type of the event, simulation time when the event occurred, source and destination nodes, packet type (protocol, action or traffic source), packet size, flags, flow id, source and destination addresses, sequence number and packet id.

In addition to tracing all events of the simulation, it is also possible to create a trace object between a particular source and a destination with the command:

$ns create-trace type file src dest

where the type can be, for instance,

- Enque – a packet arrival (for instance at a queue)
- Deque – a packet departure (for instance at a queue)
- Drop – packet drop
- Recv – packet receive at the destination

Tracing all events from a simulation to a specific file and then calculating the desired quantities from this file for instance by using perl or awk and Matlab is an easy way and suitable when the topology is relatively simple and the number of sources is limited. However, with complex

topologies and many sources this way of collecting data can become too slow. The trace files will also consume a significant amount of disk space.

A.6.2.   Monitors

With a queue monitor it is possible to track the statistics of arrivals, departures and drops in either bytes or packets. Optionally the queue monitor can also keep an integral of the queue size over time.

For instance, if there is a link between nodes n0 and n1, the queue monitor can be set up as follows:

set qmon0 [$ns monitor-queue $n0 $n1 stdout]

The packet arrivals and byte drops can be tracked with the commands:

set parr [$qmon0 set parrivals_]
set bdrop [$qmon0 set bdrops_]

Notice that besides assigning a value to a variable the set command can also be used to get the value of a variable. For example here the set command is used to get the value of the variable "parrivals" defined in the queue monitor class.

A flow monitor is similar to the queue monitor but it keeps track of the statistics for a flow rather than for aggregated traffic. A classifier first determines which flow the packet belongs to and then passes the packet to the flow monitor.

The flowmonitor can be created and attached to a particular link with the commands:

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n3] $fmon

Notice that since these commands are related to the creation of the flow-monitor, the commands are defined in the Simulator class, not in the Flowmonitor class. The variables and commands in the Flowmonitor class can be used after the monitor is created and attached to a link. For instance, to dump the contents of the flowmonitor (all flows):

$fmon dump

If you want to track the statistics for a particular flow, a classifier must be defined so that it selects the flow based on its flow id, which could be for instance 1:

set fclassifier [$fmon classifier]
set flow [$fclassifier lookup auto 0 0 1]

In this exercise all relevant data concerning packet arrivals, departures and drops should be obtained by using monitors. If you want to use traces, then at least do not trace all events of the simulation, since it would be highly unnecessary. However, it is still recommended to use the monitors, since

with the monitors you will directly get the total amount of events during a specified time interval, whereas with traces you will have to parse the output file to get these quantities.

A.7. Controlling the simulation

After the simulation topology is created, agents are configured etc., the start and stop of the simulation and other events have to be scheduled.

The simulation can be started and stopped with the commands

$ns at $simtime "finish"
$ns run

The first command schedules the procedure finish at the end of the simulation, and the second command actually starts the simulation. The finish procedure has to be defined to flush the trace buffer, close the trace files and terminate the program with the exit routine. It can optionally start NAM (a graphical network animator), post process information and plot this information.

The finish procedure has to contain at least the following elements

```
proc finish {} {
        global ns trace_all
        $ns flush-trace
        close $trace_all
        exit 0
}
```

Other events, such as the starting or stopping times of the clients can be scheduled in the following way:

$ns at 0.0 "cbr0 start"
$ns at 50.0 "ftp1start"
$ns at $simtime "cbr0 stop"
$ns at $simtime "ftp1 stop"

If you have defined your own procedures, you can also schedule the procedure to start for example every 5 seconds in the following way:

```
proc example {} {
        global ns
        set interval 5
        ….
        …
        $ns at [expr $now + $interval] "example"
}
```

A.8. Simple ns2 example

To give you a better idea of how these pieces can be put together, a very simple example script is given here. The script in itself does not do anything meaningful, the purpose is just to show you how to construct a simulation.

The script first creates the topology shown in Figure 6 and then adds one CBR traffic source using UDP as transport protocol and records all the events of the simulation to a trace file.
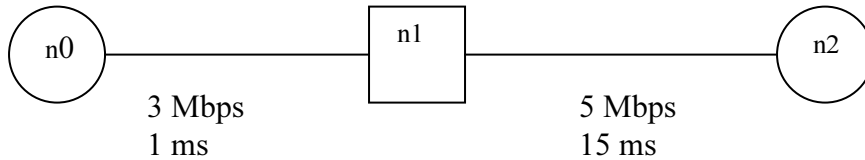
Figure 6 Example simulation topology

The example script:

```
#Creation of the simulator object
set ns [new Simulator]

#Enabling tracing of all events of the simulation
set f [open out.all w]
$ns trace-all $f

#Defining a finish procedure

proc finish {} {
    global ns f
    $ns flush-trace
    close $f
    exit0
}

#Creation of the nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]

#Creation of the links
$ns duplex-link $n0 $n1 3Mb 1ms DropTail
$ns duplex-link $n0 $n1 1Mb 15ms DropTail

#Creation of a cbr-connection using UDP
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
```

```
$cbr0 attach-agent $udp0
$cbr0 set packet_size_ 1000
$udp0 set packet_size_ 1000
$cbr0 set rate_ 1000000
$udp0 set class_ 0

set null0 [new Agent/Null]
$ns attach-agent $n2 $null0
$ns connect $udp0 $null0

#Scheduling the events
$ns at 0.0 "$cbr0 start"
$ns at $simtime "$cbr0 stop"

$ns at $simtime "finish"

$ns run
```