Bitcoin-ish Miner

CPL Project (inspired by a CMU 15-440 project)

Bernardo Toninho

November 12, 2020

Bitcoin, cryptocurrencies and the blockchain in general are extremely hot topics in Computer Science these days (possibly due to the heat generated by computers that employ such technologies).

To play along with current trends, you are going to implement a cryptocurrency miner, inspired by the Bitcoin protocol, albeit in a simplified form.

In distributed ledgers such as Bitcoin and other cryptocurrencies, a replay prevention mechanism is in place to prevent anyone from spending their currency multiple times. This mechanism is based on a so-called proof-of-work function, designed to be computationally hard to generate but easy to verify. In such systems, clients compete to be the first to find a solution to the proof-of-work, in order to attach their signature to a certain sequence of transactions. If a client "wins", they are rewarded with bitcoins. The process of finding solutions to the proof-of-work is called *mining*.

You are not going to implement the Bitcoin protocol, nor its exact proof-of-work function, but rather a simplified variant of Bitcoin mining: given a message M and an unsigned integer N, find the unsigned integer n, that, when concatenated with M generates the largest hash value, for all n between 0 and N (inclusive). These unsigned integers are referred to as *nonces*. Your task is to implement a concurrent system that implements this simplified variant of mining, adhering to some constraints specified below.

---

1 System Specification

Your system must be composed of two software components, a **client** and a **server**. Each component **must compile to a separate binary**.

1.1 Client

The client sends user-specified requests to the server, receives and prints the result to the standard output.

The client and the server communicate using a **standard TCP socket**. The client binary should take as command line arguments the server address and port, the message **M** and an unsigned integer, **max**, denoting the upper range of nonces to be checked by the server, according to the mining description given above.

To launch a client component, the command should be:

```
./client host:port message max
```

where `host:port` is the server address and port, `message` is the message that will be concatenated with the nonce and `max` is the upper limit of the range of nonces to be checked.

If a client loses contact with the server, it should print a `Disconnected from server` message to standard output and terminate.

For convenience, the provided repository contains an implementation of the messages (package `message`) you will need to use between the client and the server, as well as their marshaling and unmarshaling to-and-from JSON byte slices.

1.2 Server

The server receives requests from one or more clients, performs the corresponding mining operation and answers back with the reply.

To launch the server, the command should be:

```
./server port
```

The server **must** be able to handle client requests concurrently. This means that at any given time, the server can be performing the mining computation for different clients (and thus of various messages and nonce ranges).

From the perspective of the server, a client request of the form (using the message formats from the supplied code – see below):

```
[Request message low high]
```

can (and should) be split into multiple smaller jobs, performed by multiple workers. The partition of the request into jobs and their aggregation into a final answer is straightforward: we can partition a request simply by splitting the [`low`,`high`] interval into chunks and computing the maximum hash value of the concatenation of `message` with nonces in the chunk. A result for a chunk is the pair consisting of the hash value and the nonce. Once we have results for multiple chunks, aggregating them is easy since we can just pick the result with the largest hash value.

If a server loses contact with a client, it should stop working on any pending requests on behalf of the disconnected client (jobs that are already underway need not be forcibly terminated, just wait for them to complete and ignore the result).

1.2.1 Additional Server Requirements:

1) The server must use multiple workers (i.e. miners) to perform the client jobs. It is up to you to choose a suitable maximum job size.
2) Choosing a good way of organizing (and creating) miners, suited to the specific context of the problem is part of the challenge. There might be several equivalent options, but you should be able to argue why your choices are good.
3) **Correctness is mandatory**. The server must only produce answers to clients that are indeed correct and that match the client's request.

4) Workloads should be managed. A client should not be able to flood the server with requests, starving other existing clients of access. **Note:** This is not so much about network control (although you have the tools to control connection acceptance rate), but about not letting clients with many small requests starve clients with few but large requests, and vice-versa.

5) The server should **try** to minimize mean response time for client requests. If the server gets a very large request followed by a small request, it is reasonable to expect that the small request finishes before the large request. If the server gets several requests of similar size, the requests should finish roughly (although not necessarily) in the order they arrived. Your code should document if/how these concerns are taken into account.

---

2 Supplied Code

- `message/message.go` defines the message types needed for client/server exchanges.
- `hash/hash.go` defines the hash function that miners will use.
- `client/client.go` is where you will implement your client program.
- `server/server.go` is where you will implement your server program.

---

3 Grading

Projects are to be completed preferably in groups of two (at most three), which must match with the groups of the mini-project. The standard plagiarism rules apply and will be enforced.

The project deadline will be on the 25th of November at 23:59, enforced by Github Classroom. You must turn in your code and a **brief** report (add a

PDF to the repository), documenting the various design choices in your work. The report should try to address the questions below.

Your code **must not** use locks or mutexes. All synchronization must be done using goroutines, channels, and Go's channel-based `select`. Waitgroups may be used to ensure a program does not terminate before it should. Any of the techniques seen in lecture may be used (but they need not all be used!).

You should try to avoid using fixed-size buffers and arrays to store things that can grow arbitrarily in size. For example, using fixed-size buffers for message queues might not be the best solution (which doesn't mean you should not use buffered channels entirely!). Its often better to use a linked list (see the `container/list` package, but note that it is **not** thread safe) or some other data structure that can expand to arbitrary size.

Your project will receive a better grade according to the following criteria:

- **Correctness:** Does the server answer correctly? (Critical!)
- **Worker creation:** When and how are workers created?
- **Worker management:** How are workers organized amongst themselves and in the overall system? How is work divided among and allocated to workers? How is work aggregated?
- **Client rate management:** Are clients granted unlimited resource usage?
- **Fairness and responsiveness:** How does the server aim to minimize mean response time?
- **Overall code quality**

The criteria are not listed in any particular order. The main focus will be on correctness and worker creation/management.

Note: The project may seem scary, but its bark is worse than its bite. Relax and try to have some fun!