Lab 5: Flow Control and DNS
CS640 Fall 2022

Released: Monday, Nov 21 2022
Due: Wednesday, Dec 14 2022 11:59pm
Points: 60

Important notes

- All answers and code that you write for this lab must be your own work. Plagiarism is a serious violation of university statutes and all answers will be thoroughly checked for it.
- Ensure that your answer has proper headings and subheadings wherever necessary.
- Late policy:
    - Upto 30 minutes late — lose 0% of points
    - Upto 24 hours late — lose 10% of points
    - Upto 48 hours late — lose 30% of points
    - Upto 72 hours late — lose 60% of points
    - Beyond 72 hours — lose 100% of points

Overview

For this lab, you will first implement a Python-based data sender and receiver using the sliding window algorithm. Then you will write your own simple Java-based DNS server that performs recursive DNS resolutions, and appends a special annotation if an IP address belongs to an Amazon EC2 region.

Part 1: Flow Control
Part 2: Simple DNS Server
Submission Instructions

Learning objectives

After completing this assignment, you should be able to:

- Explain how the sliding window algorithm facilitates flow control
- Explain how the domain name system (DNS) works

===

Getting started

You can work on this lab on a Unix-based machine such as Linux, Mac or Mininet VM. All the skeleton code needed in this assignment can be downloaded by

```
    wget
    https://pages.cs.wisc.edu/~mgliu/CS640/F22/labs/lab5/lab5.tgz

    tar xzf lab5.tgz
    cd lab5
```

In the `lab5` folder, you will find two folders: `fc` and `src`, containing the code for Part 1 and 2 respectively. There is another file for Part 2 called `ec2.csv` in `lab5`, which is the list of public IP address ranges for each EC2 region.

Package Requirements:
Python version >= 3.7, Java version >= 11

===

Part 1: Flow Control

As you work on this assignment, you may want to consult the "Sliding Window" portion of Section 2.5.2 of Computer Networks: A Systems Approach.

Background

You will implement a simple sliding window protocol (SWP) in Python that transmits data in only one direction (and acknowledgements in the reverse direction).

The **sender side** of the SWP is responsible for:
(1) transmitting data packets
(2) guaranteeing the number of "in-flight packets" remains within a fixed bound
(3) retransmitting data packets if an acknowledgment (ACK) is not received within a pre-determined timeout.

The **receiver side** of the SWP is responsible for sending cumulative ACKs.

Every SWP packet exchanged between the sender receiver contains:
- The character `D` or `A`, which indicates whether it is a Data or ACK packet
- A 32-bit sequence number, which indicates which chunk of data the packet contains or which chunk of data is being ACK'd
- Up to 1400 bytes of data (only in SWP data packets)

The `SWPPacket` class defined in `swp.py` is used to represent an SWP packet. This object-based representation can be converted to a sequence of bytes to be sent across the network using the `to_bytes` method. Conversely, a sequence of bytes received from the network can be converted to an object-based representation using the `SWPPacket.from_bytes` method.

The SWP sender and receiver both interact with a simple lower layer protocol (LLP) that sends and receives SWP packets across the network on behalf of the SWP. The `LLPEndpoint` class in `llp.py` exposes a basic API for sending and receiving a packet—really just a sequence of bytes—to/from a "remote" endpoint.

NOTE
Please use the constants like _SEND_WINDOW_SIZE, _TIMEOUT & _RECV_WINDOW_SIZE given in the respective classes.

Part 1.1: SWPSender

You are responsible for implementing the sender side of SWP by completing the `SWPSender` class in `swp.py`.

The sender must do three things:

1. Buffer and send a data packet that is within the send window. If the send window is full, then the caller (e.g., an application) will be blocked until the packet can be buffered.
    a. Use the `_send` function in the `SWPSender` class, more explained below.
2. Retransmit a data packet within the send window if it is not acknowledged within a predetermined amount of time.
    a. Use the `_retransmit` function the `SWPSender` class, more explained below.
3. Discard data that has been successfully acknowledged, thus enabling new data to be buffered and sent.
    a. Use the `_recv` function the `SWPSender` class, more explained below.

As you write the code, we recommend you add some debugging statements—use the function `logging.debug` instead of `print`—to make it easier to trace your code's execution.

_send

The `_send` function is invoked by the send function which is invoked by an "application" (e.g., `client.py`). The `_send` function needs to:

1. Wait for a free space in the send window—a semaphore is the simplest way to handle this.
2. Assign the chunk of data a sequence number—the first chunk of data is assigned sequence number 0, and the sequence number is incremented for each subsequent chunk of data.
3. Add the chunk of data to a buffer—in case it needs to be retransmitted.

4. Send the data in an SWP packet with the appropriate type (D) and sequence number—use the `SWPPacket` class to construct such a packet and use the `send` method provided by the `LLPEndpoint` class to transmit the packet across the network.
5. Start a retransmission timer—the Timer class provides a convenient way to do this; the timeout should be 1 second, defined by the constant `SWPSender._TIMEOUT`; when the timer expires, the `_retransmit` method should be called.

`_retransmit`

The `_retransmit` function is invoked whenever a retransmission timer—started in `_send` or a previous invocation of `_retransmit`—expires. The `_retransmit` function needs to complete steps 4 and 5 performed by the `_send` function, which are:

1. Send the data in an SWP packet with the appropriate type (D) and sequence number—use the `SWPPacket` class to construct such a packet and use the `send` method provided by the `LLPEndpoint` class to transmit the packet across the network.
2. Start a retransmission timer—the Timer class provides a convenient way to do this; the timeout should be 1 second, defined by the constant `SWPSender._TIMEOUT`; when the timer expires, the `_retransmit` method should be called.

`_recv`

The `_recv` function runs as a separate thread—started when an `SWPSender` is created—and receives packets from the lower layer protocol (LLP) until the SWP sender is shutdown. The SWP sender should only receive SWP ACK packets—you should ignore any packets that aren't SWP ACKs.

For every chunk of data that is ACK'd, the `_recv` function needs to:

1. Cancel the retransmission timer for that chunk of data.
2. Discard that chunk of data.
3. Signal that there is now a free space in the send window.

Note: the SWP ACKs are cumulative, so even though an SWP ACK packet only contains one sequence number, the ACK effectively acknowledges all chunks of data up to and including the chunk of data associated with the sequence number in the SWP ACK.

Testing and debugging

To test your code:

1. Run the test server executable that we provide:
   ```
   ./fc/server.py -p PORT
   ```
   where `PORT` >= 1024.
2. In a separate terminal window (or terminal multiplexer link tmux) connected to the same Mininet VM, start the client Python script using the command:
   ```
   ./fc/client.py -p PORT -h 127.0.0.1
   ```
   replacing `PORT` with the port number you specified for the server.

3. Type something into the client and hit enter; the data should appear on the server.

Check rubric for tests and expected outputs.

Part 1.2: SWPReceiver

Now that you have finished part 1, you should work on implementing the receiver side of SWP by completing the `SWPReceiver` class in `swp.py`.

`_recv`

All functionality for the receiver side of SWP is implemented in the `_recv` function, which runs as a separate thread—started when an `SWPReceiver` is created—and receives packets from the lower layer protocol (LLP).

For every SWP data packet that is received, the `_recv` function needs to:

1. Check if the chunk of data was already acknowledged and retransmit an SWP ACK containing the highest acknowledged sequence number.
2. Add the chunk of data to a buffer—in case it is out of order.
3. Traverse the buffer, starting from the first buffered chunk of data, until reaching a "hole"—i.e., a missing chunk of data. All chunks of data prior to this hole should be placed in the _ready_data queue, which is where data is read from when an "application" (e.g., `server.py`) calls recv, and removed from the buffer.
4. Send an acknowledgement for the highest sequence number for which all data chunks up to and including that sequence number have been received.

Testing and debugging

To test your code:

1. Start the server Python script using the command:
   ```
   ./fc/server.py -p PORT
   ```
   where `PORT` >= 1024.
2. In a separate terminal window (or terminal multiplexer link tmux) connected to the same Mininet VM, start the client Python script using the command:
   ```
   ./fc/client.py -p PORT -h 127.0.0.1
   ```
   replacing `PORT` with the port number you specified for the server.
3. Type something into the client and hit enter; the data should appear on the server.

To test retransmission, include the command line argument **-l PROBABILITY** (that is a lowercase L) when you start the client and/or the server. Replace **PROBABILITY** with a decimal number between 0.0 to 1.0 (inclusive), indicating the probability that a packet is dropped. If you pass this option to the client, then ACK packets may be dropped. If you pass this option to the server, then data packets may be dropped.

Part 2: Simple DNS Server

For this part of the assignment you will implement your own simple DNS server in Java (NOTE: Use Java >= 11 for this lab). Your server will accept queries from clients, and issue queries to other DNS servers in order to respond to client queries. Your server will also append a special TXT record if an IP address belongs to an Amazon EC2 region. For simplicity, your server will not cache any DNS records, nor will it be responsible for storing the records for any DNS zones.

You can play around with DNS queries using the dig program, which should already be on your Mininet, you can download if needed or you can run queries from your computer. One of the formats to use dig command is

```
    dig +norecurse @name.of.dns.server record-type domain-name
        name.of.dns.server is the domain name of the DNS server
        you wish to query
        record-type is the type of DNS record you wish to
        retrieve, e.g., A
        domain-name is the domain name you seek information on
```

E.g.

```
    dig +norecurse @a.root-servers.net A www.google.com
```

You should take some time to look into how to use this and what the output looks like. Another useful tool for this lab is wireshark. You should take some time to learn how to use wireshark, and how to see/read DNS packets.
Follow http://mininet.org/walkthrough/#start-wireshark for a walk you through of wireshark in mininet, if you want to use mininet.
You can also download wireshark on your laptop or use the CS machines.

Learning About DNS Packets

Before you write any code, you should familiarize yourself with the format of DNS messages. You can read the Network Sorcery RFC Sourcebook page on DNS. Chapter 9.3.1 in our book also covers DNS.

You will need to have root (or administrator) access to capture packets, so you should issue your queries either from your own machine, or from your Mininet VM. You can use tcpdump in your Mininet VM to capture DNS packets:

```
    sudo tcpdump -n -i eth0 udp port 53 -w dnstrace.pcap
```

You can then copy the `pcap` file with `scp` to a machine with Wireshark. If using the CS machine you can run:

```
    scp dnstrace.pcap USERNAME@MACHINE.cs.wisc.edu:~
```

If you use your own machine, you can use Wireshark to both capture and view the packets.

In Wireshark, you should select the DNS packet you want to view, then look at its details in the pane in the bottom half of the Wireshark window. You should pay particular attention to the Flags, Questions, and Answers parts of the DNS packet.

Command Line Arguments

Your DNS server should be invoked as follows:

```
java edu.wisc.cs.sdn.simpledns.SimpleDNS -r <root server ip> -e
<ec2 csv>
```

`-r <root server ip>` specifies the IP address of a root DNS server. You can google root servers and use one, example 198.41.0.4

`-e <ec2 csv>` specifies the path to a comma-separated variable (CSV) file that contains entries specifying the IP address ranges for each Amazon EC2 region

Receiving Queries

You should start by writing code that receives and parses DNS queries. Your server should listen for UDP packets on **port 8053**. You should call the deserialize method in the DNS class in the `edu.wisc.cs.sdn.simpledns.packet` package to parse the payload of a UDP packet that contains a DNS query. (Hint: Look at DatagramSocket and DatagramPacket in Java)

Your server only needs to handle opcode 0 (standard query), and query types A, AAAA, CNAME, and NS. You can silently drop all other client queries. Also, your server only needs to handle one client query at a time (i.e., it does not need to be multi-threaded).

Handling Queries

When your server receives a query of type

- A : The Value field is an IPv4 address.
- AAAA : The Value field is an IPv6 address.
- CNAME : The Value field gives the canonical name for a particular host; it is used to define aliases.
- NS : The Value field gives the domain name for a host that is running a name server that knows how to resolve names within the specified domain.

with the recursion desired bit set to 1, it should recursively resolve the query, starting from the root name server. (please review 9.3.1 if you are unsure what this means) If the recursion desired bit is set to 0, it should only query the root name server.

If a query is of type A, and your DNS server successfully resolves the query, then you should check if the address(es) are associated with an EC2 region. For each address associated with an EC2 region, you should add a TXT record to the answers you provide to the client. The TXT record should contain the name of the EC2 region (from the CSV file), followed by a hyphen (-), followed by the address in dotted decimal form. For example:

```
www.code.org TXT Virginia-50.17.209.250
```

Don't forget to include the A record(s) as well! Also you can use the `DNSRdataString` class when creating a `DNSResourceRecord` of type TXT.

NOTE

Implementation for handling multiple questions is not required

For cases where you get Authority Sections without Additional sections, you should recursively fetch the IP for the name server in the Authority Section. Once you get one of the name servers IP in this authority list, you can continue with the name resolution of the original query.

Testing

You can test your code using `dig`. To force queries to use your DNS server, include the arguments "`-p 8053 @localhost`". The question, answer, authority, and additional sections output by `dig` when using your DNS server should match what is output produced by `dig` when you query your machine's default DNS server (although the addresses and name servers may be slightly different if an upstream DNS server is using round robin to select which records are returned). You can also use `tcpdump` to help you debug.

Check rubric for tests and expected outputs.

Prepare for Grading

You should create a `Makefile` under the `lab5` folder that enables us to compile and run your Part 2 Java code by executing the following commands:

```
cd lab5
make
make run
```

Also, `make clean` should remove `*.class` files. You should include a `README.md` file that contains your group members names and your Wisc IDs along with any additional information needed by the TA to run the code.

==

Submission Instructions

Assuming you are in the lab5 folder, you should pack the submission by:

```
tar czvf netid1_netid2.tgz fc/ src/ Makefile README.md
```

IMPORTANT: netid1 and netid2 should be NetID, not CS username (this is different from Lab 3)

You should submit the tar file to Canvas. Please make only one submission per team.