Computer Communications and Networks (COMN), 2018/19, Semester 2

Assignment

Overview

The overall goal of this assignment is to implement and evaluate different protocols for achieving end-to-end reliable data transfer at the application layer over the unreliable datagram protocol (UDP) transport protocol. 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 henceforth 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 over a link as a sequence of smaller messages. The underlying link is assumed to be *symmetric* in terms of bandwidth and delay 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 [1]. 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.

Virtual Machine Setup

More specifically, you need to setup a virtual machine (VM) using your DICE accounts, following the instructions in [2], to run your protocol implementations and evaluate their performance. The VM so created (*DummynetSL6*) can be used from any DICE machine and comes with *Dummynet* pre-installed. You will be able to configure *Dummynet* using `ipfw` command. 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 [2]. You should however be able to compile and run your code from inside the VM as the Java compiler (`javac`) and application launcher (`java`) are installed as part of the *dummynetSL6* 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.

Link Emulation using Dummynet

Once the above one-time VM 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 programs) inside the *DummynetSL6* VM. For example, to create a symmetric 1Mbps emulated link with 5ms one-way propagation delay (thus, 20ms in total considering both directions when the sender and receiver are in the same host; see why in the important note section below) and 0.5% packet loss rate for each direction (thus, 1% packet loss rate in total), 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 5ms plr 0.005 bw 1Mbits/s
% ipfw pipe 200 config delay 5ms 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 5ms plr 0.005 bw 10Mbits/s
```

A few additional notes about *Dummynet* and *DummynetSL6* VM follow. **Note that packets are not corrupted in transit (i.e., no bit errors) via the Dummynet emulated link, 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 (plr) setting when configuring the emulated link using *Dummynet*. For more information on *Dummynet*, please refer to [1] and the *Dummynet website*.

Important Note: In *Dummynet*, a total round-trip propagation delay would be twice as large as the specified delay when both sender and receiver are in the same host that enforces the delay. Consider a case where a sender sends a packet to a receiver and the receiver sends the packet back to the sender. Given that two pipes (one "in" pipe and one "out" pipe) are enabled, the packet goes through the "in" and "out" pipes once from the sender to the receiver, and again traverses those two pipes when it travels back to the sender.

```
% ipfw pipe 100 config delay 5ms plr 0.005 bw 1Mbits/s
% ipfw pipe 200 config delay 5ms plr 0.005 bw 1Mbits/s
```

Given that both sender and receiver are in the *DummynetSL6* VM, a total round-trip propagation delay is **20ms, not 10ms**, in the above example. Each part of the assignment specification below states the *Dummynet* configuration parameters. It is important to set the configuration parameters as described in order to do the assignment correctly.

Besides *Dummynet*, *DummynetSL6* VM has other networking utilities that you may find useful while working on this assignment. These include:

- iperf
- thrulay
- netcat

- Wireshark
- tcpdump

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.

Detailed Assignment Specification

The assignment needs to be done in two parts. The second part builds on the first part. Each part is further divided into two sub-parts as detailed below.

Assignment Part 1

Part 1a: 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 maximum payload (NB. 1KB = 1024 bytes) via *Dummynet* emulated link with two pipes (one "in" and one "out"), **each of which is configured with 10Mbps bandwidth, 5ms one-way propagation delay and 0% packet loss rate** (i.e., no packet loss). In this configuration, a total round-trip propagation delay is 20ms (See the Important Note part above in Link Emulation using Dummynet section).

In the sender code, insert, at a minimum, a 10ms gap (i.e., sleep for 10ms) after each packet transmission. The *Dummynet* sets a queue size of 50 as default (100 at a maximum). Because the sending rate from the sender is typically larger than the link speed (10Mbps) specified here, the queue is likely to overflow and hence packets losses are unavoidable. To allow the test of an ideal, reliable channel case, the 10ms gap is suggested. If packet losses continue to occur, increase the time gap. **Note that inserting the time gap is only for Part 1a. From Part 1b onwards, the sleeping part (the 10ms time gap) should be removed from the sender.**

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 the next and subsequent parts) and a byte flag to indicate the last message (i.e., end-of-file). That is, the following header format should be used:

Offset	Octet	0	1	2	3 ~ up to 1026
		16-bit sequence number		8-bit EoF flag	Data

Name the sender and receiver developed in this part as **Sender1a.java** and **Receiver1a.java** respectively. The receiver should store the transmitted data (after removing header from packet) into a local file (See Implementation Guidelines section below for more details).

- Sender program must be named as specified below and must accept the following options from the command line:
  `java Sender1a <RemoteHost> <Port> <Filename>`
  `<RemoteHost>` is IP address or host name for the corresponding receiver. Note that if both sender and receiver run on the same machine, `<RemoteHost>` can be specified as either 127.0.0.1 or localhost.
  `<Port>` is the port number used by the receiver.
  `<Filename>` is the file to transfer.
  For example: `java Sender1a localhost 54321 sfile`
- Receiver program must be named as specified below and must accept the following options from the command line:
  `java Receiver1a <Port> <Filename>`
  `<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.
  For example: `java Receiver1a 54321 rfile`
- Expected output: A successfully transferred file to the receiver; both sent and received files must be identical at a binary level.

Part 1b: Stop-and-Wait

Extend sender and receiver applications from *Part 1a* to implement a stop-and-wait protocol described in section 3.4.1 in [4], specifically rdt3.0. [**Hint**: You need two finite state machines (FSMs); one for rdt3.0 sender and the other for rdt3.0 receiver. While the sender FSM is presented in [4], there is no rdt3.0 receiver FSM. The rdt3.0 receiver FSM is the rdt2.2 receiver FSM in [4]. Convince yourself why the rdt2.2 receiver FSM is sufficient before you begin to implement the rdt3.0 protocol. Call the resulting two applications **Sender1b.java** and **Receiver1b.java** respectively. This part requires you to define an acknowledgement (ACK) 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 put in by the sender is also required. You can test the working of duplicate detection functionality in your implementation by using a small retransmission timeout on the sender side. For ACK message, use the following format:

Offset	Octet	0	1
		16-bit sequence number	

Using a **5% packet loss rate for each direction (i.e., pipe)** and rest of *Dummynet* emulated link configuration parameters as before (i.e., **10Mbps bandwidth and 5ms one-way propagation delay for each direction**), experiment with different retransmission timeouts and the corresponding number of retransmissions and throughput.

- Sender program must be named as specified below and must accept the following options from the command line:
  `java Sender1b <RemoteHost> <Port> <Filename> <RetryTimeout>`
  `<RemoteHost>` is IP address or host name for the corresponding receiver. Note that if both sender and receiver run on the same machine, `<RemoteHost>` can be specified as either 127.0.0.1 or localhost.
  `<Port>` is the port number used by the receiver.
  `<Filename>` is the file to transfer.
  `<RetryTimeout>` should be a positive integer in the millisecond unit.
  For example: `java Sender1b localhost 54321 sfile 10`
- Receiver program must be named as specified below and must accept the following options from the command line:
  `java Receiver1b <Port> <Filename>`
  `<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.
  For example: `java Receiver1b 54321 rfile`
- Expected output: (1) A successfully transferred file to the receiver; both sent and received files must be identical at a binary level, and (2) the sender must output number of retransmissions and throughput (in Kbytes/second) only in a single line; no other terminal output should be displayed; the following output implies that the number of retransmissions is 10 and the throughput is 200 Kbytes/second:
  `10 200`

Tabulate your observations in the space provided under Question 1 in the results sheet for Part 1 provided at [5]. For this, your sender implementation should count the number of retransmissions and 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.

Under Question 2 in [5], discuss the impact of retransmission timeout on number of retransmissions and throughput. Also indicate the optimal timeout value from communication efficiency viewpoint (i.e., the timeout that minimizes the number of retransmissions).

Assignment Part 2

Part 2a: Go-Back-N

Extend Sender1b.java and Receiver1b.java from Part 1 to implement the Go-Back-N protocol as described in section 3.4.3 of [4], by allowing the sender window size to be greater than 1. Name the sender and receiver implementations from this part as **Sender2a.java** and **Receiver2a.java** respectively.

- Sender program must be named as specified below and must accept the following options from the command line:
  `java Sender2a <RemoteHost> <Port> <Filename> <RetryTimeout> <WindowSize>`
  `<RemoteHost>` is IP address or host name for the corresponding receiver. Note that if both sender and receiver run on the same machine, `<RemoteHost>` can be specified as either 127.0.0.1 or localhost.
  `<Port>` is the port number used by the receiver.
  `<Filename>` is the file to transfer.
  `<RetryTimeout>` should be a positive integer in the millisecond unit. `<WindowSize>` should be a positive integer.
  For example: `java Sender2a localhost 54321 sfile 10 5`
- Receiver program must be named as specified below and must accept the following options from the command line:
  `java Receiver2a <Port> <Filename>`
  `<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.
  For example: `java Receiver2a 54321 rfile`

- Expected output: (1) A successfully transferred file to the receiver; both sent and received files must be identical at a binary level, and (2) The sender must output throughput (in Kbytes/second) only in a single line; no other terminal output should be displayed; the following output implies that the throughput is 200 Kbytes/second:
  `200`

Experiment with different window sizes at the sender (increasing in powers of 2 starting from 1) and **different one-way propagation delay values (5ms, 25ms and 100ms)** in the emulator. For the 5ms case, use the "optimal" value for the retransmission timeout identified from the previous part. Adjust the timeout suitably for the other two cases. Across all these experiments, use the following values for the other emulated link parameters: **for each direction (i.e., pipe), 10Mbps bandwidth** and **0.5% packet loss rate**. Tabulate your results under Question 1 and answer Question 2 in the results sheet for Part 2 provided at [6].

Part 2b: Selective Repeat

Extend Sender2a.java and Receiver2a.java to implement the selective repeat protocol as described in section 3.4.4 of [4]. Call the resulting two applications as **Sender2b.java** and **Receiver2b.java** respectively.

By configuring the *Dummynet* link with, **for each direction (i.e., pipe), 10Mbps bandwidth, 25ms one-way propagation delay and 0.5% packet loss rate**, experiment with different window size values and complete the table under Question 3 and answer Question 4 in [6].

- Sender program must be named as specified below and must accept the following options from the command line:
  `java Sender2b <RemoteHost> <Port> <Filename> <RetryTimeout> <WindowSize>`
  `<RemoteHost>` is IP address or host name for the corresponding receiver. Note that if both sender and receiver run on the same machine, `<RemoteHost>` can be specified as either 127.0.0.1 or localhost.
  `<Port>` is the port number used by the receiver.
  `<Filename>` is the file to transfer.
  `<RetryTimeout>` should be a positive integer in the millisecond unit.
  `<WindowSize>` should be a positive integer.
  For example: `java Sender2b localhost 54321 sfile 10 5`
- Receiver program must be named as specified below and must accept the following options from the command line:
  `java Receiver2b <Port> <Filename> <WindowSize>`

`<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.
`<WindowSize>` should be a positive integer.
For example: `java Receiver2b 54321 rfile 10`

- Expected output: (1) A successfully transferred file to the receiver; both sent and received files must be identical at a binary level, and (2) The sender must output throughput (in Kbytes/second) only in a single line; no other terminal output should be displayed; the following output implies that the throughput is 200 Kbytes/second:
`200`

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 *iperf* 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 (-F option) of the parameters to *iperf* on the client side. In addition, you should use –t option as well (refer to FAQs below for more details). Use the results of this experiment to complete the table under Question 5 and answer Question 6 in [6].

Implementation Guidelines

Your programs must adhere to the following standard with both sender and receiver application programs to be run inside the *DummynetSL6* VM:

- You can choose to have common files with functions used in different parts but you are required to submit such common files along with necessary documentation.
- You need to take appropriate measures for terminating your sender applications by considering cases where receiver finishes while sender keeps waiting for acknowledgements.
- Please use comments in your code!
- Please start each source file with the following comment line:
/* Forename Surname MatriculationNumber */
For example: /* John Doe 1234567 */

Submission

Submission deadlines for this assignment are as follows:

- **Part 1 due by 4pm on Friday, 15th February 2019:**

  For Part 1, you must submit an electronic version of your implementations for Parts 1a and 1b (Sender1a.java, Receiver1a.java, Sender1b.java, Receiver1b.java and any common files) and completed results sheet [5] (**as PDF**). Use the following submit command:

  ```
  submit comn cw1 <directory-name>
  ```

- **Part 2 due by 4pm on Friday, 22nd March 2019:**

  For Part 2, you must submit an electronic version of your implementations for Parts 2a and 2b (Sender2a.java, Receiver2a.java, Sender2b.java, Receiver2b.java and any common files), and completed results sheet [6] (**as PDF**). Use the following submit command:

  ```
  submit comn cw2 <directory-name>
  ```

Additional instruction on submission: Put all the files under the specified directory, <directory-name>. DO NOT create any new directory in that directory. For example, suppose that Sender1a.java, Receiver1a.java, Sender1b.java, Receiver1b.java and part1_results.pdf should be submitted, and <directory-name> is part1. Then, put all the files in "part1" and run the following:

```
submit comn cw1 part1
```

Late submissions of coursework will be dealt with as per the School of Informatics policy on late submission of coursework.

You are expected to work on this assignment on your own. Or else, you will be committing plagiarism (see School of Informatics guidelines on academic misconduct).

Assessment

This assignment accounts for the whole of your coursework mark (or, 40% of the overall course mark). Distribution of marks among the different parts (as percentage of the coursework mark) is given below:

- Part 1 (30%)
    - Part 1a (10%)
    - Part 1b (20%)
- Part 2 (70%)
    - Part 2a (30%)
    - Part 2b (40%)

References

1. M. Carbone and L. Rizzo, "Dummynet Revisited," SIGCOMM Computer Communication Review, Vol. 40, No. 2, pp. 12-20 Apr 2010.
2. VirtualBox VM Setup Instructions
3. Test file
4. J. F. Kurose and K. W. Ross, "Computer Networking: A Top-Down Approach" (6th edition), Pearson Education, 2013.
5. Part 1 Results Sheet
6. Part 2 Results Sheet
7. FAQs