Department of Computer Science, Box 8206
North Carolina State University, Raleigh, NC 27695

Networking Practicum

(an excerpt, adapted for use at the CAIDA 2001 Internet Technology Laboratory Workshop)

Developed by

Rudra Dutta (rdutta@csc.ncsu.edu)

ITL Demo Team

John Streck (jps@ncstate.net)
Mladen A. Vouk (vouk@csc.ncsu.edu)
Marhn Fullmer (mhfullme@eos.ncsu.edu)
Andrew Barnes (ajbarnes@eos.ncsu.edu)

12-15 June 2001

Table of Contents

SmartBits[1] Exercises – Lab Session Three

3.1. General Remarks

Note: *The exercises in this ITL document were developed for the NC State University Network Engineering Degree hands-on practicum. This practicum accompanies its introductory graduate level networking courses. The set presented at ITL is an adaptation intended to illustrate to the students elements of modern end-to-end performance measurements using a modern network measurement tool, such as SmartBits. This "SmartBits" session is intended to follow two other sessions given on the networking laboratory "pods." The preceding exercise set is attached to this document for illustrative purposes. Please note that all exercises evolve and are regularly adapted to the specific class needs (curriculum), and that they are given here only as an illustration of the content and scope ofb a typical lab time slot. For further information regarding the practicum, please contact Rudra Duta (rdutta@csc.ncsu.edu, http://www4.ncsu.edu/~rdutta, http://renoir.csc.ncsu.edu/NetLab).*

Measuring Network Characteristics

In this third lab session, we shall take a view often adopted in more practical circumstances. Because the size of a typical internet or even an intranet is large (that is it contains many routers and other network components connected in arbitrary and complicated topologies), and many streams of traffic between different network access points share these resources, the behavior of the network between any two access points is difficult to model theoretically using our understanding of the behavior of small-scale networks, such as we have been gaining in the last two lab sessions. Often, the questions that are meaningful to ask in such situations are ones that relate to the aggregate behavior of the network, such as:

- what is the average throughput between two given hosts?
- what is the delay that we can expect for a packet transmitted from one host to another, and how it is distributed?
- what is the probability that such a packet will be lost along the way?

When asking such questions, we switch to the view of the network known as "network cloud", as shown below in Figure 3.1. The representation of the network by a cloud underlines the fact that we do not know nor particularly care what the details of the network internals are. We simply look upon the network as a black box providing service to the end points. Our view is that of the hosts accessing the network at its edges and obtaining networking service from it. We are interested in characterizing the behavior and performance of the network with this looking-in-from-outside point of view.

Empirical measurements are a good way of determining such performance metrics. We can, of course, install measurements software on the hosts which are accessing the network, and use the hosts themselves for performing the measurements. However, this requires deploying potentially a large number of hosts, and disrupting normal operation on the hosts selected to make measurements. Another solution is to use a dedicated combination of hardware and software

[1] Spirent Communications (http:/www.spirent.com)

which would access the network as a host would do, or as several hosts would do. In other words, the specialized network testing equipment would emulate the behavior of hosts and record the behavior of the network in response.

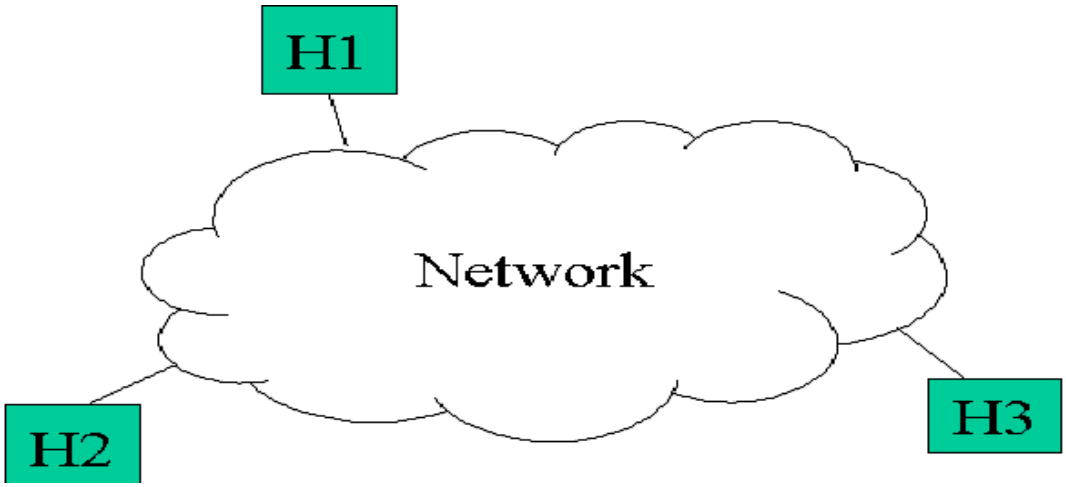


Figure 3.1 Illustration of the "network cloud" and end-clients.

Lab setup

In this lab session, we use such a network testing equipment, the SmartBits tester manufactured by Spirent Communications. For our network cloud, we will use the simple network we have used in the last two lab sessions, after making a minor change that will transform it into a network only slightly more complicated. However, this network, with three shared media and two routers, is already sufficient to demonstrate the concept. Figure 3.2 below shows the lab setup with the SmartBits tester introduced.

As before, Z refers to your pod number. Find out what this number is for your group from the lab administrator.

Notice that in the setup we will be using, the machine H3 has been transformed into a router. We shall refer to this machine as R3 instead of H3 in this lab session. It retains its original address of 10.Z.2.10, but now has an extra interface enabled, with the address 10.Z.3.1, and it has been set up to route packets. The machine H2 also has an extra interface, but it is not connected to our test network cloud. Instead, it serves as a connection to the management interface of the SmartBits tester. We will use this machine to control the SmartBits tester and perform the tests. The four extra hosts S1, S2, S3 and S4 that appear in the diagram are really SmartBits ports that will emulate hosts in our experiments. The IP addresses being used are shown in Figure 3.2.

Take a moment to look at the wiring and verify that the network components are connected as shown in the diagram. Run netstat -rn on H2 and R3 and identify the different interfaces and the routes related to them.

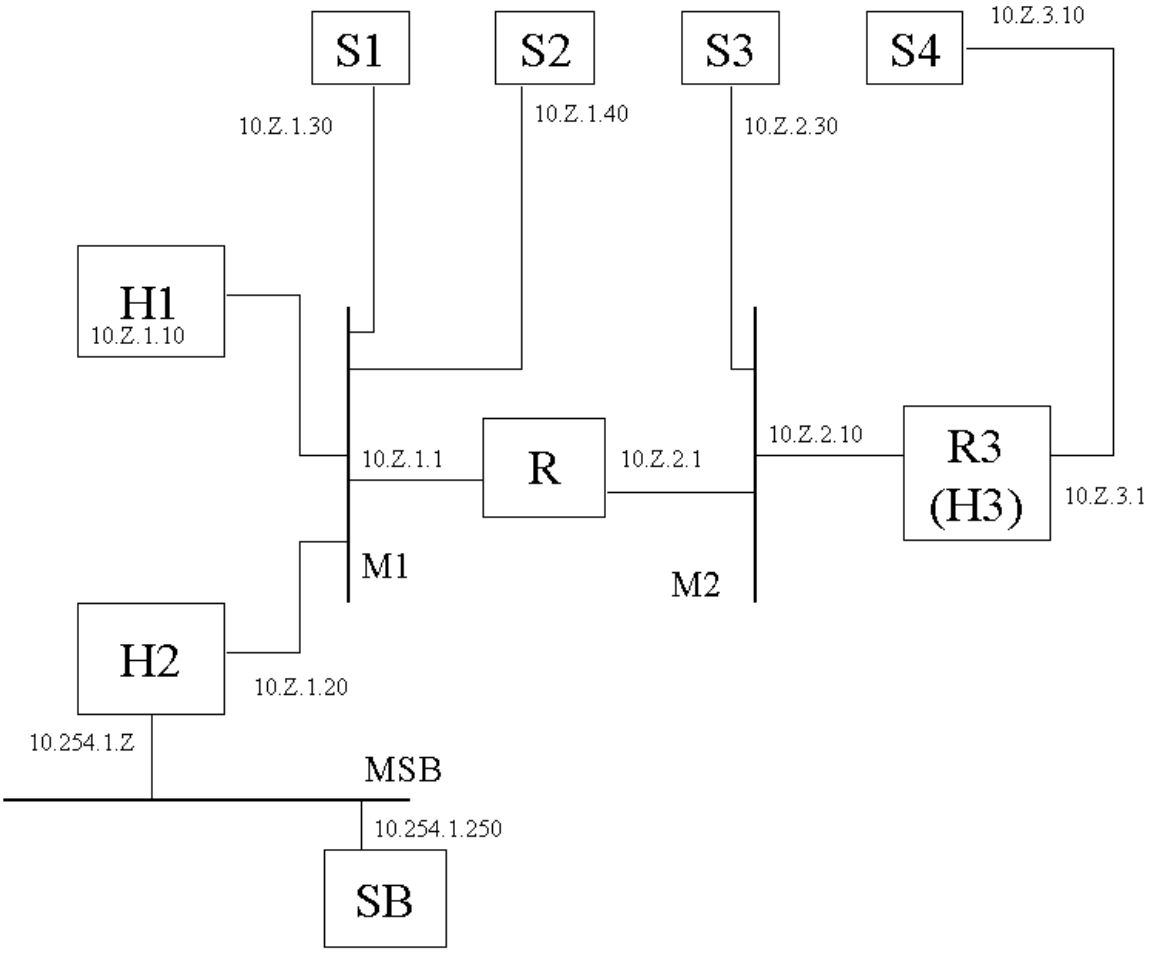


Figure 3.2 Illustration of the “pod” logical network, including SmartBits (SB) tester.

3.2. Running the initial setup

Becoming Familiar with the SmartBits Tester

In this exercise we walk through some of the capabilities of the SmartBits tester using the Windows graphical interface to the tester.

Reboot H2 by logging out of the Gnome graphical interface, and choosing the "Reboot" radiobutton and clicking the "OK" button in the dialog. After the machine reboots and you get the menu with the different operating systems, choose "Windows NT 4.0" and continue booting. When you get the login dialog, login using the same userid and password assigned to your group that you have been using to log in to Linux.

Double click on the icon called "SmartFlow" - the application should start up. It is set up to

connect to the SmartBits tester automatically. You should see the 20 ports on the SmartBits tester represented in a table. Open the file "C:\573_home\ex_3_1.flo", using the "File" menu. A simple test scenario has been saved in this file.

The ports of the SmartBits tester are listed with a name such as "xx y-z", where xx refers to the SmartBits tester itself (there is only one in the lab), y refers to the slot (1 or 2), and z refers to the port on the card in that slot. For the SmartBits 2000, each slot houses a card which has only one port, so z is always 1. You shall be using four ports on four slots of the tester. Find out which slots your group is working on from the lab administrator and concentrate on the 4 ports. These ports will be refered to as ports 1,2,3,4 in the rest of this document.

Complete the following table, writing down the slot numbers on the SmartBits tester assigned to your pod by the lab administrator, and the actual IP addresses, replacing Z by the actual value of Z assigned your group by the lab administrator.

Port	Slot number assigned	IP address in your pod
1		
2		
3		
4		

The working area in the SmartFlow window is organized by different tabs. Click the "Card Setup" tab. This tab contains information pertinent to OSI layers 1 and 2 for the different ports. Verify that ARP has been setup for all the ports, and that all the ports have different (though arbitrary) MAC addresses.

Click the "Networks" tab - this contains information pertaining to layer 3. Verify that the port IP address, network, netmask, and gateway are set for each of the ports 1 to 4 in accordance with the diagram at the top of this document.

The SmartBits tester tests the network characteristics by generating packets in a series, and feeding them into the network. The packets traverse the network and come back into the tester, at which point the tester can measure loss rate, latency, etc. for the packets. Such a stream of packets going out of one port of the SmartBits and coming in at another, for which the SmartBits packet extracts one set of measurements, is termed a **flow**. Thus a flow consists of a series of packets which are logically part of one set of measurements at the SmartBits, and packets in the same flow may have different source and destination addresses, both at MAC and IP layers. (This usage of the term "flow" is somewhat different from the usage commonly found in literature to refer to single TCP connections, but retains the same flavor of an *end-to-end* property rather than something pertaining to the network cloud internals.)

Click the "SmartFlows" tab, this is where you can set up traffic flows that the tester will generate and analyze on return. Two flows, called "sample_1" and "sample_2", are already set up. For each flow, the workspace is further divided into more tabs. Examine the information in these tabs until you can answer the questions below for **each of the two flows**. Fill out the worksheet by completing the table and marking the route taken by each of the flows.

1. What port of the SmartBits tester will source the flow? Sink it?
2. What MAC address will the frames appear to originate at? Destined to?
3. What IP addresses will these ports emulate in the flow?
4. What transport layer protocol header will be included in the packets?
5. How many bytes will be transferred in each packet?

Also answer the question:

- What network resources (hubs, switches, routers...), if any, will the two flows share?

Reserving Ports

The SmartBits tester is designed to allow different testing on different networks to proceed at the same time, using different ports of the tester. Obviously, some protocol is required to determine which controlling station controls which ports of the SmartBits, otherwise the same port might receive different control instructions from different groups attempting to perform different tests, and none of the tests will be successful. SmartBits implements a FCFS reservation protocol by which any controlling station is allowed to reserve any port not currently in use to perform tests. Once a port is so reserved, no other controller can reserve or use that port before the original controller has finished testing and released it.

- Go to the "Card Setup" tab and look at the column called "Multiuser". Verify that all ports are unreserved initially. Reserve the slots which have been assigned to your group by the lab administrator.

- To verify that the same ports cannot be reserved by more than one controller, find out from the lab administrator which ports have been reserved by another group. Try to reserve those ports and verify that you receive an error message, and are unable to reserve them.

- **Note:**If some ports were already reserved by the time you started the SmartFlow application, those ports are completely invisible to you, so you cannot even attempt to reserve them.

Running the Test

Now we shall run the different kind of tests that SmartBits is capable of performing using these flows. Note the following information about the SmartFlow application:

- The left side of the screen has *two* toolbars, each with icons such as "Throughput", "Latency", etc. - but at any time only one of these is visible. The first one is to setup characteristics of the tester and the flows we design, and to run the test. Whenever you run the test, the second toolbar comes up. This is the one in which you view results of the tests we are running. After the test is complete and you have finished viewing results, you must click on "Setup and Run" to get back to the Setup and Run screen to modify the tests and run them again.

- Results persist from one test to another. For any one type of test, such as "Latency", you

cannot view results before you have performed the test even once. However, once you have performed the test, you will see the same results any time you go to the "Results" screen and click on "Latency". If you have changed the tests, the old results are not automatically expired. Thus you *must* go back to "Setup and Run" and rerun the test if you want to see updated results.

- For the same reason, you will get a dialog box asking "Overwrite the results?" every time you run a test after the first time. Answer "Yes" to rerun the test and get new results.

- Very occasionally, you may get an error message such "Learning failed on port x" or "Step cannot be less than zero" when you try to run a test even though your test parameters are correct. This appears to be because of a flaw in the SmartFlow GUI application - it merely needs to perform some initialization that it did not do automatically. If this happens, visit all the major tabs in "Setup and Run", and all the subtabs in each, without changing any data. Then try running the test again, the error should be resolved.

- Messages appear on the status bar at the bottom of the window when you run a test, indicating that

    - Ports are being setup
    - ARP addresses are being learned
    - Flows are being setup
    - Traffic is being generated and received
    - Test is successfully concluded

  You must wait until the test is successfully concluded before interpreting the results.

- In the "Results" screen, the results appear in different formats: a 3-D graphical chart, a summary table, a detail table. Different formats are most helpful for different tests. Examine all of them to find out which is most appropriate for a given test.

While running the tests, we shall monitor the packets flowing over the network using R3. We shall use `tcpdump` for the purpose. From our earlier lab sessions, we know that `tcpdump` is a powerful tool, but in this lab we will be using it only to visually verify that packets are flowing across the medium. On R3, open a shell or a virtual console, and log in using the id/password provided. Run the following command:

- `tcpdump -n`

Leave this `tcpdump` running throughout this lab session. Terminate it using `Ctl-C` when you are finished with the lab.

Go back to H2, the machine acting as SmartBits controller. On the left of the workspace, there is a toolbar with the different tests. Click the top one, called "Throughput". You will see messages in the status bar at the bottom of the window, band see the following : When traffic is being generated, you should be able to see packets flowing by the `tcpdump` at R3. If you do not, there is an error.

After the test is concluded, the Results screen will come up. In all formats, there will be data for the two flows individually as well as a total. Look at the different results representations and answer the following questions:

1. Is the throughput 100% for either of the flows? If not, why do you think this is?
2. Which flow was able to transfer the greater amount of data?
3. Look at the `tcpdump` output on R3. Verify that the packets seen by R3 have the source and destination IP addresses and ports that we had set up in the flows.

- **Note:** The results for throughput is presented in the same units as Frame loss, that is it is presented in inverse units.

Now run the "Frame Loss" and "Latency" tests as well, and examine the results. Verify that the results are consistent with the lab setup, as well as within themselves.

3.3. Running a Single Flow

Go to the "SmartFlows" tab and delete the flow "sample_1" by selecting it in the list of flows and clicking the button with the red "X" on it. Now we only have one flow. Run the "Frame Loss" test again. Comment on the nature of the results this time as opposed to in the last exercise.

Customizing Latency Distributions

Run the "Latency Distribution" test. Once the test is completed, you will see a frequency distribution of the latency values observed. The distribution is presented by showing the number of packets that were seen to have a latency that falls into a given range of latency values or "bucket" - for example, one bucket may represent the range from 500 microseconds to 1000 microseconds, while the next may represetnt the range from 1000 to 5000 microseconds, and so onl. In the results we generated, most of the frames fall in the same bucket, so the distribution is not very informative. We can customize these buckets and fine tune them for our network cloud to get more useful information out of the test.

1. Go back to the "Setup and Run" screen and go to the "Test Setup" tab. Go to the "Individual tests" subtab. Here you can specify the buckets that should be used for the frequency distribution. Change the buckets to be used: as a first example, change the 50,000 microseconds entry to read 75,000 microseconds. Run the "Latency Distribution" test again. See if the results carry any more information this time.
2. Change the buckets a few more time until you have significant occupancy in at least four of them.

Load Varying Tests

Now we will see how SmartBits automates testing the same characteristics at different loads. Go to the "Test Setup" tab again, and then to the "Individual Tests" subtab. We have been getting only one test run each time because we had the minimum and maximum set equal. Now change the initial load to 10%, step load to 10%, and leave the maximum at 100%.

- Run the throughput test again. Notice that the results graph is now built in increments, and the flow is started and stopped ten times. Look at the `tcpdump` output, it should be visibly obvious that initially the flow is slower and speeds up on later iterations.

Emulating Many Hosts

Next we see how to emulate many hosts using just one SmartBits port. Go to the "SmartFlows" tab and the "Traffic" subtab. In the lower part of the window, "Variable Fields within Flow", check the "SRC IP address" checkbox. Enter a value of 200 as the variable count. This specifies that each time a packet is sent out, it will be sent out with a source IP address incremented by 1 in the last octet. The IP address will cycle through 200 addresses. Thus to the network being tested, there appears to be traffic coming from 200 different hosts. Run the throughput test again. The results will look the same as before, but while the flow is being transmitted, the `tcpdump` running on R3 will record the packets as coming from different IP addresses. You will see that the addresses go from 10 in the last octet to 209, in consecutive numbers.

Using the entry field for "Variable Count" as above, and entry fields on the "IP" subtab, make the flow emulate packets which come from addresses which go from 2, 4, 6, ... 200 in the last octet. Verify that you have correctly set the parameters by running the throughput test again and observing the packets on the `tcpdump` output.

3.4. Creating Several Flows

Delete the remaining flow. Now there are no flows. Add flows with the following properties, using the "SmartFlows" tab.

- **testflow_1**: From port 1 to port 4, source address 10.Z.1.30, destination address 10.Z.3.10, no next protocol to IP, no variable fields.

- **testflow_2**: From port 3 to port 2, TCP protocol, source address starting at 10.Z.2.30, last octet varying 30, 31, ... 101, source port starting at 1024 and varying, no other variation, destination port 9 (DISCARD service).

Make sure that the "Test Options" - "Individual tests" setup is as before, that is, initial load 10%, step 10%, final 100%.

Go to the "Options" tab. The three big buttons at the bottom provide a way to verify a new flow setup in stages, helping in locating errors. Click each of the three buttons in turn; for each one wait until you get a success message on the status bar or an error message before proceeding.

Once you have successfully gone through all three setup stages, you can run the various experiments and view the output. Try to understand the output intuitively. If you feel there is something you cannot explain, try to see if there is an error you can deduce.

3.5. Introducing Some Complexity

Since we have been using our simple network as a network cloud, it may have appeared that we could have obtained the results we have obtained using the SmartBits tester simply by applying analytical method to our knowledge of the network. It is true that the simple network we have

been using is amenable to analysis, though this characteristic would quickly be lost with increasing size and complexity of the network. However, even this simple network may not be easy to analyze if we assume more realistic traffic scenarios. We have already considered several streams of traffic flowing in the network along different routes. Because some resources are shared for these traffic streams, while others are not, it is not easy to predict how the network will behave from the point of view of each individual traffic stream, but it is still possible to measure it. To add another measure of realism, we drop some packets at the routers in our network to simulate congestion in this exercise.

On both R and R3, execute the following from a command line:

- `forwarding`

This is the same tool we used in the second lab session. When executed without any arguments, it will continue to switch the packet forwarding functions on the routers on and off randomly until it is killed from the command line using `Ctl-C`.

Once `forwarding` is running on both routers, run the different tests using the SmartBits again. This time the output will be less predictable, and you may see noticeably lower throughputs and higher frame loss rates. This once again points out the fact that even comparatively simple networks can be difficult to model, and measurements will be often the only practical way of evaluating network characteristics.

3.6. Scripting

While a graphical user interface as the one we have been using to control the SmartBits tester is user friendly and intuitive, it has the disadvantage that the operation is more difficult to automate for streamlining and reliable repeatability. In such situations, it is convenient to have a command or programming interface which, in effect, implements a language that can be used to program, or script, the tester. Scripts in this language can also be produced in an automatic or semi-automatic fashion. SmartBits provides such a scripting interface, and the scripts can be run to control test scenarios in both Windows and Linux operating systems. In this exercise, we walk through running a script under Linux. Exactly similar scripts can run under Windows as well.

Close the SmartFlows application. Reboot H2 by selecting "Shutdown" from the Windows Start menu, and selecting "Restart the computer". After the machine reboots and you get the menu with the different operating systems, choose "Linux" and continue booting. (You may have to make this choice twice.) When you get the login dialog, login using your usual id and password.

Under your home directory, you will find a directory called SmartBitApps/ containing some scripts. Use an editor such as `vi` or `pico` to view the file called 'run_latency'. It contains only a few lines, which look like the following:

```
if ($?LD_LIBRARY_PATH < 1) then
   setenv LD_LIBRARY_PATH /
endif


setenv LD_LIBRARY_PATH ".:$LD_LIBRARY_PATH"
echo "Ready to run TCL script "
```

```
./sctcl.out SinglePairLatency.tcl
```

This file simply sets some environment variables and runs the customized TCL shell that SmartBits uses to run its scripts. The real script name is supplied as an argument to the TCL shell, and is called 'SinglePairLatency.tcl'.

Close the file 'run_latency' and view the file 'SinglePairLatency.tcl'. The comments tell you what each piece of script code does. Verify that the sequence of various setup and running the experiment is the same as we followed using the SmartFlows application above. Identify the variables used for setup and answer the following questions. Complete the worksheet.

1. How many flows are being set up? Answer the following questions for each.
2. What port of the SmartBits tester will source the flow? Sink it?
3. What MAC address will the frames appear to originate at? Destined to?
4. What IP addresses will these ports emulate in the flow?
5. What transport layer protocol header will be included in the packets?
6. How many bytes will be transferred in each packet?
7. What test will be run by the script?

Close the file. From a virtual console, execute the script by running:

- `run_latency`

When the test is complete, some data is presented textually on the screen. This data is the result of the test, in similar format to the "Summary" representation in the SmartFlows GUI application. In a realistic scenario, this data could be captured and parsed programmatically and saved to a data repository.

Try to interpret the data. If you feel there is something you cannot explain, try to see if there is an error you can deduce.

- Change the appropriate settings in the script to measure the latency again, but vary the source IP address to cycle from 10.8.1.40 to 10.8.1.239, the last octet varying by 1 each time. Run the script.

3.7. Deliverables for Lab session 3

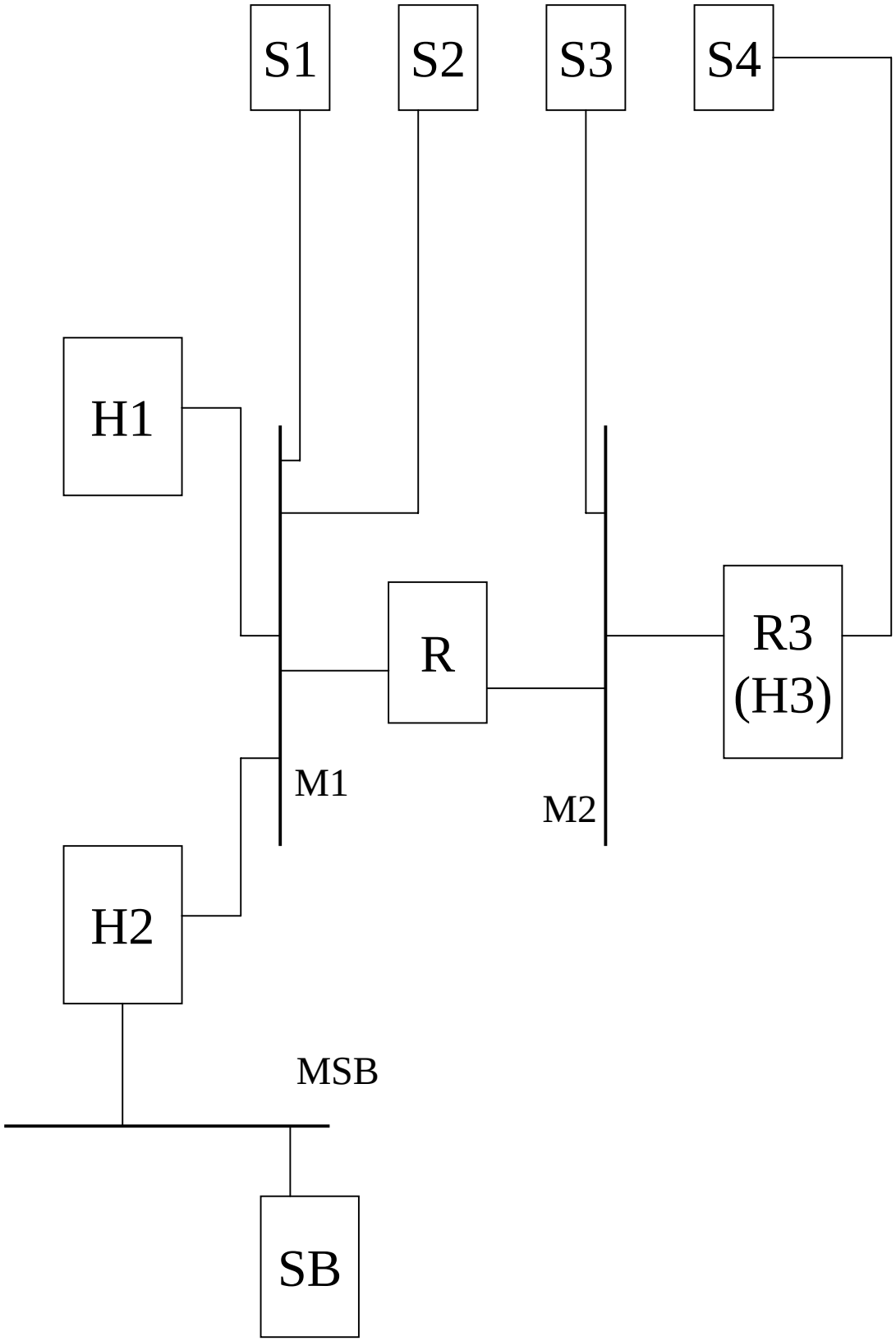
- No program output is needed. For each exercise 3.x above, provide all answers in `conclusions.3.x`

Cleanup

Please delete any temporary files you produced during Lab 3, on any of the pod machines.

4. Appendix I – Worksheets

- Network topology
- Flows
- Scripting


S1
S2
S3
S4
H1
R
R3
(H3)
M1
M2
H2
MSB
SB

Sample Flows Worksheet

Flow Name	
SmartBit source port	
SmartBit destination port	
Source MAC address	
Destination MAC address	
Source IP address	
Destination IP address	
Transport layer protocol	
Bytes transferred per packet	

Flow Name	
SmartBit source port	
SmartBit destination port	
Source MAC address	
Destination MAC address	
Source IP address	
Destination IP address	
Transport layer protocol	
Bytes transferred per packet	

Flow Name	
SmartBit source port	
SmartBit destination port	
Source MAC address	
Destination MAC address	
Source IP address	
Destination IP address	
Transport layer protocol	
Bytes transferred per packet	

Scripting Worksheet

Number of flows:
Test being run:

SmartBit source port	
SmartBit destination port	
Source MAC address	
Destination MAC address	
Source IP address	
Destination IP address	
Transport layer protocol	
Bytes transferred per packet	

SmartBit source port	
SmartBit destination port	
Source MAC address	
Destination MAC address	
Source IP address	
Destination IP address	
Transport layer protocol	
Bytes transferred per packet	

5. Appendix II – Labs I and II with Discussion Notes

- Lab1 Discussion
- Lab1
- Lab2 Discussion
- Lab2

Overview - Before You Begin

This document provides an overview to the laboratory experiment sessions for the CSC/ECE 573 Internet Protocols and Architectures course. This overview and the suggested readings should be read before beginning the lab since lab time is limited. We suggest that you bring your course textbook and a calculator to the lab.

The lab is physically in Suite 150 room 101 (Networking Lab) at Ventures II in Centennial Campus. The machines you will be working on are to your left as you enter the lab.

The lab exercises are in the following pages:

- Lab 1 Exercises
- Lab 2 Exercises

There is also a discussion page available for each of the two lab sessions:

- Lab 1 Discussion
- Lab 2 Discussion

All lab documents are available in electronic form in each lab cluster, on the machine H2 in the directory `/net_analysis/docs`. For the meaning of "cluster" in this context, see below in the **Software/OS Setup**. The exact meaning of H2 will be defined in the first exercise of Lab 1.

Authority, Accountability, Responsibility

The CSC/ECE 573 Networking Lab offers the privilege of experimenting with a computer network at the level of a network administrator, providing the opportunity to witness the theoretical principles learned in the classroom in the reality of a physical network. In particular, you will experiment with:

- ARP
- IP Routing
- TCP flow control in the presence/absence of network congestion

Realize that this privilege carries with it a certain responsibility, to behave honestly and in accordance with the instructions of the lab materials. Intentional misuse of lab equipment will result in a failing grade in the course plus possible disciplinary action. The following rules apply at all times while working in the networking lab:

- Do not go behind the tables.
- Do not remove/insert cables.
- Don't move equipment here or there.
- Don't move equipment anywhere.

Software/OS Setup

Computers and Operating Systems

The machines in the networking lab are arranged in nine identical "clusters". Each cluster is isolated from the others and from the rest of the world, giving you a small "network world" of your own to conduct your experiment. There are nine columns in the lab which each have the word "Cluster" on it, followed by one of the numbers 1,3,4,5,6,7,8,11,12. (There is no cluster 2,9 and 10.) Choose a cluster to work on and note the cluster number for use in your experiments. You will also need to provide Dr. Martin or the TA present with your cluster number so that your work can be graded. (Try to work on the same cluster each time you come in. Otherwise, provide Dr. Martin or the TA present with each cluster number.) The IP addresses assigned to the machines in your cluster depend on the cluster number. In the exercises we use the symbol `'Z'` to refer to the cluster number when necessary.

Each cluster contains four computers and two monitors. All the computers are HP-Kayak RedHat Linux 7.0.

The four computers are referred to in the exercises as H1, H2, H3 and R. The first exercise of lab 1 matches the names to the computers. All are PCs running the Linux operating system. Linux is a UNIX family operating system, which means that it is similar to other UNIX systems such as BSD, Solaris, Ultrix or FreeBSD that you might be familiar with.

Login Access

Each student will receive a userid/password combination from Dr. Martin or the TA present. Your userid will be of the form `csc573gxx` where `xx` is a non-zero two digit number; *e.g.* `csc573g53` may be the userid assigned to a particular user. Your userid and password work for all the Linux machines. Do not change the password for your group, because they are separately set on each machine.

IP Address

The machines in the lab make up a private network which uses IP addresses that are invalid on the public internet. The private network is isolated from the public internet. This is a precautionary procedure, so that simple mistakes do not result in possibly disastrous consequences on the public network. Thus this measure will protect you from inadvertently affecting the public network, while allowing you to study the network undisturbed by general internet traffic. These IP addresses are of the form `10.A.B.C` and are clearly labeled on each machine. Labels of the form `152.A.B.C` can be ignored.

Display Sharing

The three PCs H1, H2 and R share the same monitor, keyboard and mouse by means of a *Keyboard/Mouse/Monitor sharing switch* (or sharing switch for short). The sharing switch in each cluster is the black box between the machines. Pressing the "Select" button selects each of the different machines connected to the sharing switch.

The Machine H3 has its own dedicated monitor, keyboard, and mouse which are placed on the table.

Linux virtual consoles

The Linux machines are running Red Hat 7.0 with the Gnome X interface. This system provides seven virtual consoles which can be accessed at any time by pressing the key combinations `Ctl-Alt-F1` for the first virtual console, `Ctl-Alt-F2` for the second, ..., and `Ctl-Alt-F7` for the seventh. The X interface always runs on the seventh virtual console. The other consoles are the traditional VT100 "black screen" console. You may be logged onto several consoles at the same time; the Linux system regards you as multiple users who are concurrently logged in using different pseudoterminal devices.

Note that all seven consoles are completely independent. You remain logged on to one while logging out from another. It is possible to come in and directly switch to virtual console 1 and log in, without ever logging in to the X system on console 7, and vice versa. Please verify that you are logged off from all the consoles of all machines in your cluster before leaving.

If you are comfortable using a GUI, you may want to use only console 7 and leave the other consoles alone. Since you can open multiple windows (xterms) in the X system GUI, you can do everything you need to do using only this console.

Please note to avoid confusion that we are using a hardware sharing switch to share the monitor between H1, H2 and R, and among these H1, H2 and R use this virtual console mechanism (in software) to share the console between the seven virtual consoles on each machine. H3 uses the virtual console mechanism to share its dedicated console between its virtual consoles.

EOS/UNITY Connectivity

There is no access to EOS/UNITY from the lab machines, since the experimental setup is in a private network shielded from the outside.

A single EOS/UNITY public workstation may be available in the lab for CSC/ECE573 students, if time permits its installation. This is not finalized at the time of this writing. Please check when you get to the lab.

Tools You Will Use

Standard Utilities

We shall use some standard tools provided on UNIX for network management and monitoring, such as `arp, ping, netstat,` and `route`. Any standard UNIX system, such as public EOS stations running Solaris, as well as the machines in the lab running Linux, will have the manual pages for all of these installed. There may be minor differences in the options or actions of these utilities in different versions of UNIX, but the basic concepts remain the same. Whenever a lab exercise employs a specific option, the option's action will be defined. **We suggest you read the manual pages before you come to the lab.**

Some less common utilities are described below.

tcpdump

`tcpdump` is a packet capture utility on UNIX, developed by the people at Lawrence Berkley Laboratories. It puts the network interface card in full promiscuous mode, and dumps the packet header data on the standard output. Thus by running `tcpdump` on one machine you can view all the traffic in the LAN the machine is on. We shall be using it to view TCP packets as well as

other things like ARP or UDP traffic.

Each line of the output of `tcpdump` represents one packet that was observed on the network. `tcpdump` recognizes the protocol for the packet (ARP, TCP, UDP, etc.) and prints some of the relevant header information. Filters can be provided to specify that only some types of packets for some hosts or networks are to be observed, not all; such filters are not used in the exercises. Read the manual pages for `tcpdump` to find out how to interpret the output. The book *TCP/IP Illustrated, Vol I : The Protocols* by W. Richard Stevens also has material on `tcpdump` output format. The lab exercises and Discussion pages also indicate how to interpret the `tcpdump` output.

`tcpdump` is started from the command line (VT or xterm window). Once you have finished running the experiment, kill it with a `Ctl-C`.

The exercises ask you to run `tcpdump`, observe the output, and save it as part of your results. If you are comfortable using a GUI, then you can run `tcpdump` in an xterm window and cut-and-paste it to an editor. If you must use a VT console, then the best option is to redirect the output to a file and then view the file once you are done running the experiment. **The exercises may ask you to do the same for other commands besides tcpdump.**

While tcpdump can normally only be run by the root user, you are enabled to run it as trusted users. Please do not use `tcpdump` in any way other than that specifically described in the exercises.

Never run more than one instance of `tcpdump` at the same time on a machine.

The initial part of the `tcpdump` manual page, which lists the command line options, is provided below:

```
TCPDUMP(1)                                                TCPDUMP(1)

NAME
       tcpdump - dump traffic on a network

SYNOPSIS
       tcpdump [ -adeflnNOpqStvx ] [ -c count ] [ -F file ]
               [ -i interface ] [ -r file ] [ -s snaplen ]
               [ -T type ] [ -w file ] [ expression ]

DESCRIPTION Tcpdump prints out the headers of packets on a network
       interface that match the boolean expression.

       Under Linux: You must be root or it must  be  installed  setuid  to
 root.

OPTIONS
       -a     Attempt to convert network and broadcast addresses to names.

       -c     Exit after receiving count packets.

       -n     Don't convert addresses (i.e., host addresses, port numbers,
              etc.) to names.

       -q     Quick (quiet?) output.  Print less protocol information so
              output lines are shorter.

       -S     Print absolute, rather than relative, TCP sequence numbers.

       -t     Don't print a timestamp on each dump line.
```

```
-v (Slightly more) verbose output.  For example, the time to
       live and type of service information in an IP packet is
       printed.
```

More information on `tcpdump` can be found at LBNL's Network Research Group at `"http://www-nrg.ee.lbl.gov/"`.

oursock

`oursock` is a small socket utility developed specifically for this lab. You can use `oursock` to send zero or more bytes from either H1 or H2 to H3. On the command line you can specify several options: (1) whether to use TCP or UDP, (2) the total number of bytes to send, and (3) for UDP, the number of bytes to be sent per packet.

The data are sent to the `discard` service on H3 and thus play no role in the experiment. We are only interested in the traffic that is created on the network.

gnuplot

`gnuplot` is a plotting utility for UNIX and other systems. It can convert data into plots in several formats, and also display it on screen for an X windowing system. We will use gnuplot to graphically interpret bulk traffic data related to TCP data transfer.

Despite the name, `gnuplot` is not related to the FSF. You can find more information on `gnuplot` at the Gnuplot homepage at `"http://www.cs.dartmouth.edu/gnuplot_info.html"` .

You will only need to use `gnuplot` for the second lab session.

About Editing

To view the Overview, Discussion, and Lab Exercises documents, use Netscape or the Gnome Help Browser. (The Help Browser opens automatically at login; simply type in "/net_analysis/docs/overview.html"). In addition, you will need to read and edit plain text files on the Linux machines. The possibilities are:

- **vi:** The all-purpose text editor that comes with almost all versions of UNIX. It is completely based on a text terminal and can be used either on a VT or in an xterm window. Vi uses separate command and editing modes. Unless you have used vi before, we do not suggest using it in the lab.
- **emacs:** The version on Linux provides a simple menu-driven interface running on a text terminal. Even if you are not familiar with emacs, you can use the menus to find your way.
- **pico:** This is probably intermediate in difficulty to `vi` and `emacs`. It also runs on a text terminal, and provides menus to help you do your editing. Your options may be more limited than with `emacs`.

Since these are all text terminal based editors, you can run them on a "black screen" VT or in an xterm window in the X windowing system. The advantage of the latter is that it is easy to cut-and-paste text between an xterm in which you obtained the results and an xterm running the editor.

Transferring Files

During the exercises, you will sometimes produce an output file on one machine by capturing the output of a command, and then need to transfer the output to a file on another machine. You can do this easily using the `ftp` utility. **If you are not familiar with `ftp`, read the corresponding manual pages before you come to the lab.**

We provide an example of using `ftp` below. Assume you have produced the file `output1.txt` on H1 and need to transfer it to H3. On H3, you wish to save it as the file `netstat.1.3.1` in the directory `results` under your home directory. If the address of H3 is 10.4.2.10, then `ftp` to that address from the local directory which contains your file, as shown below.

```
ftp 10.4.2.10
Connected to 10.4.2.10.
220 [H3 machine private name] FTP server (Version wu-2.5.0(1) Tue Sep 21
16:48:12 EDT 1999) ready.
Name : c573gxx
331 Password required for c573gxx
Password: [Type in password, not echoed]
230 User c573gxx logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd results
250 CWD command successful.
ftp> ascii
200 Type set to A.
ftp> put output1.txt netstat.1.3.1
local: output1.txt remote: netstat.1.3.1
200 PORT command successful.
150 Opening ASCII mode data connection for netstat.1.3.1.
226 Transfer complete.
7 bytes sent in 7.3e-05 secs (94 Kbytes/sec)
ftp> bye
221-You have transferred 6 bytes in 1 files.
221-Total traffic for this session was 477 bytes in 1 transfers.
221-Thank you for using the FTP service on [H3 machine private name].
221 Goodbye.
```

Your input is indicated by bold letters above. After logging in, change to the directory where the file needs to be placed, set the transfer mode to ASCII characters, and transfer the files. Then terminate the connection.

Doing the exercises

Each exercise asks you to perform specific tasks on the machines of your cluster and then observe and possibly also save the results. Then you must answer some questions by looking at the data you have obtained, and save your conclusions in separate files. Each exercise tells you exactly what files you need to deliver for that exercise.

Every time you create a login session on one of the machines (by logging in at a console, a virtual console, etc.) and every time you exit one (by logging out) the system runs some tasks which may invalidate your observations. For this reason, you should not log in or out in any session on any of the machines while you are in the middle of an exercise. For each exercise, get all the login shells and windows you need, then perform the exercise without creating new ones or exiting existing ones, then exit as necessary.

You may need more than one time slot to complete the entire lab session. We suggest that the

first time you concentrate on performing the tasks specified in the exercises and saving your observation files. Then you can come to the lab, having though about the output you have seen, and create your conclusions files.

No textbooks or reference material will be available in the lab. You should bring your textbooks and any notes, books or other reference material you may want to use. Volumes 1 and 3 of *Internetworking with TCP/IP* by Comer and *TCP/IP Illustrated, Vol I : The Protocols* by W. Richard Stevens contain all the material you need to do the exercises.

Submitting Results

All your results should be in a directory named `results` on H3 in your cluster under your home directory. We shall refer to this directory as "Your `results` directory" in the exercises. Your home directory is `/home/csc573gxx` and you should be in this directory when you log in. All your results must be in plain text files. For the most part, this will be text you see as the output of some command you have run, or text you simply type in (such as "The average throughput is : 7 MB/s"). Each exercise tells you exactly which results are expected to be in what files.

Please do not leave any result files on any machine other than in the `results` directory on H3.

Do not save any file (even temporary ones) on the Linux machines outside your group's home directory.

Good Luck

That's it! You can go off and do those lab exercises now, and we hope you enjoy it.

Go back to the top of the page.

Discussion of Exercises for First Lab Session

Overview | Exercises for Lab 1

1.1 Becoming Familiar With the Lab Setup

The goal of this exercise is to become familiar with the machines in the cluster in terms of type, operating system, name, and network properties such as addresses, subnet, and routing. At the end of this exercise you should be able to think "H1" and immediately point to the machine in the cluster and remember its address(es) and subnet.

If you cannot decide which network a machine is in, it may be in both networks. Think about this question: Can a machine which acts as a router (gateway) *not* be on more than one network?

1.2 ARP: Address Resolution Protocol

Computer networking management for internetworks has traditionally been done on UNIX operating systems, because most computers were running UNIX when the designers built TCP/IP, the ARPANET, and the Internet. As more computers running operating systems other than UNIX connected to the Internet, similar management systems were created for these operating systems. In Windows, for example, essential network management tasks that must be carried out on every machine (such as setting the IP address and mask) are usually implemented with a GUI. Less essential tasks are carried out with programs that run from the command line and retain a distinctly UNIX flavor. Two examples are `arp` and `ping`, used in this exercise. Sometimes GUI programs are available for the same task, not as part of the operating system but as a program that must be acquired separately.

This exercise introduces `tcpdump` and uses it to examine headers of packets that are observed on the local physical network. It is important to remember that there are two physical networks in the cluster, and that `tcpdump` is running on the network connecting R and H3. `tcpdump` will therefore capture all packets traveling on that network, but not packets in the other physical network (connecting H1, H2 and R).

In a line of `tcpdump` output (also called a packet trace or packet header dump), the first quantity is the timestamp. In particular, the timestamp is the time when the packet was observed on the wire, according to the clock on the machine running `tcpdump`. Though the time is given with microsecond precision, the network interface card may not have the same accuracy, so the last digit or two may not be significant.

The rest of the line is header data for the packet. `tcpdump` interprets as much of the header as it can, recognizing many of the common protocols. Thus when an ARP packet is recognized, `tcpdump` looks in the header and reports (1) whether it is an ARP request or reply and (2) the IP addresses of the machines involved. Only a selection of the information in the packet is displayed, with the aim of making the packet trace more readable. For example, the hardware (MAC) address for the requesting host is usually sent in an ARP request, but it is not displayed in the packet trace.

When `tcpdump` observes a packet using an IP protocol, it extracts and displays the relevant header information. In this exercise `tcpdump` captures some ICMP packets and displays the

ICMP type and code field values. The ICMP traffic is generated by the `ping` command, which sends ICMP_ECHOREQUEST packets and receives the ICMP_ECHO packets. `ping` verifies that a host is up, reachable, and responding to ICMP. In later exercises, we shall see header dumps of TCP and UDP packets as well.

1.3 Discovering Routing

The three tools we use in this exercise give us information about routing, but they report different information that is acquired using different methods. You should read the manual pages for each of these if you have not already done so.

`netstat` can be used to report the network related status of a host, and in particular we use it to report the routing table. Thus the information we obtain is purely local, and has the "next-hop-only" information characteristic of the IP routing table philosophy. In contrast, `traceroute` and `ping` both send data onto the network, and extract information from the data received from other hosts or routers on the network.

In looking at the routing table, we are most interested in the first few fields of each route. Each route starts with a destination field, which holds the IP address of the destination host or network. The next few fields specify the subnet mask and the address of the next hop router (gateway) where packets should be forwarded. To answer some of the questions, you need to interpret these fields to describe the route; to answer others, you need to hand-trace the routing decisions by looking at the routing tables.

Let us consider some of the output we get from a Linux machine when we run `netstat -rn`. The output on a Windows machine is very similar and can be similarly interpreted. Since we are using subnetting (the *de facto* standard), a route of the form:

```
<Destination>     <Gateway>          <Genmask>
```

can be interpreted as: if the destination address for a packet, when AND-ed with `<Genmask>`, is equal to the `<Destination>`, then this packet may be forwarded to `<Gateway>`. As we know, there may be multiple routes to a destination, and the order of the search should be for a route to a host, then for one to a network, then for a default. (See *TCP/IP Illustrated* by Stevens or any other text for details.) With subnetting, how are such routes distinguished? Naturally, if the `<Genmask>` is complete (255.255.255.255), then we end up comparing the whole destination address to `<Destination>`; thus this route is a route to a host. If the `<Genmask>` has some trailing zeroes, then we ignore the least significant bits of the destination address before we compare it to `<Destination>`; thus all hosts in a certain network will match this route. Lastly, if the `<Genmask>` is all zeroes and so is the `<Destination>`, then any destination address we care to compare will match; such a route is a default route.

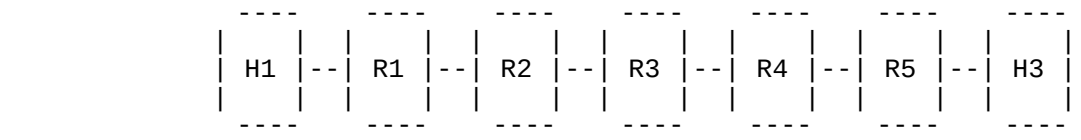
Note that for routes to networks, when `<Genmask>` has trailing zeroes, the `<Destination>` field must have at least the same number of trailing zeroes. If not, no address will ever match that route. This rule is in keeping with the rule of zeroes in the host part of an address denoting the network itself. This rule also applies to the `<Gateway>` field. For some destinations, the host may its own `<Gateway>`. This is the case, for example, for any destination on the same physical network, and for any destination which is really the host itself (either using an explicit address or the loopback address). In that case the `<Gateway>` might be listed as 0.0.0.0 , which would have the meaning "this host on this network".

By default, `traceroute` attempts to utilize the ICMP_TIME_EXCEEDED message. It sends out

a succession of datagrams destined for the address we wish to trace the route to, each with a higher value of the time-to-live field (starting at one). This scheme elicits successive ICMP_TIME_EXCEEDED messages from successive routers along the way, until finally we obtain the complete route that the datagram has taken to the destination. This mechanism is vulnerable to route changes in intermediate routers during operation. However, for a fairly stable network this is not a big problem.

As we have seen before, `ping` uses the ICMP_ECHO mechanism. In this exercise, we turn on the Record Route option on the ECHO packets. In this case, only one packet is transmitted to the destination and only one comes back, but the route (both outgoing and incoming) is recorded in its header. The manual page mentions that many systems ignore this option : to see a case in point ping R from H1. Now compare the output with the output you got when you ping-ed H2 from H1. You see H2 twice as expected (once coming and once going), but you see R only once because R ignores this option on the outgoing packet.

Between `traceroute` and `ping` (with the Record Route option), `traceroute` is more likely to work in any given situation. However, if the Record Route option works, then `ping` returns more information, because information is saved both going and coming. Any intermediate router will show up with different IP addresses on the outgoing and incoming path, because the packet goes out different interfaces. Now if the outgoing and incoming packets go through the same sequence of routers, then it is possible to conclude the IP address of both the outgoing and incoming interfaces on each router from the route recorded. Consider the following route, where two hosts H1 and H3 are connected over a sequence of routers Ri. For each Ri, the address closer to H1 is Ri1 and the address closer to H3 is Ri2.



Then the route we expect to see recorded when we ping H3 from H1 is:

```
H1-R12-R22-R32-R42-R52-H3-H3-R51-R41-R31-R21-R11-H1
```

The great disadvantage is that this method does not work beyond a few hops because there is only space for nine addresses to be recorded.

A special note with respect to the host specific route experiment of exercise 1.3.2 .
Normally, such an obviously redundant route as the one used in this exercise would not be allowed to have long-term effect on the routing in the internet. As soon as H1 sends the first packet to H2 through R, R would send an ICMP REDIRECT message to H1, instructing H1 to reach H2 directly. From there on, H1 would send messages to H2 just as if the host specific route had not been added, until the redirect expired after some time. For the purpose of demonstration, the mechanism to accept redirect messages has been disabled on the H1 machine in all the clusters. However, it appears that this was not correctly set for all the clusters, so some student groups may have conducted this experiment while the redirect was still being accepted. If you are one of these groups, then you will see the following behavior - if you run`traceroute` after you add the route using `route_H2_through_R`, you will see R in the route to H2, but if you run `traceroute` again, you will not see R included, the route will be back to normal. If you see this behavior and note it down (even better, try to explain it), it is okay. If you want to see the expected output, then you must run `traceroute` only once, right after you run `route_H2_through_R`. If you *have* to run `traceroute` again (because you did not capture the output or whatever), then you should run `clear_H2_route`, then run `route_H2_through_R`

again, and finally run `traceroute` again. You need to do this each time you want to run `traceroute` in this experiment. We apologize for the mistake and the resulting inconvenience.

In the toy "network world" which your cluster represents, all of the above are good methods to investigate routing, and they should all give consistent results because everything has been configured correctly and consistently. In the real world, we need all the tools that are available (including far more powerful ones than the ones we use here) to observe the network, and discrepancies that we may observe are valuable indicators to things that might be wrong in the network.

The last two questions of this section are purely thought-experiments, but some (*not* all!) parts of the answer have already been obtained in exercise 1.2 . We hope that going through this sequence mentally after understanding the routing will make things clearer for you.

1.4 The Transmission Control Protocol (TCP)

1.4.1 TCP Synchronization (handshake)

In this exercise, we use `oursock` for the first time, and capture the packet headers sent by it. The internal checkpoints of `oursock` correspond to the following:

1. Parsed input
2. Initialized some internal data
3. Initialized address information
4. Created socket
5. Connected socket
6. Sent full packets of data
7. Sent any leftover data
8. Closed connection

If you get any errors from `oursock`, it is probably an internal error you can do nothing about. Note the checkpoint where the error happened, since this information can be used to fix this problem later.

When dumping the packet headers for TCP or UDP packets, `tcpdump` uses the convention of including the port number as a fifth dotted element in the dotted decimal address string. Under normal circumstances, each packet has two sequence numbers displayed by `tcpdump` as explained in the exercise. For the acknowledgment, there is a single sequence number. It is important to note that the acknowledgment sequence number is the sequence number of the next byte the receiver expects to receive, not one that it has already received. Also, the SYN and FIN packets themselves incur sequence number increments; look closely at the output produced in this exercise to see this.

1.4.2 Sequence Numbers

In this exercise we ask you to follow the growth of sequence numbers. TCP computes the MSS taking into account the fact that 20 bytes of IP header will be incurred in each packet after TCP finishes building it. TCP sends MSS bytes of user data in each packet while it still has data to send. The actual amount of user data in a given packet can be obtained by taking the difference of the two sequence numbers provided for data packets. For your convenience, `tcpdump` lists this difference in parentheses after the starting and ending sequence numbers. To find out which

acknowledgment acts as the acknowledgment for any particular byte, we can compute its sequence number by adding the order number of the byte to the initial sequence number, and then noting the first acknowledgment that has a higher sequence number.

The version of TCP used in this lab is an implementation of TCP-SACK. For more details, read RFC 2018. However, for the purpose of doing the exercises, you should remember that TCP-SACK uses the RFC 1323 tcp timestamp options in the header. It also uses other header options to send sequence numbers for selective acknowledgments, but only when out of order packets have been received by the receiver.

This means that the size of the TCP header is variable. The length of the variable part is at minimum 3 four byte words (that is, a total of 12 bytes). This minimum occurs when no out of order packets have been seen. Thus the TCP header length of any packet is at least 12 bytes more than the normal TCP header size of 20. If the timestamp option is turned off (at a system level) then these extra 12 bytes would not need to be used in the header, and the exactly MSS bytes of user data would be transferred in each packet.

An earlier version of this document incorrectly stated that the variable header length is due to the selective acknowledgment sequence numbers. We apologize for the mistake. If you used that material to answer the question pertaining to the difference between the MSS and the actual user data transferred, you will get full credit for that question.

It is difficult to trace a much longer connection than this by hand. In lab 2, we shall attempt to capture traces of longer connections and more data transfer, and then use graphical methods to extract aggregate information.

1.4.3 Timeout and Retransmission

Note that we ran `tcpdump` without the `-S` option, so that relative sequence numbers can be seen. Only 10 bytes were to be sent before closing the connection, so that there was an attempt to close the connection before the first timeout occurred. Therefore all the retransmissions attempt to resend the same 10 bytes, but are also FIN packets.

1.5 User Datagram Protocol

In this exercise we see UDP datagrams being split into fragments by the underlying network. The relevant information is in the part of the packet header dump which looks like:

```
(frag x:y@z+)
```

This means that the packet seen on the network is a fragment of the datagram with id `x`, and contains `y` bytes of the datagram data at an offset of `z`. The `+` at the end, if present, indicates that there are more fragments for this datagram, so we expect to see this flag on every fragment but one (the last one in fragmentation order).

Note that the numbers reported all refer to user data, including the UDP header (which is user data to IP). But each packet also has an IP header 20 bytes long, so a fragment of size `y` is really an Ethernet packet of size `y + 20`. This value of this sum for the largest value of `y` is actually the MTU of the Ethernet, which should be a familiar number.

UDP adds 8 bytes of header data to the amount of data we asked it to send, as expected. And as we know, UDP is a connectionless protocol, so there is no connection setup/tearing down to be

observed.

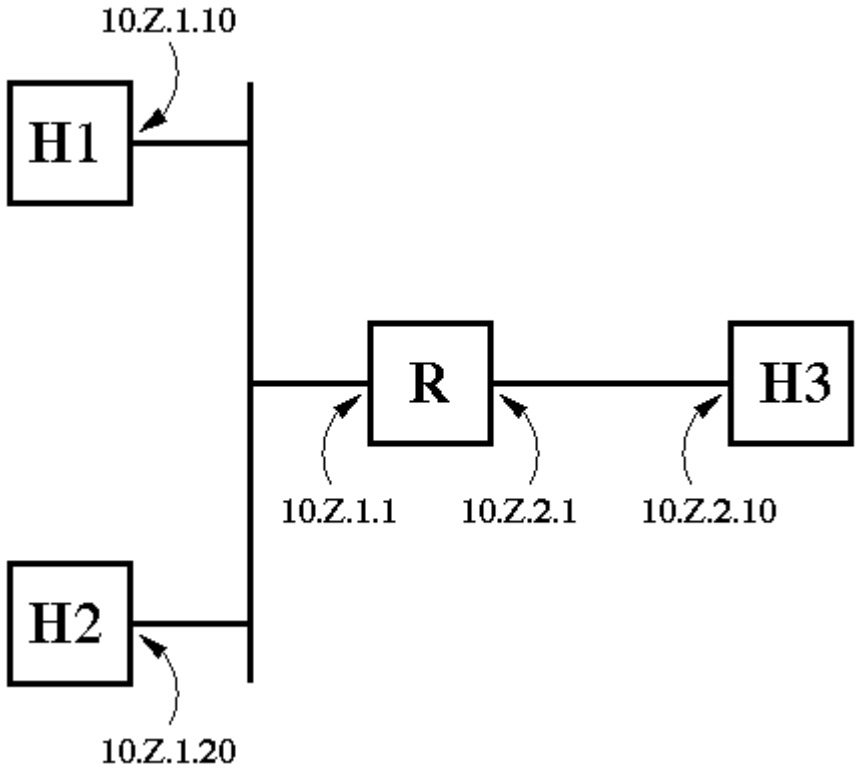
Overview | Exercises for Lab 1

Exercises for First Lab Session

Click here for Overview | Click here for Discussion on Lab 1

1.1 Becoming Familiar with the Lab Setup

A single cluster consists of Four Intel-based PCs. These are connected to form the isolated networks as shown in the following diagram.



1.1.1 The Machine H3

H3 has its own dedicated keyboard, mouse, and monitor. Log onto H3 using the group ID and password your group has received. With the help of the labels on H3, determine:

1. name (e.g. shakespeare; read from label)
2. IP address(es) (read from label)
3. identity (e.g., H1, H2, H3, or R) (match IP address in above diagram)

Make a directory called `results` under your home directory on H3 machine in your cluster if such a directory is not already there. **Place the answers to the above questions (and all remaining questions in section 1.1) in a file called `conclusions.1.1` in the `results` directory.** We suggest that you open an editor now for this file and leave it open for the duration of the exercise. To access the `emacs` editor, open an xterm window (click on Footprint, Utilities, Color XTerm) and type:

```
emacs -fg black -bg AntiqueWhite -font 10x20 mylab1.txt &
```

1.1.2 The Three PCs (H1, H2 and R)

The three PCs share the other Keyboard/Mouse/Monitor set by way of a Keyboard/Mouse/Monitor Sharing Switch (hereafter referred to as the Sharing Switch). Log onto each of the 3 PCs and, for each, determine as above:

1. name (e.g., shakespeare; read from label)
2. IP address(es) (read from label)
3. identity (e.g., H1, H2, H3, or R) (match IP address in above diagram)

Convince yourself that the labels on the Sharing Switch match the machine labels by one of the following procedures:

- ON A LINUX MACHINE: Open an XTerm window (click on Footprint, Utilities, Color XTerm) and then eject the CD-ROM by entering:

  ```
  eject
  ```

  Make sure that the CD-ROM tray ejects for the correct computer. Close the CD-ROM tray by pressing it back in gently.

1.1.3 The Two Networks

All the machines in the cluster are in the subnet 10.Z.0.0. The four machines in the cluster are then further organized into two physical networks, as seen in the above diagram. Machine R can be viewed as a router that divides the 10.Z.0.0 subnet by using the third octet of the IP address as a subnet address. In this manner, R could potentially reach 254 subnets; however, only two networks are physically connected. These are viewed as two subnets.

Based on this representation of the networking topology, determine the following for each of the two subnets in your cluster:

1. The subnet address.
2. The hosts on the network (for each host, indicate both the identity---H1, H2, H3, or R---and the IP address).
3. The network part and subnet part of the IP address.
4. The subnet mask used by machines in that subnet.

You may want to write the machine name information you have obtained on a copy of the diagram above for easy reference during the rest of the lab. **If you use different clusters on different visits to the lab for this lab session, perform exercise 1.1 once on each cluster.**

(Click here for Discussion on this exercise)

1.2 ARP: Address Resolution Protocol

Place all answers to questions in this section in a file called `conclusions.1.2` in your `results` directory.

1.2.1 Filling the ARP cache

Log onto R. Open a XTerm window.
We wish to determine the IP-to-Physical Address mappings for H1 and H2 from R. Perhaps the ARP cache at R contains these mappings. Examine the current ARP cache at R by entering the command:

```
arp -a
```

Record the result in the file you opened earlier (conclusions.1.2).

1. The ARP cache at R is likely empty because entries in the table expire after a short timeout. (If it is not empty, wait two minutes and re-examine the ARP cache. Repeat this procedure until the ARP cache is empty.) Now ping H1 by entering:

   ```
   ping <H1's IP addr>
   ```

   Re-examine the ARP cache and record H1's physical (MAC) address. Now do the same for H2: ping H2, examine the ARP cache, and record H2's physical (MAC) address.
2. Finally, examine the ARP cache to find which interface of R is connected to the Ethernet that contains H1 and H2; record the IP address of that interface.

1.2.2 Using `tcpdump` to capture packets from `ping`

Now we will use the `tcpdump` program to observe the packet traffic that is generated by the `ping` command. We will generate IP traffic on the 10.Z.2.0 subnet and capture packets at H3. Open an XTerm window at H3 for the purpose of running `tcpdump`.

1. At R, check the ARP cache to make sure that it DOES NOT contain the IP-to-Physical Address mapping for H3 (if it DOES, wait 5 minutes before proceeding).
2. At H3, start the `tcpdump` program by entering the command:

   ```
   tcpdump -n
   ```

   The `-n` option specifies that IP addresses will not be converted to names; hence the output will show only IP addresses.
3. At R, `ping` H3; watch the XTerm window at H3 to see the packets as they are captured by `tcpdump`. Once the packets have stopped arriving (approximately 5 seconds), press `Ctl-C` to halt the `tcpdump` program. Record the output.
4. The first two lines of your `tcpdump` output should be similar to the following (if NOT, wait 5 minutes then start over at part 1, section 1.2.2):

```
13:12:18.852792 arp who-has 10.4.2.10 tell 10.4.2.1
13:12:18.852962 arp reply 10.4.2.10 is-at 00:d0:bc:ed:23:64
```

```
<...time....> <..type..> <.....description.....>
```

Answer the following questions about the first 2 lines of YOUR `tcpdump` output:

1. What is the IP address of the host that broadcasts the ARP request?
2. Which IP address does the requesting host wish to resolve?
3. What is the IP address of the host that unicasts the ARP reply?
4. How many milliseconds after the ARP request is broadcast does the ARP reply appear on the Ethernet? Compute the difference between timestamps.

5. The next 8 lines of your `tcpdump` output should be similar to the following pair of lines (times 4):

```
18:08:20.918160 10.4.2.1 > 10.4.2.10: icmp: echo request
18:08:20.918396 10.4.2.10 > 10.4.2.1: icmp: echo reply
```

```
<..time..> <..source..> <..destination..> <...type...>
```

Answer the following questions about these 8 lines of YOUR `tcpdump` output:

1. How many milliseconds elapse between successive transmissions of ICMP echo requests? Compute the average of your observations.

2. How many milliseconds after the ICMP echo request is transmitted does the ICMP echo reply appear on the Ethernet? Compute the average of your observations.

Any additional lines of ARP traffic after these lines can be ignored.

6. Finally, examine the ARP cache at R and verify that the physical address listed for H3 matches the physical address given in the ARP reply of your `tcpdump` output. (If the ARP cache at R does not contain the IP-to-Physical Address mapping for H3, the timeout has expired. Simply ping H3 again to rediscover the mapping.)

(Click here for Discussion on this exercise)

1.3 Routing

In this exercise we verify that the routing table is correctly set up on each of the machines and experiment with changing the routing table. Machine R acts as a *router* in that it forwards IP packets according to its routing tables, whereas the other three machines are *hosts*. The subnet mask on all machines in the cluster is 255.255.255.0. Any two addresses differing only in the last octet will be assumed by each machine to be on the same physical network. In contrast, any two addresses differing in a higher (more significant) position will be assumed to be on different physical networks. Look at the diagram at the top of this page and convince yourself that this subnetting scheme is consistent with the network.

1.3.1 Discovering Routing

We first use three different tools to study the routing tables set up in the network machines. Questions follow at the end of the section.

Using `netstat`

- Go to H1 and run:

  ```
  netstat -rn
  ```

  The `-r` option requests current routing information (`netstat` can provide other information related to network status), and the `-n` option specifies that IP addresses should not be converted to names.
- Do the same for H2 and H3.

Put the output in files called `H1_netstat.1.3.1`, `H2_netstat.1.3.1`, `H3_netstat.1.3.1` in your `results` directory.

- Go to R and from a command line (Xterm window) run:

```
netstat -rn
```

Do not put the output in any file. Simply use the output to answer some of the questions below.

Using `traceroute`

- Go to H1 and successively run the commands:

```
traceroute <IP address of H2>
traceroute <IP address of H3>
traceroute <IP address of R>
```

For R, choose the IP address that is in the same subnet as H1.
- Do the same for H2 and H3, from each machine tracing the route to the other three machines. Each time, choose the IP address for R that is in the same subnet as the machine running `traceroute`.

Put the output in files called `H1_traceroute.1.3.1, H2_traceroute.1.3.1, H3_traceroute.1.3.1` in your `results` directory. Each file should contain the output from all `traceroute` commands that were run on that machine. Since some of the output is produced on machines other than H3, transfer these files to H3 using `ftp`, as described in the overview.

Using `ping`

- Go to H1 and successively run the commands:

```
ping -n -c 1 -R <IP address of H2>
ping -n -c 1 -R <IP address of H3>
```

The `-R` option sets the "Record Route" option on the ICMP packet, so that we receive a list of the IP addresses of network interfaces that the ICMP_ECHOREQUEST and ICMP_ECHO go through. The `-n` option is once again to specify IP addresses only, and the `-c` option specifies a single ping, or ICMP_ECHOREQUEST.

Put the output in files called `ping.1.3.1` in your `results` directory.

By looking at the `netstat, traceroute`, and `ping` outputs you have gathered, answer the following questions. **Put the answers in a file called `conclusions.1.3.1` in your `results` directory.** When asked to describe a route, do not merely quote the output of `netstat`; instead, report it in English, such as: "There is a route on this machine for packets destined for the network 10.4.0.0, and the next-hop router (gateway) is specified as 10.4.4.4".

1. Look at the routing table of H1 as reported by `netstat`. What is the default gateway (router) for H1?
2. Are there any routes in the routing table on H1 which
    1. Do not have H1 itself as the destination, and
    2. Are not default routes?

   If so, describe these routes.
3. Look at the routing table of R as reported by `netstat`. What is the default gateway (router) for R?
4. Are there any routes in the routing table on R which
    1. Do not have R itself as the destination, and
    2. Are not default routes?

   If so, describe these routes.
5. Look at the routing table of H1 and R as reported by `netstat`. What would be the route taken by a packet traveling from H1 to H2?
6. What would be the route taken by a packet traveling from H1 to H3?
7. Look at the output of the `traceroute` you ran from H1 to H2 and H3. Is that output in agreement with your two answers above? If not, what is the difference?
8. Look at the output of the `ping` you ran from H1 to H2 and H3. Is that output in agreement with your two answers above? If not, what is the difference?

1.3.2 Host Specific Routes

In this exercise we add a host specific route to the routing table of H1. Although this action will not be an improvement over the default routing that already exists, it will demonstrate how host specific routes can be used to augment a more general routing table.

- Go to H1 and run:

  ```
  route_H2_through_R
  ```

  This command causes a route to be added in the routing table of H1 stating that traffic to H2 should be forwarded to R. In particular, this command is equivalent to:

  ```
  route add -host <IP address of H2> gw <IP address of R>
  ```

  We use the customized command since `route` can only be used by the root user on all systems.
- On H1, run:

  ```
  netstat -rn
  ```

  Compare the result with the `netstat` output obtained in exercise 1.3.1, saved in the file `H1_netstat.1.3.1`.

  Put the output in a file called `netstat.1.3.2` in your `results` directory.
- On H1, run the command:

  ```
  traceroute <IP address of H2>
  ```

  Compare the result with the `traceroute` output obtained in exercise 1.3.1, saved in the file `H1_traceroute.1.3.1`.

  Put the output in a file called `traceroute.1.3.2` in your `results` directory.

- Run the command:

  ```
  clear_H2_route
  ```

  This command deletes the route we added above. In particular, it is equivalent to:

  ```
  route del <IP address of H2>
  ```

  You may run `netstat -rn` again to check that the route has been removed from the routing table. You do not need to save the result.

By looking at the `netstat` and `traceroute` outputs, answer the following questions. **Put the answers in a file called `conclusions.1.3.2` in your `results` directory.**

1. Look at the output of `netstat` and `traceroute`. Are they consistent with each other? If not, what is the difference?
2. The route to H2 now appears to consist of two hops instead of only one as before. However, earlier we did not have a specific route for H2, and the default router was (and still is) the route we specifically added for H2. How was traffic being routed to H2, so that it took less hops?

1.3.3 Defective Routes

In this exercise we attempt to demonstrate that it is easy to add routes that are defective and cause the network to stop functioning. We will add a host specific route to the routing table of H1, as before.

- Go to H1 and run the command:

  ```
  route_H3_through_H2
  ```

  Now a route has been added to the routing table of H1 which routes traffic destined for H3 through H2. In particular, this command is the equivalent of running:

  ```
  route add -host <IP address of H3> gw <IP address of H2>
  ```

- On H1, run:

  ```
  netstat -rn
  ```

  Compare the result with the `netstat` output you obtained in exercise 1.3.1, saved in the file `H1_netstat.1.3.1`.

  Put the output in a file called `netstat.1.3.3` in your `results` directory.
- On H1, run the command:

  ```
  traceroute <IP address of H3>
  ```

  If the command does not terminate in about 10 seconds, kill it with a `Ctl-C`. Compare the result with the `traceroute` output obtained in exercise 1.3.1, saved in the file `H1_traceroute.1.3.1`.

  Put the output in a file called `traceroute.1.3.3` in your `results` directory.
- Run:

```
clear_H3_route
```

This command deletes the route we added above. In particular, it is equivalent to running:

```
route del <IP address of H3>
```

By looking at the `netstat` and `traceroute` outputs, answer the following questions. **Put the answers to the following questions in a file called `conclusions.1.3.3` in your `results` directory.**

1. Were you able to trace the route from H1 to H3? If not, then no route can be established from H1 to H3 and H3 is said to be *unreachable* from H1.
2. If H3 was not reachable, explain why not. Note that we only added a route to the routing table; we did not delete any routes. All the routes that were in the routing table for exercise 1.3.3 (including the default through R) remained unchanged. Your answer should explain why H3 is not reachable using that route (the default through R), as well as why it is not reachable using the new route.

(Click here for Discussion on this exercise)

1.4 The Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) implements guaranteed and reliable delivery over the best-effort Internet Protocol (IP) services. The rest of the exercises in this lab as well as the next lab will focus on TCP.

1.4.1 TCP Synchronization (handshake)

In this exercise we observe the TCP three-way handshake in detail. In the following, `tcpdump` produces a considerable amount of information on each line (packet header dump). If you are running `tcpdump` from an xterm, you should size it so that it gives you a large number of columns to avoid line wrap. If you are running from a VT, then you probably will not encounter line wrap and there is no way to avoid it if you do.

- First go to H3 and run:

  ```
  tcpdump -n -S
  ```

  By default `tcpdump` attempts to (after the three-way handshake is seen) convert sequence numbers to a base of zero, which makes it easy to follow the progress of the connection. We shall use this feature in later exercises, but here we need to observe the sequence numbers in detail, so we use the `-S` option to turn this feature off.

  In this and later exercises, you may see some ARP packets being observed by `tcpdump`. You should ignore these lines.

  Put the output in a file called `tcpdump.1.4.1` in your `results` directory.
- Then go to H1 and run:

  ```
  oursock -tcp
  ```

  This command runs the `oursock` utility using TCP. Specifically, the `-tcp` option specifies that a connection-oriented or TCP socket is to be opened to H3. Since the command does

not specify a number of bytes to transfer, the default number of bytes (zero) will be transferred.

Your output on H1 should be similar to the following:

```
Checkpoint 1
Checkpoint 2
Checkpoint 3
Checkpoint 4
Checkpoint 5
Checkpoint 6
Checkpoint 7
Checkpoint 8
```

These are internal checkpoints that the utility runs through. (Click here for Discussion)

- Now go to H3, terminate the `tcpdump` process, and observe the output. There should be a total of seven lines of output; the first three lines represent the TCP three-way handshake which sets up the connection, while the remaining lines correspond to the termination of the connection. A SYN packet in the `tcpdump` output can be recognized by an `S` on the line. Immediately following is the initial sequence number, the same number follows once more after a colon (:).

By looking at the `tcpdump` output, answer the following questions. **Put the answers to the following questions in a file called `conclusions.1.4.1` in your `results` directory.**

1. Which port number does the `oursock` utility use on H1?
2. To which port number does it connect (i.e., the `discard` service)?
3. What is the initial sequence number chosen by `oursock`?
4. What is the initial sequence number chosen by the `discard` service?
5. What are the Maximum Segment Sizes advertised by each end of the connection?
6. Consider the FIN packets which terminate the connection. What is the arithmetic difference between the final acknowledgement sequence number sent by `discard` in the FIN packet and the initial sequence number sent by `oursock` in the SYN packet? Why?

(Click here for Discussion on this exercise)

1.4.2 Sequence Numbers

In this exercise we observe the growth of sequence numbers in detail.

- First go to H3 and run:

  ```
  tcpdump -n -S
  ```

  Put the output in a file called `tcpdump.1.4.2` in your `results` directory.
- Then go to H1 and run:

  ```
  oursock -tcp -b 15000
  ```

  This runs the `oursock` utility using TCP as before, specifying this time that 15000 bytes of user data are to be transmitted using the TCP connection before it is terminated.
- Now go to H3 and terminate the `tcpdump` process and observe the output. As before, you will see the TCP synchronization at the beginning and the connection termination at the end. The lines in between show the data transfer packets. On each line for such a packet sent from H1 to H3, after the `P` indicating data transfer, there are two numbers separated

by a colon (:). These are the sequence numbers of the earliest byte and one more than the latest byte in that packet. On each line for an acknowledgement, there is a single number after the `ack` which denotes the acknowledgement sequence number. This is the next byte that the receiver expect to receive.

By looking at the `tcpdump` output, answer the following questions.**Put the answers to the following questions in a file called `conclusions.1.4.2` in your `results` directory.**

1. What is the initial sequence number chosen by `oursock`?
2. What is the initial sequence number chosen by the `discard` service?
3. What was the time at H3 at which the acknowledgement for the 5000-th byte of user data was sent out?
4. How much data is sent in each packet by `oursock` while it has more data to send?
5. By how much does this number differ from the MSS advertised? Why?
6. Is there a packet which is both a FIN packet and also carries data? If so, explain how this is accomplished by the TCP protocol.
7. If there is such a packet, how many bytes of user data are carried in it?
8. If there are no retransmissions, the send sequence numbers should only go up, never down (ignoring the rare wrap-around). Trace the successive sequence numbers only on the packets sent from H1 to discover whether retransmission occurred or not. List the ending sequence numbers on data packets sent, and state how many retransmissions occurred together with the times they occurred.

(Click here for Discussion on this exercise)

1.4.3 Timeout and Retransmission

In this exercise we examine some details of the TCP timeout and retransmission mechanism. Such an experiment takes a significant amount of time and requires network management privileges to simulate failure. Therefore we have performed the experiment for you and have provided you with the resulting `tcpdump` output. The following output is from `tcpdump` running on machine X. After Machine X established a connection to Machine Y, the network between the endpoints of the connection went down.

```
01:21:29.142113 10.4.2.10.1117 > 10.4.1.20.9: S 1383812820:1383812820(0) win 32120
01:21:29.142717 10.4.1.20.9 > 10.4.2.10.1117: S 3785554645:3785554645(0) ack 1383812
01:21:29.142963 10.4.2.10.1117 > 10.4.1.20.9: . ack 1 win 32120  (DF)
01:21:39.165965 10.4.2.10.1117 > 10.4.1.20.9: P 1:11(10) ack 1 win 32120  (DF)
01:21:39.167876 10.4.2.10.1117 > 10.4.1.20.9: F 11:11(0) ack 1 win 32120  (DF)
01:21:42.162402 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:21:48.162362 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:22:00.162372 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:22:24.162372 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:23:12.162388 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:24:48.162387 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:26:48.162371 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:28:48.162387 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:30:48.162370 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:32:48.162390 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:34:48.162388 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:36:48.162388 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:38:48.162391 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:40:48.162388 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:42:48.162390 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
01:44:48.162387 10.4.2.10.1117 > 10.4.1.20.9: FP 1:11(10) ack 1 win 32120  (DF)
```

By looking at the above `tcpdump` output, answer the following questions. **Put the answers to**

the following questions in a file called `conclusions.1.4.3` in your `results` directory.

1. How many retransmissions were tried in all before the connection was finally terminated?
2. How much time elapsed between the original transmission of the packet and the first retransmission?
3. How much time elapsed after the network went down until the connection was terminated (that is the last retransmission was attempted)?

(Click here for Discussion on this exercise)

1.5 User Datagram Protocol

Though Labs 1 and 2 concentrate on TCP connection-oriented service, we will use UDP for comparison purposes in Lab 2. Therefore, this exercise provides a preview of UDP traffic in a `tcpdump` output. In particular, this exercise demonstrates how fragmentation works on UDP packets.

- First go to H3 and run:

  ```
  tcpdump -n
  ```

  Note that we have removed the `-S` option since this has no significance for UDP traffic.

  Put the output in a file called `tcpdump.1.5` in your `results` directory.
- Then go to H1 and run:

  ```
  oursock -udp -b 20000 -p 5000
  ```

  This command runs the `oursock` utility using UDP and specifies that 20000 bytes of user data are to be sent in datagrams, each of which can contain a maximum of 5000 bytes of user data. We therefore expect that the UDP protocol will attempt to send four datagrams, each 5000 bytes long.
- Now go to H3 and terminate the `tcpdump` process and observe the output. There should be four sets of packet dumps, each representing one of the datagrams that UDP attempted to send. The lines in each set represent fragments that were created by the underlying protocol (here, Ethernet). The last part of each line contains the datagram ID of the fragment and the fragment offset information, specifically, the number after the `frag` denotes the amount of data in this fragment and the number following the `@` after that denotes the fragment offset. Finally, one line of each set indicates the total amount of data in the packet, located immediately after the `udp` indicating UDP traffic.

By looking at the `tcpdump` output, answer the following questions. **Put the answers to the following questions in a file called `conclusions.1.5` in your `results` directory.** Except for the first question, all questions refer to the first datagram of the four; the answers would be the same for each datagram.

1. What are the datagram IDs of the four datagrams that UDP attempted to send?
2. Into how many fragments was the datagram fragmented?
3. Look at the fragment of the datagram which has offset zero. What is the port number `oursock` used on H1?
4. How many bytes of user data was the Ethernet able to accommodate in a single fragment?
5. Is this number the same as the MSS used by the TCP connection in exercise 1.4.2 ? If not,

why is it different?

6. One fragment should be smaller than all the others. What was the order of fragmentation of this smallest fragment (1,2,3,4) ? Here we are concentrating on the *fragmentation order*, as given by the fragment offset from the beginning, rather than the *transmission order*.
7. If the bytes in a datagram are numbered from 0 to 4999, in which fragment was byte 1999 of a datagram sent?

(Click here for Discussion on this exercise)

Deliverables for Lab session 1

- 1.1 - all answers in `conclusions.1.1`
- 1.2 - all answers in `conclusions.1.2`
- 1.3.1 - `netstat` outputs in `H1_netstat.1.3.1, H2_netstat.1.3.1, H3_netstat.1.3.1`, `traceroute` outputs in `H1_traceroute.1.3.1, H2_traceroute.1.3.1, H3_traceroute.1.3.1`, `ping` output in `ping.1.3.1`, answers to questions in `conclusions.1.3.1`
- 1.3.2 - `netstat` output in `netstat.1.3.2`, `traceroute` output in `traceroute.1.3.2`, answers to questions in `conclusions.1.3.2`
- 1.3.3 - `netstat` output in `netstat.1.3.3`, `traceroute` output in `traceroute.1.3.3`, answers to questions in `conclusions.1.3.3`
- 1.4.1 - `tcpdump` output in `tcpdump.1.4.1`, answers to questions in `conclusions.1.4.1`
- 1.4.2 - `tcpdump` output in `tcpdump.1.4.2`, answers to questions in `conclusions.1.4.2`
- 1.4.3 - Answers to questions in `conclusions.1.4.3`
- 1.5 - `tcpdump` output in `tcpdump.1.5`, answers to questions in `conclusions.1.5`

Click here for Overview | Click here for Discussion on Lab 1

Discussion of Exercises for Second Lab Session

Click here for Overview | Click here for Exercises for Lab 2

2.1 Slow-start and Congestion Avoidance

The performance enhancement measures of TCP are sometimes subtle and not trivial to appreciate. This is because all the performance measures are designed to be in the details of TCP's operation (that is, operate with temporal locality), and yet produce effects that determine the long-term behavior of TCP. In this exercise we see the effects of such measures when nothing untoward happens. Since retransmissions are costly, TCP attempts to send data utilizing the smallest chunks possible, and speed up this rate exponentially. This allows TCP to quickly get to the point of transmitting data at a high rate, but not lose packets at the beginning of the connection due to overestimation of the bandwidth available. In effect, the bandwidth is estimated dynamically, and the initial estimate is very small.

Rather than continue this dynamic increase until packet losses are observed, TCP avoids congestion and loss by estimating the round trip time and using the sliding window sizes. This is why the transmission rate stabilizes to a constant value (approximately) after some time. Details on this behavior can be found in your text or any reference material.

How well does this mechanism work? We can see that there are very few or no losses in the network between the TCP connection endpoints. Let us focus on the average throughput obtained by the TCP connection. Remember that the throughput we can calculate from the plot is in bytes/second. After we convert, it should be pretty close to about 8 Mbits/second, which is about the maximum we expect to get out of the 10 Mbits/sec bandwidth of the ethernet which is the limiting factor on the bandwidth. This tells us that TCP did a good job of utilizing the bandwidth without suffering losses.

2.2 Bandwidth Sharing for Multiple Connections

When two connections, similar to each other and to the above case we just considered, operate simultaneously, they effectively share the bandwidth available in the ethernet. Since both use exactly the same mechanism for estimating bandwidth, it is no surprise that both end up with the same value for bandwidth. This means that both connections will receive the same fraction of the total bandwidth utilized, which has to be 50% (to a high degree of accuracy). Thus the connections "share" bandwidth, and do so in a fair manner. An interesting point to note is that each TCP connection operates independently, without any feedback from the other. Thus for an individual TCP connection, the problem remains to determine what bandwidth it should utilize, in the presence of whatever other network traffic needs the same physical resources. But at a global level, this ends up being a fair sharing mechanism if all other traffic on the network also follows the same policy. This is why TCP is a "nice" protocol.

A more simple-minded protocol that does not make a similar effort to estimate bandwidth and does not possess the detail performance measures that lead to bandwidth sharing cannot have this "nice"characteristic. This is demonstrated in the second part of this exercise, where the second connection using the same network is a UDP connection. As we can see, the TCP

connection continues to try and estimate the bandwidth available, but the UDP connection just blasts the data it needs to send, and thus grabs the lion's share of the bandwidth. All the estimation and "niceness" costs a lot of design and implementation complexity.

You may notice that `oursock` skips the checkpoint numbered 6 when you run it in the endless mode. This is because in the endless mode there is no distinction between full packets and leftover packets. Checkpoint 7 now stands for "transmitted all data", and checkpoint 8 denotes the socket close as before. These actions are still performed when the endless send is terminated with a `Ctl-C` .

2.3 Congestion Control

The `forwarding` utility uses the `sysctl` interface to the networking functions on R to turn routing on and off. It utilizes the `nanosleep` function to implement the delays between turning routing off and on, and the delays between successive turn-off-on sequences. The amount of sleep time can be specified with a precision of nanoseconds, but of course depending on the system capabilities the accuracy of the sleep time is far less. The manual page for `nanosleep` states that any sleep time less than 2 ms will be performed with a busy wait and therefore will have a high accuracy.

The `forwarding` utility has been programmed to perform two kinds of pauses, each with a random duration. The length of the short pauses are centered on 1 ms with a maximum variation of 400 microseconds either way. The long pauses are centered on 1 second, with a maximum variation of 50 ms either way. Each pause is randomly chosen to be a long or a short pause, with the bias heavily towards a long pause at the beginning but quickly shifting to shorter pauses for subsequent pauses. The intervals between pauses are also random variables.

If `forwarding` received an error from the system attempting to sleep, it responds with a message which begins with:

```
ERROR attempting to suspend for precise time:
```

You should abort the exercise and start the exercise from the beginning if you see this message.

It may be difficult to see the slow-start behavior after each loss because of the scale of the plot. You can try to find the corresponding line in the `tcpdump` output, then create a small file by extracting that line together with about 30 lines before and about 50 lines after it, in order. Then you can run `extract_plot_data` with the small file you have produced, and then `plot_all_traffic`. This will only plot the events near the retransmission, and you will be able to observe the details of the plot.

To find the corresponding line in the `tcpdump` output, you should first go to the `allevents.list` file, and search for the time index provided by the `awk` script using the search facility of the editor you are using. Once you locate that line, note its line number. Then edit `tcpdump_output.2.3` and go to the same line number. Since the lines of `allevents.list` correspond to the lines of `tcpdump_output.2.3` very closely (only SYN, RST and pure FIN lines are eliminated), you should be able to find the line you need by inspection around this line. This is a slightly tedious process, and you are not required to do it. But it provides a better picture of the retransmission characteristics.

Click here for Overview | Click here for Exercises for Lab 2

Exercises for Second Lab Session

Click here for Overview | Click here for Discussion on Lab 2

General Remarks on Lab 2

In this second lab session, we shall concentrate on details of the TCP data transfer mechanism. In 2.1, we attempt to see the slow-start and congestion avoidance behavior of TCP under nominal conditions. In 2.2, we try to observe the bandwidth sharing properties of TCP --- the so-called "nice" behavior by which TCP attempts to utilize only a fair share of bandwidth in the presence of multiple connections over a network. We then contrast this behavior with the performance of a more simple-minded protocol, UDP. Finally in 2.3, we examine the TCP congestion control mechanism in the presence of network congestion and loss.

During these exercises, you may be asked to perform one or more steps and observe the result, but not save the output. Though there are no deliverables for these tasks, we strongly suggest you complete them, since they will facilitate your understanding of the material.

It is not essential to choose the same cluster for lab 2 as you did for lab 1. As before, inform Dr. Martin or the grading TA of the cluster you worked on. Each cluster is set up as it was during lab 1**.** Use the same userid and password as for exercise 1.

For the experiments in this lab session, we want to see packets on the physical network containing R, H1 and H2. All results must be left in your `results` directory on H3. H3 still performs the function of running the `discard` service to sink data transmitted by H1 and H2.

The `tcpdump` output in these experiments will be much larger than for lab 1 experiments. We shall use scripts that parse the output and extract aggregate data. The scripts require that the output be in the exact format generated by `tcpdump`. For this reason, **you should run `tcpdump` exactly as specified in the exercises,** redirecting the output directly to a file. If you attempt to view the output on the screen and then save it to a file using cut-and-paste or some other method, you may lose a lot of the data or make the output impossible to parse.

We shall use the graphical application `gnuplot` to view aggregate characteristics of bulk data in these experiments. We provide scripts that use `gnuplot` to produce plots on the screen. You do not need to run `gnuplot` yourself, though you can do so if you are familiar with its interactive command language. The scripts produce GIF files as well as on-screen plots; to answer some questions about the data transfer, use the `GQview` image viewer to view those images in detail. There is no need to save or submit any of the plots. The GIF files produced are quite large, so **`GQview` may take a significant amount of time (30 seconds or more) to display each file.**

The GIF files that you produce, as well as other temporary files such as `tcpdump` output, take up a great deal of disk space. At the end of these exercises, there is a list of temporary files you are likely to have produced, and you should make sure to delete them. As in Lab 1, all the *deliverable* files should be saved in your `results` directory on H3.

2.1 Slow-start and Congestion Avoidance

In this exercise we observe the slow-start mechanism of TCP, in which the data transfer rate

begins at a small value, grows exponentially, and finally levels off to avoid congestion.

- First go to H1 and run:

  ```
  tcpdump -n -tt -s 54 > tcpdump_output.2.1
  ```

  Note that we are no longer using the `-S` option because we want the convenience of having sequence numbers represent bytes, which is effectively what happens since `tcpdump` converts sequence numbers to a base of zero for each connection. In the rest of this lab, we shall mean "zero based sequence numbers", or effectively "bytes transferred on the TCP connection" when we refer to sequence numbers. We have also added the new options `-tt`, which asks for a timestamp in seconds rather than hours, minutes and seconds, and `-s`, which specifies the number of bytes to capture from each packet header. 54 bytes are enough for our purposes, and reducing the number of bytes captured (from the default of 68) reduces the chance of `tcpdump` dropping packets.
- From another xterm or VT session on H1, run:

  ```
  oursock -tcp -b 500000
  ```

  After this command ends, terminate the `tcpdump` you started using a `Ctl-C`.
- From the same directory you were in while running `tcpdump`, run:

  ```
  extract_plot_data tcpdump_output.2.1
  ```

  This command runs several `awk` scripts on the `tcpdump` output, and creates five files. The `tcpdump` output must be found in the current directory, and the files which are created are placed in the same directory. Of these, the main one is called `allevents.list` --- this contains almost the same information as the `tcpdump` output itself, but better formatted for subsequent scripts. You can view this file; there are at most five fields separated by a single space on each line, the first is time in seconds from the time when the first TCP handshake was seen to be completed, the next two are source and destination IP address with ports, the rest are protocol specific information for TCP send, TCP ack and UDP packets in the same format as `tcpdump` output.

  The rest of the files each contain data in two columns. The first column of each contains time in seconds from the same zero base. The second column contains data to be plotted against time. The data in this column for the different files are:
  1. `H1_send.dat` - Send sequence numbers for the connection from H1 to H3.
  2. `H1_ack.dat` - Acknowledgment sequence numbers for the connection from H1 to H3.
  3. `H2_send.dat` - Send sequence numbers for the connection from H1 to H3, or cumulative number of bytes, if the data transfer used UDP.
  4. `H2_ack.dat` - Acknowledgment sequence numbers for the connection from H2 to H3.

  The files for H2 are empty if no data packets were flowing from H2 to H3, as in this experiment. The acknowledgment file for H2 is empty if data packets are flowing from H2 to H3, but using the UDP protocol. Any files by the above names already existing in the current directory are deleted, whether new files are produced or not.

  When running this command, you may see one or more messages of the form:

  ```
  sequence number reversal at time index 3.616185
  ```

  This message indicates that the `awk` script has detected what is probably a TCP

retransmission. It is not very likely that you will see this message before we get to exercise 2.3. In that exercise, we shall utilize this information to look into the details of the retransmissions. In all the other exercises, you should ignore this message if you see it.

Open each of the above mentioned files in your current directory and examine the first few lines in each file and compare them. Convince yourself that you see data packets where they may be expected from the above description, and that they are consistent.

Put the `allevents.list` file as a file called `allevents.list.2.1` in your `results` directory.

- From the same directory, run:

```
plot_all_traffic
```

This command runs `gnuplot` on all the data files produced above. The result is displayed on the screen as well as saved to a GIF file. The plots produced consist of lines joining the points obtained by plotting the sequence numbers against time. To obtain a plot in which the actual data points are highlighted as well as joined by line segments, run `plot_all_traffic -points` . This is useful only for data sets containing comparatively few points, such as this exercise. For the much larger number of points obtained in later exercises, this option is not useful because the points cannot be rendered separately.

For each plot actually displayed on the screen, a GIF file is also produced. You can view the GIF files using the `GQview` application. To access this image-viewing application, click on Footprint, Graphics, GQview. Once you open the GIF file, you can expand the window or choose the full screen display mode from the menubar. To return from the full screen display, press the `Esc` key. You can choose to zoom in to observe some specific area of the plot. Clicking and dragging the image area allows you to move about and examine different areas of the plot.

For this exercise, we recommend using `GQview` to examine the plot in detail.

A maximum of three plots are drawn (and three GIF files are produced) from one run of `plot_all_traffic`. These are:

1. `H1_sendack.gif` - Send and acknowledgment sequence numbers for the connection from H1 to H3.
2. `H2_sendack.gif` OR `H2_send.gif` - Send and acknowledgment sequence numbers for the connection from H2 to H3, or the cumulative bytes transferred if the connection is UDP.
3. `H1&H2_send.gif` - Send sequence numbers for the connection from H1 to H3, and either the send sequence numbers or cumulative bytes transferred (if the transfer used UDP) for the connection from H2 to H3.

In all the plots, bytes of transferred data are plotted against time in seconds. Only plots for which the corresponding data are available are produced. Any files by the above names already existing in the current directory are deleted, whether new files are produced or not. In this exercise, you should see just one plot, for the send and acknowledgment sequence numbers for the TCP connection from H1 to H3.

By looking at the `allevents.list` file produced, or the plot, or both, answer the following questions. **Put the answers to the following questions in a file called `conclusions.2.1` in your `results` directory.**

1. We specified that 500000 bytes of data be transferred using the TCP connection. Verify from the plot how many bytes were actually transferred.
2. From the plot, what is the approximate time it took to transfer the above amount of data?
3. Verify the above two numbers from the `allevents.list` file.
4. From the above two numbers, what is the average rate of data transfer of the TCP connection? This should also be the average slope of the plot for sending sequence numbers.
5. At a large scale, ignoring local variations, what is the behavior of the plot for bytes transferred? For example, is it a straight line, or is it growing or decaying, or what?
6. What does this say about the long term characteristics of the TCP connection?
7. What is the nature of the plot for the acknowledgment sequence numbers in relation to the sending plot (above, below, parallel, converging, diverging, ...) ?
8. For this and the rest of the questions in this exercise, you need to view details of the plot using `GQview`. If you have not already done so, you probably also want to regenerate the plot with points by running the `plot_all_traffic` command again with the `-points` option.
   The vertical difference between the two plots is the amount of outstanding data which the sending side has transmitted but which has not yet been acknowledged by the receiving side. This should correspond to the receiver window size.
   Estimate the average vertical difference between the two plots, expressed in bytes.
9. View the `tcpdump_output.2.1` file. What is the receiving window size advertised in the return SYN packet? Is your answer to the previous question close to this window size?
10. Focus on the first few data points in the graph. You should see bursts of packets sent by the sending TCP. In this context a burst refers to a number of packets sent almost at the same time, or back-to-back. The burst size starts at a small value, like 1 or 2, and for some time predominantly increases. Then the burst size stabilizes at a large value, though it will experience small fluctuations throughout the life of the connection.
    At what value, approximately, does the burst size stabilize?
11. When the burst size stabilizes, TCP switches from slow-start to congestion avoidance mode. How many seconds into the life of the connection does this happen?
12. During this period of slow-start, what is the behavior of the plot for acknowledgment sequence numbers with respect to the sending sequence number plot?

(Click here for Discussion on this exercise)

2.2 Bandwidth Sharing for Multiple Connections

In this exercise we observe bandwidth sharing between two connections. First we study the case where both connections use TCP, then the case where one uses TCP and the other UDP.

2.2.1 TCP Connections Only

- First go to H1 and run:

  ```
  tcpdump -n -tt -s 54 > tcpdump_output.2.2.1
  ```

- From another xterm or VT session on H1, run:

  ```
  oursock -tcp -e
  ```

  In this exercise we use the `-e` option of `oursock` which sends an endless stream of data, instead of a specific number of bytes as we have been sending so far.

- Go to H2 and run:

  ```
  oursock -tcp -b 10000000 -d 5
  ```

  The `-d` option specifies a wait of 5 seconds after the connection is established and before data transmission begins. Thus this command asks `oursock` to establish a TCP connection to H3, wait 5 seconds, send ten million bytes of data, and then close the socket and end.

  After this command ends, go to H1. Wait approximately 5 seconds. (As little as 2 seconds is probably enough to obtain a good plot, but much longer than 7 seconds is going to make the plot less readable.) First terminate the `oursock` sending the endless stream, and then terminate the `tcpdump` you started, using `Ctl-C` for each termination.
- From the same directory from which you ran `tcpdump`, run:

  ```
  extract_plot_data tcpdump_output.2.2.1
  ```

  This command will run the `awk` scripts we mentioned before. This time we should see data in all four `.dat` files, since H1 and H2 each had a TCP connection to H3.

  Open each `.dat` file and verify that it contains data.
- From the same directory, run:

  ```
  plot_all_traffic
  ```

  This time you should see three plots on the screen. The corresponding GIF files are also produced, though they are not needed since we are not interested in details for this exercise.

By looking at the `allevents.list` file produced, or the plots, or both, answer the following questions. **Put the answers to the following questions in a file called `conclusions.2.2.1` in your `results` directory.**

All questions for this exercise as well as exercise 2.2.2 refer to the large scale characteristics of the plot obtained. For the purpose of these exercises, we divide the plot into three regions: (A) the period from the beginning of the plot until the second connection starts up, (B) the period when two connections operate simultaneously and (C) the period after the second connection ends until the end of the plot. We ask only for approximate answers which you can read off the plot.

1. How many bytes of data does the single TCP connection transfer in region A?
2. What is the data transfer rate for the single connection in region A?
3. How many bytes of data does the first TCP connection transfer in region B?
4. What is the data transfer rate for the first connection in region B?
5. How many bytes of data does the second TCP connection transfer in region B?
6. What is the data transfer rate for the second connection in region B?
7. How many bytes of data does the first TCP connection transfer in region C?
8. What is the data transfer rate for the first connection in region C?
9. What is the total data transfer rate of the two TCP connections in region B?
10. Is this rate less than, equal to or more than the data transfer rate of the first connection only in region C?
11. What would you say is the reason for the answer to the above question?
12. What is the percentage of total bandwidth utilization that the first TCP connection received during region B?

2.2.2 TCP and UDP

- First go to H1 and run:

  ```
  tcpdump -n -tt -s 54 > tcpdump_output.2.2.2
  ```

- From another xterm or VT session on H1, run:

  ```
  oursock -tcp -e
  ```

- Go to H2 and run:

  ```
  oursock -udp -b 10000000 -p 1000 -d 5
  ```

  This time we use UDP to transfer the ten million bytes of data, and also specify that they should be sent in UDP packets each containing 1000 bytes of user data.

  After this command ends, go to H1. Wait approximately 5 seconds. First terminate the `oursock` sending the endless stream, and then terminate the `tcpdump` you started.
- From the same directory from which you ran `tcpdump`, run:

  ```
  extract_plot_data tcpdump_output.2.2.2
  ```

  This time we should see data in both the send and acknowledgment files for H1, since H1 had a TCP connection to H3, but only in the send and *not* the acknowledgment file for H2, since UDP does not use acknowledgments.

  Open each `.dat` file and verify that the above is true.
- From the same directory, run:

  ```
  plot_all_traffic
  ```

  You should again see three plots on the screen. This time the plot for H2 shows only cumulative bytes sent, since there are no acknowledgments to plot. Once again, we are not interested in details for this exercise.

By looking at the `allevents.list` file produced, or the plots, or both, answer the following questions. **Put the answers to the following questions in a file called `conclusions.2.2.1` in your `results` directory.**

1. How many bytes of data does the single TCP connection transfer in region A?
2. What is the data transfer rate for the single connection in region A?
3. How many bytes of data does the TCP connection transfer in region B?
4. What is the data transfer rate for the TCP connection in region B?
5. How many bytes of data does the UDP connection transfer in region B?
6. What is the data transfer rate for the UDP connection in region B?
7. How many bytes of data does the TCP connection transfer in region C?
8. What is the data transfer rate for the TCP connection in region C?
9. What is the total data transfer rate of the two connections in region B?
10. What is the percentage of total bandwidth utilization that the TCP connection received in region B?

(Click here for Discussion on this exercise)

2.3 Congestion Control

In this exercise we observe how TCP deals with lost packets, in the presence of network congestion. To simulate network congestion, we shall cause some packets to be dropped at the router R by simply refusing to forward them. This loss is detected by the sending TCP when the acknowledgments for lost packets do not come back; TCP then retransmits these packets.

It is difficult to observe this behavior in a small network with no background traffic and no real chance of loss, either in the medium or in a buffer at an intermediate router. It is also unlikely that we shall be able to observe the loss of a single packet (which would allow us to see SACK-specific behavior); to see such a loss, we would need to observe packets at the kernel level and make decisions to drop or route them based on the experiment we wish to perform. Since our mechanism for dropping packets is simply to turn packet forwarding on and off, it is likely that we shall always see multiple packet losses. Accordingly, we expect to see TCP going back to slow start every time a loss occurs.

- **Preparatory Step:** In this step we become familiar with the `forwarding` utility, which serves as our means of turning forwarding on and off at R. This is a utility developed specifically for this lab. Normally there would not be a similar utility on a UNIX system, and no user except the superuser would have the privilege of turning routing on and off.
  1. Go to H1 and run:

     ```
     ping <IP address of H3>
     ```

     This will start an endless series of ping's from H1 to H3, so you should see output consisting of multiple lines that are each of the form:

     ```
     64 bytes from 10.4.1.10: icmp_seq=2 ttl=255 time=0.5 ms
     ```

     You should see one new message every second, and the ICMP sequence number should go up by one each time. Leave this process running.
  2. Go to R and run:

     ```
     forwarding -off
     ```

     This uses the `forwarding` utility to simply turn all IP packet forwarding off at R. You should see the following message, indicating that IP packets are no longer being forwarded:

     ```
     net.ipv4.ip_forward = 0
     ```

  3. Go to H1, and observe the `ping` process. You will see that the process has stopped providing any output. Convince yourself that this is the case (you should wait several seconds).
  4. Go back to R and turn routing back on with the command:

     ```
     forwarding -on
     ```

     You should again see a message, this time indicating that packet forwarding has been turned on.
  5. Go to H1, and observe that the `ping` process has again started providing output. There should be a discontinuity in the ICMP sequence numbers when the output restarts. The missing numbers correspond to ICMP_ECHOREQUEST packets

which were sent from H1 but never reached H3 because they were not forwarded at R. Naturally, there were no ICMP_ECHO packets corresponding to these sent from H3 to H1, and the `ping` process marked these as lost packets.

6. Repeat the above four steps as many times as necessary to convince yourself that you can turn packet forwarding at R on and off using the `forwarding` utility.
7. Go to R and run:

   ```
   forwarding -n 10
   ```

   This time we are using the `-n` option to specify that the forwarding is to be turned off and then back on, and that this sequence is to be repeated 10 times. The time the forwarding remains turned off each time and the time between successive turnoffs are random variables. This process always exits with the forwarding turned on, even on an interrupt.
8. Once this command terminates, go to H1 and see if there are any missing ICMP sequence numbers in the `ping` output, which would indicate that an ICMP_ECHOREQUEST or ICMP_ECHO packet was lost during one of the 10 times that forwarding was turned off at R. (You may well not see such a phenomenon, since the average time the forwarding remains off in this mode is very small, and hence it is unlikely that it will coincide with one of the ICMP_ECHOREQUEST or ICMP_ECHO packets.)
9. Terminate the `ping` process on H1 using `Ctl-C`. Now we are ready to observe the behavior of TCP under similar disruptions in packet forwarding.

- First go to H1 and run:

  ```
  tcpdump -n -tt -s 54 > tcpdump_output.2.3
  ```

- From another xterm or VT session on H1, run:

  ```
  oursock -tcp -e
  ```

- Go to R and run:

  ```
  forwarding -n 3
  ```

  After this command ends, go to H1. First terminate the `oursock` sending the endless stream, and then terminate the `tcpdump` you started, using `Ctl-C` for each termination.
- From the same directory from which you ran `tcpdump`, run:

  ```
  extract_plot_data tcpdump_output.2.3
  ```

  This time we expect to see some packets lost and retransmitted. In the `allevents.list` file, these can be spotted easily by noting the successive sending sequence numbers. Normally we expect the sending (either the beginning or ending) sequence number in successive packets to have successively larger values. On a retransmission, however, TCP resends a packet already transmitted once, so this packet would have a lower sequence number than the immediately preceding one. The `extract_plot_data` script flags these occurrences with messages of the form:

  ```
  sequence number reversal at time index 7.335743
  ```

  This enables you to look in the `allevents.list` file and quickly locate the retransmitted packets. The time index provided is the first field in each column.

Put the `allevents.list` file as a file called `allevents.list.2.3` in your `results`

directory.

Put the output of the `extract_plot_data` command above as a file called `sequence_reversals.2.3` in your `results` directory.

- From the same directory, run:

```
plot_all_traffic
```

For this exercise, we recommend using `GQview` to examine the plot in detail.

By looking at the `allevents.list` file, the list of sequence number reversals and the plots, answer the following questions. **Put the answers to the following questions in a file called `conclusions.2.3` in your `results` directory.**

1. How may sequence number reversals do you see?
2. Using the time indices you have saved in the file `sequence_reversals.2.3`, locate the TCP retransmissions in the plot. Each of the retransmissions should show up as a pause in the upward trend of the plot, with a small or large flat region which is actually sloping slightly downward. From the plot, list the width of the flat region for each of the retransmission regions you see.
3. Some of the retransmission regions are comparatively large and some are small. You should see at least one large region. (If you see no large region, you should abandon this output, and start this exercise again.) There are multiple attempts at retransmission in a large region, whereas the first attempt at retransmission succeeds for the small regions. For each large region, find the corresponding time index in the `allevents.list` file, and list the number of retransmissions attempted by TCP during that region, together with the time indices of the retransmission attempts.
4. Pick a particular retransmission region. Look at the segment of the plot immediately following the "flat" region. What is the relative behavior of the send and acknowledgment sequence number plots in this segment?
5. Is this relative behavior more like the long term behavior or more like the slow-start behavior of the plot you obtained in exercise 2.1 ?

(Click here for Discussion on this exercise)

Deliverables for Lab session 2

- 2.1 - the eventlist in `allevents.list.2.1`, all answers in `conclusions.2.1`
- 2.2.1 - all answers in `conclusions.2.2.1`
- 2.2.2 - all answers in `conclusions.2.2.2`
- 2.3 - the eventlist in `allevents.list.2.3`, the list of sequence number reversals in `sequence_reversals.2.3`, all answers in `conclusions.2.3`

Cleanup

Please delete all temporary files you produced during Lab 2. In particular, delete the following files from your home directory on H1 (or elsewhere), if they exist:

- 2.1 - the `tcpdump` output `tcpdump_output.2.1`
- 2.2.1 - the `tcpdump` output `tcpdump_output.2.2.1`
- 2.2.2 - the `tcpdump` output `tcpdump_output.2.2.2`

- 2.3 - the `tcpdump` output `tcpdump_output.2.3`
- General - the plot data and plot files `*.list, *.dat, *.gif`

Click here for Overview | Click here for Discussion on Lab 2