Computer Communications and Networks (COMN) Course, Spring 2013

Coursework

The overall goal of this assignment is to develop application layer support for end-to-end reliable data delivery over the unreliable datagram protocol (UDP) using the sliding window approach. In particular, you are asked to implement in Java three different sliding window protocols – *Stop-and-Wait*, *Go Back N* and *Selective Repeat* – at the application layer using UDP sockets. Note that the stop-and-wait protocol can be viewed as a special kind of sliding window protocol in which sender and receiver window sizes are both equal to 1. For each of the three sliding window protocols, you will implement the two protocol endpoints referred to as *sender* and *receiver* respectively; these endpoints also act as application programs. Data communication is *unidirectional*, requiring transfer of a *large file* from the sender to the receiver *as a sequence of smaller messages*. The underlying link is assumed to be *symmetric* in terms of bandwidth, delay and loss rate characteristics.

To test your protocol implementations and study their performance in a controlled environment on DICE machines, you will need to use the *Dummynet* link emulator [2]. Specifically, the sender and receiver processes for each of the three protocols will run *within the same (virtual) machine* and communicate with each other over a link emulated by *Dummynet*. For this assignment, you only need the basic functionality of *Dummynet* to emulate a link with desired characteristics in terms of bandwidth, delay and packet loss rate.

More specifically, you will use the virtual machine (VM) created on a DICE machine in the labs, following the instructions in [1] for running your implementations and evaluating the performance of different protocols. The VM so created (*dummynetSL6*) can be used from any DICE machine, has *Dummynet* included and you will be able to configure it using ipfw commands. More on this shortly. Since *dummynetSL6* VM does not include Eclipse, we suggest you develop your protocol implementations outside it and save them within the *dummynetshared* subdirectory of your assignment directory. That way, the files will be accessible from within the VM via mount

command described under "Shared folder" in [1]. Note that the Java compiler (javac) and application launcher (java) are installed as part of the *dummynetSL6 VM*, so you should be able to compile and run your code from inside the VM. You can use the /work space within the *dummynetSL6* VM for storing any temporary files you would like to keep across various executions of the VM.

Once the above one-time setup part is done, you can configure and use the *dummynet* to realize an emulated link between two communicating processes (e.g., your sender and receiver application programs) inside the DummynetSL6 VM. For example, to create a symmetric 1Mbps emulated link with 100ms one-way propagation delay and 1% packet loss rate, you create two dummynet pipes for each direction and configure them as follows (as root):

```
% ipfw add pipe 100 in
% ipfw add pipe 200 out
% ipfw pipe 100 config delay 100ms plr 0.005 bw 1Mbits/s
% ipfw pipe 200 config delay 100ms plr 0.005 bw 1Mbits/s
```

You can verify this configuration by using the following commands:

```
% ipfw list
% ipfw pipe 100 show
% ipfw pipe 200 show
```

You can use the following command to flush all previous configuration rules:

```
% ipfw flush
```

Note that in the above, the pipe identifiers (100 and 200) are arbitrarily chosen. You could instead use different numbers and still get the same effect. If a configuration for a pipe needs to be updated, then you reissue the corresponding "config" command with the modified value(s). For example, if you want the bandwidth for pipe 200 (corresponding to the outgoing direction of traffic) to be changed to 10Mbps instead, then you run the following command:

```
% ipfw pipe 200 config delay 100ms plr 0.01 bw 10Mbits/s
```

A few additional notes about *Dummynet* and *dummynetSL6* VM follow. Note that we assume packets are not corrupted in transit (i.e., no bit errors) via the *Dummynet* emulated links, so there is no need to implement error detection functionality such as checksum at the endpoints. Whole packets, however,

can be lost over the emulated link as determined by the packet loss rate setting when configuring the emulated link using *dummynet* (i.e., plr portion of the ipfw command above). Also note that in the above example, we implicitly assumed the packet loss rate in the forward and reverse directions of the link are equal to 0.5% to make up a total link packet loss rate of 1%, but the same result can be achieved by having any combination of packet loss rates for the forward and reverse directions that add up to 1%. For more information on D*ummynet*, please refer to [2] and the dummynet website.

Besides *Dummynet*, *dummynetSL6* VM has other networking utilities you may find useful in relation to this assignment. These include:

- iperf
- thrulay
- netcat
- Wireshark
- tcpdump
- tcptrace

.

Note that these tools are explicitly mentioned so that you know they are available to use. Except for *iperf*, you are not required to use the rest of them for this assignment.

Your implementation needs to follow the sequence of steps described below.

Step 1: Basic framework (large file transmission under ideal conditions):

Implement sender and receiver endpoints for transferring a large file given at [3] from the sender to the receiver on localhost over UDP as a sequence of small messages with 1KB payload (NB. 1KB = 1024 bytes) via Dummynet emulated link configured with ***10Mbps bandwidth, 10ms one-way propagation delay and 0% packet loss rate***. The exchanged data messages must also have a header part for the sender to include a 16-bit message sequence number (for duplicate filtering at the receiver in subsequent steps) and a bit/byte flag to indicate the last message (i.e., end-of-file). Name the sender and receiver developed in this step as ***Sender1.java*** and ***Receiver1.java*** respectively.

Step 2: Stop-and-Wait

Extend sender and receiver applications from the previous step to implement a stop-and-wait protocol as described in [5], specifically rdt3.0. Call the resulting two applications ***Sender2.java*** and ***Receiver2.java*** respectively. This step requires you to define an acknowledgement message that the receiver will use to inform the sender about the receipt of a data message. Discarding duplicates at the receiver end using sequence numbers is also needed. Test the duplicate detection functionality using a small retransmission timeout on the sender side.

Using a **10% packet loss rate** and rest of Dummynet emulated link configuration parameters as in the previous step (i.e., **10Mbps bandwidth and 10ms one-way propagation delay**), experiment with different retransmission timeouts. Tabulate the observed number of retransmissions in the space provided under Question 1 in the results sheet provided in [4].

Also study the impact of retransmission timeouts on the average throughput and answer Question 2 in [4]. For this, modify your sender implementation to measure average throughput (in KB/s) which is defined as the ratio of file size (in KB) to the transfer time (in seconds). Transfer time in turn can be measured at the sender as the interval between first message transmission time and acknowledgement receipt time for last message. Before the sender

application finishes and quits, print the average throughput value to the standard output.

Step 3: Go-Back-N

Extend Sender2.java and Receiver2.java from the previous step to incorporate the Go-Back-N strategy as described in [6], by allowing the sender window size to be greater than 1. *Use the "optimal" value for the retransmission timeout* obtained from the previous step. Experiment with different window sizes at the sender (increasing in powers of 2 starting from 1) and **different one-way propagation delay values** (**10ms, 100ms and 500ms**) in the emulator. Across all these experiments, use the following values for the other emulator parameters: **10Mbps bandwidth and 1% packet loss rate**. Tabulate your results under Question 3 and answer Question 4 in [4]. Name the sender-receiver pair implemented in this step as ***Sender3.java*** and ***Receiver3.java***.

Step 4: Selective Repeat

Extend Sender3.java and Receiver3.java to implement selective repeat strategy as described in [7]. Call the resulting two applications as ***Sender4.java*** and ***Receiver4.java*** respectively. By configuring the emulator with **10Mbps bandwidth, 100ms one-way propagation delay and 1% packet loss rate**, experiment with different window size values and complete the table under Question 5 and answer Question 6 in [4].

As a part of this step, also carry out an equivalent experiment using iperf with TCP within the dummynetSL6 VM, i.e., both iperf client and server running inside it. Use –M option in iperf to set the maximum segment size to 1KB and vary the TCP window sizes using the –w option. Note that TCP actually allocates twice the specified value and uses the additional buffer for administrative purposes and internal kernel structures. But this is normal because effectively TCP uses the value specified as the window size for the session, which is the parameter to be varied in this experiment. You also need to specify the file to be transferred (i.e., the one given at [3]) as one of the parameters to iperf on the client side. Use the results of this experiment to complete the table under Question 7 and answer Question 8 in [4].

Implementation Guidelines

Your programs must adhere to the following standard with both sender and receiver application programs to be run inside the dummynetSL6 VM:

- The sender program must be named as specified below and must accept the following options from the command line:
  java SenderX localhost <Port> <Filename> [RetryTimeout] [WindowSize]
  where <Port> is the port number used by the corresponding receiver. <Filename> is the file to transfer. The RetryTimeout should be specified for steps 2-4, whereas WindowSize option is only relevant for steps 3 and 4. For example:
  java Sender1 localhost 54321 sfile
- The receiver program must be named as specified below and must accept the following options from the command line:
  java ReceiverX <Port> <Filename> [WindowSize]
  where <Port> is the port number which the receiver will use for receiving messages from the sender. <Filename> is the name to use for the received file to save on local disk. The WindowSize is relevant only for step 4 as it is implicitly equal to 1 for steps 2 and 3. For example:
  java Receiver1 54321 rfile
- Please start each source file with the following comment line:
  /* Forename Surname MatriculationNumber */
  For example:
  /* John Doe 1234567 */
- Please use comments in your code!

Submission and Assessment

The deadline for this coursework is 4pm on Tuesday, 26th March 2013. No late submissions are allowed, except under extenuating circumstances. You must submit an electronic version of your implementations for steps 1 – 4 (including Sender1.java, Receiver1.java, Sender2.java, Receiver2.java, Sender3.java, Receiver3.java, Sender4.java, Receiver4.java), corresponding executable and completed results sheet [4] (as DOC, ODT or PDF) using the submit command as follows: *submit comn cw1 <directory-name>*. You are expected to work on this coursework on your own. Any kind of copying will result in punitive action.

This coursework accounts for the whole of your coursework mark (or, 40% of the overall course mark). Distribution of marks among the various steps is as follows:

- 10% of the *coursework mark* for successful completion of step 1.
- 30% of the coursework mark for successful completion of steps 1 and 2.

- 60% of the coursework mark for successful completion of steps 1, 2 and 3.
- 100% of the coursework mark for successful completion of steps 1, 2, 3 and 4.

References

1. VirtualBox VM Setup Instructions
2. M. Carbone and L. Rizzo, "Dummynet Revisited," *SIGCOMM Computer Communication Review,* Vol. 40, No. 2, pp. 12-20 Apr 2010.
3. Test file to be used for this assignment
4. Results Sheet
5. Section 3.4.1 in J. F. Kurose and K. W. Ross, "Computer Networking: A Top-Down Approach" (5th edition), Pearson Education, 2009.
6. Section 3.4.3 in J. F. Kurose and K. W. Ross, "Computer Networking: A Top-Down Approach" (5th edition), Pearson Education, 2009.
7. Section 3.4.4 in J. F. Kurose and K. W. Ross, "Computer Networking: A Top-Down Approach" (5th edition), Pearson Education, 2009.