Programming Assignments for Graduate Students using GENI

Exploring the Network Awareness of TCP using GENI

Copyright © 2012 Purdue University

Please direct comments regarding this document to fahmy@cs.purdue.edu.

1 Introduction

This project leverages resources on the ProtoGENI aggregate, using the ProtoGENI test scripts [7] or the Omni GENI client [5]. The ProtoGENI tutorial [8] is a good starting point to become familiar with the ProtoGENI aggregate. General documentation on the GENI project and its available resources is found on the GENI wiki [2].

GENI resources are shared resources provided by members of the networking community. Please release your slivers when you are done with them, or if you are going to have to leave them for an extended period of time. Remember that in many cases you may be able to perform an experiment, copy the data off to another host, and release the sliver.

1.1 Objectives

The objective of this assignment is to familiarize you with the details of TCP congestion control, and the impact of network conditions on the TCP congestion control algorithms. You will specifically learn about:

- The NewReno [11] loss recovery and intertwined congestion control mechanisms
- The CUBIC [10] congestion control mechanism
- The impact of the `ssthresh` and `cwnd` TCP state variables on TCP network performance
- Measuring the performance of TCP flows
- The impact of path delay and bottleneck bandwidth on TCP performance
- TCP fairness concerns
- The Linux pluggable congestion control module interface

1.2 Tools

The two primary tools used in this assignment are:

1. **Traffic Control (tc)**

   The `tc` command is available in the GNU Linux distributions on ProtoGENI nodes, found in the `/sbin` directory. This command manipulates the Linux network forwarding tables, allowing for configuration of *queuing disciplines*, which change the policies controlling which packets are forwarded in what order and which are dropped; and *network emulation*, which allows the Linux kernel to emulate various network conditions such as delay or loss. These two effects are provided by the `qdisc` and `netem` subcommands, respectively.

   In these exercises, `tc` will be used to modify network conditions and enable different scheduling policies. Example command lines will be provided.

2. **Iperf [3]**

   Iperf is available on the ProtoGENI nodes, located at
   `/usr/local/etc/emulab/emulab-iperf`. Iperf is used to measure the bandwidth performance of Internet links. In these exercises, it is used to study the behavior of TCP in the face of changing link characteristics.

   Iperf runs as both a server and a client. The server is started with the `-s` command line option, and listens for connections from the client. The client is started with the `-c <server>` command line option, and connects to the server and sends data at either the fastest possible rate (given the underlying network) or a user-specified rate. The `-u` option causes the sender or receiver to use UDP instead of TCP. Various other options will be required for these exercises, and provided in the appropriate sections.

   All Iperf measurement data should be recorded from the TCP receiver (server) side.

2 Setting up the Experiment

After creating a slice using the ProtoGENI test scripts, create a sliver using the RSpec request in Figure 1. This RSpec is included in:
`http://www.cs.purdue.edu/homes/fahmy/geni/geni-tcp_exp.tar.gz`

The sliver created from this RSpec will contain a set of hosts in a “star” topology having 4 nodes each connected to a center node with 100 Mbps links, as shown in Figure 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<rspec xmlns="http://www.protogeni.net/resources/rspec/2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.protogeni.net/resources/rspec/2
    http://www.protogeni.net/resources/rspec/2/request.xsd"
    type="request" >
   <node client_id="center"
        exclusive="true">
       <sliver_type name="raw-pc">
       <disk_image name="urn:publicid:IDN+emulab.net+image+emulab-ops//FEDORA10-STD" />
        </sliver_type>
       <interface client_id="center:if0" />
       <interface client_id="center:if1" />
       <interface client_id="center:if2" />
       <interface client_id="center:if3" />
    </node>
   <node client_id="left"
        exclusive="true">
       <sliver_type name="raw-pc">
       <disk_image name="urn:publicid:IDN+emulab.net+image+emulab-ops//FEDORA10-STD" />
        </sliver_type>
       <interface client_id="left:if0" />
    </node>
   <node client_id="right"
        exclusive="true">
       <sliver_type name="raw-pc">
       <disk_image name="urn:publicid:IDN+emulab.net+image+emulab-ops//FEDORA10-STD" />
        </sliver_type>
       <interface client_id="right:if0" />
    </node>
   <link client_id="leftLink">
       <interface_ref client_id="left:if0" />
       <interface_ref client_id="center:if0" />
    </link>
   <link client_id="rightLink">
       <interface_ref client_id="right:if0" />
       <interface_ref client_id="center:if1" />
    </link>
   <node client_id="top"
        exclusive="true">
       <sliver_type name="raw-pc">
       <disk_image name="urn:publicid:IDN+emulab.net+image+emulab-ops//FEDORA10-STD" />
        </sliver_type>
       <interface client_id="top:if0" />
    </node>
   <node client_id="bottom"
        exclusive="true">
       <sliver_type name="raw-pc">
       <disk_image name="urn:publicid:IDN+emulab.net+image+emulab-ops//FEDORA10-STD" />
        </sliver_type>
       <interface client_id="bottom:if0" />
    </node>
   <link client_id="topLink">
       <interface_ref client_id="top:if0" />
       <interface_ref client_id="center:if2" />
    </link>
   <link client_id="bottomLink">
       <interface_ref client_id="bottom:if0" />
       <interface_ref client_id="center:if3" />
    </link>
</rspec>
```


Figure 1: RSpec request used in this assignment

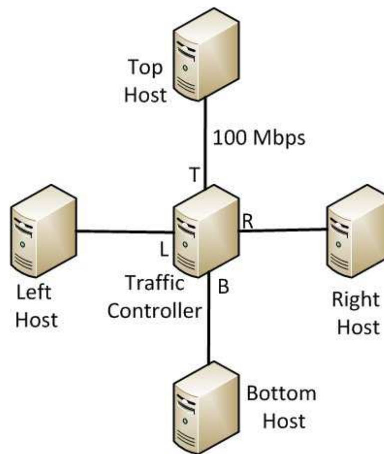

Figure 2: Star topology of ProtoGENI nodes

The nodes in the topology are named *Left*, *Center*, *Right*, *Top*, and *Bottom*. The Center node connects to the nodes Left, Right, Top, and Bottom via the interfaces *L*, *R*, *T*, and *B*, respectively. The network path properties between end nodes are tweaked by controlling the four interfaces *L*, *R*, *T* and *B* on the *Center* node. These interfaces are usually named from `eth1` to `eth4`. You can determine the physical interfaces that correspond to *L*, *R*, *T* and *B* by comparing the IP addresses of the interfaces on the *Center* node and the ping responses from the Leaf nodes to the *Center* node.

The parameters of the interfaces on the *Center* node will be modified to mimic different network path properties between the other hosts. Several TCP parameters and congestion control mechanisms will be varied at the end hosts to compare performance.

After the sliver is ready, you can begin the exercises.

3 Exercises

3.1 Comparison of Reno and CUBIC

ProtoGENI nodes provide two TCP congestion control algorithms, CUBIC and Reno, that can be chosen at run-time. The list of available algorithms are listed in the file `/proc/sys/net/ipv4/tcp_available_congestion_control`. The "Reno" congestion control provided by the Linux kernel is actually the NewReno [11]

algorithm, but we will refer to it as Reno here to be consistent with Linux terminology. Note that congestion control actions are very similar between Reno and NewReno, but NewReno has a more nuanced approach to loss recovery.

These congestion control algorithms can be chosen by placing the keywords `reno` or `cubic` in the file `/proc/sys/net/ipv4/tcp_congestion_control`. For example, to configure a host to use the Reno algorithm, use:

```
echo reno | sudo tee /proc/sys/net/ipv4/tcp_congestion_control
```

The `tc` command will then be used to set up network conditions for observation and testing. For example, if `eth1` is the physical interface representing the link *L* in Figure 2 on the *Center* node, the following command on the *Center* node will add a 200 ms delay to all packets leaving the interface:

```
sudo /sbin/tc qdisc add dev eth1 root handle 1:0 netem \
delay 200ms
```

Specific network setup commands will be provided as needed.

Run an Iperf server on the *Left* node. The Iperf client will be run on the *Right* node. The duration for an Iperf session (-t option) is 60 seconds unless otherwise mentioned. Note carefully that some exercises require a much longer duration. Ensure that your sliver lifetimes are long enough to capture the duration of your experiment. All of the experiments should be repeated at least a 5 times (especially when the interfaces include random delays or losses) to ensure confidence in the results, as transient conditions can cause significant variations in any individual run.

1. **Question:** What are the goodputs when the Reno and CUBIC algorithms are used on the network with no emulated delay or loss? Which is better?

2. **Question:** Qualitatively, under what conditions does BIC/CUBIC perform better than Reno's AIMD?

3. **Question:** Change the delay to of interface *L* to 300 ms using the following command, and run an Iperf session for 1800 seconds.
   ```
   sudo /sbin/tc qdisc add dev L root handle 1:0 netem \
   limit 1000000000 delay 300ms
   ```
   What are the goodputs of Reno and CUBIC? Which performed better? What do you conclude?

4. **Question:** Repeat the above experiment with 30 parallel connections and 1800 seconds for each algorithm by using the `-P 30` option on Iperf. How do CUBIC and Reno differ? What do you conclude?

5. **Question:** Remove the netem queueing discipline which causes delay and add a loss of 5% by using the following commands on the center node. Replace *L* with the appropriate physical interface. Alternatively, one can *change* a queueing discipline instead of deleting and adding a new one.
   ```
   sudo /sbin/tc qdisc del dev L root
   sudo /sbin/tc qdisc add dev L root handle 1:0 netem loss 5%
   ```
   How do the goodputs of Reno and CUBIC differ under loss for 60 s Iperf sessions?

3.2 Ensuring Fairness Among Flows

Restore the network state with the following command:
```
sudo /sbin/tc qdisc del dev L root
```
Run an Iperf client on the *Right* node with 10 parallel TCP connections (use the `-P` option), connecting to an Iperf server on the *Left* node for 60 seconds. Simultaneously, run a 20 Mbps UDP Iperf client on the *Top* node connecting to an UDP Iperf server session running on the Left node for 60 seconds.

1. **Question:** What are the throughput shown by the UDP and TCP Iperf server sessions? *Why* are they what they are?

2. **Question:** Provide the necessary steps and commands to enable queueing disciplines that enforce fairness among all the 11 flows in the network, and demonstrate that your solution is effective.

3.3 Reordering

Delete the previous queuing discipline and use the following `netem` configuration on interface *L* to create an 100 ms delay:
```
sudo /sbin/tc qdisc del dev L root
sudo /sbin/tc qdisc add dev L root handle 1:0 netem delay 100ms
```
As before, run a TCP Iperf client on the *Right* node connecting an Iperf server on the *Left* for 60 seconds.

1. **Question:** What is the TCP goodput?

2. **Question:** Introduce packet reordering, adding a 75 ms delay variance to

the interface *L* with the following command:
`sudo /sbin/tc qdisc change dev L root handle 1:0 \`
`netem delay 100ms 75ms`
What is the TCP goodput now?

3. **Question:** By tweaking the parameters in the file `/proc/sys/net/ipv4/tcp_reordering`, how much can the TCP goodput be improved? What is the best goodput you can show? Why is too high or two low value bad for TCP?

3.4 Performance of SACK under Lossy Conditions

Using Cubic as the congestion avoidance algorithm, set the loss characteristics on interface *L* using the following commands:
`sudo /sbin/tc qdisc del dev L root`
`sudo /sbin/tc qdisc add dev L root handle 1:0 netem loss 10%`

1. **Question:** What kind of goodput do you get using CUBIC with SACK (the default configuration)? Why do you see this performance?

2. **Question:** Disable SACK at the sender using this command:
   `echo 0 | sudo tee /proc/sys/net/ipv4/tcp_sack`
   What is the goodput without SACK? In what circumstances is SACK most beneficial? Remember that, due to the random nature of loss events, these experiments must be repeated at least five times to draw any conclusions.

3.5 An Experimental Congestion Avoidance module for Linux

In this exercise, you will develop and evaluate a TCP congestion control module for the Linux kernel. Linux provides a pluggable interface for TCP congestion control, which allows named congestion control modules to manipulate its sending rate and reaction to congestion events. You have already used the `reno` and `cubic` modules, and in this exercise you will create one named `exp`.

Linux kernel modules must be compiled against kernel source that matches the kernel into which the module will be loaded. In order to prepare your ProtoGENI host for kernel module development, follow these steps:

1. Comment out the line:
   `exclude=mkinitrd* kernel*`
   in the file `/etc/yum.conf`, to allow `yum` to install kernel headers.

2. Install the required packages with this command:

   ```
   sudo yum install kernel-devel kernel-headers
   ```

3. Fix up the kernel version in the installed headers to match the running kernel; this can be tricky, but these steps should handle it.

   (a) Find your kernel sources. They are in `/usr/src/kernel`, in a directory that depends on the installed version. As of the time this handout was created, that directory is `2.6.27.41-170.2.117.fc10.i686`. We will call this directory `$KERNELSRC`.

   (b) Identify your running kernel version by running `uname -r`. It will be something like `2.6.27.5-117.emulab1.fc10.i686`. The first three dotted components (`2.6.27`, in this case) are the *major*, *minor*, and *micro* versions, respectively, and the remainder of the version string (`.5-117.emulab.fc10.i686`) is the *extraversion*. Note the extraversion of your kernel.

   (c) In `$KERNELSRC/Makefile`, find the line beginning with `EXTRAVERSION`. Replace its value with the extraversion of your kernel.

   (d) Update the kernel header tree to this new version by running the command:

   ```
   sudo make include/linux/utsrelease.h
   ```

   More details to handle version issues are provided at [1].

A Makefile for compiling the module and the source for a stub TCP congestion control module are included in: `http://www.cs.purdue.edu/homes/fahmy/geni/geni-tcp_exp.tar`

The module is named `tcp_exp` (for experimental TCP), and the congestion control algorithm is named `exp`. Comments in the provided source file explain the relationship between the various functions, and more information can be found in [6].

The compiled module (which is built with `make` and called `tcp_exp.ko`) can be inserted into the kernel using `insmod`. It can be removed using the command `rmmod tcp_exp` and reloaded with insmod if changes are required.

Once the module is complete and loaded into the kernel, the algorithm implemented by the module can be selected in the same manner that `reno` and `cubic` were selected in previous exercises, by placing the keyword `exp` in `/proc/sys/net/ipv4/tcp_congestion_control`.

3.5.1 Algorithm Requirements

The experimental congestion control module is based on Reno, but has the following modifications:

- It uses a Slow Start exponential factor of 3. Reno uses 2.
- It cuts `ssthresh` to $3 \times \text{FlightSize}/4$ when entering loss recovery. Reno cuts to $\text{FlightSize}/2$.

3.5.2 Hints

These hints and suggestions may help you get started.

- The existing congestion avoidance modules are a good start. See `net/ipv4/tcp_cong.c` in the Linux source for the Linux Reno implementation.
- The file `net/ipv4/tcp_input.c` is a good place to learn how the congestion avoidance modules are used and invoked.
- RFC 5681 [9] specifies the Reno congestion control actions in detail, and may be helpful in understanding the kernel code.
- The Linux Cross Reference at `http://lxr.linux.no/linux` may be useful for navigating and understanding how the code fits together.
- If one of the hosts becomes unresponsive due to a bug in your congestion control module, you can restart the sliver to reboot it.
- The Linux Kernel Module Programming Guide [4] provides a good introduction to kernel module programming in general.

3.5.3 Evaluation

Once you have implemented the algorithm described above, answer the following questions:

1. **Question:** Discuss the impact of these algorithmic changes in the context of traditional Reno congestion control.

2. **Question:** Compare the convergence time and fairness of your algorithm with Reno and Cubic under (a) high delay (500 ms) and (2) high loss (5%) conditions. Use Jain's fairness index [12], or some other quantitative measure of fairness, in your comparison.

References

[1] Building modules for a precompiled kernel. http://tldp.org/LDP/lkmpg/2.6/html/x380.html.

[2] GENI wiki. http://groups.geni.net/geni.

[3] Iperf. http://iperf.sourceforge.net/.

[4] The Linux kernel module programming guide. http://tldp.org/LDP/lkmpg/2.6/html.

[5] Omni. http://trac.gpolab.bbn.com/gcf/wiki/Omni.

[6] Pluggable congestion avoidance modules. http://lwn.net/Articles/128681/.

[7] ProtoGENI test scripts. http://www.protogeni.net/trac/protogeni/wiki/TestScripts.

[8] ProtoGENI tutorial. http://www.protogeni.net/trac/protogeni/wiki/Tutorial.

[9] M. Allman, V. Paxson, and E. Blanton. TCP congestion control. RFC 5681, September 2009.

[10] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42:64–74, July 2008.

[11] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno modification to TCP's fast recovery algorithm. RFC 6582, April 2012.

[12] R. Jain, D. W. Chiu, and W. R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Equipment Corporation, September 1984.