School of Computing and Information Systems
COMP30023: Computer Systems

Project 2

Due date: No later than 6:00pm on Friday, May 26. **Weight:** 15%

1 Overview

The aim of this project is to provide you with experience in writing (a) a program that interacts with other programs over a network (socket programming), and (b) multi-threaded applications. Specifically, you will write a server-based *proof-of-work* solver for the *Hashcash* mining algorithm used in Bitcoin.

This requires you to implement the functions in the Simple Stratum Text Protocol (SSTP), which is used to format messages between clients and your `server` program. The SSTP is described in Section 3. Background information related to the the Bitcoin mining process is described in Section 2. Detailed requirements and implementation guidelines are listed in Section 4.

Your program must be written in standard C. Use the POSIX pthread library for multi-threaded tasks. Other dynamic and static libraries are prohibited.

Your program must run on the School of Engineering Linux server `digitalis` or `digitalis2`.

2 Background

2.1 Introduction to Bitcoin

Bitcoin is the world's first successful cryptocurrency and a payment network that is currently valued at more than 20 billion dollars. The system follows a fully decentralised peer-to-peer network topology and transactions take place between users directly without an intermediary. Transactions are verified by network nodes and recorded in a distributed ledger, commonly known as the *blockchain*.

Bitcoins are created as a reward in a competition where miners offer their computing power to record bitcoin transactions into the blockchain. This activity is called *mining* and successful miners are rewarded with transaction fees and the ability to create Bitcoins to pay to themselves[1].

In this project, we will not explore the purpose of mining in detail. However, we will focus on the steps necessary for you implement a server program that can execute the mining process using the SHA-256 algorithm. To get you started, please read the document

`https://en.bitcoin.it/wiki/Proof_of_work`

It is important to note, that in general, it is very difficult to produce a valid *proof-of-work* solution of the mining process as there are 2^{256} possible outputs for some random input. Therefore, this is a problem that should be solved with some means of multi-threading, which enables you to distribute the workload.

[1]details: you can lookup coinbase transaction in Bitcoin if you are interested

2.2 Hashcash Algorithm

The Hashcash algorithm used in the Bitcoin mining process is described below:

Find `x` such that `H(H(x)) = y` and `y` $<$ `target`

- H is the SHA-256 hash function (see subsection 2.3)
- `target` is a 256-bit unsigned integer, which represents the largest acceptable hash value of some input.[2]

  See Section 4–Stage B for details on how to calculate the `target` value from an appropriately formatted input protocol message (as described in Sections 3)
- `x` is defined as a `seed` value concatenated with `nonce` i.e., `seed | nonce`

  where:

  `seed` is a 64 byte array, which is provided by a client through a protocol message (see Section 3 for details)

  `nonce` is a 64 bit unsigned integer (`nonce` basically means "number once" ie., an arbitrary number that may only be used once.)[3]

Hence, the goal is to find a `nonce` value, which when concatenated with `seed` creating `x`, yields a hash that is less than `target`. To do this, the search starts from some initial `nonce` value and checks whether the conditions listed above hold. If they do, great, you have a solution. If not, increment the `nonce` value and try again (and again and again ... until a solution is found).

2.3 Cryptographic Hash Function SHA-256

Bitcoin's mining algorithm, Hashcash, utilises a cryptography hash function known as the Secure Hash Algorithm (SHA-2). SHA-2 is a collection of six hash functions with digests (hash-values) that are 224, 256, 384 or 512 bits. For the purposes of our task, we will only be using the SHA-256 to produce a *proof-of-work*.

Knowing the specifics of how SHA-256 works is not necessary for the purposes of this project. A reference implementation of SHA-256 has been provided for you. *The source code can be downloaded from the LMS.*

[2]we have intentionally elected not to describe the input, as the actual input used in Bitcoin will differ from the project specification – to find out more about the actual input used, you should consult the Bitcoin wiki page on Mining

[3]to learn about what the actual format of the input is for a proof of work function, check out: https://en.bitcoin.it/wiki/Proof_of_work

3 Simple Stratum Text Protocol (SSTP)

Before working through the SSTP messages listed below, you should familiarise yourself with the primitive types provided by the standard C library. In this project, you will also make use of:

- `BYTE : unsigned char`
- `uint256 :  BYTE[32]` *You can download source code for* `uint256` *from the LMS*

Your server-based *proof-of-work* solver will use the Simple Stratum Text Protocol (SSTP).

Every protocol message should be delimited with the carriage return, followed by a line feed (i.e., `\r\n`). You may assume that every protocol message has a pre-defined length. Any message exceeding this pre-defined length will be considered an invalid message and you must handle this correctly with an `ERRO` message. Every message is comprised of a message header (4-bytes in length) followed by its payload (the length of which depends on the actual message). All network messages must be encoded and parsed in network byte order (i.e., big endian order).

There are seven messages in the protocol – **your task in this project is to implement all of them**

3.1 Ping Message `PING`

```
PING
```

On receive, the `server` replies with a `PONG` message.

3.2 Pong Message `PONG`

```
PONG
```

Your `server` should reply with an error message `ERRO` informing the client that `PONG` messages are strictly reserved for `server` responses.

3.3 Okay Message `OKAY`

```
OKAY
```

Your `server` should reply with an error message `ERRO` explaining that is not “okay” to send `OKAY` messages to the `server`.

3.4 Error Message `ERRO`

```
ERRO reason:  BYTE[40]
```

Example:

```
ERRO with an appropriate explanation
```

On receive, your `server` should reply with an error message `ERRO` indicating that this message should not be sent to the server.

3.5 Solution Message `SOLN`

```
SOLN difficulty:uint32 seed:BYTE[64] solution:uint64
```

Example:
```
SOLN 1fffffff 0000000019d6689c085ae165831e934ff763ae46a218a6c
172b3f1b60a8ce26f 10000000232123a2
```

(*Note*: line wrap used in `seed` simply for formatting on the project spec)

On receive, your `server` should check if the concatenation of the `seed` and the `solution` (which is actually a particular `nonce` value) does in fact produce a hash which satisfies the `target` requirement derived from the `difficulty` value (see Section 4–Stage B below for details). Here, an 8 byte hexadecimal representation has been used for the `difficulty`.

If it is a valid *proof-of-work*, then you should reply with `OKAY`. Otherwise, reply with `ERRO` with a 40-byte string describing what the error is.

3.6 Work Message `WORK`

```
WORK difficulty:uint32 seed:BYTE[64] start:uint64 worker_count:uint8
```

Example:
```
WORK 1fffffff 0000000019d6689c085ae165831e934ff763ae
46a218a6c172b3f1b60a8ce26f 1000000023212399 02
```

(*Note*: line wrap used in `seed` simply for formatting on the project spec)

Once again, an 8 byte hexadecimal representation has been used for the `difficulty`. `start` is the initial `nonce` value[4] and `worker_count` is the number of threads to use in the computation task. See section 2.3 for an explanation of the computation steps.

On receive, your `server` should queue this work in a work queue. Once a *proof-of-work* solution is found, reply with the `SOLN` message with the correct `nonce` (i.e., the solution field in the `SOLN` message) which produces the *proof-of-work* solution.

3.7 Abort Message `ABRT`

```
ABRT
```

On receive, your `server` should traverse the work queue and remove all pending work for the client that sent this message. It should also pre-maturely terminate all active proof of work worker threads. Once this is completed, an `OKAY` response should be sent.

[4]for example, if `start` was set to 2 instead of 0, you may ignore testing if 0..2 may be a valid nonce for your proof of work.

4 Requirements

We strongly recommend that you complete the project in four clearly defined stages listed below.

Stage A

Create a simple text-based `server` capable of handling multiple concurrent requests. This basically means extending the TCP `server` program introduced in the labs so that it can correctly reply to simple messages received from client programs concurrently.

You will have to make a decision as to the best way to deal with concurrent requests from clients. Begin by implementing a simple plain-text server that is capable of receiving and sending `PING`/`PONG` messages. Then add the `OKAY` and `ERRO` messages.

Your `server` should also be able to receive requests (messages) from *more than* one client at any given time and still be able to manage the processing of protocol messages between all connected clients.

Stage B

Extend your `server` program from Stage A so that when an appropriately formatted message is received from a client, your `server` program will parse the message and perform appropriate computations on the parsed message to verify whether the message is a valid solution for the *proof-of-work*. That is, your `server` must be capable of validating whether a properly formatted `SOLN` message contains a valid *proof-of-work*.

To make things easier for you in Stages B (and Stages C and D) helper functions (i.e., uint256 functions and functions to execute SHA-256) have been provided for you. *You can download the source code from the LMS.*

Given the fact that the `target` requires 64 bytes of text space within a block protocol message, a mapping between the `target` and `difficulty` in the `SOLN` message is used. Here, `difficulty` is represented using a 32 bit unsigned integer and the corresponding mathematical formula to compute the `target` is:

$$\texttt{target} = \beta \times 2^{8\times(\alpha-3)}$$

where: α and β correspond to particular bit values in the `difficulty`. That is, $\alpha =$`difficulty[0..7]` and $\beta =$`difficulty[8..31]` (or bits 0-7 and 8-31 respectively). Assume that `difficulty` is encoded in big endian byte order. *Hint*: use bit manipulation to extract α and β.

When your `server` is processing a `SOLN` message, it still should be able to receive requests from multiple clients and manage the processing of protocol messages between all connected clients. For example, your `server` must be capable of immediately responding to `PING` messages with a `PONG` for a separate client.

Stage C

Your `server` program will be extended again such that it can compute a *proof-of-work* for a given `WORK` message by parsing the input received from the client and creating a specific number of threads to do the actual computations. The solution verification tasks completed in the previous Stage will also be used here.

To complete this task, you will have to maintain a work queue based on the current submitted client `WORK` messages. You will need to implement a `server` function that finds a valid *proof-of-work* for each job in the queue (in turn) by iteratively incrementing the `nonce` based on a properly formatted `WORK` message. When you find a solution, the `server` replies with the `SOLN` message. To keep things simple, in this Stage you should just use one thread to do the computation.

While searching for the *proof-of-work*, your `server` implementation must still be able to respond to all requests from clients. For example, while the search for a *proof-of-work* solution is carried out, your server must be capable of immediately responding to `PING` messages with a `PONG` etc.

Stage D: Bonus marks

In this final bonus mark Stage, you will further extend your `server` program so that multiple threads (workers) can be used to do the computation (or searching for a *proof-of-work*). The actual number of threads to be used will be given in the `WORK` message.

Your `server` will need to handle incomplete messages (i.e., messages that have not ended with a delimiter). Your `server` must store incomplete messages in a buffer and continue to receive the subsequent segments of incomplete messages until the message is delimited or exceeds the maximum message length.

On client disconnection, all queued work from the client should be removed from the work queue. Active worker threads working on searching for *proof-of-work* solutions should also stop working immediately and should start working on the next work item in the work queue.

Server Log File

You must continually write to a log file (`./log.txt`)[5] detailing each (and every) interaction between the `server` and individual clients. Each log file entry will include:

- a time-stamp (system date/time)
- IP address of the client or `0.0.0.0` for the `server`
- a socket id (or file descriptor) for the client connection
- SSTP message for the exchange between `server` and client

Note: Your log file should reflect the Stages of the project task that you have completed. If your `server` crashes you may not have any entries in your log file – so take care.

Limits

The following limits will be used:

- maximum number of clients: 100
- maximum number of pending jobs: 10
- maximum number of processors: unbounded
- maximum number of threads: unbounded

One final comment: there is no need to handle unsolvable `WORK` messages. We will not be giving you any unsolvable work.

[5]Every time your start your `server`, you should create a new `log.txt` file

Program execution / command line arguments

To run your `sever` program on `digitalis2.eng.unimelb.edu.au`

```
prompt: ./server [port_number]
```

where `[port_number]` is a valid port number (e.g., 12345) entered via a command line argument

To test your `server` program, you can use `telnet` or `nc`. Alternatively, you can use a simple `client` program (available on the LMS).

To test multiple concurrent clients, you can write a short Linux script to call `client` multiple times.

5 Submission Details

Please include your *name* and *login_id* in a comment at the top of each file.

Submission will follow the same protocols as Project 1: your SVN repository (including your log file) will be harvested after the due date.

`https://svn.eng.unimelb.edu.au/comp30023/username/project2`

Please add/commit **all** source code files `.c` and `.h` (including the helper files `uint26` and `sha256`) and a Makefile.

Using SVN is an important step in the verification of authorship.

Our plan is to also set up `submit` on the School of Engineering Linux servers. However, testing multi-threaded programs in an automated way is challenging (especially when submitted program have problems / timeout / seg fault).

See the LMS for updates.

Late submissions: will be possible. However, a late submission penalty will apply. A *flagfall* of 2 marks, and then 1 mark per 12 hours late will be applied. If you submit late, please email the lecturer Michael Kirley <`mkirley@unimelb.edu.au`> when you submit.

Extension policy: If you believe you have a valid reason to require an extension you must contact the lecturer, Michael Kirley <`mkirley@unimelb.edu.au`> at the earliest opportunity, which in most instances should be well before the submission deadline. It is university policy that projects/assignment work cannot have a due date that falls during 'Swotvac' (the week before exams starts). Given the fact that the project was released at the end of Week 9, extensions will only be awarded under extreme circumstances. Requests for extensions are not automatic and are considered on a case by case basis. You will be required to supply supporting evidence such as a medical certificate. Asking for an extension on the due date (even with a medical certificate covering Friday May 26) will almost certainly received a negative response.

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

6 Assessment

Code that does not compile and run on `digitalis.eng.unimelb.edu.au` may be awarded zero marks.

If you do not use your SVN repository for the project you may be awarded zero marks.

An automated script will be used to verify whether your program output matches the expected output for given test cases. In some circumstances, your program may be testes manually.

Sample test cases are included in the `client` program (available on the LMS). Each test explicitly examines whether your `server` program can handle a request based on a single simple message (e.g., `PING`) as well as verifying whether your program correctly evaluates a given solution example (`SOLN`). We will also test whether your program finds a solution for a *proof-of-work* (`WORK`) message.

In all cases, we will test whether your `server` program can handle multiple concurrent requests.

Your submission will be tested and marked with the following criteria:

Log file

- The log file documents "basic" interactions between your `server` and a single client using the requirements listed above - **1 Mark**.
- The log file correctly documents interactions between your `server` and multiple concurrent requests from multiple clients - **1 Mark**.

Stage A

- Working networking code for `server` (capable of processing PING/PONG/OK/ERRO messages) - **2 Marks**.
- `server` is capable of handling concurrent requests from clients - **3 Marks**.

Stage B

- Correct processing of a `SOLN` message - **3 Marks**
- `server` is capable of handling concurrent requests from clients - **1 Mark**.

Stage C

- Correct handling and processing of a `WORK` message with one worker/thread - **2 Marks**.
- `server` is capable of handling concurrent requests from clients - **1 Mark**.
- Correct full implementation of all application protocol messages, including appropriate ERRO message and ABRT - **1 Mark**.

Stage D: Bonus marks

- Correct handling and processing of a `WORK` message using multiple worker/threads - **2 Marks**.

7 Bug Reports

If you find a bug in the the provided source code, send an email message to one of the subject tutors, Renlord Yang <`rnyang@student.unimelb.edu.au`> with a detailed description of how to replicate the bug and a sample source snippet. Your toolchain, distribution and compilation flags should also be included as part of your bug report.

8 Help Me!

Tutorials and labs in Week 10 are allocated to the project.

You are encouraged to post questions on the LMS to seek help for this project, especially if you are unsure of how *proof-of-work* actually works in the context of this project.

9 Just for fun ..... ZERO marks

Please contact Renlord Yang <`rnyang@student.unimelb.edu.au`> if you are interested in attempting to solve a *very difficult* Bitcoin puzzle.