Name 1: _____ Group number: _____
Name 2: _____

COMPUTER NETWORKING
LAB EXERCISES (TP) 4
TCP AND TRAFFIC SHAPING

September 14, 2009

Abstract

TCP is the most important transport layer protocol of the Internet. Its performance is greatly influenced by its congestion control mechanism. In this lab session, you will run one or more TCP sessions between your workstation and a server and you will study the impact of congestion control on the performance of TCP. In a second step, you will learn how the TCP source code of your workstation's operating system can be modified in order to implement a more aggressive congestion control policy.

Further, you will get the chance to play around with *Traffic Shaping*. Traffic Shaping is vital for an ISP, to ensure that his customers are getting the service (i.e., bandwidth) they paid for, neither more nor less. You will apply traffic control policies at your router and study their effect on user traffic.

1 ORGANIZATION OF THE LAB EXERCISE

Two groups of students will work together. Each group works at one work space, representing one Internet Service Provider (ISP). On every work space, one workstation PC models a customer network of that ISP.

1.1 TP REPORT

The report is due on **December 17** (before the lecture) in the lecture room (INM 202). During the TP, fill in your answers directly in the spaces provided in this document. That will be your TP report (one per group). Don't forget to write your names and group number on the first page of the report.

2 PREPARING YOUR WORKSPACE

Reboot both machines into the lab4 default kernel. That is, execute the command `reboot` and then select the proper entry when the boot menu appears.

2.1 REMOVE OLD CONFIGURATIONS

First, you need to reset the network configuration to the default one and make sure that there are no stale routing processes running from the previous TP. Load the lab1 default configuration, and kill all routing processes that might be running on the machines (both router and workstation):

```
# killall zebra ripd ospfd bgpd
```

2.2 UNPACK THE SCRIPTS

On both the workstation PC *and* the router PC, there should be a file `tcp.tgz` in the directory `/root/lab4`. Uncompress it.

```
# cd /root/lab4
# tar xvfz tcp.tgz
```

This will create a directory `tcp/` with the source code for TCP sender and receiver.

```
# ls tcp
Makefile tcpreceiver.c tcpsender.c
```

3 HOW DOES THE TCP CODE WE ARE GOING TO USE LOOK LIKE?

For the remainder of the document we adopt the following notation:

- $cwnd$ refers to the congestion window as it is introduced in the lecture notes
- $twnd$ refers to the target window as it is introduced in the lecture notes

3.1 TCPSENDER AND TCPRECEIVER: TWO APPLICATIONS COMMUNICATING VIA TCP

Throughout this lab you are going to use two application layer programs that communicate via TCP. Both of these programs are very simple: `tcpsender` generates TCP traffic and sends it to `tcpreceiver` who receives it.

The source code of the `tcpreceiver` can be found in the file `tcpreceiver.c`. You can have a quick look at it, but it is not that important how it works exactly. Essentially, `tcpreceiver` listens on some predefined port for TCP connections. If a client connects, a child process who handles the connection is forked off. Whenever the child process receives data, it is written to a buffer. Before you can use `tcpreceiver` and `tcpsender` for the first time, you have to compile them. For this purpose we already created a Makefile, so you can compile by just going to your `tcp` directory and typing $make$:

```
# cd tcp
# make
```

You can start a `tcpreceiver` listening on port number `Port` by issuing the following command in your `tcp` directory:

```
# ./tcpreceiver Port
```

The source code of the `tcpsender` can be found in the file `tcpsender.c`. Unlike for the receiver, you should more or less understand what the sender does. The syntax for starting a `tcpsender` is the following:

```
# ./tcpsender IPAddress Port [IncFactor]
```

Here `IPAddress` denotes the IP address of the receiver you want to connect to and `Port` is the port number the receiver is listening to. There is also an optional argument `IncFactor` which determines the increase factor to use in the congestion avoidance phase. As you have learned in the lecture, the standard TCP implementation uses an increase factor of one *i.e.*, during congestion avoidance the $cwnd$ is increased by *one* for every full $cwnd$ received. The Linux kernel that you booted into has a change in the TCP code in order to also accept increase factors other than one. `tcpsender` communicates the increase factor to be used for the socket it creates to the Linux kernel by means of a so-called socket option. You can see this on the following line in the code of `tcpsender`:

```
setsockopt(sockfd, SOL_TCP, TCP_CWNDINC, (void *) &inc_factor,
           sizeof(int));
```

This tells the kernel that it should set the TCP socket option number 23 (note that `TCP_CWNDINC` equals 23) to the value `inc_factor` for this socket. Don't worry, you do not have to know exactly how socket options are implemented (in fact we already added the option 23 to the kernel you are using).

What you however should know later when you will be asked to modify the Linux TCP code is that the socket option we implemented for you adds a field having the value of `inc_factor` and called `snd_cwnd_inc_factor` to the `struct tcp_sock`. Like this, the increase factor set by the `tcpsender` is made available in the TCP code of the Linux kernel. In other words, assuming that in the TCP code of the kernel you have a variable `tp` which is a pointer to the current socket, then `tp->snd_cwnd_inc_factor` contains the value of `inc_factor` that corresponds to the one you entered as an argument to `tcpsender`. For the remainder of the lab those of you who are unfamiliar with C and/or have forgotten everything about it, can assume for the sake of simplicity that `tp` is an object representing a socket and that with `tp->snd_cwnd_inc_factor` you access a member variable of this object called `snd_cwnd_inc_factor` (in Java you would write something like `tp.snd_cwnd_inc_factor`).

Please note that this socket option is not part of the standard Linux kernel. It is an option we implemented solely for this lab. The parameter `IncFactor` will thus only play a role when you use the modified kernel.

If this now sounds all strange and confusing, just go on and come back here when you are asked to do the modifications to the Linux TCP implementation.

3.2 A First Look at the Linux TCP Code

The file `net/ipv4/tcp_cong.c` in a Linux kernel source tree contains the Reno implementation of TCP congestion control. You do not however have to study the whole file. The most important function for us is `tcp_reno_cong_avoid`. This function is called whenever a non-duplicate ack is received and it determines

by how much to increase the congestion window in this case. It thus implements the "increase" part of the TCP slow start and congestion avoidance phases. You should try to understand what this function does before going on with the lab. Here are some hints to get you started:

- `tp->snd_cwnd` is the congestion window (`cwnd`)
- `in_flight` is a speciality of the Linux TCP implementation. For the sake of simplicity you can assume for this lab that it is equal to the number of sent but not yet acknowledged packets. The first `if` clause in the code below consequently makes sure that $cwnd$ is not increased if it is not fully used.

And here is a slightly simplified version of the function:

```
/*
 * TCP Reno congestion control
 * This is special case used for fallback as well.
 */
/* This is Jacobson's slow start and congestion avoidance.
 * SIGCOMM '88, p. 328.
 */
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 rtt, u32
in_flight, int flag) {
    struct tcp_sock *tp = tcp_sk(sk);

    if (in_flight < tp->snd_cwnd)
        return;

    if (tp->snd_cwnd <= tp->snd_ssthresh) {
          /* In "safe" area, increase. */
          if (tp->snd_cwnd < tp->snd_cwnd_clamp)
              tp->snd_cwnd++;
    } else {
        /* In dangerous area, increase slowly.
         * In theory this is tp->snd_cwnd += 1 / tp->snd_cwnd
         */
        if (tp->snd_cwnd_cnt >= tp->snd_cwnd) {
            if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                tp->snd_cwnd++;
            tp->snd_cwnd_cnt = 0;
        } else
            tp->snd_cwnd_cnt++;
    }
}
```

What did the coder mean by "safe" area? What did he mean by "dangerous" area?

The target window ($twnd$) never explicitly appears in the code. Which variable plays the role of $twnd$?

What is the role of the variable `tp->snd_cwnd_clamp`?

What is the role of the variable `tp->snd_cwnd_cnt`?

3.2.1 TCP WINDOW SCALE OPTION

Another subtlety of the Linux TCP implementation is the TCP Window Scale Option. It is defined in RFC1323 (TCP Extensions for High Performance) and implemented in the Linux version we are using in this lab. It is thus likely that you will encounter situations in this lab where the TCP Window Scale Option comes into play.

The TCP header uses a 16 bit field to report the receive window size to the sender. Therefore, the largest window that can be used is $2^{16} = 65K$ bytes. To circumvent this problem, RFC1323 defines a new TCP option. This option defines an implicit scale factor, which is used to multiply the window size value found in a TCP header to obtain the true window size.
The scale factor is carried in a new TCP option, Window Scale. This option is sent *only* in a SYN segment (a segment with the SYN bit on), hence the window scale is fixed in each direction when a connection is opened.

4 CONGESTION CONTROL EXPERIMENTS

In order to create TCP connections (flows) and run different experiments related to congestion control, we will use the network setup as shown in Figure 1. In all the experiments, a group of two students using one Workstation PC and one Router PC will have to synchronize and collaborate with exactly one other group. In Figure 1 and in the rest of the document, X denotes your group and Y denotes the group you collaborate with.

4.1 EXPERIMENT 1: ONE TCP FLOW OVER TWO HOPS (OVER ONE ROUTER)

4.1.1 SETUP THE NETWORK AND GENERATE TCP TRAFFIC

Setup the network as shown and explained in Figure 1. Try to ping both PCs of Y in order to make sure both you and group Y set everything correct.

To be sure TCP Reno congestion control is used, execute the following command on **both Router and Workstation PCs**:

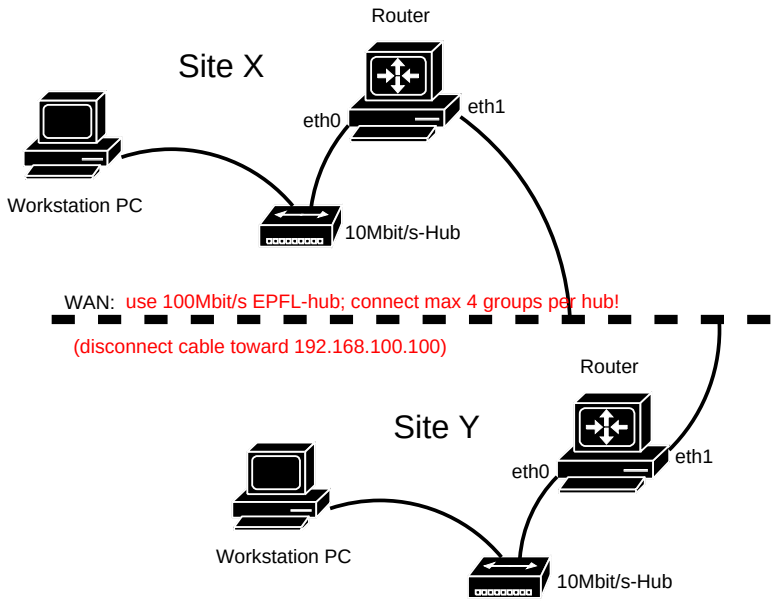

Figure 1: *Initial network setup for the congestion control experiments*: Use prefix 192.168.X.0/24 for your local LAN, and prefix 192.168.100.0/24 for the LAN interconnecting the Router PCs. To interconnect Workstation and Router PCs of your workplace, use a 10Mb/s hub. To interconnect the Router PCs use an ”EPFL” hub. Make sure that used the ”EPFL” hub has a speed of 100 Mbit/s (and not 10Mbit/s), and also make sure that not more than 4 groups connect over one ”EPFL” hub. Do not forget to activate forwarding on the Router PCs (sysctl -w net.ipv4.ip_forward=1).

```
# echo reno > /proc/sys/net/ipv4/tcp_congestion_control
```

Use the previously introduced sender and receiver application programs to create a TCP flow F1 from your Workstation to the Router of group Y as shown in Figure 2. Group Y should do the same from their Workstation toward your Router (flow F2). Use port number 20202 to send to and receive on, and use the normal increase factor 1 when starting the tcpsender program. Make sure the other group has started their receiver before you start your sender.

4.1.2 FIND WHAT IS GOING ON

How many bytes does the tcpsender program write into the socket upon each *write()* (look at the tcpsender.c file)? How many bytes of TCP data is sent in each TCP segment (in our experiment, i.e with the hardware and software you use)? Explain the difference between these two numbers.

What is the size (in bytes) of the MTU used by the interfaces? What is the size (in bytes) of Ethernet frames that are used to send the data? Explain the difference of these two numbers. What is then the maximum

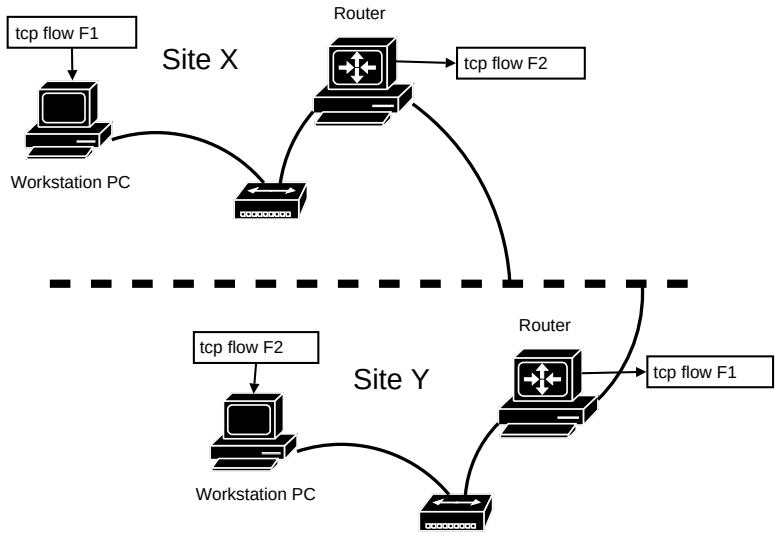

Figure 2: Traffic setup for the experiment 1.

theoretical application data throughput a flow could achieve? Explain how you compute.

How can you calculate the upper limit on the rate at which TCP is sending data from the average *cwnd*, the RTT, and the packet (data part) size? Compare the calculated value to the one in previous question.

Note: The achieved throughput printed by the tcpsender program is something smaller than the above computed values (probably because the tcp connection doesn't manage to fully utilize cwnd all the time).

Explain what limits the value of *cwnd* in this experiment? Refer to appropriate lines of code and observed parameter values that matter for the explanation. Show the computation that confirms your answer, and explain your conclusion. (Hint: think about sending window, cwnd, offered window, scale option(mentioned in the introduction), and in_flight variable (also mentioned in the introduction).)

Where is the bottleneck in the system regarding the data throughput that can be achieved by flow F1? Does F1 (when in stationary regime, i.e. look after some time upon starting tcpsender) experience losses because of this bottleneck or not and why?

In this system the main component of the RTT comes from the queuing of the packets.

In this experiment all the queuing probably happens at the sender's outgoing interface buffer. Why? Then what constrains the size of this queue, and how you can compute the component of the RTT that come from this queue? Show your computation.

4.2 Experiment 2: Two TCP flows over two hops (over one router)

Setup the network as in the Experiment 1 (Figure 1).

To be sure TCP Reno congestion control is used, execute the following command on **both Router and Workstation PCs**:

```
# echo reno > /proc/sys/net/ipv4/tcp_congestion_control
```

Start TCP flows as in Figure 3 (still we use standard TCP/IP congestion control).

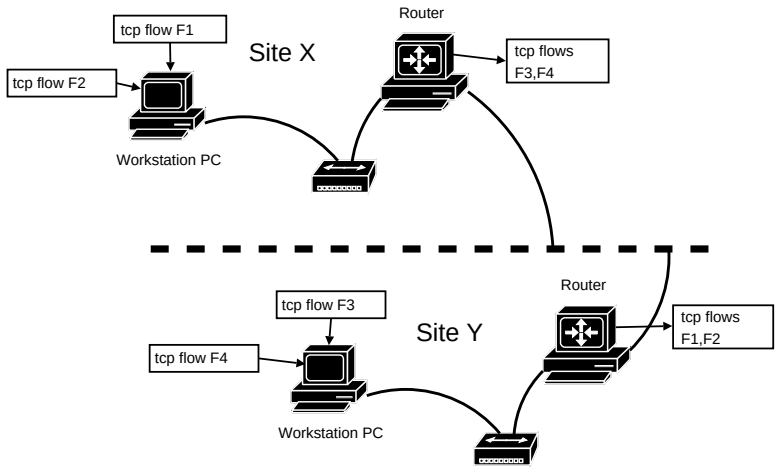

Figure 3: Traffic setup for the experiment 2.

Explain the throughput experienced by F1 (also give the value). What happens with cwnd and RTT of F1 compared to Experiment 1 and why?

4.3 Experiment 3: Aggressive TCP flow versus normal TCP flow over two links (over one router)

In this Experiment we want you to create TCP flows that in the congestion avoidance phase increase cwnd not by 1 per RTT as usual, but by `IncFactor`, where you can control the `IncFactor` from applications using these TCP flows.

4.3.1 Creating your own Linux with aggressive TCP and additional socket option to control a TCP flow from the application

Put yourselves in the following position: You want to change the function *tcp_reno_cong_avoid* such that the TCP flow in the congestion avoidance phase increases cwnd not by 1 per RTT as usual, but by the `IncFactor` parameter of the tcpsender program that will use that flow.

Explain which changes you would do and why. Give the appropriate lines of the code you would change.

4.3.2 Setup the network and generate TCP traffic

Setup the network as in the Experiment 1 (Figure 1).

To be sure TCP Reno congestion control is used, execute the following command on the rebooted machine:

```
# echo reno > /proc/sys/net/ipv4/tcp_congestion_control
```

Start TCP flows as in Figure 3, and do it 3 times for 3 different values of the cwnd increase factor used by flow F2 (F4): a) 1, b) 4, c) 10. F1 (F3) should each time use standard cwnd increase factor (equal to 1).

4.3.3 Find what is going on

In each case wait until the flows stabilize. You should observe the same values for cwnd, RTT, and achieved throughput as in the Experiment 2, for both flows, and both flows get the same throughput though F2 now uses bigger increase factor.

Explain why the more aggressive flow does not get more share of the throughput.

5 CONGESTION CONTROL EXPERIMENTS IN SCENARIOS WITH TRAFFIC SHAPING

Traffic shaping allows a network operator to enforce policies concerning the traffic he forwards. For instance, an ISP that sells Internet connectivity with a capacity of 1Mbit/s to a customer will restrict the customer's traffic to this rate even though the underlying hardware might be able to support a much higher rate. More or less sophisticated tools exist for the common routing platforms to support such policing (FreeBSD: dummynet, Linux: tc, Cisco: Committed Access Rate (CAR), Juniper: Policer, ...).

5.1 A VERY SHORT INTRODUCTION TO LINUX TRAFFIC CONTROL (TC)

The Linux `tc` command allows you to add various types of queues and policies to an interface. Queues or queuing disciplines are usually used to control outgoing packets whereas controlling incoming traffic is called policing. Policing is mainly used to limit incoming traffic to a certain rate. In this lab session we concentrate on queuing disciplines *i.e.*, on shaping outgoing traffic.

Further, we are only going to use one of the many available queuing disciplines under Linux, the Token Bucket Filter (TBF) (Those of you interested in the topic, and wanting to know more about the options you have with Linux, are referred to the "Linux Advanced Traffic Control & Routing HOWTO", in particular section 9 on queuing disciplines and sections 14, and 15 on more advanced filters and examples. The document can be found at `http://www.lartc.org/lartc.pdf`).

5.1.1 THE TOKEN BUCKET FILTER (TBF)

The TBF is a filter whose main purpose is to limit the rate of outgoing packets. It is always associated with an interface, so it limits the rate of packets going out through the associated interface.

In the following we explain the main idea of the TBF. To every TBF is associated a bucket with a certain size measured in bytes. This bucket constantly fills up with tokens at a certain rate. Every token represents permission to send one byte. The number of tokens the bucket can hold is limited by its size, tokens arriving when the bucket is full are lost. Additionally, a TBF also has an associated queue of a certain size in bytes. Now, if an outgoing packet of size s bytes arrives at the TBF, the latter checks whether the bucket contains at least s tokens. If this is the case, s tokens are removed from the bucket and the packet is sent. If there are not enough tokens, the packet is queued if there is enough space for it in the queue, otherwise it is dropped.

A TBF thus has three parameters that are important to us: the rate at which the bucket is filled with tokens, the size of the bucket and the size of the queue.

The syntax to install a TBF is the following:

```
# tc qdisc add dev IF root tbf limit LIM_B burst BURST_B rate RATEkbit
```

where IF denotes the interface to which you want to attach the TBF, $RATE$ denotes the rate at which the bucket fills with tokens and is given in kbit/s, $BURST\_B$ denotes the size of the bucket in bytes, and finally $LIM\_B = BURST\_B + queuesize$ which is also given in bytes. Thus $LIM\_B = BURST\_B$ equals a scenario where there is no queue. Values of LIM_B smaller than $BURST\_B$ default to $BURST\_B$. So, as an example

```
# tc qdisc add dev eth1 root tbf limit 20000 burst 4000 rate 1000kbit
```

associates a TBF with interface eth0, having a bucket of size 4kB, a queue of size 16kB and tokens are generated at a rate of 1Mb/s.

Assume packets arrive at the TBF with an average rate of 10Mb/s. Assume that the TBF is the one in the example above (*i.e.*, rate = 1Mb/s, burst = 4kB, limit = 20kB). What is the average rate of the outgoing packets and why?

Assume packets of size 1500B arriving at a TBF with an average rate of 10Mb/s. Further assume that the TBF has the parameters $rate = 1.5Mb/s$, $burst = 1kB$, $limit = 15kB$. What happens to these packets? What is the average rate of the outgoing packets?

To show what queuing disciplines are installed you can use the command

```
# tc qdisc show
```

To remove a previously installed TBF you can use the command

```
# tc qdisc del dev IF root tbf
```

where you replace IF by the interface the TBF is associated to.

To change the parameters of a previously installed TBF you can use the command

```
# tc qdisc replace dev IF root tbf limit LIM_B burst BURST_B rate
RATEkbit
```

where the parameters are the same as when you install a TBF.

Be careful: it turns out that replacing a TBF usually works fine if you change $burst$ or $rate$. However, if you change the parameter $limit$, chances are the change will not be made properly. Therefore, rather remove a previously installed TBF and add a new one instead of using $replace$, especially if you are playing around with the $limit$ parameter!

5.2 Experiment 4: Aggressive TCP flow versus normal TCP flow(s) over two hops, but over the router with constrained rate and queue size

Setup the network as in the Experiment 1 (Figure 1).

To be sure TCP Reno congestion control is used, execute the following command on the rebooted machine:

```
# echo reno > /proc/sys/net/ipv4/tcp_congestion_control
```

Install the queuing discipline of interface eth1 of your Router PC with a TBF with rate parameter 3000kbit, burst parameter 40000, and limit parameter 40000.

WARNING: If the filter was already installed, replace it by **deleting it, then installing it again!**.

Then go back to your workstation PC and stop ongoing senders if any. Then start two TCP senders, one with the default increase behavior and one with the increase factor of 4.

In theory, an increase factor four times larger than normal should give you a throughput that is 4 times larger as well. However, we will see that in practice this is not the case in our small setting with only very few competing flows.

What is the size of the congestion window and the RTT of the two connections? What average rate do the flows achieve? Explain the values of observed `cwnd`-s? Also explain the values of RTTs and why F2 throughput is larger than F1 throughput.

Give an explanation why in our setting, the more aggressive flow does not achieve four times the throughput of the normal TCP flow.

Open two new terminal windows and start a TCP flow with the normal increase behavior of 1 packet / RTT in each of them. You should now have 3 normal TCP flows and 1 more aggressive TCP flow running. Also change the rate to 5000kbit/s, the burst to 60kbytes, and the limit to 60 kbytes so that the $cwnd$ and RTT values for the flows remain roughly the same. Wait a bit until the sending rates of the TCP connections have stabilized at their new values.

What rates are achieved by the normal TCP flows compared to the more aggressive TCP flow?

Do the four TCP flows share the bottleneck capacity differently now, i.e., is the throughput ratio aggressive_flow/normal_flow larger/smaller than before

Replace the aggressive TCP flow with an even more aggressive TCP flow with an increase factor of 20. You should now have 3 TCP flows with normal congestion control and 1 TCP flow with an AIMD increase of 20 running.

What rates are achieved by each flow? Why the more aggressive flow does not get 20/23 of the total throughput?

Replace the aggressive TCP flow by one with an increase factor of 120.

What rates are achieved and why?

6 CLEAN-UP YOUR WORK SPACE

Reboot the PCs by typing:

```
# reboot
```

Unplug the 10baseT cables and disconnect the power plug of your hub. Leave the long cable on the desk and connect the router and workstation PC to the hub located between the machines (i.e., leave the workplace in exactly the condition you found it).