ECE361 Lab 3
Flow Control and Congestion Control

Due Mar. 7, 2020 @ 12:59 PM

1 Overview

Congestion control and flow control are important features of the TCP protocol. They play an important role in reducing network congestion and improving the efficiency of the channel utilization. In this lab, you will implement flow control and a simple version of congestion control. In the last lab, you implemented a Send Many, Receive One ARQ protocol. In this lab, you will implement a Send Many, Receive Many ARQ protocol to provide support for flow control.

2 Lab Initialization

Similar to lab 2, this lab provides starter code and a custom-built Python 3 library called `ece361`. A Python virtual environment will be provided once the lab is initialized. All Python code in this lab should be run in the provided virtual environment. To run the initialization script for lab 3, open a terminal and execute `ece361-lab-init 3`. You should see a similar output as listing 1.

```
ubuntu@ece361:~$ ece361-lab-init 3
Finding available UG EECG host...
Warning: Permanently added 'ug251.eecg.utoronto.ca,128.100.13.251' (ECDSA) to the list of known
    hosts.
Creating working directory for lab 3 at /home/ubuntu/lab3
Creating Python3 virtual environment...
Installing libraries...

...

Done. Now run source /home/ubuntu/lab3/sourceMe to activate the virtual environment.
```

Listing 1: Initializing lab 3.

The initialization script creates a working directory for this lab, located at `~/lab3`. To activate the virtual environment, run the source command as shown in listing 1. Once the virtual environment is activated, you should see a `(.venv)` appear at the beginning of your prompt.

You will need to activate the virtual environment **in each terminal** that runs code belonging to this lab.

The `ece361` library used in this lab is different than the ones from the previous two labs. Your code from the previous labs will not work in this new virtual environment.

In your lab 3 working directory, you should have the following files:

- `lab3-createNet`, `lab3-enterHost`, `add-load`, and `scripts` directory
  - Various scripts for setting up and configuring the lab's virtual network.
- `sender.py` and `receiver.py`
  - The primary files you will run during this lab. You will need to complete the code in `receiver.py`.
- `senderbase.py`, `flow_control_sender.py`, and `congestion_control_sender.py`
  - Base sender class and two derived classes (which you will have to finish implementing).
- `config.json`
  - Configuration file that will be read by the sender and receiver.
- `binary_file_large`, `text_file_small.txt`, `text_file_medium.txt`, `text_file_large.txt`, and `video1.mp4`
  - Test files you will use during this lab. More specifically, these are files that can be sent by the sender to the receiver.
- `test_configs` directory
  - A directory containing configuration files that you can used to test your implementation.

If you discover a bug in any of the files provided, please report it on Piazza.

3 Provided Scripts

This section will provide an overview of the scripts provided to you as part of this lab, and show examples of their usage.

3.1 Virtual Network

Similar to lab 2, you will create a virtual network consisting of two hosts connected via a single switch[1], all fully emulated within your virtual machine (VM). A script named `lab3-createNet` has been provided to you within the lab working directory, which will create the virtual network with the topology described. When you run the script, you should see something similar to listing 2.

```
ubuntu@ece361:~/lab3$ ./lab3-createNet
Creating new virtual network (2 hosts, 1 switch)...
Configuring host IP addresses...
Done.


Created two hosts with (ID: IP):
```

[1] A network switch is a type of packet forwarding device

```
      h1: 192.168.1.1
      h2: 192.168.1.2
```

Listing 2: Creating the virtual network.

The virtual network's two hosts, as seen in listing 2, has been assigned IPs `192.168.1.1` and `192.168.1.2`.

Different from lab 2, the virtual network created in this lab limits the bandwidth between the two hosts to 10 Mbps. This configuration will allow you to more easily test and see the effects of the congestion control protocol.

3.2 Entering a Virtual Host

To run commands from within one of the virtual hosts, you will need to first "enter" it using the `lab3-enterHost` script provided to you. See listing 3 for an example of how to run the script.

```
ubuntu@ece361:~/lab3$ ./lab3-enterHost h1
ubuntu@ece361-h1:~/lab3$
ubuntu@ece361-h1:~/lab3$ ifconfig
h1-eth0   Link encap:Ethernet  HWaddr 1a:8f:9d:d2:0c:99
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
...
```

Listing 3: Entering into one of the virtual hosts.

In the example shown in listing 3, note the changed command-line prompt, which indicates that the terminal is now within virtual host h1. Running `ifconfig` as shown, note the interface has an IP of `192.168.1.1`, as expected for this particular host.

To test the communication between virtual hosts h1 and h2, first enter h1 on an open terminal, and ping[2] `192.168.1.2`. If the virtual network is functioning correctly, the ping should work, as seen in listing 4. If the ping is not working, exit the host and try re-creating the virtual network to re-test.

```
ubuntu@ece361-h1:~/lab2$ ping -c 4 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.056 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.046 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=0.076 ms

--- 192.168.1.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.046/0.059/0.076/0.013 ms
```

Listing 4: Pinging h2 from within h1.

If you wish to exit the virtual host at any point, simply type `exit` on the command-line.

[2]"ping" is a command-line utility to test connectivity between hosts. It is also often used as a verb.

3.3 Creating Network Load

For congestion control, we need to evaluate the protocol's performance under a congested network. To create background network traffic, we have provided you with a script called `add-load`. This script will create UDP traffic that takes up a certain amount bandwidth in your network. Using the `add-load` script, you can create two kinds of network load: 1. static load with fixed (i.e. constant) bandwidth; and 2. dynamic load where the bandwidth changes between 7 Mbps and 10 Mbps.

1. **Static Load**: To create static background traffic, you can invoke the command in the following way:

   ```
   add-load [load size in mbps] [duration in seconds]
   ```

   Example: To create a network load with 5 Mbps that runs for 1 minute, you do:

   ```
   ./add-load 5m 60
   ```

2. **Dynamic Load**: To create dynamic background traffic that switches between 7 Mpbs and 10 Mbps, you can run the commend as follows:

   ```
   add-load dynamic [duration in seconds] [bandwidth change interval (optional)]
   ```

   The bandwidth change interval refers to how often the background bandwidth changes. It is optional and the default interval is 3 seconds.

   Example: To create a dynamic network load that runs for 30 seconds and switch the background traffic bandwidth every 5 seconds, you do:

   ```
   ./add-load dynamic 30 5
   ```

3. **Stop Load**: To stop the network background traffic generation, you can run the following:

   ```
   ./add-load stop 0
   ```

4 Configuration File

Similar to lab 2, this lab uses a JSON configuration file to customize the behaviour of the senders and the receiver. We have added the following new fields to support the requirements of this lab:

- **sender_max_window_size**
  - The maximum window size for the sender window (intended mainly for the congestion control portion of the lab in Section 6).
- **sender_window_size**
  - The default or initial window size for the sender window.
- **receiver_window_size**
  - The receiver window size.
- **application_wait_time**
  - The time interval between two consecutive reads performed by the application process.

- **application_max_read_bytes**
  - The maximum number of bytes the application process can read each time. -1 means no limit on the number of bytes to read.
- **keepalive_timeout**
  - Timeout for the keep-alive feature. You can keep this set to 0.1 and you should not need to use this configuration parameter.
- **use_flow_control**
  - A flag for enabling/disabling flow control.
- **protocol**
  - This field exists in lab 2. The options in this lab for this field are different. The options are: `flow_control` and `congestion_control`.

5 Flow Control

In this section, you will implement flow control on top of UDP.

5.1 Send Many, Receive Many Protocol

In lab 2, you implemented a Send Many, Receive One ARQ. To further improve the efficiency and use flow control, we will first change the receiver to have a window size greater than 1 and implement the Send Many, Receive Many protocol.

5.1.1 Detailed Description

The version of Send Many, Receive Many protocol you are implementing is similar to the one used by TCP. The following are some of the main points:

- **Cumulative ACK**: The receiver sends cumulative ACKs to the sender with the ACK number being the next sequence number the receiver expects to receive. Outstanding sequence numbers prior to the received ACK number are implicitly ACKed as well. There is no NACK in this system.
- **Per-Byte Sequence Number**: In lab 2, for simplicity, each sequence number represented a single frame. In this lab, we move to a more realistic scenario where sequence numbers are per-byte. For example, if a sender sends a frame with a payload size of 10 bytes and sequence number 2, the receiver should send back an ACK with sequence number 12.
- **Per-Frame ACK**: The receiver sends ACK only when a complete frame is successfully delivered. It does not send ACK if only part of a frame is received, even though the sequence number is per byte.
- **Fixed Receiver Window Size**: In this part of the lab, you will consider a receiver with a fixed window size.

- **Single Timeout Timer**: In this lab, you will implementing a Send Many, Receive Many protocol with a single timeout timer (similar to TCP's implementation). We call this timer a Retransmission Timeout Timer (RTO Timer). The details of this timer will be discussed in Subsection 5.1.2. A separate timer is used for RTT measurements.

5.1.2 Retransmission Timeout (RTO)

When using only a single timer, we need to consider how we manage the timeout. The following is how you will use the timer:

- **Start RTO timer**: When a frame is sent and the RTO timer is not running, start the RTO timer.
- **Restart RTO timer**: When an ACK for a new sequence number is received, restart the timer.
- **Stop RTO timer**: When end-of-file is reached, and all the messages in the send queue are acknowledged, stop the RTO timer.
- **Timeout**: When timeout happens, retransmit the first unacknowledged frame and restart the timer.

To implement RTO timer and RTT timer, you may find the SenderBase.TimerStatus helpful. Refer to the comments in the SenderBase file for more details.

5.1.3 Round Trip Time (RTT) Calculation

To avoid conflict with the RTO timer, you will use a separate timer for measuring RTT. Since we are also using a single timer for measuring RTT, the RTT is calculated based on the time difference between sending one frame and receiving its ACK. Since the receiver uses cumulative ACKs, it will not explicitly ACK every frame; so if an ACK implicitly confirms the frame we are measuring, we can use the received time of that ACK. A detailed illustration of how RTT is calculated is show in Figure 1.

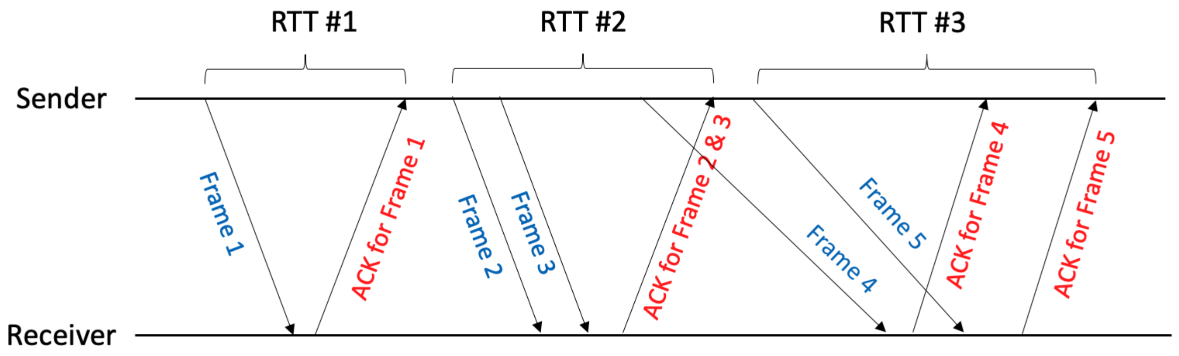

Figure 1: Single timer RTT calculation

Note that you will need to ignore retransmitted frames as the RTT will not be accurate for that frame.

5.1.4 Implementing the Receiver

You will begin by completing the receiver, implemented within the `receiver.py` file. We have provided you the main structure of the receiver, and you will need to complete some functions. A `"Your code here"` comment is shown at locations where you will add your code. The correctness of the following functions may be tested in the exercise and marking, so make sure you do not change the interfaces of the functions. Functions that may be tested:

- `_get_flow_control_window()`
- `insert_packet_to_queue(seqnum, data)`
- `_check_if_seqnum_is_valid(seqnum)`
- `update_r_next()`
- `application_get_data(max_bytes=None)`

You may find the **`NetworkBuffer`** data structure to useful when implementing the receiver. Refer to the Section A.1 for details regarding the `NetworkBuffer` data structure.

Note that the `ApplicationProcess` class is provided in the `ece361` library and you will not be able to change it. This is desirable as the network layer's functionality should not depend on the operation of the application.

Since the receiver uses a polling based method to check for incoming frames, it will have a higher CPU utilization. To reduce CPU utilization, you may uncomment the `time.sleep(0.0001)` line towards the end of the receiver file. You can adjust the time, but make sure you comment it when running exercise and submit for marking. Same applies for the sender files.

5.1.5 Implementing the Sender

In this section, you will complete the implementation of the sender in the `flow_control_sender.py` file. The locations where you will need to add your code is shown in the file's comments. The RTO timer and RTT measurement need to be implemented in the sender. The correctness of the following functions in the flow control sender may be tested:

- `_update_send_queue(ack_seqnum)`
- `calc_window_size()`
- `_send_frames_in_queue(max_frames=None)`
- `_check_timeout()`

5.1.6 Execute your code and measurement performance

Once you have finished the above implementation, you can run it to measure the performance. You will run the receiver in h1 and the sender in h2. Similar to lab 2, you will need to open two terminals, one for h1 and one for h2.

In the **terminal for h1**, you need to execute the following:

1. Enter the h1 host: `./lab3-enterHost h1`
2. Activate virtual environment: `source sourceMe`
3. Run receiver:
```
python3 receiver.py received --config-file \
test_configs/config_mm_32768_vs_4096_slow.json
```

In the **terminal for h2**, you need to execute the following:

1. Enter the h2 host: `./lab3-enterHost h2`
2. Activate virtual environment: `source sourceMe`
3. Run receiver:
```
python3 sender.py text_file_medium.txt --config-file \
test_configs/config_mm_32768_vs_4096_slow.json --verbose
```

The above commands for the sender and receiver use the test configuration files provided along with the skeleton code. You can also customize `config.json` and use that for your development and testing. The `--verbose` flag is use to periodically (every 5 seconds) print the sender's statistics and show its progress. It is intended to help you with your implementation and debugging. This flag only works for the sender. The `--debug` is also available, and it works similar to lab 2.

5.2 Flow Control

You will now implement flow control on top of the Send Many, Receive Many protocol.

5.2.1 Implementing the Sender and Receiver

To implement flow control on top of the Send Many, Receive Many protocol, the receiver will need to send information regarding its flow control window size to the sender. The `send_ack()` function supports sending ACKs containing information about the flow control window size. When you call the `send_ack()` function, you simply provide the flow control window size as the argument. The sender will need to act accordingly based on the receiver's flow control window size.

There is a constructor parameter in the `FlowControlSlidingWindowSender` class called `use_flowcontrol` which is a boolean flag intended for enabling/disabling flow control. Make sure you provide flow control in the sender only when this flag is set to `True`.

5.2.2 Executing Your Code and Measuring Performance

To measure the performance, you can follow the procedure from Subsection 5.1.6 with the `test_configs/config_fc_32768_vs_4096_slow.json` config file. That is:

Run the receiver in **h1** with:

```
python3 receiver.py received --config-file \
test_configs/config_fc_32768_vs_4096_slow.json
```

Run the sender in **h2** with:

```
python3 sender.py text_file_medium.txt --config-file \
test_configs/config_fc_32768_vs_4096_slow.json --verbose
```

How does the result with flow control differ from the result with only the Send Many, Receive Many protocol in Subsection 5.1.6? Do you see higher efficiency (i.e. frames delivered / frames sent)?

Note that it is still reasonable to see frame drop with flow control if you assume the initial sender window size is high. If you assume initial sender window is low, then you should not see any frame drop under a network with not loss.

6 Congestion Control

In this section, you will implement a simple version of congestion control. In the lectures, you have learned about two phases of congestion control in TCP: slow start and congestion avoidance. In this lab, you will implement a simple one with only slow start. Recall that the slow start method increments the sender window by one frame size for each valid ACK received.

6.1 Implementing Congestion Control

We will implement congestion control on top of the flow control sender and receiver you have implemented. For the sender, we will use add your implementation to the `congestion_control_sender.py` file. Note that this file is very similar to `flow_control_sender.py`, and you can feel free to copy your code from `flow_control_sender.py` to this file.

6.2 Executing Your Code and Measuring Performance

You can follow the procedure from Subsection 5.1.6, but with different configuration files. You can configure the sender window in the configuration file to be the same as the frame size as the initial value, and let the congestion control algorithm dynamically manage the sender window size for you.

To see the effect of the congestion control algorithm, you will need to use the `add-load` script to add background traffic to your virtual network.

To help you compare congestion control and fixed sender window size, we have provided some reference configuration files in the `test_configs` directory. The following is a description of the configuration files:

- **config_cc.json**: congestion control configuration file with sender window initialized to the same as the frame size
- **config_fc_ws_4096.json**: configuration file with fix sender window size of 4096 (i.e. only flow control is enabled)
- **config_fc_ws_8192.json**: configuration file with fix sender window size of 8192 (i.e. only flow control is enabled)
- **config_fc_ws_16384.json**: configuration file with fix sender window size of 16384 (i.e. only flow control is enabled)
- **config_fc_ws_32768.json**: configuration file with fix sender window size of 32768 (i.e. only flow control is enabled)

The following are the steps to run these configurations:

Run receiver in **h1** with:
```
python3 receiver.py received.mp4 --config-file [config file path]
```
Run sender in **h2** with:
```
python3 sender.py video1.mp4 --config-file [config file path] --verbose
```
You can also send other files available in the lab3 folder such as binary_file_large, text_file_large.txt, etc.

Under dynamic load or static load, does congestion control perform better than the fixed sender window flow control implementation from Subsection 5.2.2?

To help you understand how the congestion control algorithm behave, it would be a good idea to plot how the congestion control window changes over time. You can log your output to a file and plot it using tools such as Excel or Google Sheets.

7 Exerciser

You can use the exerciser to help test the correctness of your implementation. The exerciser will run a set of public test cases against your code. However, note that the tests in the exerciser is not complete and it is your responsibility to test your code to make sure it conforms to the requirement of the lab. Ensure you have the following files all within the same directory:

- **receiver.py**: Your completed receiver code.
- **flow_control_sender.py**: Your completed flow control sender code.
- **congestion_control_sender.py**: Your completed congestion control sender code.

In terminal, browse to the directory containing the files and type `ece361-exercise 3`.

8 Submission

Once you are confident of your implementation, you can run the submission process (which will invoke the exerciser before submitting). Only one person in the group needs to submit.

From the same directory as where you ran the exerciser, type `ece361-submit submit 3`.

You can then verify the submission by typing `ece361-submit list 3`.

At some point after the lab's due date, private test cases will be run against your submission to calculate your final mark.

A Appendix

A.1 The NetworkBuffer Class

To simplify the implementation for the receiver, we have provided you with a ring buffer data structure with indexing capability called `NetworkBuffer`. The source code of this data structure can be found at the following location:

`/home/ubuntu/lab3/.venv/lib/python3.5/site-packages/ece361/network_buffer.py`

The `NetworkBuffer` data structure provides functionality similar to a ring buffer. It has a fixed length, and each index of the buffer stores one byte. You can insert a frame's payload at any index, and the `NetworkBuffer` will handle the index wrap-around for you. The `NetworkBuffer` will also store the indexes of the first byte of each piece of data inserted and its related metadata.

The following are the available API methods of `NetworkBuffer`:

- **is_start_of_frame(idx)**: Returns `True` if the provided index of the buffer is the start of a frame payload, else it returns `False`. The argument is:
  - **idx**: A buffer index.
- **get_buffer()**: Returns the buffer list object. This can be used to see what is in the buffer.
- **frame_exist(start_idx)**: Returns `True` if a frame exist at the provided start position of the buffer (same as the is_start_of_frame), else it returns `False`. The argument is:
  - **start_idx**: The start buffer index of the frame.
- **pop_frame(start_idx)**: Returns the frame at the specified index from the buffer and remove it from the buffer. Returns `None` if no frame starts at the specified index. The argument is:
  - **start_idx**: The start buffer index of the frame.
- **get_frame(start_idx)**: Return a copy of the frame at the specified index, while keeping the frame in the buffer. Returns `None` if no frame starts at the specified index. The argument is:
  - **start_idx**: The start buffer index of the frame.
- **insert_frame(start_idx, data, frame_metadata={})**: Insert frame at the provided start index with optional metadata to be inserted along with the frame payload. Returns `True` if the insert is successful, else it returns `False`. The arguments are:
  - **start_idx**: The start buffer index of the frame.
  - **data**: The data to be inserted into the buffer (only the payload of a frame should be inserted).
  - **frame_metadata**: A dictionary[3] containing key-value pairs of any metadata/header information you would like to store with this frame (e.g. sequence number, etc.).
- **get_frame_length(start_idx)**: Return the length of the frame at the specified index. Returns `None` if no frame starts at the specified index. The argument is:
  - **start_idx**: The start buffer index of the frame.

[3]Dictionary data structure: https://docs.python.org/3.5/library/stdtypes.html#typesmapping

- **get_frame_metadata(start_idx, metadata_name)**: Return the metadata related to the frame at the specified index. Returns `None` if no frame starts at the specified index, or if no metadata with the specified name is found. The arguments are:
  - **start_idx**: The start buffer index of the frame.
  - **metadata_name**: The name (key) of the metadata value you want to return.

  You can enable debug for this class by initializing it with the debug flag set to True. For example:

  ```
  frameBuffer = NetworkBuffer(10, debug=True)
  ```

  If you have any question on this data structure, please post it on Piazza.

A.2 Frame Class and ApplicationProcess Class

For the `Frame` class and `ApplicationProcess` class, please refer to the comments and descriptions in the source code for detail information.

The Frame class can be found at the following location:

`/home/ubuntu/lab3/.venv/lib/python3.5/site-packages/ece361/frame.py`

The ApplicationProcess class can be found at the following location:

`/home/ubuntu/lab3/.venv/lib/python3.5/site-packages/ece361/application.py`