Reliable UDP

This project implements a client-server layer on top of UDP. This was made for CIS 427 - Networking.

Install

This example uses node.js as the vm to run the files. Install the latest version of node.js from here.

To get the example project running, first clone this repo, `cd` into it, and install the dependencies:

```
git clone https://github.com/ricokahler/cis427-p2-kahler-rico
cd cis427-p2-kahler-rico
npm install
```

Then to run the alice example, start the server by running this command:

```
npm start
```

Then in a separate terminal or command prompt, start the client by running this command:

```
npm run client
```

You should get screens like the following:

Screenshots

Tests

Though the alice example shows reliable udp working, it doesn't show the desired actions of the protocol.

**Please see the `tests.ts` test file and the corresponding `tests_output.txt` to show the actions of the protocol at work.

You can also run the tests by running the following command:

```
npm test
```

If you'd like to see logs of the protocol, uncomment the `logger` property in the `tests.ts` file for any corresponding test you'd like to see. For example

```
// rest of test...
const receiver = createReceiver({
  dataSegmentStream,
  sendAckSegment,
  segmentSizeInBytes,
  //
=====================================================================================
======
  logger: console.log.bind(console), // uncomment to enable logging
  //
=====================================================================================
======
});
// rest of test...
```

Usage

This project exposes the following interface for any arbitrary application to use:

```
interface ReliableUdpServerOptions {
  port: number,
  segmentSizeInBytes: number,
  windowSize: number,
  segmentTimeout: number,
  logger?: (logMessage: string) => void,
}

interface ReliableUdpServer {
  connectionStream: Observable<ReliableUdpSocket>,
  rawSocket: Udp.Socket,
  close(callback?: () => void): void,
  bind(callback?: (port: number) => void): void,
}
```

```
interface ReliableUdpClientOptions {
  hostname: string,
  port: number,
  segmentSizeInBytes: number,
  windowSize: number,
  segmentTimeout: number,
  connectionTimeout: number,
  logger?: (logMessage: string) => void,
}

interface ReliableUdpSocket {
  info: Udp.AddressInfo,
  messageStream: Observable<Buffer>,
  sendMessage: (message: string | Buffer) => Promise<void>,
}
```

And are used like so:

Here are the alice client and server examples in full:

server-example.ts

```
import { createReliableUdpServer } from '../rudp';
import * as fs from 'fs';
import * as path from 'path';
import { oneLine } from 'common-tags';

const aliceTxt = fs.readFileSync(path.resolve(__dirname, './alice.txt'));

const rudpServer = createReliableUdpServer({
  segmentSizeInBytes: 100, // 1kB for segment size
  logger: console.log.bind(console),
});

rudpServer.connectionStream.subscribe(async connection => {
  console.log('Client connected! ', connection.info);

  connection.messageStream.subscribe(message => {
    console.log('MESSAGE FROM CLIENT: ', message.toString());
  })

  console.log('Sending alice.txt...');
  await connection.sendMessage(aliceTxt);
});

rudpServer.bind(port => console.log(
  oneLine`Reliable UDP server running on port: ${port}.
  Run the command 'npm run client' in another terminal to start the download.`
));
```

client-example.ts

```
import { connectToReliableUdpServer } from '../rudp';

async function main() {
  try {
    const socket = await connectToReliableUdpServer({
      logger: console.log.bind(console),
      segmentSizeInBytes: 100, // 1kB
    });

    console.log('Connected to server! Downloading all of alice.txt...');
    const aliceTxt = await socket.messageStream.take(1).toPromise();

    console.log(aliceTxt.toString());

    socket.messageStream.subscribe(message => {
      console.log('MESSAGE FROM SERVER: ', message.toString());
    });

    socket.sendMessage('Hello server!'); // can send messages bi-directionally
  } catch (e) {
    console.error('======');
    console.error('ERROR: Could not connect to server. Ensure that it is running
first.');
    console.error('======');
    console.error(e);
    process.exit(1);
  }
}

main();
```

Design

Segment interfaces

Since this rUdp implementation does *not* pipeline segments (i.e. does *not* send data and acks in the same segment), there are two interfaces for the two types of segments:

The **`DataSegment`** which is sent by the sender and received by the receiver.

```
interface DataSegment {
  messageId: string,
  seq: number,
  data: Buffer,
  last?: true,
}
```

And the **`AckSegment`** which is sent by the receiver and sent by the receiver.

```
interface AckSegment {
  messageId: string,
  ack: number,
}
```

Sender and receiver

The above segment interfaces are used by the `sender` and `receiver`. The sender and receiver are implemented by the two functions `createSender` and `createReceiver`.

`createSender` in **`sender.ts`**

The `createSender` is the factory that creates the sender. The sender is a function that takes in a message and returns a promise that will resolve when the messages has been completely sent with acknowledgement.

The `createSender` function takes in a configuration object with the following interface:

```
interface SenderOptions {
  sendDataSegment: (dataSegment: DataSegment) => Promise<void>,
  ackSegmentStream: Observable<AckSegment>,
  segmentSizeInBytes: number,
  windowSize: number,
  segmentTimeout: number,
  logger?: (logMessage: string) => void,
}
```

The `createSender` function returns a function that sends data segments by using the `sendDataSegment` function given through the `SenderOptions`.

The `createSender` function then consumes the given `ackSegmentStream` to move the sliding window and send more data segments.

See the example below:

```
async function sendHelloWorld() {
  const sendMessage = createSender({
    sendDataSegment: (dataSegment: DataSegment) => /* some UDP implementation */,
    ackSegmentStream: udpStream.filter(rawMessage => /* some condition to keep only
segments*/),
    segmentSizeInBytes: 100, // number of bytes of data in each segment
    windowSize: 6,
    segmentTimeout: 1000,
    logger: logMessage => console.log(logMessage),
  });

  await sendMessage('hello world!');
  console.log('finished sending!');
}
```

The sender creates a sliding window by sending as many segments as the window size. Each segment is sent with the function `sendDataSegmentAndWaitForAck` that creates promise. This promise resolves when it has been properly ACKed. In order to do this, the `ackSegmentStream` is filtered upon until it finds an `ACK` that is greater than segment's `SEQ` number:

```
(segmentStreamForThisMessage
  .filter(ackSegment => ackSegment.ack > segment.seq)
  // ...
)
```

When the segment is sent and resolved with an ACK, it procedes to send the next segment that hasn't been sent. If the segment fails to send (either by timeout or some other error), the same function will re-transmit the segment. This will implictly implement the sliding window just by initially sending as many segments as the window size.

`createReceiver` in `receiver.ts`

The `createReceiver` function returns a stream (an RxJS `Observable`) that we can subscribe to to get the messages from a receiver.

The `createReceiver` function takes in a configuration object with the following interface:

```
interface ReceiverOptions {
  sendAckSegment: (segment: AckSegment) => Promise<void>,
  dataSegmentStream: Observable<DataSegment>,
  segmentSizeInBytes: number,
  logger?: (logMessage: string) => void;
}
```

Where the `sendAckSegment` is a function that takes a an `AckSegment` object and pushes it down UDP, and `dataSegmentStream` is an `Observable` of `DataSegment`s.

See the example below:

```
const messageStream = createReceiver({
  sendAckSegment: (ackSegment: AckSegment) => /* some UDP implementation */,
  dataSegmentStream: udpStream.filter(rawUdpMessage => /* some filter to only keep
DataSegments*/),
  segmentSizeInBytes: 100, // must match sender's
  logger: (logMessage: string) => console.log(logMessage),Ï
});
```

The receiver uses the function `findNextSeqNumber` to find gaps in an array buffer so that it can `ACK` correctly. This function is located inside the `receiver.ts` file.

That's it!

enjoy!

(please see the tests if you haven't yet.)

MIT license