# Tcl and ns2: Tutorial

*Neal Charbonneau* *nealc.com*

## Outline

network simulator 2 (ns2) is a popular open source discrete event simulator for computer networks. It is often used by researches to help evaluate the performance of new protocols or validate analytical models. ns2 allows you to setup a computer network, consisting of nodes/routers and links. You can then send data (packets) over the network using a variety of different protocols at different layers. You may want to use ns2, for example, if you have designed a new protocol to replace or augment TCP. Using ns2, you can implement your new protocol and compare its performance to TCP. This allows you to test ideas before trying real-world experiments.

This document will provide an introduction to using ns2 for networking research. In order to use ns2, you must first understand the programming language Tcl, which is discussed in the next section. With Tcl covered, we will then discuss how to use ns2, including how to collect and plot data. The Tcl section is a very short introduction, but the section on ns2 should reinforce the concepts from the Tcl section. The document will introduce a number of concepts through examples. The source code for the ns2 examples is available along with the PDF. The goal is to show you how to setup an ns2 simulation and how to go about collecting and plotting data from your simulations. It'll focus on a sample simulation network. Many of the details will be ignored, the purpose here is just to provide a starting point. With the examples and sample code you should be able to find more details elsewhere on the web. This will not introduce the concepts of discrete event simulation, how ns2 works internally, or how to interpret the data. This document assumes that ns2 is already installed and that you have some familiarity with Linux, other programming languages, and basic networking concepts.

## OTcl Basics

In order to setup the simulation network in ns2, you must use a language called Tcl. It actually uses an extension of Tcl, called OTcl, which incorporates objects into Tcl. We'll ignore how ns2 uses OTcl for now. The purpose of this section is just to get a basic understanding of OTcl. This tutorial assumes you already know some programming language like C/C++ or Java. It won't introduce programming concepts, but just show you how they work in OTcl. Everything up until the last subsection of this section is in standard Tcl, but I will use OTcl when talking about the language.

Once ns2 is installed, you can access an interactive OTcl prompt by running the *ns* command (from a Linux shell or Cygwin on Windows, for example). In the following examples I'll represent the OTcl prompt by '%'. In this prompt, you can enter your OTcl

code and it will interpret it for you. You can also save your OTcl code in a separate file and use the command *ns <filename>* to execute the entire file at once.

### Everything is a command

Everything in Tcl is a command followed by a number of arguments, which are separated by whitespace. Every line in your OTcl code will be based on the template *command arg1 arg2 ... argn*. For example, there is a *puts* command that takes two arguments. The first argument is the output stream and the second argument is what should be printed to that output stream. Try the following:

```
% puts stdout Hello
Hello
%
```

The command here is *puts*, the first argument is *stdout*, and the final argument is *Hello*. Now, lets say we want to print "Hello World" instead of Just "Hello". We may try:

```
% puts stdout Hello World
bad argument "World": should be "nonewline"
```

As you see, this results in an error. OTcl is treating *World* as a third argument to the *puts* command. We really want *Hello World* to be a single argument. To do that, we need to use a concept called *grouping*. Grouping can be done using either double quotes or brackets.

```
% puts stdout "Hello World"
Hello World
% puts stdout {Hello World}
Hello World
%
```

We will discuss the difference between double quotes and brackets shortly. Lets look at another command called *expr*. This command is used to evaluate expressions. It takes a variable number of arguments, concatenates them together, then evaluates the result.

```
% expr 3 + 2 * 5
13
% expr ( 3 + 2 ) * 5
25
%
```

Note, the OTcl prompt echos the value of the command. This output isn't actually part of the program's output. If you added these two statements to a file and then ran the file, you would not see any output.

In the first case, we give the *expr* command 5 arguments: 3, +, 2, *, and 5. It concatenates them and evaluates the expression. The second examples adds '(' and ')' as additional arguments. It is important to remember the concept of commands and arguments in Tcl. Everything, including control structures like *if* statements, is just a command followed by a number of arguments, which may be within the grouping structures we just discussed.

### Variables

Tcl is a dynamically typed language. This means that we do not have to declare variable types in our code. A variable can store anything, it does not have a specific type like "integer". Type checking is done at runtime.

We can define variables in Tcl using the *set* command. The *set* command takes two parameters. The first is the name of the variable and the second is the value you want to store in the variable.

```
% set x 5
5
% set y Hello
Hello
% set z {Hello World}
Hello World
%
```

Remember that OTcl will treat anything separated by whitespace as a separate argument, so to store "Hello World" in a variable, we must use grouping (with the brackets in this case) for variable $z$. We'll use the variable assignments here in the next couple of examples.

In order to get the value of a variable, we use the concept of *substitution*, where we substitute the variable *name* with its *value*. This is done with the $ symbol in Tcl. You can think of $ as meaning "get the value of".

```
% puts $x
5
% puts $y
Hello
%
% puts y
y
```

Notice in the last line, we attempt to output the variable $y$, but forget to include the substitution symbol. OTcl outputs $y$ instead of getting the value of $y$. Also note that we did not include the *stdout* argument. The output stream argument is optional for *puts* and if not included it defaults to *stdout*.

We can now explain one of the differences between using brackets and double quotes for grouping. The double quotes allow for substitution, while the brackets do not. See the following example.

```
% puts "x is $x"
x is 5
% puts {x is $x}
x is $x
%
```

In the first case, the $ symbol works as expected to substitute $x$ for its value. The brackets in the second case disable substitution so $ is not treated as a special symbol.

### Evaluating and defining commands

So far we've seen how to execute some built-in commands. This is somewhat conceptually similar to executing functions in most programming languages. Most commands return some value that we are interested in. We need some special notation for getting the value that a command returns.

Say we want to print the result of the *expr* command. We need some way of evaluating the value of the *expr* command and then passing that value to the *puts* command. For example, we cannot do:

```
% puts expr 6 * 7
wrong # args: should be "puts ?-nonewline? ?channelId? string"
```

This is giving the *puts* command the arguments: expr, 6, *, and 7. What we really want to do is first evaluate the *expr* command. To do this, we wrap the command with square brackets in OTcl.

```
% puts [expr 6 * 7]
42
% set answer [expr 6 * 7]
42
% puts $answer
42
```

This will first evaluate the *expr* command with the arguments 6, *, and 7, get its value, and then use that value as a parameter to the *puts* command, exactly what we need. The square brackets essentially get the return value of a command. The second OTcl command gets the return value and stores it in a variable, which is then written to the output.

OTcl allows you to define your own commands, which you can think of as procedures or functions in other languages. To define a new command, you must use a command called *proc*, which takes three arguments. The first is the name of the new command, the second is the list of arguments, and the last is the body of the command (what your command will do). Consider the following example. We'll define a command called *triple*, which takes a single argument and returns a value that is triple the argument.

```
% proc triple arg1 { return [expr $arg1 * 3] }
% triple 9
27
% set t [triple 9]
27
% puts $t
27
```

The first argument to the *proc* command is the name of our new command, *triple*. The second argument is the list of arguments our new command will take (which is just one in this case), *arg1*. The third argument is the body of the command. Here, we need the brackets to group the body into a single argument to *proc*. The *return* works just as it does

in regular programming languages. When we wrap our new command in square brackets, we will get whatever is defined by the *return* statement. After we define our command, we can use it just like any other command is OTcl, as shown in the second line of the example.

The definition of our new command may not look very familiar to what functions in C/C++ or other languages look like. To help make this look more like a function or at least more readable, we usually spread this across multiple lines and use brackets around the arguments as well.

```
% proc triple { arg1 } {
     return [expr $arg1 * 3]
}
%
```

This definition is equivalent to the first one, but is much more readable. It is important to remember that *proc* is still a command expecting three arguments though. Because of this, the closing bracket after *arg1* and the next opening bracket *must* be on the same line. Otherwise, OTcl will think that there is a newline being passed to the *proc* command and give an error.

### Built in data types

Most scripting languages come with a number of built in data types. Arrays and lists are the common data types used in Tcl. We'll just briefly discuss arrays here. For a nice introduction to lists, see [1]. We typically won't be coding anything too complex in OTcl for ns2, so just covering arrays should be enough to get started with ns2.

You do not need to define the size of the array like you would in other programming languages. The indices for the array are also not restricted to integers like most languages. Arrays here are more like hash maps. A few quick examples should give you an idea of how to use arrays.

```
% set arr(0) 30
30
% set arr(1) 50
50
% puts $arr(1)
50
% set arr("something") 99
99
% puts $arr("something")
99
% set x 0
0
% puts $arr($x)
30
%
```

From the commands above, we can see that arrays are very similar to normal variables except with the parenthesis to denote the index. OTcl has a command, called *array* to deal with arrays. For example, if we wanted to get the length of the array, we can issue the following command.

```
% set length [array size arr]
3
%
```

The *array* command performs a number of different operations based on its first argument (*size* in this case). For more, see the Tcl documentation.

### Control structures

Tcl comes with a number of built in commands for control structures like *if* statements and different kinds of loops. We'll just look at a few examples and it should be obvious how these commands work. They are very similar in concept the control structures of typical programming languages.

```
1 set x 5
2 if { $x == 1 } {
3         puts "Not true"
4 } else {
5         puts "True"
6 }
```

A quick note about using these commands. It is important to remember that this is a command with arguments and not some built in functionality of the language. This means that the following will give you an error.

```
1 set x 5
2 if { $x == 5 }
3 {
4         puts "True"
5 }
```

This is because the *if* command takes at least two arguments: the condition and the body. In the above example, one of the arguments is a newline character. You have to put the close bracket and open bracket between arguments on the same line. (This is similar to what we saw earlier when defining new commands with *proc*).

You can also use create if/elseif statements in Tcl.

```
1 if { $x == 1 } {
2     puts "first"
3 } elseif { $x == 2 } {
4     puts "second"
5 } else {
6     puts "third"
7 }
```

Lets look at an example of a while loop. The *while* command takes two arguments. The first is a condition it will check on each iteration and the second is the body of the loop.

```
set value 1
set fact 2
while { $fact <= 5 } {
    set value [expr $value * $fact]
    incr fact
}
puts $value
```

Lastly, we can also create for loops using the *for* command. The for loop works the same way it does in C or Java. The first argument to the *for* command is evaluated once. The second is checked after each iteration, the third is executed after each iteration, and the last argument is the body of the loop.

```
set sum 0
for { set i 0 } { $i < 10 } { incr i } {
    set sum [expr $sum + $i]
}
puts $sum
```

## Objects

In this last section, we'll discuss how to use classes and objects. Tcl itself does not support objects, but the OTcl extension, used by ns2, does. It is best to describe how to create a class through an example. We'll create a counter class with a constructor and three methods. The constructor will initialize an instance variable to zero. There will be two methods to increment and decrement this variable. There will be one more function to get the value of the variable. The code is show in the listing below.

```
Class Counter

Counter instproc init { } {
    $self instvar value
    set value 0
}
Counter instproc increment { } {
    $self instvar value
    set value [expr $value + 1]
}
Counter instproc decrement { } {
    $self instvar value
    set value [expr $value -1 ]
}
Counter instproc getValue { } {
    $self instvar value
    return $value
}

set count [new Counter]
$count increment
$count increment

puts [$count getValue]
```

To define the class, we first give it a name, as shown on line 1. Next, we define the constructor on line 3. The constructor is just a regular function with a special name: *init*. To add functions to the class, we use *instproc*. On line 3 we define a new function called *init* that takes no arguments. Defining functions of the class is just like defining commands, except instead of *proc* you use *<ClassName> instproc*. These functions can be defined anywhere, even in separate files. Classes can have instance variables that are accessed through the *self* variable as shown in line 4. All functions of the class have access to any instance variables. Line 4 says that we want to access an instance variable called *value*. In the rest of the function, we can use *value* as a regular variable and anything we do to it here will be seen by all other functions of the class. In this case, on line 5 we initialize it to zero. After *init* is called, any other function accessing the *value* variable will see its value as zero.

Lines 7-18 define the other three functions. Notice they all access the same instance variable. Other than the special *instproc* and *self instvar*, defining functions of a class is the same as defining your own commands. On line 20, we create an instance of the Counter class and store it in the *count* variable. Using *new* will call the constructor (the *init* method). The remaining three lines show how we can call functions on the object. The square brackets are again used to get the return value.

## Using ns2

With the basics of OTcl, we can start discussing how to use ns2 to setup network simulations. The command we've been using in the previous section to execute OTcl code, is actually the main ns2 binary, which means ns2 is essentially just an interpreter for OTcl. ns2 provides a number of OTcl objects that are used to setup a simulation environment (e.g. there are objects representing network nodes). Using these objects, we setup our topology and use a special object that will actually execute the simulation. The entire simulation does not take place in OTcl, but much of it is done in C++. There is a lot material on the web discussing how ns2 works behind the scenes. This section does not discuss how ns2 works, but how to use ns2 to setup a simulation.

### First script

We'll start by creating a simulation script step-by-step. The completed script will be shown at the end of the section (the source is also available along with this document). We are going to create the network topology shown in Figure 1. The circles in the figure represent nodes in the network. The two leftmost nodes have an FTP application running on them, both are sending data to the right-most node. The lines between the nodes represent links in the network, where each link has a specified bandwidth and delay.

The first thing we typically do is create an instance of the *Simulator* object, which is a class provided by ns2. The *Simulator* objects provides functions for creating nodes, links, and running the actual simulation.

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

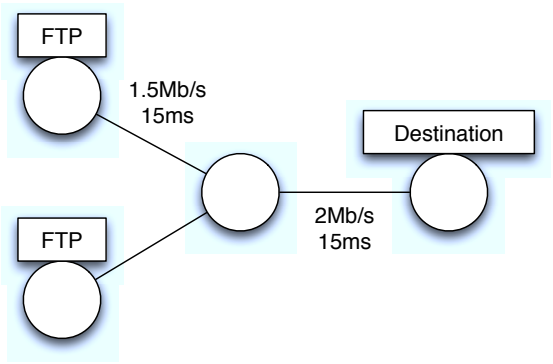


Figure 1: Simulation topology we will create. The circles are the nodes connected by links. We will have an FTP sender application on the two left-most nodes sending data to the right-most node.

We've seen the syntax for creating instances of objects before. Note, you can view the source of the *Simulator* object, in case you are unsure of how something works. It is available in the file tcl/lib/ns-lib.tcl within the ns2 source code.

With the *Simulator* object, we create create nodes and links. In Figure 1 we see that we will need four nodes and three links. They can be created as follows.

```
set s1 [$ns node]
set s2 [$ns node]
set router [$ns node]
set d [$ns node]
$ns duplex-link $s1 $router 1.5Mb 15ms DropTail
$ns duplex-link $s2 $router 1.5Mb 15ms DropTail
$ns duplex-link $router $d 2Mb 15ms DropTail
```

As we can see, the *Simulator* object has a function called *node* to create new nodes. $s1$ and $s2$ represent the left-most nodes, *router* is the middle, and $d$ is the right-most node. The *Simulator* object also has a function called *duplex-link*, which creates a link to send data on between two nodes. Its parameters are the source node, the destination node, the bandwidth, the delay, and the queuing discipline. The code above will setup the physical topology of our network. The next thing to do is setup the transport and application layers.

As shown in the figure, we have two FTP applications sending data to the destination node. In order to use FTP, we need a transport layer protocol to transfer the data. We will use TCP to do this. The *Simulator* object has a function called *create-connection* that can be used to setup TCP connections between two nodes.

```
set tcpSender1 [$ns create-connection TCP/Reno $s1 TCPSink $d 0]
set tcpSender2 [$ns create-connection TCP/Reno $s2 TCPSink $d 1]
```

A TCP connection in ns2 is defined by a sending agent and a receiving agent. These mimic the behavior of a source node sending data and a destination node responding with acknowledgments. The first argument to the *create-connection* function is the sending agent.

The second argument is the source node. The third is the receiving agent while the fourth is the destination node. The last argument is a unique identifier for this flow (which we'll use later). In this case, we want to setup two TCP connections. One between $s1$ and $d$ and another between $s2$ and $d$. The code above does exactly that using TCP Reno. The function returns a reference to a TCP sender object. What this function does is ensure that when data is given to the TCP sender object, its destination address and port is that of the TCP receiver object at the destination node (we don't have to worry about routing).

Now, we have to define the FTP applications which will use the TCP sender objects to send data from the source to the destination. In ns2, the FTP application simulates a data transfer by sending a constant stream of data to the TCP sender.

```
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcpSender1
set ftp2 [new Application/FTP]
$ftp2 attach-agent $tcpSender2
```

We start by creating a new FTP OTcl object, one for each sender. The FTP object has a function, *attach-agent*, which is used to tell FTP who to give its generated packets to. In this case, we tell each FTP sender to send its packets to the TCP sender, which will then ensure the packets reach the destination. Underneath TCP is the routing layer, which we do not have to worry about, ns2 will setup routes and move packets around for us.

We have now defined everything we need to send packets over the network using FTP and TCP. The next thing we have to do is actually start and stop the simulation. Right now, nothing would happen if we ran the code. The FTP agents have a function called *start*, which will make them constantly generate packets. With ns2, we can specify at what time we want events to happen. For example, we could have one FTP agent start sending at time X and the other at time Y. To specify when events happen, we use the *at* function of the *Simulator* object.

```
$ns at 0.0 "$ftp1 start"
$ns at 0.0 "$ftp2 start"
$ns run
```

The first line specifies at time 0 (in seconds), tell the first FTP sender to start sending data. The second line does the same thing for the second FTP sender. At this point, these two events are added to the simulator's event queue, but the simulation hasn't started yet. To actually begin the simulation, we use the *run* command. This command sets up the routing, performs some other tasks, then begins executing the events in the event queue including the two events we just defined.

Right now, if we ran this code, it would run forever. The FTP senders will keep sending data indefinitely. We need to tell the simulation when to stop. To do this, we need two things. First, we'll define a command to be called when the simulation is complete. Next, we need to tell the simulator when to call that command and exit the simulation. Here is our command to finish the simulation.

```
proc finish { } { puts "Complete."; exit 1 }
```

When it is called, it will simply write “Complete” to the console and exit the simulation. We now need to tell the simulator when to execute this command. This is also done with *at*.

```
$ns at 5.0 "finish"
```

This specifies that after five seconds of simulation time have passed, the simulator must call the *finish* command, which will then exit the simulator. We can now execute the simulation, but it wouldn't do anything useful. Right now it will only output "Complete" when it finishes. As the simulation is running, we can tell ns2 to output various types of information. We'll go into more detail in the next section, but for now we'll add some more commands to our script to output some data. We'll write the output to a file called “trace.tr”.

```
set trace [open trace.tr w]
$ns trace-all $trace
```

The *open* command is a built in Tcl command which opens a file. *trace-all* is part of the *Simulator* object and tells the simulator to write a line to the output for each event. Again, we'll discuss this in more detail later.

Putting this all together we get the following script, which implements the simulation setup shown in Figure 1.

Listing 1: ns2 script implementing Figure 1

```
1  set ns [new Simulator]
2
3  set nf [open "nam.out" w]
4  set trace [open "trace.tr" w]
5  $ns namtrace-all $nf
6  $ns trace-all $trace
7
8  Agent/TCP set window_ 50
9
10 proc finish {} {
11     global nf trace ns
12
13     $ns flush-trace
14     close $nf
15     close $trace
16
17     puts "Simulation complete";     exit 1
18 }
19
20
21 set s1 [$ns node]
22 set s2 [$ns node]
23 set router [$ns node]
24 set d [$ns node]
25
26 $ns duplex-link $s1 $router 1.5Mb 15ms DropTail
27 $ns duplex-link $s2 $router 1.5Mb 15ms DropTail
```

```
28 $ns duplex-link $router $d 2Mb 15ms DropTail
29 set tcpSender1 [$ns create-connection "TCP/Reno" $s1 "TCPSink" $d 0]
30 set tcpSender2 [$ns create-connection "TCP/Reno" $s2 "TCPSink" $d 1]
31
32 set ftpSender1 [new Application/FTP]
33 $ftpSender1 attach-agent $tcpSender1
34
35 set ftpSender2 [new Application/FTP]
36 $ftpSender2 attach-agent $tcpSender2
37
38 $ns at 0.0 "$ftpSender1 start"
39 $ns at 0.0 "$ftpSender2 start"
40 $ns at 5 "finish"
41 $ns run
```

We can run this using the *ns* command from the command line. Once the simulation is complete, you will see two files: "trace.tr", which we discussed, and "nam.out" (see lines 3 and 5 above). We'll talk about the "trace.tr" file in the next section. The "nam.out" file is a visual trace of the simulation. ns2 comes with a program called *nam* that allows you to visually see the packets moving around the network. Executing the command *nam nam.out* at the command prompt will open up a GUI that allows you to start/stop/pause/rewind etc., the simulation that ns2 just finished.

There are a few things in the code above that we did not mention previously. First, notice that in the *finish* command we close all of our files and call *flush-trace*. Also note that to access those variables from within *finish* we need to declare them as global on line 11. You'll also see line 8 is new. TCP's flow control functionality includes the concept of an advertised window, which is the maximum amount of data the receiver is willing to receive. This can be simulated in ns2 by setting a special variable in the OTcl TCP sender object. This is what line 8 is doing. We are setting the advertised window to 50 packets.

### Data collection

In this section we'll consider how to deal with the trace file output of the previous section. We'll use the program *gnuplot* to make some graphs of the data as well. *gnuplot* is open source and comes installed on most Linux distributions.

The trace file we got in the previous section (trace.tr) is a standard format used by ns2. In ns2, each time a packet moves from one node to another, or onto a link, or into a buffer, etc., it gets recorded in this trace file. Each row represents one of these events and each column has its own meaning. The full documentation for this file can be found at [2].

We'll use an example to see how we can extract some useful information from this trace file. Let's say we want to record the number of bytes received at the destination per second for each of the FTP senders. In other words, we want to record each sender's goodput in Mb/s. We will also graph this using *gnuplot*. The first step is to collect the data we need, the goodput in Mb/s, every second of the simulation. We need to know when a packet reaches the destination from each sender and how big that packet is. We can simply sum this up for each second of the simulation. This information is in our trace file. The columns we are interested in are:

- Column 1: the type of event: (packet received, dropped, etc)

- Column 2: the time the event happened
- Column 4: the destination node
- Column 5: the type of packet (TCP, UDP, etc)
- Column 6: size of the packet received
- Column 8: the flow ID

We want to know when a packet is received at the destination, which we can get from columns 1 and 4. We can differentiate between the two senders by using column 8. The flow ID is the last parameter to the *create-connection* function we discussed earlier. We want only care about TCP packets, which we can determine from column 5. Lastly, to calculate the data in Mb/s, we can use a combination of columns 2 and 6. Column 4 requires some additional explanation. Nodes in ns2 do not have IPv4 or IPv6 addresses. Instead, they are assigned integer addresses starting from 0. The first node you create will have address 0, the next will have address 1, and so on. In our example above, $s1$ has address 0, $s2$ has address 1, *router* has 2, and $d$ has 3.

Clearly, all the information we need is there, but we need to put it into a useful format. Let's say that we want to store the goodput of each connection in a separate file with four columns. The first column is the time, the second is the goodput of flow 1, the third is the goodput of flow 2, and the last column is the total goodput. Storing the data in column format will make it easier to display in a graph later. What we want to do this is transform the trace file "trace.tr" into a file that looks like the following.

```
1 0.76576 0.84896 1.61472
2 0.22464 1.77216 1.9968
3 0.67392 1.3312 2.00512
4 0.59904 1.39776 1.9968
5 1.14816 0.4576 1.60576
```

Since we are dealing with column oriented data, AWK is probably the easiest tool we can use to format our data. AWK is a simple scripting language that scans through a file line by line. It allows you to access any column in the current line by using special variables \$1, \$2, \$3, etc. for the first, second and third columns. The definition of each column of the trace file is shown above, so we can use the AWK script to check the value of each column and collect the data we need. A sample AWK script is shown below.

Listing 2: AWK script used to parse the trace file.

```
1 #!/usr/bin/awk -f
2
3 BEGIN {
4     last = 0
5     f1 = 0
6     f2 = 0
7     total = 0
8 }
9 {
```

```
10      if ($5 == "tcp" && $1 == "r" && $4 == "3") {
11          if ($8 == "0") {
12              f1 += $6
13          }
14          if ($8 == "1") {
15              f2 += $6
16          }
17          total += $6
18      }
19
20      #every second
21      if ($2 - 1 > last) {
22          last = $2
23          print $2, (f1*8/1000000), (f2*8/1000000), (total*8/1000000)
24          f1 = 0
25          f2 = 0
26          total = 0
27      }
28  }
29  END {
30      print $2, (f1*8/1000000), (f2*8/1000000), (total*8/1000000)
31  }
```

The BEGIN and END sections are only executed once (before and after the file has been processed). The middle section is executed for each line of the file. The AWK script keeps three variables to store the goodput in Mb/s for flow 1, flow 2, and the total. For each line of the trace file, it checks to see if a TCP packet ($5 == "tcp") is received ($1 == "r") at node 3 ($4 == "3"), which is our destination node. If so, it increments the count for the appropriate flow, using the size of the particular packet (in bytes) from column 6. After each second, it prints the total while converting bytes to Mb.

With this AWK script, we can parse the tracefile by running *./tp.awk trace.tr > tp.tr*, for example. The file "tp.tr" will contain the goodput in the column format we need. Now we want to graph this data. *gnuplot* is a tool that will allow us to create graphs from data stored in column format, which is why we stored our goodput in columns. *gnuplot* is pretty self-explanatory, so we'll just show the code below.

Listing 3: gnuplot script to process the output of our AWK script.

```
1  set term postscript eps enhanced color
2  set title "Throughput"
3  set xlabel "Time (s)"
4  set ylabel "Throughput (Mbps)"
5  set output "tp.eps"
6
7  plot "tp.tr" using 1:2 title "Flow 1" with linespoints,\
8  "tp.tr" using 1:3 title "Flow 2" with linespoints
```

The *gnuplot* script above will generate an encapsulated postscript file. You may need to convert this to another format to open it on Windows (or output to a different format such as png). To run the file, you can use the following command: *gnuplot tp.gp*. Once this is executed, you should have a file "tp.eps" in the same directory. This is the graph of your goodput. Let's look at how it works. The first line just specifies the type of output. Line

2-4 specifies the labels and title of the resulting graph. Line 5 specifies the output file name. Lines 7 and 8 are the lines that tell *gnuplot* how to plot the data. We are telling it to use data in the "tp.tr" file, which is the output of our AWK script. Line 7 says use columns 1 and 2 to draw a line with the title "Flow 1". Column 1 (time) will be used as x-axis values while column 2 (goodputin Mb/s) will be used as the y-axis values. Line 8 does the same thing but uses column 3 for goodput.

### Custom data collection

In the previous section we parsed some data from the standard trace file produced by ns2. What if the data we need is not in that trace file? We can modify our ns2 script to get some additional information while the simulation is running and output it to a file. We'll go through an example of collecting the TCP senders' congestion window values over the course of the simulation.

Each ns2 object implemented in OTcl has a number of variables that we can read (and change) while the simulation is running. The TCP sender object (TCP/Reno) has a variable called *cwnd_*, which represents that sender's current congestion window. As the simulation is running, the value of this variable is updated. What we want to do is get the value of this variable during the simulation and write it to a file so that we can plot it when the simulation is done running.

First, we'll create a new file to store this extra data.

```
set cwndTrace [open cwnd.tr r]
```

Next, we'll create a new command that will be called repeatedly while the simulation runs. Each time it is called, we'll get the value of the congestion window and write it to the file we just opened.

```
1 proc record { } {
2   global tcpSender1 tcpSender2 ns cwndTrace
3
4   $ns at [expr [$ns now] + 0.6] "record"
5   set cwnd1 [$tcpSender1 set cwnd_]
6   set cwnd2 [$tcpSender2 set cwnd_]
7   puts $cwndTrace "[$ns now] $cwnd1 $cwnd2"
8 }
```

In order to access variables outside of the command's body, we can use *global*, as shown in the example above. We need to access the TCP senders, the *Simulator* object, and our trace file.

Let's start with line 5 and 6. Here, we get the value of the congestion window of each TCP sender object. As we mentioned previously, each TCP sender object has an instance variable called *cwnd_*. We can get its value as shown on lines 5 and 6. We are then going to write those values to our trace file on line 7. The *now* function of the *Simulator* object gets the current time. Our custom trace file will have three columns. The first is the time and the second and third are the current values of the TCP sender's congestion windows. Just calling this command once isn't any good. We need to call it repeatedly over the simulation. That is what line 4 will do. We are telling the *Simulator* object to schedule another event

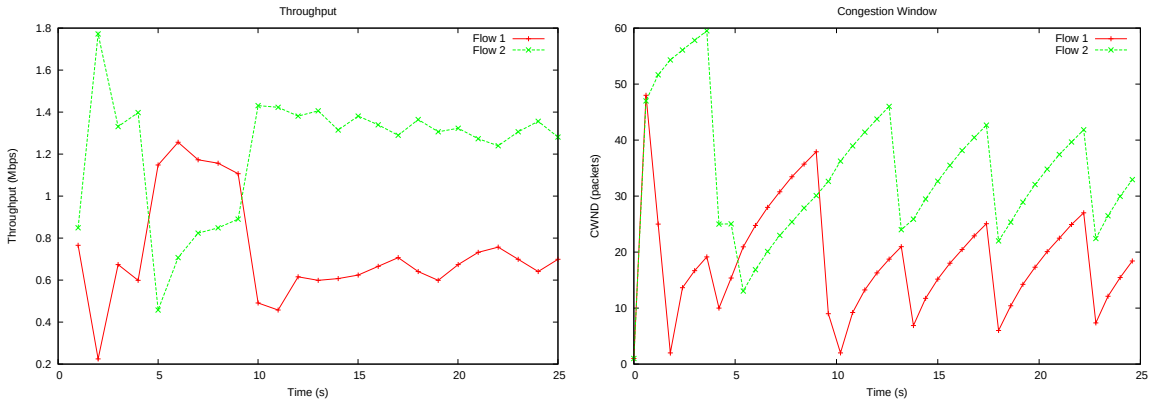


Figure 2: Sample *gnuplot* graphs of throughput and congestion window from the simulation topology in Figure 1.

600ms from the current time. This event is to call the *record* function again. After *record* is executed once, it will continuously execute itself every 600ms, as long as the simulation is running. The last thing we need to do is call this function at the start of the simulation.

```
ns at 0.0 "record"
```

Once the simulation finishes, the "cwnd.tr" file will contain the TCP sender's congestion window values for every 600ms of the simulation. We can then write a small *gnuplot* script to display the data as a graph. See the sample code along with this document. Sample graphs can be seen in Figure 2.

Many OTcl objects have variables like this that we can use to collect data. The file tcl/lib/ns-defaults.tcl shows all of the variables in all the objects that we can access in this way. If you need to collect data that isn't in the trace file or is readily available like TCP's congestion window, you typically have to modify the C++ source code and re-compile ns2.

# References

[1] K. Waclena, "Lists and keyed lists." [Online]. Available: http://www2.lib.uchicago.edu/keith/tcl-course/topics/lists.html

[2] "Ns-2 trace formats." [Online]. Available: http://nsnam.isi.edu/nsnam/index.php/NS-2_Trace_Formats