RPI Graph and Statistics Package for `ns`
User Manual and Tutorial

Internal Distribution Only
RPI Computer Science Department
RPI Electrical, Computer, and Systems Engineering Department

November 2, 2001

Abstract

This user manual contains an installation guide and a short tutorial for the RPI graphing tools for ns.

1 Introduction

The RPI graph package provides a set of classes for generating commonly used graphs. These graphs may appear in gnuplot, xgraph, xdvi, ghostview, or acroread windows or the graphs objects may simply output to encapsulated postscript for later use such as inclusion in a latex document.

2 Installation

Presumably you already have the distribution and have installed ns-2. The graph package has been tested with ns-2.1.b5. There may be incompatibilities with other ns versions. If you have not downloaded the graph package then get it from
`http://www.ecse.rpi.edu/~harrisod/graph-v6.0.4.tar.gz`.
(The URL is likely to change before external distribution).

In order for the statistics and graphing packages to function, you must define the `NS` environment variable pointing to the root of the `ns` distribution. The following directions apply to `ns` version 2.1b5. Make a backup of your `ns` distribution before attempting this installation.

NOTE: If you installed graphing and statistics packages when you installed the farmer-worker then you do not need to perform any of the following steps. You can skip to the tutorial section 3.

1. Move to a temporary directory

```
> gunzip graph-v6.tar.gz
> tar -xf graph-v6.tar
> cd graph-v6
```

2. Move directories from the distribution into your $NS distribution.

```
> cp -r rpi $NS
> cp -r tcl/rpi $NS/tcl/rpi
> cd $NS
```

3. Insert the following lines in your `ns` Makefile

```
[...]
INCLUDES= \
    -I. -Irpi \
[...]
LIB= \
    -lpthread \
[...]
OBJ_CC= \
[...]
rpi/byte-counter.o rpi/delay-monitor.o rpi/file-tools.o \
        rpi/rate-monitor.o rpi/rpi-flowmon.o rpi/rpi-queue-monitor.o \
[...]
```

4. Rebuild `ns`

```
> cd $NS
> rm gen/*
> make depend
> make
```

3 Tutorial

All of these examples assume a working knowledge of `ns`. If you have not created `ns` scripts before then consult the `ns` documentation and write a few test scripts before proceeding here.

Currently the graph package provides the graphs shown in table 1.

CWndVersusTime	TCP congestion window size versus time
FlowQLenVersusTime	sampled per-flow queue contribution versus time
PointToPointDelayVersusTime	delay between two points on the network
QDelayVersusTime	sampled queuing delay versus time
QLenVersusTime	sampled queue length versus time
RateVersusTime	arrival rate versus time over constant
REDQueueVersusTime	avg queue, instant queue versus time
RTTVarianceVersusTime	TCP variance in round-trip-time versus time
RTTVersusTime	TCP round-trip time versus time
Sequence	TCP sequence number versus time
SRTTVersusTime	TCP smoothed round-trip time versus time
UtilizationVersusTime	utilization versus time over constant interval length

Table 1: Provided Graph classes

3.1 Example 1: Graphing link statistics

To include the graph package we preface every experiment script with the following:

```
source $env(NS)/tcl/rpi/graph.tcl
```

The graph package contains several graphs for gathering statistics about traffic passing through a link including queue length versus time, utilization versus time, rate versus time, and queue delay versus time. In this example we will show how to graph utilization over given fixed intervals versus time.

Assume we have created a topology with a known bottleneck spanning ns nodes `n0` and `n1`. Next we add the lines shown in figure 1 to our script. To collect statistics for our graph we instantiate a `Graph/UtilizationVersusTime` object. The arguments `$n0` and `$n1` tell the graph object to collect statistics for the link spanning between nodes `n0` and `n1`. The 0.1 argument tells the graph object that it should measure utilization over 0.1 second intervals. When the simulation finishes, we call `$util_graph display` to tell the graph object to show a window containing our utilization versus time plot generated using `gnuplot`. We show the complete source code for this script in `graph-v6/examples/ex1.tcl`, and the generated graph in figure 2.

3.2 Example 2: Outputing to Various Plot Devices

The graph package can output graphs to windows or files using various applications or formats for displaying/saving graphs. We show the current set of support “plot devices” in table 2.

```
[...]
proc finish {} {
  global util_graph
  [...]
  $util_graph display
  exit 0
}
[...]  ;# define nodes n0 and n1 and link spanning them.
# create graph of utilization vs time for linkk [n0,n1].
set util_graph [new Graph/UtilizationVersusTime $n0 $n1 0.1]
```

Figure 1: Utilization versus time script: ex1.tcl

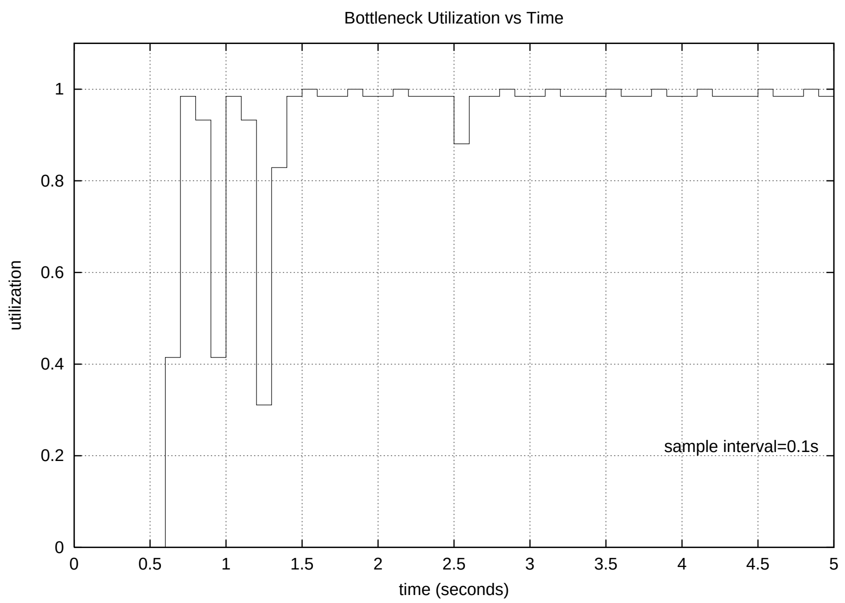

Figure 2: Graph generated by ex1.tcl

In `graph-v6/examples/ex2.tcl` we provide an example script that uses an `xdvi` plot device to display multiple graphs in the same window in the designated order. One of the first things we do in this script is to create a plot device:

```
Graph set plot_device_ [new xdvi]
```

Here we create an instance of the `xdvi` class and assign it to the `Graph plot_device_` class member. Whenever a graph object is instantiated it by default uses the plot device defined in the `Graph plot_device_` class member. All of the classes can be instantiated by passing a plot device object to their respective `init` instprocs. This allows you to use a different plot device for each graph object if you so wish.

acroread	When closed, display all graphs plotted with this device using acroread.
fig	Output to fig file via gnuplot.
ghostview	Output eps via gnuplot then display using ghostview
gnuplot	Show plot in X window using gnuplot.
gnuplot35	Show plot in X window using gnuplot 3.5 (no comments).
latex	Output to eps file and create latex file that includes the graphs.
pdf	Output to pdf file from eps using epstopdf.
postscript	Output to encapsulated postscript via gnuplot.
postscript35	Output to eps via gnuplot 3.5 (no comments).
xdvi	When closed, display all graphs plotted with this device using xdvi.
xgraph	Show plot in X window using xgraph.

Table 2: Supported plot devices

We show the relevant snippets from the `ex2.tcl` script in figure 3. Not shown in figure 3 we create a topology containing a bottleneck between nodes `$n0` and `$n1`. We then define two TCP connections that pass through the bottleneck. The first of these two connections has the TCP agent object `$tcp0`. Shown in figure 3, we create three graph objects of which the first two install statistics gathering objects into the bottleneck link from `n0` to `n1`. The last graph object traces the congestion window of `tcp0`. When the simulation completes we output each of the graphs to encapsulated postscript by calling each object's `display` instproc. Next we close the plot device causing the plot device to compile a latex file that references the eps files generated by each graph object. The latex compiler outputs an dvi file which the plot device then opens using xdvi.

3.3 Example 3: Bottleneck link statistics

We now move on to demonstrating how to collect statistics using the statistics functions included with the graph package. The objective of these statistics gathering tools is to simplify statistics gathering over the standard methods provided by `ns`.

To include the link statistics gathering functions do the following:

```
source $env(NS)/tcl/rpi/link-stats.tcl
```

To begin gathering statistics on a link spanning from nodes `n0` to `n1`, do the following:

```
set stats [new LinkStats $n0 $n1]
```

We show code snippets from `examples/ex3.tcl` in figure 4. The calls to `get-utilization`, `get-mean-queue-delay`, and `get-packet-arrivals` returns the link statistics gathered from the moment the link-stats object was instantiated. In this case, that means reporting statistics gathered from the beginning of the simulation. There are many more statistics we could report by simply calling any of the instance proceduers in table 4.

If you wish to gather statistics starting from some time into the simulation then simply instantiate the `LinkStats` object at that time. If you want to gather statistics over consecutive time intervals (.e.g, from the beginning of the simulation and from after some time for initial transients) then you can reset the link statistics by calling the `LinkStats reset` instance procedure. If you

```
[...]
Graph set plot_device_ [new xdvi]
[...]
proc finish {} {
  global util_graph qlen_graph cwnd0_graph

  $util_graph display
  $qlen_graph display
  $cwnd0_graph display

  [Graph set plot_device_] close

  run-nam
  exit 0
}
[...]  ;# define a bottleneck between n0 and n1.

set util_graph [new Graph/UtilizationVersusTime $n0 $n1 0.1]
$util_graph set title_ "Bottleneck Utilization vs Time"

set qlen_graph [new Graph/QLenVersusTime $n0 $n1]
$qlen_graph set title_ "Bottleneck Queue Length Versus Time"

set cwnd0_graph [new Graph/CWndVersusTime $tcp0]
$cwnd0_graph set title_ "cwnd of flow 0 versus Time"
[...]
```

Figure 3: Example 2: Display multiple graphs using xdvi

want to collect a link's statistics for overlapping intervals then create two `LinkStats` objects each at a time when you want to begin collecting statistics then retrieve the statistics at the end of their respective intervals. Installing more than one `LinkStats` object on a link is particularly useful when you want to record statistics for the whole simulation as well as for the steady-state (i.e., skipping some time to eliminate initial transients).

We provide the code for this complete example in `graph-v6/ex/ex3.tcl`.

```
[...]
proc finish {} {
  global n0 n1 tcp0 tcp1 stats

  # output link statistics
  puts "Bottleneck statistics: "
  puts "Utilization: [$stats get-utilization]"
  puts "Number of drops: [$stats get-packet-drops]"
  puts "Mean queue delay [$stats get-mean-queue-delay]"
  puts "Number of arrivals [$stats get-packet-arrivals]"
  [...]
}
[...]  ;# create topology w/ bottleneck between n0 and n1.

set stats [new LinkStats $n0 $n1]
[...]
```

Figure 4: Example 3: Gathering link statistics

`init` $n0$ $n1$ [$qmon$]	Instantiates a LinkStats object spanning between $n0$ and $n1$. You must have previously created these two nodes and a link spanning these nodes. More than one LinkStats object can span between $n0$ and $n1$. By default the LinkStats object uses an instance of the `RPIQueueMonitor` class to collect statistics. You can install your own queue monitor using the optional third argument.
`reset`	resets all statistics to the initial state.

Table 3: Special instance procedures for LinkStats

`get-utilization`	Returns the link utilization as the number of bytes departing the queue as the number of bytes that could have departed the queue.
`get-packet-utilization` *avgpktsz*	Returns the link utilization as the number of packets that departed the bottleneck link times the average packet size over the number of average sized packets that could have departued the link. The argument specifies the average packet size.
`get-throughput`	Returns the number of byte arrivals at the tail of the queue * 8 over time.
`get-packet-arrivals`	Returns the number of packet arrivals at the tail of the queue.
`get-byte-arrivals`	Returns the number of byte arrivals at the tail of the queue.
`get-packet-drops`	Returns the number of packets dropped.
`get-byte-drops`	Returns the number of bytes dropped.
`get-packet-departures`	Returns the number of packets that departed the queue.
`get-byte-departures`	Returns the number of bytes that departed the queue.
`get-mean-queue-delay`	Returns the mean delay experienced by packets transitting the queue.
`get-queue-delay-variance`	Returns the variance in delay experienced by packets transitting the queue.
`get-queue-delay-stddev`	Retursn the standard deviation in delay experienced by packets transitting the queue.
`get-mean-packet-queue-length`	Returns the mean packet queue length. This is not the same as the mean queue length seen by an arriving packet which would not take into account idle times in the mean.
`get-mean-byte-queue-length`	Same as `get-mean-packet-queue-length` except the result is returned in units of bytes.
`get-max-packet-queue-length`	Returns the maximum queue length in packets.
`get-max-byte-queue-length`	Returns the maximum queue length in bytes.
`get-min-packet-queue-length`	Returns the minimum queue length in bytes. Typically this would be zero except when measured over some small interval.
`get-min-byte-queue-length`	Same as `get-min-byte-queue-length` except the result is returned in units of bytes.

Table 4: LinkStats statistics instance procedures

3.4 Example 4: TCP statistics

The most direct way to obtain TCP statistics is to simply query the bound variables defined for the Agent/TCP class. For example, if we define a TCP agent `$tcp0` then we can do the following without extending `ns-2`:

```
puts "Number of data packets transmitted by flow 0: \
  [$tcp0 set ndatapack_]"
puts "Number of data packets transmitted by flow 1: \
  [$tcp1 set ndatapack_]"
puts "Retransmission timeouts for flow 0: \
  [$tcp0 set nrexmit_]"
puts "Retransmission timeout for flow 1: \
  [$tcp1 set nrexmit_]"
```

When using our TCP statistics gathering functions it is necessary to include the `tcp-stats.tcl` package. This loads the definitions shown in tables 5 and 6:

```
source $env(NS)/tcl/rpi/tcp-stats.tcl
```

Then initialize various counters by calling `init-stats`. `init-stats` can also be called to simply reset the the statistics for both our extensions and the statistics gathered by `ns-2.1b5`.

```
[...]
$tcp init-stats
[...]
```

At some later time, such as when `finish` is called, you can output tcp-statistics as follows:

```
puts "Goodput Variance: \
  [get-goodput-stddev "$tcp0 $tcp1"]"
puts "Total TCP packets: \
  [get-total-data-packets "$tcp0 $tcp1"]"
```

In the example above, `get-goodput-stddev` and `get-total-data-packets` operate on a list of tcp agents. In this case returning the standard deviation in goodput and the sum of the data packets sent respectively across the passed list of agents. The above examples can be found in `examples/ex4.tcl`.

`get-useful-packets`	Returns the number of packets transmitted containing new data. over time.
`get-useful-bytes`	Returns the number of bytes transmitted containing new data. The number of useful bytes in each useful packet is the packet size minus the TCP and IP headers. The header size is determined by `tcpip_base_hdr_size_` defined in `$NS/tcl/lib/ns-default.tcl`. In ns-2.1b5, `tcpip_base_hdr_size_` is set to 40.
`get-goodput-bps`	Returns the rate of useful bits transmitted by the network. Uses get-useful-bytes to determine the total number of useful bits. Packets that have been lost but not yet detected by the source are counted as useful because our measure is based on state maintained by the TCP source. In a long simulation this should have negligible impact on the goodput measure.
`get-goodput`	Returns goodput in bps over the bottleneck capacity.

Table 5: TCP statistics gathering instance procs

`get-mean-goodput` *tcplist*	Calculates mean goodput in bps across the passed TCP agents (i.e., connections).
`get-goodput-variance` *tcplist*	Calculates variance in goodput (bps) across the passed TCP agents (i.e., connections).
`get-goodput-stddev` *tcplist*	Calculates the standard deviation in goodput in bps across the passed TCP agents (i.e., connections).
`get-goodput-cov` *tcplist*	Calculates the Coefficient of Variation (C.O.V.) across the passed TCP agents. C.O.V. is standard deviation over the mean.
`get-total-data-packets` *tcplist*	Calculates the sum of the packets sent across the passed TCP agents (i.e., connections).
`get-total-retransmitted-packets` *tcplist*	Calculates the sum of the packet retransmissions across the passed TCP agents (i.e., connections).
`get-total-retransmission-timeouts` *tcplist*	Calculates the tume of the retransmission timeouts across the passed TCP agents (i.e., connections).

Table 6: TCP statistics functions operating on lists