ECE361 Lab 2
Automatic Repeat Request (ARQ) Protocols

Due Feb. 18, 2020 @ 11:59 PM

1 Overview

In many applications reliable data transmission over an unreliable service is required. ARQ protocols are for exactly that. In this lab, you will be given 2 virtual hosts and an unreliable channel between them. With that you will first enhance the Stop-and-Wait protocol implemented in lab 1 to work on an unreliable channel. After that you will implement a better, more efficient sliding window protocol, in particular, the Send Many, Receive One protocol.

1.1 Lab Initialization

This lab includes starter code that uses a custom-built Python 3 library called `ece361`. A Python virtual environment will be created, that will include the library, where you will do all the work in this lab. Virtual environments must be "activated" before you attempt to run the code inside. We have provided an initialization script for you that will setup this virtual environment, along with the starter code, and the custom library.

It's highly recommended to update your VM before starting each lab. Open a terminal (in the VM), and type `ece361-update`

To run the initialization script for lab 2, open a terminal and type `ece361-lab-init 2`. You should see something similar to listing 1.

```
1  ubuntu@ece361:~$ ece361-lab-init 2
2  Finding available UG EECG host...
3  Warning: Permanently added 'ug251.eecg.utoronto.ca,128.100.13.251' (ECDSA) to the list of known
       hosts.
4  Creating working directory for lab 2 at /home/ubuntu/lab2
5  Creating Python3 virtual environment...
6  Installing libraries...
7
8  ...
9
10 Done. Now run source /home/ubuntu/lab2/sourceMe to activate the virtual environment.
```

Listing 1: Initializing lab 2.

The initialization script creates a working directory for this lab, located at `~/lab2`. Within that directory, there is a file called `sourceMe` which is used to start the virtual environment. To activate the virtual environment, run the `source` command as shown in listing 1. You should see a `(.venv)` appear in your prompt, confirming the environment is activated.

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 one from lab 1. Your lab 1 code will not work in this new virtual environment.

In your lab 2 working directory, you should have the following files:

- `lab2-createNet`, `lab2-enterHost`, and `lab2-adjustLoss`,
  - Various scripts for setting up and configuring the lab's virtual network
- `example` directory, with `client_improved.py`, `client_simple.py`, and `server.py` within
  - Sample code showing how to use the `ece361` library code (documented in Appendix A)
- `sender.py` and `receiver.py`
  - The primary files you will run during this lab
- `senderbase.py`, `stopwaitsender.py`, and `slidingwindowsender.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` and `text_file_small.txt`
  - Test files you will use during this lab

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

2 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.

2.1 Virtual Network

In this lab 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 `lab2-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.

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

```
ubuntu@ece361:~/lab2$ ./lab2-createNet
Creating new virtual network (2 hosts, 1 switch)...
Configuring host IP addresses...
Done.


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

2.2 Entering a Virtual Host

To run commands from within one of the virtual hosts, you will need to first "enter" it using the `lab2-enterHost` script provided to you. See listing 3 for an example of how to run the script.

```
ubuntu@ece361:~/lab2$ ./lab2-enterHost h1
ubuntu@ece361-h1:~/lab2$
ubuntu@ece361-h1:~/lab2$ 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.

```
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.

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

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

2.3 Emulating Frame Loss

To see the effects of ARQ, we need an unreliable channel between the two virtual hosts. To emulate frame loss, we have provided a script called `lab2-adjustLoss` which allows you to control the loss rate of frames entering a particular host. For example, listing 5 shows how to set a frame loss rate of 10% for frames entering h1.

The `lab2-adjustLoss` script will **not** work if the terminal is within a virtual host, so double-check your prompt.

```
ubuntu@ece361:~/lab2$ ./lab2-adjustLoss h1 10%
```

Listing 5: Emulating 10% frame loss for frames entering h1.

To test the effect of the loss rate, we will use a tool called `iperf`. With two terminals, enter h1 in one terminal, and enter h2 in the other. In terminal h1, run an `iperf` server (the terminal will pause, waiting for a client to connect and send it data), as shown in listing 6.

```
ubuntu@ece361:~/lab2$ ./lab2-enterHost h1
ubuntu@ece361-h1:~/lab2$ iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
```

Listing 6: Running an `iperf` server in h1.

In a second terminal, we enter h2 and run `iperf` client to send traffic to h1 at a rate of 100 Mbps. Listing 7 show the client in action with the end result highlighted, verifying the frame loss is being properly emulated.

```
ubuntu@ece361:~/lab2$ ./lab2-enterHost 2
ubuntu@ece361-h2:~/lab2$ iperf -c 192.168.1.1 -u -b 100M
------------------------------------------------------------
Client connecting to 192.168.1.1, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 46722 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   120 MBytes   101 Mbits/sec
[  3] Sent 85471 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec   108 MBytes  90.5 Mbits/sec   0.001 ms 8539/85470 (10%)
[  3]  0.0-10.0 sec  6 datagrams received out-of-order
```

Listing 7: Running an `iperf` client in h2.

After this test, you may kill the `iperf` server in the h1 terminal by pressing `Ctrl+C`.

For the remainder of this lab, we assume you have opened at least 3 terminals: one outside of the virtual hosts, one within host h1, and the other within host h2.

3 A Simple Client-Server Example

Before starting this section, it is highly recommended that you go through Appendix A to get familiar with the lab 2 library we provided to you.

We have provided you with a simplified example to illustrate some of the concepts of this lab. In all 3 terminals, browse into the `example` directory. We need some frames to be dropped for this example to be meaningful. Add a drop rate of 20% to both hosts.

```
ubuntu@ece361:~/lab2/example$ ./lab2-adjustLoss h1 20%
ubuntu@ece361:~/lab2/example$ ./lab2-adjustLoss h2 20%
```

Now from the h1 terminal, run: `python3 server.py`. This starts a simple server on host h1. This server is a simple receiver that will reply to each frame it receives with an ACK contianing the next sequence number it expects.

Now from the h2 terminal, run: `python3 client_simple.py` This simple client, which uses the stop-and-wait ARQ protocol, will transmit 10 frames and report which ones got ACKed and which ones timed out. The timeout is hardcoded to 1 second. When the client program finishes, it prints out the result like that seen in listing 3.

```
Frames delivered: 8
Frames timedout: 2
Total transmission time: 0:00:02.025744
```

As you can see, in this implementation, the total transmission time depends on the number of frames that time out. The more frames that time out, the longer it takes for the client to complete the task. By using stop-and-wait ARQ, the client must wait for the acknowledgement of each frame before sending the next one. Thus, this implementation is quite inefficient.

To see an improved client implementation, run `python3 client_improved.py` in h2. You may keep the server in h1 running.

The output should look something like this:

```
ubuntu@ece361-h2:~/lab2/example$ python3 client_improved.py
b'This is message # 5' DELIVERED. ACK: 6 rtt: 0:00:00.002422
b'This is message # 4' DELIVERED. ACK: 5 rtt: 0:00:00.003443
b'This is message # 2' DELIVERED. ACK: 3 rtt: 0:00:00.004136
b'This is message # 0' DELIVERED. ACK: 1 rtt: 0:00:00.004697
b'This is message # 8' DELIVERED. ACK: 9 rtt: 0:00:00.002196
b'This is message # 9' DELIVERED. ACK: 10 rtt: 0:00:00.002368
b'This is message # 3' TIMED OUT.
b'This is message # 1' TIMED OUT.
b'This is message # 6' TIMED OUT.
b'This is message # 7' TIMED OUT.
Frames delivered: 6
Frames timedout: 4
Total transmission time: 0:00:01.004118
```

Notice that the acknowledgements no longer come in order. More importantly, the total transmission time is at most 1 timeout period regardless of the number of frames that time out. To achieve this performance, `client_improved.py` no longer uses stop-and-wait. Instead, it simultaneously transmits all the frames and then waits for the acknowledgements. The same idea can be applied to improving the stop-and-wait ARQ: transmit more frames in parallel (and wait for multiple acknowledgements) to keep the channel busy.

You should have a high-level understanding of these client implementations before proceeding to the next section. Open the files `client_simple.py` and `client_improved.py` and try to understand the code. When in doubt, refer to Appendix A for more information.

4 Stop-and-Wait ARQ

In this section you will explore the stop-and-wait ARQ, first in an errorless channel, then in a error-prone channel. You will see the effects of frame loss in the forward direction (sender to receiver), as as well as the reverse direction (receiver to sender, in the case of ACKs). Then you will see how to handle these errors.

4.1 Stop-and-Wait (Errorless Channel)

We begin by introducing stop-and-wait when there iss no risk of frame loss (similar to lab 1). Open the configuration file, `config.json`, and make sure it matches the following (which should be the default):

```
ubuntu@ece361:~/lab2$ cat config.json
{
    "sender_address": "192.168.1.2",
    "receiver_address": "192.168.1.1",
    "receiver_port": 6789,
    "arq_protocol": "stopandwait",
    "timeout": null,
    "frame_size": 1,
    "window_size": 2,
    "maxseqnum": 0
}
```

Listing 8: Configuration For Small File

First, let's ensure we have an errorless channel. Using the `lab2-adjustLoss` script, set the frame loss rate in both directions to 0%.

From the h1 terminal, run:

```
ubuntu@ece361-h1:~/lab2$ python3 receiver.py received.txt
```

From the h2 terminal, run:

```
ubuntu@ece361-h2:~/lab2$ python3 sender.py text_file_small.txt
```

When the sender is finished, verify that the file got successfully:

```
ubuntu@ece361-h2:~/lab2$ cat received.txt
I only have a few bytes.
```

Unlike the simple ACK server in section 3, you cannot keep the receiver running between experiments. If you want to start a new transmission, you have to stop the receiver using `Ctrl-C`. This ensures the destination file is closed before starting the next transmission, otherwise it will simply append to the file and won't yield the desired result.

4.2 The `--debug` Option

We have provided you with a debugging feature that prints out the progress on the sender and receiver frame by frame. This is very useful for either debugging or just to understand how the protocol works.

Try the previous stop-and-wait example with the `--debug` flag set and try to understand the debug messages. From the h1 terminal, run `python3 receiver.py --debug received.txt`. Then from the h2 terminal, run `python3 sender.py --debug text_file_small.txt`

4.3 Round Trip Time (RTT)

Round trip time, or RTT, is the time it takes for the sender to send a frame and receive an ACK back for that particular frame. You will need to update the sender code to calculate the RTT.

Let's start by trying to send a larger file. We need to increase the frame size so the transmission doesn't take forever. Modify the `confg.json` file and set the frame size to 1024 (or 1K bytes).

Do not use the `--debug` option when sending large files! The debug messages will flood your screen.

From the h1 terminal, run:

```
ubuntu@ece361-h1:~/lab2$ python3 receiver.py received
```

From the h2 terminal, run:

```
ubuntu@ece361-h2:~/lab2$ python3 sender.py binary_file_large
```

Verify that the file was successfully transmitted using `diff`. Note that no output from the `diff` utility indicates that both files are byte-by-byte identical.

```
ubuntu@ece361-h1:~/lab2$ diff binary_file_large received
```

When the transmission ends, the sender will print some statistics (e.g. see listing 9.

```
ARQ protocol: stopandwait
Frames sent: 9106
Frames delivered: 9106
Transmission time: 0:00:23.292909
Average RTT: 0:00:00
Maximum RTT: 0:00:00
```

Listing 9: Example of sender statistics

We have implemented the `Frames sent`, `Frames delivered` and `Transmission time` statistics for you. We have left the RTT statistics for you to implement.

Complete the `_update_rtt` class method in `stopwaitsender.py` so that it updates the `rtt_max` and `rtt_total` member variables.

After you are done, re-run the previous `binary_file_large` experiment and make sure you get the RTT statistics printed out.

4.4 Handling Frame Drop

We will now see the effects of frame drops. After you get the basic stop-and-wait working, enable frame drops for packets going into h1. Keep the reverse direction (frames from h1 going to h2) errorless for now. For example, set the frame drop rate to 5%:

```
ubuntu@ece361:~/lab2$ ./lab2-adjustLoss h1 5%
```

Modify the configuration file and change the frame size back to 1 for now. Re-run the `text_file_small.txt` with the `--debug` option. What is the result and why?

4.4.1 Choosing a Timeout

When a frame is lost, the sender would be waiting forever for an the acknowledgement forever. To fix this, the sender needs an appropriate timeout for each frame, such that when the timeout occurs, the sender retransmits the previous frame. Based on your RTT calculation in the previous section, change the `"timeout"` field of the configuration file from `null` to an appropriate value (**Note:** The unit is in seconds, and floating point numbers are accepted).

Should the timeout be based on the average or maximum RTT? Why?

Re-run the `text_file_small.txt` experiment and test if the transmission is successful. After you get the small file working, change the frame size to 1024 and try transmitting `binary_file_large`.

Can you figure a mathematical relationship between the expected frames sent, frames deliverd and the frame drop rate? Does the experimental result agree with your formula?

4.5 Handling Lost ACKs

Now we will see the effect of frame loss for ACKs. Set a frame loss rate of 5% for frames entering into h2.

```
ubuntu@ece361:~/lab2$ ./lab2-adjustLoss h2 5%
```

Change the frame size configuration back to 1 and re-run the `text_file_small.txt` experiment. Examine the resulting file created by the receiver. You should see something similar to listing 10. The actual result varies from run to run, but the key here is duplicated frames. If you don't see duplicated frames then increase the frame drop rate (e.g. 10%) and try the experiment again.

```
ubuntu@ece361-h1:~/lab2$ cat received.txt
I only havee a few bytess.
```

Listing 10: Example of duplicate frames being received.

Pause here and think if you can figure out why this happens.

When an ACK frame is lost, the sender will eventually time out and re-send a frame. Without the use of sequence numbers to distinguish subsequent frames, the receiver cannot tell if a frame is a duplicate.

The implementation we have provided already supports sequence numbers, you just need to enable it in the configuration file. Set the `"maxseqnum"` field of the configuration file to any positive integer (e.g. 64) and re-run the `text_file_small.txt` experiment. You should see that the file gets successfully transmitted without duplicated frames.

Finally, let's test the fully implemented stop-and-wait ARQ protocol on `binary_file_large`. Set the frame size back to 1024 and re-run the experiment. You might need to wait for a while for the transmission to finish. When it does, verify that the file gets successfully transmitted using `diff`.

If the previous experiment succeeds, congratulations! You have a fully working stop-and-wait ARQ. Now open the `stopwaitsender.py` file and have a look at the `_arqsend` function. You should understand the code at a high level before proceeding to the next section. You only need to understand what the API calls in that function are doing (you do not need to understand the details of how the APIs work under-the-hood). Also, you only need to look at that one single function, you do not need to read other parts of the code. However, if you are interested, you can read the entire code-base since we made all the source code available to you.

5 Sliding Window ARQ (Send Many, Receive One)

In this section, you will complete the implementation of a sender that uses the Send Many, Receive One ARQ protocol. Section 5.1 shows how to configure the sender to use the Send Many, Receive One protocol. Section 5.2 provides a technical overview of the protocol, and section 5.3 describes your task.

5.1 Configuring the Sender to Use Sliding Window ARQ

Update the configuration file. Change the `"arq_protocol"` field of the configuration file from `"stopandwait"` to `"slidingwindow"`. Make sure `"window_size"` is an integer greater than 1. Make sure `"maxseqnum"` is at least `"window_size"`. Keep other parameters (e.g. timeout) the same as in the stop-and-wait section.

```
{
    "sender_address": "192.168.1.2",
    "receiver_address": "192.168.1.1",
    "receiver_port": 6789,
    "arq_protocol": "slidingwindow",
    "timeout": <your choice>,
    "frame_size": 1,
    "window_size": 2,
    "maxseqnum": 64
}
```

Listing 11: Sliding Window Configuration

If you set the window-size to be 1, what does the Sliding Window ARQ become? Why must the `"maxseqnum"` be at least `"window_size"`?

Try running the `text_file_small.txt` experiment with `--debug` the same way as before. From the h1 terminal, run: `python3 receiver.py -debug received.txt`. From the h2 terminal, run: `python3 sender.py -debug text_file_small.txt`.

As you can see, the programs will hang without any debug outputs, which means no frame is being sent or received. This shouldn't surprise you since sliding window ARQ has not been implemented yet! Open the `slidingwindowsender.py` file and find the `_arqsend` method.

Your task is to complete the `_arqsend` method. We have started the implementation for you by creating a send queue.

Apart from this method, the rest of the sender functions are already implemented for you.

We have given you some high level guidelines on what to do as well in the comments. Before starting, make sure you have read section 5.2, which explains our implementation (it may differ in minor details from what you have learned).

5.2 Protocol Description

As you can see from Figure 1, stop-and-wait ARQ is inefficient because the channel is idle during the period where the sender waits for acknowledgement from the receiver. It would be much more efficient if the channel could be kept busy during this waiting period. This is where sliding window ARQ comes in. Figure 2 illustrates how sliding window works on a perfect, errorless channel.

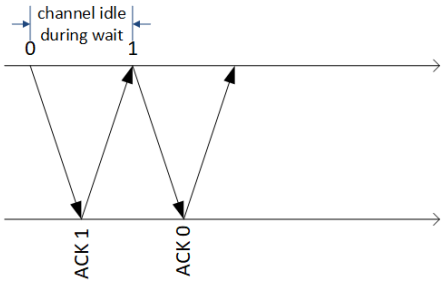

Figure 1: Stop-and-Wait ARQ Timing Diagram

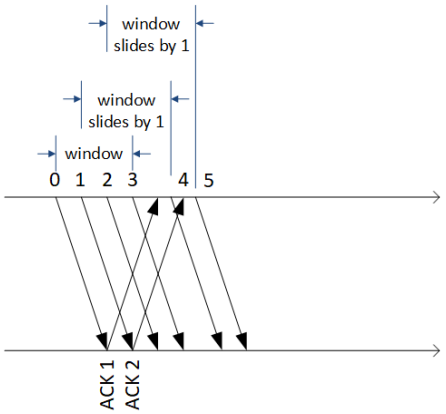

Figure 2: Sliding Window ARQ Timing Diagram

A sliding window can be implemented on both the sender and the receiver side, but for this lab, we only consider sliding window on the sender side. The receiver still accepts one frame at a time, similar to a stop-and-wait receiver. Thus, our protocol behave like "Send Many, Receive One".

Note that one needs to make sure that the sliding window is implemented correctly. A typical incorrect implementation is illustrated in Figure 3 where frames 1, 2 and 3 should not be resent.

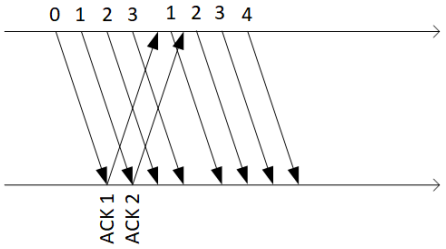

Figure 3: Sliding Window ARQ Incorrect Implementation

5.2.1 Error Control

The most basic error control mechanisims in sliding window ARQ are acknowledgement and timeout, much like stop-and-wait. Upon receiving a frame with the expected sequence number, the receiver

sends back an ACK with the next sequence number. The acknowledgements are independently sent to every frame. The sender maintains a timer for each frame, which starts right after the frame is sent. If, after some timeout period, the frame does not receive its corresponding ACK, the frame is retransmitted. As illustrated in Figure 4, frame 0 is retransmitted after some timeout period.

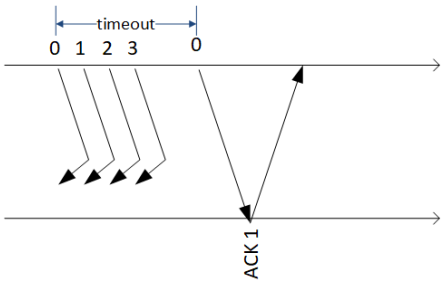

Figure 4: Sliding Window ARQ Error Control

Consider the case in Figure 5. Frames 0, 1, 2, and 3 all transmitted successfully but the ACKs are all lost. From the sender's point of view there is no difference to the case where the frames are lost, so it retransmits frame 0. When the receiver receives the duplicate frame 0, it should send acknowledgements for frames 0, 1, 2 and 3 since those frames have all been received. However, a more bandwidth efficient way is to use what is called a "culmulative ACK", in which the receiver only sends an ACK for frame 3 (ACK 4), implying that frames 0 through 3 have all been received. When the sender receives this ACK, it immediately sends frame 4, as illustrated in the figure. Note that if the sender does not know about culmulative ACKs it won't be able to correctly communicate with the receiver.

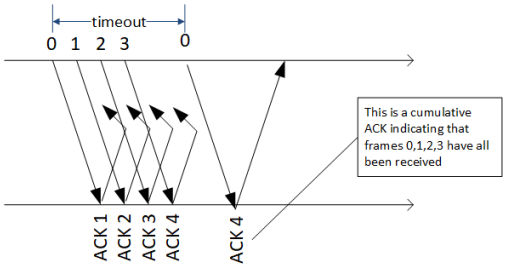

Figure 5: Culmulative ACK

In the same way, when an ACK for a certain frame is received, it is implied that all frames earlier than that frame were also correctly received, even if their ACKs never arrived. This case is illustrated in Figure 6. Upon receiving the ACK, the sender can immediately start transmitting the next frame.

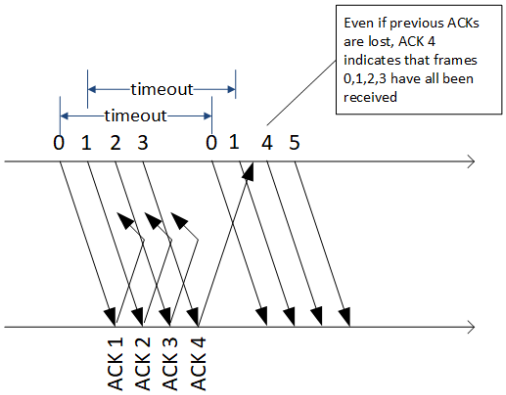

Figure 6: Culmulative ACK Handling

There is an implementation detail on what the sender should do upon receiving a culmulative ACK half way through transmitting the send window. The best implementation is shown in Figure 6, where the sender aborts the current send window and immediately start sending the frame with the correct sequence number. However, this is sometimes challenging to implement. Therefore, in this lab, you will do a simpler but less efficient implementation as illustrated in Figure 7, where the sender is allowed to finish the current send window and then start sending the correct frame. This is less efficient due to extra duplicated frames being sent, but easier to implement.

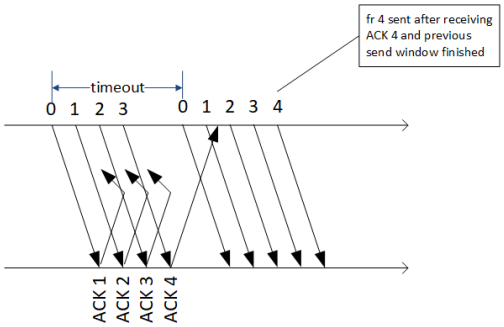

Figure 7: Culmulative ACK Handling (Simpler Implementation)

Finally, there is one last case we need to cover, which is frames that arrive out of order. As illustrated in Figure 8, if frames 0 is lost, frames 1, 2, 3 will be considered to be out of order by the receiver. Thus, it rejects the frame and sends a NACK with the sequence number of the frame it expects (in this case, frame 0). In our implementation, NACKs are always sent for out of order frames.

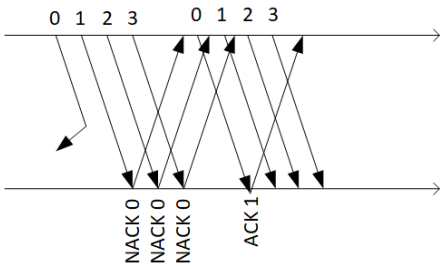

Figure 8: Out of Order Frames and NACKs

Note that in this lab, implementation-wise, there is no difference between a NACK and a resend of the ACK for the last received frame. You may have learned that the term "NACK" is also used in the context of indicating frame corruption, but you don't have to worry about that in this lab. You can assume that if a frame arrives at the receiver, it always arrives without error.

If we do not send any NACKs, does the sliding window ARQ still work? If the answer is yes, what mechanism will it rely on for the sender to know to re-transmit the correct frame, and why are NACKs useful then?

5.3 Implement the Sliding Window ARQ

Based on what you've learned so far, implement the `_arqsend` function in `slidingwindowsender.py`. During the debugging stage, it is recommended to test transmission using `text_file_small.txt`, with frame size of 1, and `--debug` on. However, the `DEBUG` messages on the sender side needs to be implemented by you. After testing your code on the small text file, test sending the `binary_file_large` file using a frame size of 1024 and 5% channel loss in both directions. After the transmission is done, use the `diff` command to make sure that the file is transmitted successfully.

If you get stuck, take a look at `client_improved.py` in the example directory and understand it in detail. A lot of the ideas used in that progam are directly applicable to implementing sliding window ARQ.

5.4 Getting the Optimal Window Size

Start with the following configuration:

```
{
    "sender_address": "192.168.1.2",
    "receiver_address": "192.168.1.1",
    "receiver_port": 6789,
    "arq_protocol": "slidingwindow",
    "timeout": <your choice>,
    "frame_size": 1024,
    "window_size": 2,
    "maxseqnum": 64
}
```

Listing 12: Sliding Window Configuration

Transmit the `binary_file_large` file. Do you see an improved performance compared to Stop-and-Wait?

If you keep increasing the window size, what do you see? Can you explain why?

Your task is to find an optimal window size. What is your minimum file transmission time?

You can try to experiment with other binary files types (e.g. images, videos etc.). When the implementation is incorrect, do you see corruption?

5.5 Goal

Your target is to transmit the `binary_file_large` file under 10 seconds, on a channel with 5% loss both ways, and a frame size of 1024. Your sliding window ARQ implementation has to conform to descriptions in Section 5.2. We will not only test the time it takes to transmit the file but also test every aspect of the protocol described in Section 5.2. Our additional tests for protocol conformation may change parameters such as channel loss, frame size, and window size, so you should test with different parameters, not just the ones given to you in the lab.

6 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 Section 5.2. Ensure you have the following files all within the same directory:

- **stopwaitsender.py**: Your completed stop-and-wait sender code from section 4.
- **slidingwindowsender.py**: Your completed sliding window sender code from section 5.

In terminal, browse to the directory containing the files and type `ece361-exercise 2`.

7 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 2`.

You can then verify the submission by typing `ece361-submit list 2`.

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 Frame Class

In lab 1 you worked with a `Socket` class. In this lab you will be working with a `Frame` class which encapsulates the concept of a frame. For this lab we have encapsulated all the functionalities in this class so you do not need to work with sockets directly.

A.2 Importing the Library

Similar to lab 1 you will need an "import" statement in order to use the library:

```
from ece361.lab2.frame import Frame
```

Listing 13: Import Statement

A.3 Creation

To create a `Frame` class object, at least 2 parameters are required: `seqnum` (sequence number) and `data`, which are self explanatory. The other parameters are optional and illustrated below:

```
# Instantiate a new Frame object
new_frame = Frame(
                  # sequence number
                  seqnum,
                  # bytes storing the actual data
                  data,
                  # optional: (ip_address, port) tuple for destination
                  destination,
                  # optional: expected acknowlegement number
                  expected_ack,
                  # optional: timeout for receiving acknowlegement
                  timeout
                  )
```

Listing 14: Frame Object Instantiation

A.4 Methods you Need in Order to Complete the Lab

A.4.1 status()

This is one of the key methods for completing this lab. It is for checking the current status of the `Frame` object. A frame can be in one of the following 4 states: `notsent`, `inflight`, `timedout`, `ack_nacked`. To check the current status of the frame, you need to write something similar to the following:

```
# Assuming 'frame' is an object instance of the Frame class
if (frame.status() == Frame.Status.notsent):
    # Code here...
```

Listing 15: Frame status

A.4.2 send()

As the name suggests, this method sends the frame to its destination. It only works if the destination field is not empty when the `Frame` was created. Note that unlike lab 1, there is no need to pass in the destination address since we assume that a `Frame`'s destination is determined upon creation (and cannot be modified). If you want to send to a different destination you have to create a new `Frame` object.

A.4.3 wait_for_ack_nack()

This method will block until an ACK/NACK for the frame comes back.

A.4.4 wait_for_multiple_ack_nacks(frames_list)

This is an essential function for completing Section 5 of the lab. This is the "non-blocking" counterpart of `wait_for_ack_nack()`, where a list of frames is passed in and will return if any of the frames in the list receives an ACK/NACK. If none of the frames in `frames_list` receive an ACK/NACK, then this function will block. This is a static function. In order to use it one needs to proceed its name with `"Frame"`:

```
1 Frame.wait_for_multiple_ack_nacks(frames_list)
```

Listing 16: Frame status

Note that although this function returns when at least one the frames gets an ACK, it does not tell which frame(s) in the `frames_list` is acknowledged. After this function returns one needs to go through the entire `frames_list` again and check the status of each to know which ones got acknowledged.

A.5 Other Methods

You do not need the following methods for completing this lab. They are used in skeleton and example code, so we include them here for reference purposes.

A.5.1 sendtime()

Returns a `datetime` object indicating the time the frame object is sent. If there are multiple send attempts, it returns the time for the most recent send attempt. Returns `None` if the frame was never sent.

A.5.2 retrieve_ack_nack()

Retrieves the ACK/NACK of the frame (if ACK/NACKED), which is a number. This method only works if the frame is in `ack_nacked` state. Since there are no true NACKs in this lab you do not really need this function because ACK is always the next sequence number. Making sure the frame is in `ack_nacked` state is sufficient.

A.5.3 acktime()

Returns a `datetime` object indicating the time when the ACK was received. Returns `None` if no acknowledgement has been received.

A.5.4 pack_data()

Returns the raw "packed" version of the `Frame` object, which is a sequence of bytes with first 4 bytes being the sequence number, followed by the frame data. The "packed" version is useful for transmitting the frame over the network.

A.5.5 unpack_data()

This is exactly the opposite of `pack_data()` where a sequence of bytes is converted to a `Frame` object, assuming the same format: 4 bytes sequence number followed by data. Each time this method is called a new `Frame` object is returned.