Summary

This project (partially) implements a clock chain: a blockchain where each block in the chain has an associated timestamp; for clock ticks where no data is inserted, a blank block will be inserted (which still has a timestamp).

The chain must be managed by a server, which can be connected to via TCP by clients. Clients send commands to the server indicating what to add to the chain, or retrieving data from the chain.

Progress Report

This project is currently unfinished.

Here's what I *have* done:

- Implementation of a blockchain
- Server/Client communication protocol, including two commands (see **How To Use** and **Design Choices**):
    - add: makes a new entry into the chain
    - get: queries the chain
- Automatic blockchain persistence to file in CSV format
- Very basic code demonstrating the concept behind a repeating timer (i.e., a clock) using boost::asio::steady_timer.

Here is what I *have not* done:

- Initializing blockchain from a pre-written CSV file (continuing from where server previously left off)
- Integrate clock with blockchain (see **Design Choices**)

Design Choices

Client/Server Communication Protocol

- Messages from client are *sent* synchronously, since the client does not need to worry about performance
- Clients' messages are suffixed with a single newline character. This has the unfortunate side-effect that multi-line messages cannot be sent. (A better protocol would append some other string (like null-byte) which would never appear in a valid string)
- Server *reads* messages asynchronously. Tasks are processed as they become available, using boost's io_service asynchronous task management service.
- Server *responses* are suffixed with two newline characters. The full server response is displayed by the client program.

Server Commands

- The server allows the following two commands: add and get.
- "add" causes the creation of a new block, with the data consisting of whatever occurs between "add" (not incl. initial space after command name) and the terminal '\n' (not

incl. the '\n'). The server's response contains information about the block that was created.

- "get" queries a block by its number, where 0 is the genesis block, 1 is the first block after genesis, etc.
- Ideally, it should also be possible to query a block by its hash. This would probably take me an additional 2-3 hours.

Blockchain Storage

- Blockchains are stored in CSV format, where comma-separated fields represent the private non-function attributes from the Block class, and rows represent different blocks.
- Any commas appearing in the _sData field of Block are first escaped with '\' before being written to a file.
- (When this step is completed,) reading Blocks from CSV will require un-escaping the '\' chars which were inserted.
- The procedure for escaping commas was written without using <regex> or <boost/regex.hpp>, because I had a hard enough time figuring out how to use boost/asio.hpp, and didn't want to spend 3 hours figuring out how to write a proper substitution regex in C++.
- Escaping newline characters was not necessary, because the communication protocol made it impossible for clients to send multi-line data. It would require only one line of change to escape newline chars.

Clockchain Procedure

- At the moment, the chain is not clock-oriented at all, except for the fact that the blocks each contain a timestamp.
- The obvious next step is to add a procedure to automatically insert a block into the chain each minute. There are two ways this could happen:
  - Queueing each "add" so that all queued items within a minute appear in the same block.
  - Writing independent blocks for each added item.
- The second approach is easier to implement. I am not sure if it is the intended result.

Installation

Assuming you already have g++-7 and boost libraries, you should simply be able to run make:

```
$ make
```

This creates a directory (`obj64/`), which contains all executables.

How To Use

Executables

`server.exe` is the server application. It can be run without arguments, in which case it will default to running on port 4000. Another port can be chosen with the `-p` option.

client.exe is the client application. It must be run like this:

```
     $ ./client.exe hostname port_num
```

`main.exe` is the test program from the blockchain tutorial. It can be run without arguments.

`timer_test.exe` is a clock proof-of-concept. It is run without args, and will print a message once every second.