CS 638 Lab 6: Transport Control Protocol (TCP)

Joe Chabarek and Paul Barford
University of Wisconsin –Madison
`jpchaba,pb@cs.wisc.edu`

The transport layer of the network protocol stack (layer 4) sits between applications (layers 5-7) and the network (layer 3 and below). The most basic capability that is enabled by transport is multiplexing the network between multiple applications that wish to communicate with remote hosts. Similar to other layers of the network protocol stack, transport protocols encapsulate packets with their own header before passing them down to layer 3 and decapsulate packets before passing them up to applications.

The most simple transport protocol is the User Datagram Protocol (UDP). UDP provides a multiplexing/demultiplexing capability for applications but not much more. Most significantly, UDP provides no guarantees for reliability, which is unacceptable for applications like a file transfer, Web or Email, in which people would very much like to get complete files. For this and other reasons, the Transport Control Protocol (TCP) is the most widely used transport protocol in the Internet today. Over 70% of the traffic on the Internet today is carried by TCP.

TCP is complicated, and there are actually many different versions of the protocol. There are literally thousands of academic research papers on TCP and TCP variants, many IETF RFC's on TCP, and entire books written about TCP's implementation and its behavior. TCP includes many features such as bi-directional connection management, ACK-based reliability, receiver flow control, and congestion control, all of which make it an attractive choice for a wide variety of applications. An easy way to think of what TCP does (beyond multiplexing/demultiplexing) is that it governs how and when packets are transmitted between a sending host and receiving host. As such it has a very big influence on the transmission performance as measured by either throughput (bytes transmitted over a specified time period) or response time (elapssed time from transmission start to transmission end). The flow of data between hosts is determined by *windows* which dictate how many packets can be *in flight* between sender and receiver at any point in time.

1 Overview and Objectives

Unlike prior labs, the focus of lab #6 is is on learning more about experimental tools and on observing the behavior of the various mechanisms that are part of TCP. The reason for this is because in moving from layer 3 to layer 4, we are moving away from the network per se, and into end hosts. Furthermore, network administrators usually don't spend a lot of time messing around with TCP since there is no programming or management interface to TCP on end hosts. The exception is content providers who may make tweaks in an attempt to get better performance on large file transfers.

In terms of experimental tools, a focus of this lab is on learning about traffic generation. Since the transport layer is concerned with facilitating the transmission of packets between two end hosts, hopefully, it is clear that we need tools for transmiting data in order to observe how TCP behaves. On one hand, we could simply have you generate traffic by using an application like the Web, and then tracking packet traffic generated by your browsing. However, it would be difficult to have you do enough browsing by hand to generate traffic sufficient to cause congestion on even low bandwidth links, which is essential in order to observe key aspects of TCP behavior.

Another tool that you will use in this lab is the Click Modular Router. Click is a utility that enables high performance personal computers to operate like a network device. Click offers a simple programming framework that you can implement many different kinds of new functions on top of a high performance packet processing system. This is a capability that is extremely useful for lab experiments since commodity routers and switches don't offer programming APIs.

Wireshark will be used to observe the details of TCP behavior. Since TCP governs the transmission of packets by end hosts, the focus of the lab configurations will be to generate different kinds of traffic that exercise different aspects of TCP's algorithms. In particular, you will be able to observe the details of TCP connection setup and tear down, flow control, reliability and congestion control. Before you proceed to the lab description, take a minute and think about how you might configure the lab in order to observe these chacteristics. We will experiment with two different versions of TCP in this lab: TCP-Reno and TCP-Vegas.

Finally, TCP is entirely implemented on end hosts. This means that you can examine how TCP is implemented (*e.g.,* in open-source software), make tweaks to existing implementations or even create your own implementation. While we will not be doing any TCP hacking in this lab, you may be interested in doing this at some point in the future.

Upon completion of the lab exercises below, students will be able to:

- *Understand the basic packet transmission behavior of TCP.*
- *Distinguish the differences between TCP-Reno and TCP-Vegas.*
- *Configure and operate simple traffic generators.*

2 TCP Variants

As mentioned above, there are many different variants of the Transmission Control Protocol (TCP). In most cases, the primary differentiating feature of a TCP variant is its congestion control mechanism. The reason for this is that the congestion control mechanism has a large impact on performance (defined in terms of either throughput or response time). Remember, the congestion control mechanism governs the size of CWND, which on paths with congestion will almost always determine the number of unacknowledged packets in flight (as opposed to RWND which is likely to govern on uncongested paths).

The two TCP variants that we are concerned with in this lab are TCP Reno and TCP Vegas. TCP Reno was one of the first of the Jacobson-based versions of TCP. Reno is considered an agressive version of TCP since it is always probing for additional available capacity until a loss event takes place, in which case it backs off (multiplicative decrease) and recovers the lost packets. The New-Reno and SACK versions of TCP improved the ability to quickly recover from losses of multiple packets, which is not uncommon. Collectively, all of these variants of TCP attempt to *control* congestion in the network.

An alternative approach is to attempt to *avoid* congestion all together. The idea is that if there is some signal that indicates that congestion is starting, then adjust the sending rate to keep it from overloading the path. So, the question is, how might a sender infer congestion other than by simply packet loss, which is what is used by TCP Reno? The answer is that when a link begins to be overloaded, packet will enqueue in the congested router's output buffers. The effect of this enqueuing is to lengthen the RTT of data/ACK pairs (since packets now have to spend time in the queue before being transmitted). The direct implication of lengthened RTT values is a *decrease in end to end throughput* since fewer packets will be sent over a given period of time. The basis for the Vegas version of TCP is that if a decrease in throughput is measured, then send rate should be adjusted to stay just below the rate that led to the decrease. This approach to managing congestion is called *congestion avoidance* since it does not rely on packet loss to adjust send rate.

2.1 TCP Reno

TCP Reno has all the required elements of a reliable, connection oriented, full duplex protocol which implements flow control and conjestion control. For flow control, a sliding window algorithm is used to ensure that the sender does not overrun the receiver. For conjestion control TCP Reno uses Jacobson/Karls to compute the estimated round trip time and the RTO. It also included slow start, additive increase/multiplicative decrease and the fast retransmit/fast recovery mechanisms for managing CWND.

TCP Reno is the default TCP protocol variant used in the FC6-STD image, so you don't have to do anything special to use Reno in your experiments. You will, however, need a mechanism for generating traffic that will then utilize the version of TCP Reno running on the Schooner hosts. Note that traffic is generated by applications (or in our case traffic generation utilities); those used in this lab are described below.

2.2 TCP Vegas

Like Reno, TCP Vegas has all the required elements of a reliable, connection oriented, full duplex protocol which implements flow control and congestion management. However, instead of reacting to congestion in an after-the-fact fashion like Reno (*i.e.,* when loss takes place), Vegas attempts to avoid congestion through careful measurement and management of send rate. While the ideas behind Vegas are interesting and certainly make sense, it can be the case that Vegas is unsuccessful at avoiding congestion and packet loss takes place. In this case, Vegas' default behavior in terms of loss recovery and window management is identical to TCP Reno.

TCP Vegas along with a few other major TCP variants are distributed with the linux kernel. Modern 2.6 kernels implement a modular framework which makes changing from TCP Reno to TCP Vegas is relatively painless. See the commands below on how to find the current TCP algorithm in use and to change beteen Reno and Vegas for your experiments.

There are a number of differences between Reno and Vegas (See PreLab readings) that you will be measuring. **NOTE:** It may be easier to complete all of the tasks/experiments with Reno described below, then change to Vegas to conduct the remainder of the tasks/experiments.

```
sh-3.1# sysctl net.ipv4.tcp_congestion_control
net.ipv4.tcp_congestion_control = reno
sh-3.1# sysctl -w net.ipv4.tcp_congestion_control=vegas
net.ipv4.tcp_congestion_control = vegas
sh-3.1# sysctl net.ipv4.tcp_congestion_control
net.ipv4.tcp_congestion_control = vegas
sh-3.1# sysctl -w net.ipv4.tcp_congestion_control=reno
net.ipv4.tcp_congestion_control = reno
```

3 Experimental Tools

In this section, brief overviews of the tools that will be used in this lab are provided. More details can be found in the readings specified in the Pre Lab.

3.1 Measuring and Observing Packet Traffic

In a laboratory test environment, one of the major goals while running experiments is to observe traffic without disturbing the performance of the system. To this end we describe a standard technique for measuring packet traffic. Remember, however, that another critical task in conducting experiments with network traffic is to identify useful vantage points in the network *i.e., where* you should place your instrumentation for measuring traffic.

Wireshark, a tool that you have used in prior experiments, is one of a number of applications that uses a system library named `libpcap` to capture packets. The library efficiently creates copies packets as they are received from the network card (set in promiscuous mode) and sends them to user programs. Perhaps the most well know application that uses *libpcap* as a mechanism for creating a stream of packet copies is Tcpdump (which is utility available on most systems).

Wireshark is a popular tool because of its graphical user interface and easy to use features. For our exercises, we will be using a graphing module to observe the sequences of packets generated by TCP variants. To use the graphing module, find the “TCP Stream Graph” entry in the “Statistics” menu of the Wireshark GUI. Try each of the graph types and think about what they are showing you. You can also use a Wireshark display filter to limit the traffic you are graphing. You are encouraged to play around with the Wireshark GUI to gain familiarity with this observation technique. Be sure to ask questions early if you need help.

3.2 Generating Traffic

In order to examine the behavior of the TCP Reno and Vegas protocols we must be able to generate packet traffic. This is normally done by users accessing application such as Email or the Web that result in files being transfered between a client and a server. However, having real users accessing real applications in lab experiments is cumbersome and does not lend itself to repeatability or to experiments that demand large traffic loads (*e.g.,* for experiments on how TCP behaves under congested conditions) since hundreds or thousands of users who be required!

You will be using two tools for generating traffic in this lab. The first is `ttcp`, which is a simple tool that generates a single TCP or UDP stream. We will use `ttcp` with Wireshark to gather statistics about different implementations of TCP. However, `ttcp` does not address our need to recreate traffic that is a composite of perhaps thousands of users. For this purpose, the *Harpoon* traffic generator will be used. Harpoon is a sophisitcated tool which we will use in our experiments to generate a sufficient amount of representative “background” traffic that to create congestion on a link.

3.2.1 ttcp

Ttcp is a simple tool that starts a TCP or UDP flow between two systems and reports some basic statistics about the flow. A few other features are also available in the utility – see the man page for these additional options. To initiate a transfer, you need a receiver and a transmitter. The receiever is started on one machine with **ttcp -s -r** on the transmitting machine the command is **ttcp -s -t IPAddress**. **NOTE:** use this utility to generate the streams that you will visualize in Wireshark for your experiments.

3.2.2 Harpoon

Harpoon is a sophisticated tool which can mimic the type and intensity of traffic on a network. At its core, Harpoon is a tool that runs on a two systems - clients and servers. The clients determine how and when to request file transfers. The servers send data to the clients based on what has been requested. An easy way to think about this is as an automated Web environment. The important aspects of Harpoon are that the client request streams can be automatically configured to generate traffic that it identical to what has been observed in an actual network.

The Harpoon software distribution comes with an extensive user manual that describes installation and configuration procedures. A link to this manual is provided in the Pre-Lab. Harpoon is available on the standard FC6-STD Operating System image that you will use for this lab. So, you will only need to familiarize yourself with how to configure and run Harpoon in your experiments.

NOTE: In the interest of time, Harpoon will not be used for the required tasks of this lab. However, you are welcome and encouraged to install it and use it in your experiments if you are interested.

3.3 Simulating Adverse Network Conditions

The Initial experiments that you will run will primarily be concerned with observing TCP in ideal conditions *i.e.,* with no congestion on the path between clients and servers. However, we are also interested in how the two different variants of TCP behave when a link is congested and packet loss occurs. Normally, to create congestion, the aggregate demand on a set of input links must exceed the capacity of an output link. For example, if you are using a router in an experiment with 1 Gbps links, then the input on some set of input links (where the set must be at least size = 2), must be greater than 1 Gbps. In real life, it takes *a lot* of clients to generate 1 Gbps of traffic. For example, the average traffic rates for the entire UW-Madison campus is usually only several hundred Mbps! The good news is that we have several options in Schooner in terms of how we experiment with congestion.

Using Schooner, we actually have the ability to specify different link characteristics. Three characteristics that we are particularyly interested in are *bandwidth*, *propagation delay* and *packet loss rate*. There is a popular BSD-based operating system utility called `dummynet`, which is used to augment traffic flows with different characteristics. By specifying a bandwidth, `dummynet` can enable an output link that has 1 Gbps native capacity to behave as if it has *e.g.,* 1 Mbps. The implication is that with `dummynet`, we can create a test environment in which it is much easier to create congested conditions. Alternatively, we can use dummynet to simply specify a loss rate or RTT on a link. In this case, we need not even bother with creating an input load that exceeds an output link's capacity in order to get a sending host to react to congestion. However, this approach is not nearly as realistic and will result in behavior that is not very similar to what would be seen in real networks. `dummynet` is available by default in Schooner through the standard network configuration GUI, which by clicking on a link, enables you to specify link characteristics including bandwidth capacity and loss rate.

When you set any sort of traffic shaping component (delay, loss, possibly bandwitdh) Schooner interposes a hidden node on the link running dummynet. At experiment swapin the traffic shaping node will use the parameters in your NS file. During experimentation you can use the "Modify Traffic Shaping" link on the experiment webpage which will allow you to dynamically change the shaping parameters of your experiment.

3.4 The CLICK Modular Router

CLICK is a utility and programming framework that enables you to configure a PC to behave as a high performance packet processing system. The basic primitive in Click is called an *element*. A set of elements can be composed in Click to create a packet processing device. Many different kinds of devices can be created including routers, switches, firewalls, intrusion detection systems, etc. Click is easy to use and has a growing library of preconfigured elements.

A Click of elements are composed in a configuration file that is read by the Click engine. There will be three different configurations used in the experiments that enable you to create different conditions for your experiments. The first configuration is a standard router with drop tail queues. The second configuration implements a RED queue. The third configuration delays each packet for 50us with a drop tail queue. In each configuration file you will need to change the strings PC1_IP, PC2_IP, PC3_IP, ... PC8_IP, MAC1, MAC2, MAC3, MAC4, MAC5, MAC6, MAC7, MAC8 to the appropriate values.

NOTE: In the interest of time, Click will not be used for the required tasks of this lab. However, you are welcome and encouraged to install it and use it in your experiments if you are interested.

4 Tasks

Create Topology #1 described in the Pre Lab and determine where to gather measurements for your tests. **NOTE: In the interest of time, topology #2 will NOT be used in this lab.** This topology will be used in all of the experiments described below. Load the standard FC6-STD disk image and include the RPM for ttcp along with the list of RPMs you have been using for Wireshark**NOTE:** A lot of data will be generated in this lab so make sure that you have a method for organizing the data before you actually gather it so you don't mix up traces from different experiments.

4.1 Baseline Performance Characteristics

We will begin by examining throughput and RTT's, and taking a look at sequence numbers in packets to get an idea of how TCP Reno and TCP Vegas each perform. Transfer three different sized files of your own choosing (from about 100Bytes to 1MB) with `ttcp` from PC1 to PC2, and generate graphs for the statistics that we listed above. Wireshark should enable you to do this. Report the results of the experiments in your lab notebook. Detailed tasks include:

1. Using TCP Reno, `ttcp` and Wireshark, transfer each of the three files from PC1 to PC2 and record the sequence number versus time, for throughput, and RTT for the connection.
2. Using TCP Vegas, `ttcp` and Wireshark, transfer each of the three files (same sizes as before) from PC1 to PC2 and record the sequence number versus time, throughput, and RTT for the connection.

These measurements and statistics will serve as a baseline for performance in an uncongested environment.

5 Performance in Congested Conditions

Next, run a set of tests using the same file sizes as above, this time using Schooner to specify a loss rate and or a propagation delays on the link between sender and receiver. Report of each experiment in your lab notebook. Detailed tasks include:

1. Using TCP Reno, `ttcp` and Wireshark, transfer each of the three files from PC1 to PC2 configured with a loss probability of 0.1 on the link, and record the sequence number versus time, throughput, and RTT for the connection.
2. Using TCP Reno, `ttcp` and Wireshark, transfer each of the three files from PC1 to PC2 configured with a loss probability of 0.1 and a RTT delay of 30ms on the link, and record the sequence number versus time, throughput, and RTT for the connection.
3. Using TCP Vegas, `ttcp` and Wireshark, transfer each of the three files from PC1 to PC2 configured with a loss probability of 0.1 on the link, and record the sequence number versus time, throughput, and RTT for the connection.
4. Using TCP Vegas, `ttcp` and Wireshark, transfer each of the three files from PC1 to PC2 configured with a loss probability of 0.1 and a RTT delay of 30ms on the link, and record the sequence number versus time, throughput, and RTT for the connection.