Computer Communications and Networks (COMN)
2024/25, Semester 2

Assignment 2

Please carefully read this whole document before getting started on this 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 Python 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 these three sliding window protocols, you will need to 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, **you will need to use your COMN coursework virtual machine (VM) [1].** Specifically, the sender and receiver processes for each of the three protocols will run *within the same VM* and communicate with each other over a link *emulated* using *Linux Traffic Control (TC)* [2]. For this assignment, you only need the basic functionality of *TC* to emulate a link with desired characteristics in terms of bandwidth, delay and packet loss rate.

Since the coursework VM does not include a graphical text editor, we suggest you develop your protocol implementations outside of it in the directory/folder containing the 'Vagrantfile'. These files would appear under "/vagrant" within your VM, from where you can run them.

Link Emulation using TC

Within your COMN coursework VM, you can configure and use the *TC* utility to realize an emulated link between two communicating processes (e.g., your sender and receiver programs). TC is a very useful tool that allows you to configure the kernel packet scheduler to emulate packet delay and loss, and limit bandwidth for UDP or TCP applications. Because sender and receiver processes for this assignment are going to be within the same VM, they would communicate with each other through the ***loopback interface (lo) on 127.0.0.1***.

The following outlines how the sender-receiver link via lo can be configured using TC to emulate different link conditions. For example, the following `tc` command adds a rule to the loopback interface scheduler to realize a link with *10ms one-way propagation delay* (equivalently, *20ms round-trip propagation delay* given our symmetric link assumption), *0.5% packet loss rate in each direction* (or, *~1% packet loss rate overall*), and *5 Mbit/s bandwidth limit*:

```
% sudo tc qdisc add dev lo root netem loss 0.5% delay 10ms rate 5mbit
```

You can verify the effect of the above rule by running the following command:

```
% ping 127.0.0.1
```

You will find that `ping` reports an average RTT of 20ms and a packet loss that is approximately 1%.

Note that using the loopback interface and configuring it with `tc` as above allows emulation of a symmetric link between sender and receiver. A packet sent from sender to receiver goes through the loopback interface once; and one more time for the return (receiver to sender) communication, essentially resulting in round-trip propagation delay and total packet loss rate that are double that of the settings used in the `tc` command for configuring these two parameters (as shown in the example above).

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

```
% sudo tc qdisc del dev lo root
```

We assume that packets are not corrupted in transit (i.e., no bit errors). 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 over the loopback interface,

as determined by the packet loss rate setting when configuring it with the `tc` command. For more information on *TC*, please refer to [2].

Besides *TC*, the coursework VM comes with other networking utilities that you may find useful while working on this assignment. These include:

- tcpdump
- iperf
- netcat

Note that these tools are explicitly mentioned so that you know they are available to use. The tcpdump tool is very useful to verify the correctness of the implemented protocols, whereas you'll need iperf for part 4 of this assignment.

Note that each part of this assignment specification (detailed below) states the *TC* configuration parameters to be used. It is important to set the parameters as specified in order to correctly do this assignment.

Detailed Assignment Specification

This assignment is divided into *four* parts. These parts are related but distinct as detailed below.

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

Implement sender and receiver endpoints for transferring a large file given (test.jpg in the coursework git repository) 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 *the loopback interface,* **configured with 10Mbps bandwidth, 5ms one-way propagation delay and 0% packet loss rate** (i.e., no packet loss). In this configuration, the total round-trip propagation delay is 10ms.

Each data message from sender to receiver would have to be 1027 bytes long – 3 bytes for the "header" and 1024 bytes of data. The header in turn consists of 2 bytes of sequence number (for duplicate detection at the receiver) and 1 byte end-of-file (EoF) flag to indicate the last message. **Note that the header must be at the beginning of each packet.**

The sender and receiver developed in this part, respectively, are **Sender1.py** and

Receiver1.py. The receiver should store the transmitted data (after removing header from packet) into a local file.

- Sender program must be named as specified above and must accept the following options from the command line:

  `python3 Sender1.py <RemoteHost> <Port> <Filename>`

  `<RemoteHost>` is the 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: `python3 Sender1.py localhost 54321 sfile`

- Receiver program must be named as specified above and must accept the following options from the command line:

  `python3 Receiver1.py <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 the local disk. For example: `python3 Receiver1.py 54321 rfile`

- Expected output: A successfully transferred file to the receiver; both sent and received files must be identical at a binary level when checked using the `diff` command.

Part 2: Stop-and-Wait

Extend sender and receiver applications from *Part 1* to implement a stop-and-wait protocol described in section 3.4.1 in [3], 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 [3], there is no rdt3.0 receiver FSM. The rdt3.0 receiver FSM is the rdt2.2 receiver FSM in [3]. Convince yourself why the rdt2.2 receiver FSM is sufficient before you begin to implement the rdt3.0 protocol.]* Call the resulting two applications **Sender2.py** and **Receiver2.py,** 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. ACK messages have to be 2 bytes each to hold the sequence number.

- Sender program must be named as specified above and must accept the following options from the command line:

  `python3 Sender2.py <RemoteHost> <Port> <Filename> <RetryTimeout>`

`<RemoteHost>` is the 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, representing retransmission timeout setting in milliseconds.

For example: `python3 Sender2.py localhost 54321 sfile 10`

- Receiver program must be named as specified above and must accept the following options from the command line:

  `python3 Receiver2.py <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 the local disk. For example: `python3 Receiver2.py 54321 rfile`

- Expected output: (1) A successfully transferred file to the receiver. Note that 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`

Using a **5% packet loss rate** while leaving the rest of *TC* configuration parameters as before (i.e., **10Mbps bandwidth and 5ms one-way propagation delay**), experiment with different retransmission timeouts to measure the corresponding number of retransmissions and throughput. Tabulate your observations in the space provided under **Question 1 in the worksheet (included with the git repository)**. 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.

Then for **Question 2 in the worksheet**, discuss the impact of retransmission timeout on the number of retransmissions and throughput. Also indicate the optimal timeout value from a communication efficiency viewpoint (i.e., the timeout that minimizes the number of retransmissions while ensuring a high throughput). Please clearly explain your observations.

Part 3: Go-Back-N

Extend Sender2.py and Receiver2.py from Part 2 to implement the Go-Back-N protocol as described in section 3.4.3 of [3], by allowing the sender window size to be greater than 1. Name the sender and receiver implementations from this part, respectively, as **Sender3.py** and **Receiver3.py**.

- Sender program must be named as specified above and must accept the following options from the command line:
  `python3 Sender3.py <RemoteHost> <Port> <Filename> <RetryTimeout> <WindowSize>`
  `<RemoteHost>` is the 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, representing retransmission timeout in milliseconds.
  `<WindowSize>` should be a positive integer.
  For example: `python3 Sender3.py localhost 54321 sfile 10 5`
- Receiver program must be named as specified above and must accept the following options from the command line:
  `python3 Receiver3.py <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 the local disk. For example: `python3 Receiver3.py 54321 rfile`
- Expected output: (1) A successfully transferred file to the receiver. Note that 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. For example, 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)**. For the 5ms case, use the "optimal" value for the retransmission timeout identified from part 2. The timeout values for the other two cases should be justified clearly. Across all these experiments, use the following values for the other *TC* parameters: **10Mbps bandwidth** and **5% packet loss rate in each direction**. Tabulate your results under

Question 3 and answer **Question 4** in the results sheet.

Part 4: Selective Repeat

Extend Sender3.py and Receiver3.py to implement the selective repeat protocol as described in section 3.4.4 of [3]. Call the resulting two applications, respectively, as **Sender4.py** and **Receiver4.py**.

- Sender program must be named as specified above and must accept the following options from the command line:

  `python3 Sender4.py <RemoteHost> <Port> <Filename> <RetryTimeout> <WindowSize>`

  `<RemoteHost>` is the 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, representing the retransmission timeout in the milliseconds.

  `<WindowSize>` should be a positive integer.

  For example: `python3 Sender4.py localhost 54321 sfile 10 5`

- Receiver program must be named as specified above and must accept the following options from the command line:

  `python3 Receiver4.py <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 the local disk.

  `<WindowSize>` should be a positive integer.

  For example: `python3 Receiver4.py 54321 rfile 10`

- Expected output: (1) A successfully transferred file to the receiver. Note that 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`

By configuring the *TC* link with **10Mbps bandwidth, 25ms one-way propagation delay and 5% packet loss rate**, experiment with different window size values and complete the table under **Question 5** and answer **Question 6** in the worksheet.

As another step in this part, also carry out an equivalent experiment using *iperf* with TCP within your COMN coursework 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 as one of the parameters to *iperf* on the client side (-F option). In addition, you should use the –t option as well. Use the results of this experiment to complete the table under **Question 7** and answer **Question 8** in the worksheet.

Implementation Guidelines

- Your programs must adhere to the following standard with both sender and receiver application programs to be run inside the COMN coursework VM.
- Note that this assignment will be marked using a fresh installation of COMN coursework VM. This means that any extra packet that you install within your VM and use in your code will produce an exception during marking, and in turn would affect your mark. **So, please do not install any additional Python packet or library.**
- You need to take appropriate measures for terminating your sender applications by considering cases where the receiver finishes while the sender keeps waiting for acknowledgements.
- You have been provided with template python files for all four part, that read the arguments from the command line.
- 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.
- Please start each source file with the following comment line:
  # Forename Surname MatriculationNumber
  For example: # John Doe 1234567
- As a general guideline, in the worksheet clearly explain any observations related to the results.

Assessment

This assignment is worth 30% of the overall course mark. Distribution of marks among

the different parts (as a percentage of the overall course mark) is as given below. The reasoning for the unusual percentages is the change in the percentage of this coursework from 35% (previous years) to 30% (this year):

- Part 1 (4.2%)
- Part 2 (8.6%)
- Part 3 (8.6%)
- Part 4 (8.6%)

Submission

Deadline: This assignment is due by **midday (1200 UTC) on Monday, 31th March 2024**.

You must submit an electronic version of your implementations for parts 1-4 (i.e., Sender1.py, Receiver1.py, Sender2.py, Receiver2.py, Sender3.py, Receiver3.py, Sender4.py, Receiver4.py and any common files). **Submit a gzipped tar file named cw2.tar.gz of the directory containing the above files.** To prepare the submission file, use the following command in your coursework VM just outside (i.e., one level above) the directory containing those files:

```
tar czf cw2.tar.gz <directory-name>
```

Separately, you must submit the completed version of the worksheet that is included in the coursework git repository. The completed worksheet should be (1) **exported as a PDF document**, and (2) submitted through gradescope.

Further details on the exact submission process will be made available in due course.

Late submissions of coursework will be dealt with as per **Rule 1** of 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).

References

1. COMN Coursework Virtual Machine Setup
2. Linux Traffic Control (TC)
3. J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach (8th Edition), Pearson Education, 2021.