Open in Colab

Notebook Setup

You must execute the cell below ***once*** before proceeding...

```
%%bash
#rm -rf *
git clone https://github.com/jlegatheaux/RC2020-assignments.git RC 2> /dev/null || git -C RC pull
mv -f RC/assignment-2/* .

# Fetch the CNSS repository and compile it
git clone https://github.com/jlegatheaux/cnss.git 2> /dev/null || git -C cnss pull
javac -d cnss-classes cnss/src/*/*/*.java
```

Laboratory project - Reliable Data Transmission over an Unreliable Network with the FT20 Protocol

Goal

This assignment concerns the problem of delivering information reliably across a network where a degree of packet loss is expected.

The goal is to understand how the performance of sliding window protocols is impacted by parameter choices, such as fixed timeouts, adaptative timeouts, windows sizes, as well as the flow control technique employed etc.

Context

This assignment will use a simple file transfer protocol, between a client and a server, implemented in the CNSS simulator.

FT20 Protocol

The FT20 protocol, developed for this assignment, allows for the reliable transfer of a file from a client to a server, over an unreliable network. This protocol comprises a total of 5 types of packets, namely `UPLOAD`, `ERROR`, `DATA`, `ACK` and `FIN`.

FT20 Packets

Each packet type is identified by an 1-byte numeric *opcode*. The format and purpose of each packet is as follows:

- ***UPLOAD***

  `|1|timestamp|filename|`

  Initiates the transfer of the file with the given `filename. This packet has 0 as its implicit *sequence number*.

- ***ERROR***

  `|2|errormessage|`

  Reports a fatal error.

- ***DATA***

  `|3|timestamp|seqN|data|`

  Represents a block of file data. The block is identified by a *sequence number* (`seqN´), starting at 1, for the first block of a file.

  `data` - the file block payload encoded as raw bytes.

- ***ACK***

  `|4|timestamp|cSeqN|sSeqN|`

  Confirms correctly received packets.

  `cSeqN` - Represents a **cumulative sequence number** that acknowledges **all** packets up to and including the given value.

  `sSeqN` - Acknowledges the **given** packet, identified by this `sSeqN` value. A value of ***-1** means this field is *invalid/not present*.

- ***FIN***

  `|5|timestamp|seqN|`

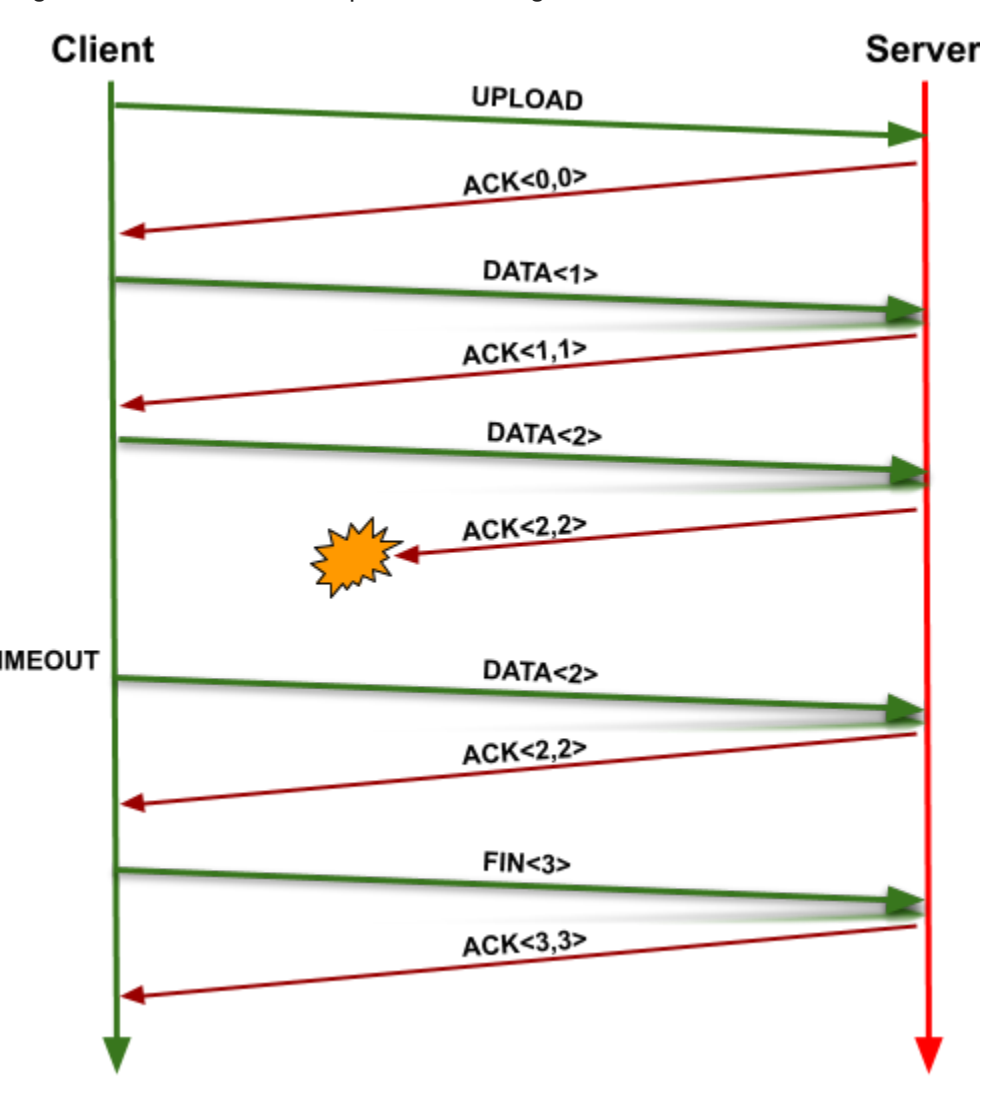
  Signals the file transfer is complete. The sequence number (`seqN`) should be 1 past the last block of the file.

When present, the `timestamp` field is expected to contain the time the packet was sent. For ACK packets, the `timestamp` is obtained/copied from the packet that generated that ACK packet. Used in this way, the `timestamp` can be used to sample the RTT between the client and server nodes.

The Cumulative Sequence Numbers issued by the server support the **Go back N** (GBN) version of the sliding window protocol. Selective Sequence Numbers can be used to support the **Selective Repeat** version of the same protocol.

FT20 File Transfer

The figure below illustrates the packet exchange between the client and server, using the `Stop&Wait` protocol.



To achieve a reliable file transfer, `UPLOAD`, `DATA` and `FIN` packets are retransmitted until acknowledged by the server.

CNSS Simulation

FT20 Packets

To implement the FT20 Protocol in CNSS, we require a way to encode FT20 packets as CNSS data packets.

FT20Packet.java is a Java class to encode FT20Packets into arrays of bytes, which can be used as the payloads of CNSS DataPackets. Conversely, the same class allows an array of bytes to be decoded into an FT20Packet. The only relevant public methods are `encodeToBytes()` and `decodeFromBytes()`.

Companion classes: FT20_UploadPacket, FT20_DataPacket, FT20_AckPacket, FT20_FinPacket, FT20_ErrorPacket model each specific FT20 packet type. These classes expose FT20 Packet fields as public **immutable** data. In these classes, the public constructors is the only way to assemble new FT20 packets.

The box below reproduces the FT20_UploadPacket class, which models the UPLOAD packet.

```
%%writefile src/ft20/FT20_UploadPacket.java

package ft20;

public class FT20_UploadPacket extends FT20Packet {
        public int timestamp;
        public final String filename;

        FT20_UploadPacket(byte[] payload) {
                super(payload);
                this.timestamp = super.getInt();
                this.filename = super.getString();
        }

        public FT20_UploadPacket(String filename, int timestamp) {
                super(UPLOAD);
                this.timestamp = timestamp;
                this.filename = filename;
                super.putInt(timestamp).putString(filename);
        }

        public String toString() {
                return String.format("UPLOAD<%s, %d>", filename, timestamp);
        }
}
```

FT20 Nodes

The *client* and *server* nodes of the FT20 protocol can be implemented in CNSS directly on top of the ApplicationAlgorithm or AbstractApplicationAlgorithm classes.

Alternatively, the client and server nodes can extend the FT20AbstractApplication class. This way, the handling of FT20 packets is greatly simplified. Namely, this class uses the FT20_PacketHandler interface to provide specific *upcalls* for each of the FT20 packet types.

Stop&Wait Example

- ***SERVER***

```
%%writefile src/FT20Server.java

import java.io.*;
import java.util.*;

import cnss.simulator.*;
import ft20.*;

public class FT20Server extends FT20AbstractApplication implements FT20_PacketHandler {

        private int windowSize; // by default in blocks

        private SortedMap<Integer, byte[]> window = new TreeMap<>();

        private int nextSeqN;
        private String filename;
        private FileOutputStream fos;

        public FT20Server() {
                super(true, "   FT20-Server");
        }

        @Override
        public int initialise(int now, int nodeId, Node self, String[] args) {
                super.initialise(now, nodeId, self, args, this);
                if( args.length != 1 ) {
                        System.err.println( this.getClass().getSimpleName() + " missing windowSize argument [in config file]");
                        System.exit(-1);
                }
                this.windowSize = Integer.parseInt(args[0]);
                this.fos = null;
                this.nextSeqN = 0;
                return 0;
        }

        @Override
        public void on_receive_upload(int now, int client, FT20_UploadPacket upload) {
                if (nextSeqN <= 1) {
                        super.sendPacket(now, client, new FT20_AckPacket(0, 0, upload.timestamp));
                        nextSeqN = 1;
                        window.clear();
                        filename = upload.filename;
                } else
                        super.sendPacket(now, client,
                                        new FT20_ErrorPacket("Unexpected packet type...[Already initiated a transfer...]"));
        }

        @Override
        public void on_receive_data(int now, int client, FT20_DataPacket data) {

                // outside the window.
                if (data.seqN < nextSeqN || data.seqN > nextSeqN + windowSize)
                        super.sendPacket(now, client, new FT20_AckPacket(nextSeqN - 1, -1, data.timestamp));
                else {

                        window.putIfAbsent(data.seqN, data.block);

                        //try to slide window and flush to disk.
                        byte[] block;
                        while ((block = window.remove(nextSeqN)) != null) {
                                writeBlockToFile(block);
                                nextSeqN += 1;
                        }
                        super.sendPacket(now, client, new FT20_AckPacket(nextSeqN - 1, data.seqN, data.timestamp));
                }
        }

        @Override
        public void on_receive_fin(int now, int client, FT20_FinPacket fin) {
                if (window.isEmpty() && nextSeqN == fin.seqN)
                        super.printReport( now );

                super.sendPacket(now, client, new FT20_AckPacket(fin.seqN, fin.seqN, fin.timestamp));
        }

        private void writeBlockToFile(byte[] data) {
                try {
                        if (fos == null)
                                fos = new FileOutputStream("copy-of-" + filename);
                        fos.write(data);
                } catch (Exception x) {
                        System.err.println("FATAL ERROR: " + x.getMessage());
                        System.exit(-1);
                }
        }
}
```

- ***CLIENT***

```
%%writefile src/FT20ClientSW.java

import java.io.*;

import ft20.*;
import cnss.simulator.*;

public class FT20ClientSW extends FT20AbstractApplication implements FT20_PacketHandler {

        static int SERVER = 1;

        enum State {
                BEGINNING, UPLOADING, FINISHING
        };

        static int DEFAULT_TIMEOUT = 1000;

        private File file;
        private RandomAccessFile raf;
        private int BlockSize;
        private int nextPacketSeqN, lastPacketSeqN;

        private State state;

        public FT20ClientSW() {
                super(true, "FT20-ClientSW");
        }

        public int initialise(int now, int node_id, Node nodeObj, String[] args) {
                super.initialise(now, node_id, nodeObj, args, this);

                raf = null;
                file = new File(args[0]);
                BlockSize = Integer.parseInt(args[1]);

                state = State.BEGINNING;
                lastPacketSeqN = (int) Math.ceil(file.length() / (double)BlockSize);

                sendNextPacket(now);
                return 0;
        }

        private void sendNextPacket(int now) {
                switch (state) {
                case BEGINNING:
                        super.sendPacket(now, SERVER, new FT20_UploadPacket(file.getName(), now));
                        break;
                case UPLOADING:
                        super.sendPacket(now, SERVER, readDataPacket(file, nextPacketSeqN, now));
                        break;
                case FINISHING:
                        super.sendPacket(now, SERVER, new FT20_FinPacket(nextPacketSeqN, now));
                        break;
                }
                self.set_timeout(DEFAULT_TIMEOUT);
        }

        public void on_timeout(int now) {
                super.on_timeout(now);
                sendNextPacket(now);
        }

        @Override
        public void on_receive_ack(int now, int client, FT20_AckPacket ack) {
                switch (state) {
                case BEGINNING:
                        state = State.UPLOADING;
                case UPLOADING:
                        nextPacketSeqN = ack.cSeqN + 1;
                        if (nextPacketSeqN > lastPacketSeqN)
                                state = State.FINISHING;
                        break;
                case FINISHING:
                        super.log(now, "All Done. Transfer complete...");
                        super.printReport( now );
                        return;
                }
                sendNextPacket(now);
        }

        private FT20_DataPacket readDataPacket(File file, int seqN, int timestamp) {
                try {
                        if (raf == null)
                                raf = new RandomAccessFile(file, "r");

                        raf.seek(BlockSize * (seqN - 1));
                        byte[] data = new byte[BlockSize];
                        int nbytes = raf.read(data);
                        return new FT20_DataPacket(seqN, timestamp, data, nbytes);
                } catch (Exception x) {
                        throw new Error("Fatal Error: " + x.getMessage());
                }
        }
}
```

- ***Configuration***

```
%%writefile configs/config-2.1.txt
# A network with a sender node and a receiver node interconnected
# by a direct link. The link has 2 Mbps bandwidth and 20 ms latency


# uncomment if you want to see control algorithms traces
# parameter trace


Node 0 1 cnss.lib.EndSystemControl FT20ClientSW earth.jpg 1000
Node 1 1 cnss.lib.EndSystemControl FT20Server 5

Link 0.0 1.0 2000000 20 0.25 0.0
```

- ***Execution***

```
%%bash
javac -cp .:cnss-classes -d ft20-classes src/*/*.java src/*.java

java -cp .:cnss-classes:ft20-classes cnss.simulator.Simulator configs/config-2.1.txt > results.txt ; cat results.txt
```

Assignment-2 Deliverables

There are three deliverables:

- Delivery 1: GoBackN: MANDATORY, evaluation up to 15 marks
- Delivery 2: Selective Repeat: OPTIONAL, evaluation up to 3 marks
- Delivery 3: Adaptative Timeouts, either for GoBackN or for Selective Repeat, OPTIONAL, evaluation up to 2 marks

IMPORTANT: study chapter 6 of the book supporting the course to fully understand these protocols.

First Deliverable - GoBackN

Implement the sliding window version of the FT20Protocol, using the **GoBackN** technique.

Your *client* should be implemented in a class named `FT20ClientGBN`. It has to accept 3 arguments, in this order:

`filename` - the file being transferred;

`blocksize` - the size of file blocks sent in each `DATA` packet;

`windowsize` - the capacity of the window in number of blocks.

For tests use the provided server FT20Server. Your client should work correctly with the provided server. You must ***not change*** the server in any way.

Use a fixed, default timeout value of 1000 ms.

Watch out for corner cases, such as: files that fit in a single block, files with length that is whole multiple of the blocksize.

Test your implementation against configurations config-2.1 config-2.2 config-2.3

Your solution should be as general as possible. It will be tested against a variety of configurations in addition to those provided.

Confirm the transfered file is identical to the original. You can use the `diff` command for that.

Important Notes

- Look at the implementation constraints at the end of this notebook. Violating them will penalize your final evaluation.
- If you only implement this deliverable, your work evaluation will be at most 15 marks.

Optative Deliverable - Selective Repeat.

Improve your previous delivery by using the techniques introduced by the **Selective Repeat** version of the protocol.

Your *client* should be implemented in a class named `FT20ClientSR`.

Use a fixed default timeout value of 1000 ms.

Leverage the `sSeqN` field present in `ACK` packets to selectively retransmit expired packets.

Using the provided configuration files, compare the results of this version against those for the GBN version of the first deliverable.

Also test your implementation against configuration config-2.4

Important Notes

- Look at the implementation constraints at the end of this notebook. Violating them will penalize your final evaluation.
- If you implement this optative deliverable, your work evaluation can be increased by up to 3 marks.

Optative Deliverable - Adaptative timeout

You can leverage the `timestamp` field present in most packets to estimate RTT and improve the performance of your best solution (the GBN one if you have not implemented the SR one, or the SR one if you have implemented it) using an adaptative timeout.

Your *client* should be implemented in a class named `FT20ClientGBN_DT` or `FT20ClientSR_DT` depending on your best solution.

The fixed timeout value used in the previous solutions should be replaced by a dynamic value based on an estimate of the network RTT between client and server.

IMPORTANT: please study section 7.2 of the book supporting the course for more information on TCP dynamic timeouts which you may use as inspiration

Compare the results of this version against those for the fixed timeout versions of the previous deliverables, using the provided configuration files.

Important Note

- If you implement this optative improvement, your work evaluation can be be increased by up to 2 marks.

Statistics

A statistics report should be presented at the end of each file transfer.

Refer to the provided reference Stop&Wait version FT20ClientSW to see how to produce the statistics report.

Implementation Constraints

You **must not change** the provided **FT20_* classes** in any way. Note that for evaluation purposes, your code will be tested against the **original** versions of the files.

You are only allowed to send 1 packet, on each 1 ms `on_clock_tick` round. Retransmitting *expired packets* takes priority compared to sending fresh packets added to the sliding window. In other words, *older packets* take priority.

You cannot send `DATA` packets in the `on_timeout` upcall. `UPLOAD` and `FIN` packets are allowed.