Project 1: LiteMiner

Due: 11:59 PM, Feb 12, 2019

Contents

1 Introduction

Welcome to CS 1380! This project is a self-contained introduction to many of the concepts that you'll be using frequently in the CS 1380. You'll be implementing a real distributed system while getting some practice using concurrency control mechanisms in Go. Before you dive into this project, make sure you've signed and turned in the collaboration policy and that you've read the coding guidelines that we've posted on the website.

2 Cryptocurrencies

2.1 Decentralization

A cryptocurrency is a digital currency that enables trusted and secure financial transfer without the involvement of a centralized entity. Many cryptocurrencies use a distributed consensus algorithm called the *blockchain* which we will study later. Key to cryptocurrencies is the use of peer-to-peer networking principles, the notion of proof of work, and the use of cryptography to maintain security and authenticity. In this assignment, we will focus on the peer-to-peer and proof of work aspects.

Central to peer-to-peer systems is coordination between a dynamically set of changing nodes. In the case of cryptocurrencies, these nodes are miners which may drop in and out of the network for various reasons: a miner may crash (or reboot), or a new miner may be added. These changes to the network happen in an unscheduled and unpredictable manner and special care must therefore be taken to ensure that no information is lost. To further complicate peer-to-peer systems, the underlying networks used for communication between these nodes can be highly unpredictable: messages can get lost, reordered, or duplicated. In designing a cryptocurrency mining system, one must tackle all of these challenges.

2.2 Proof of Work

In cryptocurrencies, each transaction is validated by performing a computationally intensive task. This is often referred to as *proof of work*. A common proof of work scheme involves finding a random number, referred to as a *nonce*, that when concatenated to a string and hashed results in a new string that has a certain property. In Bitcoin, for example, the proof of work entails finding the nonce that, when concatenated with the transaction data and hashed, results in a string that starts with a series of zeros. The fundamental insight is that doing the work to find such a nonce is computationally difficult and requires a lot of resources. In order to "cheat" the system, one needs to have significantly more resources than other participants.

2.3 Miners

A node in a cryptocurrency network that attempts to verify transactions by performing proof of work is generally referred to as a miner. Given a financial transaction encoded as a string M and an unsigned integer N, a LiteMiner miner will need to find the nonce between 0 and N, inclusive, that, when concatenated with M and hashed, results in the *lowest* hash value. For example, consider the following nonces for the transaction $M =$ "msg", $N = 2$:

- `LiteMiner.Hash("msg", 0)` = 13781283048668101583
- `LiteMiner.Hash("msg", 1)` = 4754799531757243342
- `LiteMiner.Hash("msg", 2)` = 5611725180048225792

In this case, nonce 1 generates the least hash value, and thus the final result consists of the least hash value 4754799531757243342 and nonce 1. Note that you need not worry about the details of how these hash values are computed – the TAs have provided a function in the stencil code for you to use to calculate these hashes.

2.4 Pools

A pool is a collection of miners that aggregate their resources and coordinate to ensure faster mining. In essence, a pool is a divide-and-conquer approach to tackling proof of work as it splits the proof of work problem among different miners, thus enabling it to solve the problem faster. In this assignment, you will develop a pool and its corresponding miners with the following properties:

- The pool must keep track of the miners under its control and allow miners to come and go at will.
- The pool must take client requests and shard these requests (distribute the work) across all of its miners.
- The pool must aggregate miner responses and return the proof of work to the client.
- The miners must take work distributed by the pool, perform the work, and return proof of work to the pool.

2.4.1 Distribution of Work

In order to find the nonce that yields the lowest hash value for a given message, you will need to perform a brute force search of all nonces between 0 and the upper bound, inclusive. In a centralized system, this process can take a considerable amount of time. In a distributed system, you will be able to divide up the work, but you must take care when dealing with faulty miners (i.e., miners that drop in and out). One reasonable approach to this is for your pool to divide each transaction request from a client into a discrete set of intervals, each interval having a lower and upper bound which together represent a subset of the entire search space. A potential algorithm for your pool to divide up the work for a transaction request is to distribute these intervals to available miners and record the lowest hash within each of them. Once all intervals have been accounted for, you can then return the nonce which corresponds to the lowest of those hashes.

2.5 Edge cases

2.5.1 Dynamic Addition and Removal of Miners

Pools would be far less useful if they were not designed to withstand the dynamic addition and removal of miners. Thus, your pool should be able to cope with the intermittent failure of various miners and take advantage of the addition of new miners. We will be testing for this functionality!

2.5.2 Miner Liveliness

A pool needs a way to keep track of which of its miners are avalible for work and which have crashed or are unreachable due to network partitions or outages. One way to accomplish this is through status updates from miners while they are working.

These updates act as a heartbeat that the pool can listen for to ensure that a particular miner is still alive and performing work. These heartbeats are sent by miners at a set interval (in the case of LiteMiner, every second). Your pool should consider a miner to be dead if it has not received a heartbeat from them in 3 seconds.

In the stencil code for LiteMiner, we have provided a set of helper functions to facilitate the sending of heartbeats, however, it is your job to actually implement the logic detailed above.

2.5.3 Slow Miners

Some miners may have more or less processing power than others, and may therefore run slower or faster than them. You do not need to consider this to have a pool which works properly, but it is a problem a real pool would probably want to solve. For the competition / performance part of the assignment, finding a reasonable solution to this will probably be necessary to receive full credit.

3 The Assignment

You will be implementing a simple pool and its corresponding miners. The TAs have written a significant amount of support code for you. The code you must write is marked with `// TODO: Students` comments.

3.1 Code Overview

There are three main moving parts to this assignment, and each part has a very specific set of responsibilities. These three parts are as follows:

Client The client is responsible for receiving user-specified transactions and forwarding these transactions to the set of pool(s) it is connected to.

Pool The pool is responsible for receiving transactions from the client, dividing up the corresponding work and distributing it to its connected miners, and aggregating and returning the final proof of work.

Miner The miner is responsible for receiving work from the pool, performing that work, and sending the resulting proof of work back to the pool.

For your convenience, the TAs have provided a fully-functional client. As mentioned above, you will be implementing the core pool and miner logic.

A more detailed overview of the project structure can be found below, with short descriptions of each file in the repository.

- `cmd/`
  - `liteminer-client/`
    - `liteminer-client.go`: Implements a CLI for the client so that users can issue mining requests. Running `go install` will create a `liteminer-client` executable in `$GOPATH/bin`.
  - `liteminer-miner/`
    - `liteminer-miner.go`: Implements a CLI for the miner so that users can spin up new miners from the command-line and view debugging statements. Running `go install` will create a `liteminer-miner` executable in `$GOPATH/bin`.
  - `liteminer-pool/`
    - `liteminer-miner.go`: Implements a CLI executable for the pool so that users can spin up new pools from the command-line, query the state of a pool, and view debugging statements. Running `go install` will create a `liteminer-pool` executable in `$GOPATH/bin`.
- `liteminer/`
  - `client.go`: Implements the client logic for LiteMiner.
  - `hash.go`: Implements the hash function you will be using when mining.
  - `interval.go`: Contains the `Interval` struct and a method to divide up an interval, which you will be implementing.
  - `listener.go`: Implements all network listener logic.
  - `logging.go`: Implements serveral loggers which you may find useful.
  - `miner.go`: Contains the core miner logic, most of which you will be implementing.
  - `network_manager.go`: Contains functions for creating connections and sending and receiving messages over the network.
  - `pool.go`: Contains the core pool logic, most of which you will be implementing.
  - `proto.go`: Contains the LiteMiner protocol, some of which you will be implementing.
  - `basic_test.go`: Contains an example test.

3.2 Networking and Types of Messages

In this assignment, we define a very simple type of network connection called a `MiningConn` in `liteminer/network_manager.go`. A `MiningConn` consists of a Go network connection, an encoder (for sending messages through the connection), and a decoder (for reading messages from the connection). This `MiningConn` is how clients, miners, and pools communicate.

To facilitate consistent communication, we have also defined a `Message` struct and a set of different message types (see `MsgType`) in `proto.go`. See below for more detail.

- **Client → Pool**
  - `ClientHello: { Type }`
  - `Transaction: { Type, Data, Upper }`

- **Pool → Client**
  - `ProofOfWork: { Type, Data, Nonce, Hash }`
  - `BusyPool: { Type }`
- **Pool → Miner**
  - `MineRequest: { Type, Data, Lower, Upper }`
- **Miner → Pool**
  - `MinerHello: { Type }`
  - `ProofOfWork: { Type, Data, Nonce, Hash }`
  - `StatusUpdate: { Type, NumProcessed }`

3.3 Functions

Below is a comprehensive list of all the functions you are responsible for implementing (either partly or fully):

- `liteminer/miner.go`
  - `func (m *Miner) sendHeartBeats(conn MiningConn)`
  - `func (m *Miner) Mine(data string, lower, upper uint64) (nonce uint64)`
- `liteminer/pool.go`
  - `func (p *Pool) handleClientConnection(conn MiningConn)`
  - `func (p *Pool) handleMinerConnection(conn MiningConn)`
- `liteminer/interval.go`
  - `func GenerateIntervals(upperBound uint64, numIntervals int) (intervals []Interval)`
- `liteminer/proto.go`
  - `func StatusUpdateMsg(numProcessed uint64) *Message`
  - `func MineRequestMsg(data string, lower uint64, upper uint64) *Message`
  - `func TransactionMsg(data string, upper uint64) *Message`
  - `func BusyPoolMsg() *Message`

3.4 Performance Requirement and Optional Competition

Performance Requirement: A small portion of your grade (5%) will be reserved for how well your pool distributes work. This grade will just require that you beat the (very poor) TA solution, which takes around 3 minutes and 20 seconds to complete the task. For full credit, you should aim to be faster than 2 minutes and 30 seconds.

To test your code without and understand it's performance characteristics without competing in the competition, you should use this command: *compete-local*. Below we have information, on how to use these comments.

Optional Competition: There is also a competition which you can participate in (for non-profit): you will be able to see competition results at http://cs.brown.edu/courses/csci1380/s19/board.html.

To participate in the competition, on the department machines, you should run a command we have created: liteminer-competition. In order to figure out how well your pool performs, you must first launch an instance of your pool (liteminer-pool) in a terminal window. Then in a separate terminal window, run liteminer-competition.

liteminer-competition launches a shell with two commands: *compete* and *compete-local*.

- ***compete command*** takes the address of your pool (on a department machine this will look like <machine name>:port, and you can find the port in the shell for your pool).

  compete also takes a list of 5 pairs of words to mine and the expected result for each, and your "nickname". While the server-name and port are seperated by a ":", the word and appropriate result are seperate by a space.

  compete `<Server-address>:<Port> <Word_1> <Result_1> <Word_2> <Result_2> <Word_3> <Result_3> <Word_4> <Result_4> <Word_5> <Result_5> <Nickname>`

  So an example call might look like

  compete pokemon:38023 josh 3586653 sam 3948011 will 3848253 jim 3420565 tom 782614 TA-Code.

  Note that the "expected results" need to be the correct result of mining the given data string up to 5,000,000. This is necessary so that *compete* doesn't need to spend extra time figuring out what the correct responses are in order to check the results your pool produces. For testing purposes, the values used in the example are correct.

  compete will create a client connected to your pool, as well as several miners, and then mine each of your given words in sequence and make sure they return the correct value. As it does this it will print out how long it has taken. Once *compete* finishes, it will publish your time to a website where you can track and compare scores: your score will be recorded under nickname.

  Note: Please keep nicknames appropriate: we can tell who made each submission!

  Note: You can run *compete* as many times as you want, but there is an api key that has 10,000 uses on it, so if you are planning to compete a lot of times for testing (more than 10), please use local-compete for that purpose and just use compete once you are done.

- ***local-compete*** is identical to *compete*, except that it does not take a username and does not push anything to the site. Thus, it allows you to test the performance of your codebase without publishing results to the website or participating in the competition.

  Given that, *local-compete* takes a list of 5 pairs of words to mine and the expected result for each. While the server-name and port are seperated by a ":", the word and appropriate result are seperate by a space.

  compete-local `<Server-address>:<Port> <Word_1> <Result_1> <Word_2> <Result_2> <Word_3> <Result_3> <Word_4> <Result_4> <Word_5> <Result_5>`

So an example call might look like

local-compete pokemon:38023 josh 3586653 sam 3948011 will 3848253 jim 3420565 tom 782614.

Note: We will test your code against both "good" miners and "bad" miners. Where bad miners will help you explore edge-cases. For example, a "bad" miner may mine extremely slowly – compared to a "good" miner. Also, a "bad" miner, may not disconnect or may wait an arbitrarily long amount of time before disconnecting. Your pool should be able to handle these "bad" miners and still perform correct (and faster than the TA implementation). Our pool takes about 3.5 minutes to mine all the nonces. This is a fairly poor result, since our pool is not well optimized to cope with the slow miners. We think it is possible to get this time down to only a few seconds, but again you do not need to do this well to get full credit (which you will receive if you are faster than 2.5 minutes).

3.5 Reporting Problems

The competition site and the competition code (i.e., *compete*) are newly added to the course and may not work (or work incorrectly) due to a number of reasons including local environment setup and the correctness of your code. Below are a few helpful hints:

- **Issues with "local-compete":** If compete-local doesn't work, it probably means you have something wrong with your pool implementation.
- **Issues with "compete":** If compete doesn't work and local-compete does work, it probably means the HTAs messed something up. If this happens, do not worry about it. After everyone hands in we will be using compete-local to get peoples times for grading purposes, and after doing so we will also update the site to include everyones final times.

3.6 FAQ

We will have an FAQ up on Piazza – Please consult the FAQ to see if problems have been identified.

3.7 Project Difficulty

The difficulty of this project lies in understanding how LiteMiner works and writing correct code that is both thread-safe code and that accounts for faulty miners. Essentially, the difficulty is not in the quantity of the code but rather the quality of the code.

4 Demo

TA implementations of the LiteMiner pool, miner, and client are available at

```
/course/cs1380/pub/liteminer/{darwin,linux,windows}/liteminer-pool
/course/cs1380/pub/liteminer/{darwin,linux,windows}/liteminer-miner
/course/cs1380/pub/liteminer/{darwin,linux,windows}/liteminer-client
```

Below is a sequence of commands to startup a pool, connect two miners, and connect a client to the pool (each command should be run in a different terminal).

```
/course/cs1380/pub/liteminer/linux/liteminer-pool -p 1234 -d on
/course/cs1380/pub/liteminer/linux/liteminer-miner -c localhost:1234 -d on
/course/cs1380/pub/liteminer/linux/liteminer-miner -c localhost:1234 -d on
/course/cs1380/pub/liteminer/linux/liteminer-client -c localhost:1234 -d on
```

Once your implementations of miner and pool are sufficiently functional, you should test them with the TA implementation for interoperability.

5 Testing

We expect to see good test cases. This is going to be worth a portion of your grade. You can use the provided CLI programs to test you project as you are developing it, but you are required to submit more exhaustive tests with your handin. You can check your test coverage by using Go's coverage tool[1].

- `cmd/liteminer-miner/liteminer-pool.go`

  This is a Go program that serves as a console for interacting with the pool, and to check its state. We have kept the CLI simple, but you are welcome to improve it as you see fit.

  You can pass the following arguments to `liteminer-pool`:

  - `-p(ort) <port>`: The port to listen on
  - `-d(ebug) <on|off>`: Toggle debug statements on or off

  You get the following set of commands available to you in the terminal:

  - `miners`
    - Prints any connected miner(s)
  - `client`
    - Prints the currently connected client, if one exists
  - `debug <on|off>`
    - Toggle debug statements on or off

- `cmd/liteminer-miner/liteminer-miner.go`

  This is a Go program that serves as a console for interacting with the miner. We have kept the CLI simple, but you are welcome to improve it as you see fit.

  You can pass the following arguments to `liteminer-miner`:

  - `-c(onnect) <pool address>`: Mining pool address to connect to
  - `-d(ebug) <on|off>`: Toggle debug statements on or off

  You get the following set of commands available to you in the terminal:

[1] `http://blog.golang.org/cover`

  - `shutdown`
    - Shuts down the miner
  - `debug <on|off>`
    - Toggle debug statements on or off

- `cmd/liteminer-client/liteminer-client.go`

  This is a Go program that serves as a console for interacting with the client. We have kept the CLI simple, but you are welcome to improve it as you see fit.

  You can pass the following arguments to `liteminer-client`:

  - `-c(onnect) <pool addresses>`: List of mining pool addresses to connect to (comma-seperated)
  - `-d(ebug) <on|off>`: Toggle debug statements on or off

  You get the following set of commands available to you in the terminal:

  - `connect <pool addresses>`
    - Connect to the specified pool(s)
  - `mine <data> <upper bound on nonce>`
    - Send a mine request to any connected pool(s)
  - `pools`
    - Print the pools that the client is currently connected to
  - `debug <on|off>`
    - Toggle debug statements on or off

- Go provides a tool for detecting race conditions[2], which can be helpful for detecting and debugging concurrency issues. To run all Go tests in your current directory with the race detector on, run:

  ```
  go1.11 test -race
  ```

  Note that you can also use the race detector while building or running your Go programs.

6 Style

CS 1380 does not have an official style guide, but you should reference "Effective Go" for best practices and style for using Go's various language constructs.

Note that naming conventions in Go can be especially important, as using an upper or lower case letter for a method name affects the method's visibility outside of its package.

At a minimum, you should use Go's formatting tool `gofmt` to format your code before handing in.

You can format your code by running:

```
gofmt -w=true */*.go
```

This will overwrite your code with a formatted version of it for all go files in the current directory.

[2] `http://blog.golang.org/race-detector`

7 Getting Started

Before you get started, please make sure you have read over, understand, and have set up all the common code.

Note: if you're working on a department machine, use go1.11 instead of go for all the following commands (e.g., go1.11 get or go1.11 test).

To get started, run the following command:

```
go get github.com/brown-csci1380/<team-name>
```

where `<team-name>` is the name of the team repository created for you and your partner by the course staff. This command will download the stencil code for LiteMiner. (If you get the error "terminal prompts disabled", you need to set the git prompt environment variable by running `export GIT_TERMINAL_PROMPT=1`.)

Make sure to go through the files in the `cmd/` folder, and change the imports statements

`github.com/brown-csci1380/stencil-s19/liteminer/liteminer` to

`github.com/brown-csci1380/<team-name>/liteminer/liteminer`.

Next, navigate to the project directory `github.com/brown-csci1380/<team-name>/liteminer` where the stencil code was pulled to (This is determined by the `$GOPATH` environment variable. For more information on setting up Go, check out our "Get Going with Go" guide!) and run the following command:

```
go get -u ./...
```

You can ignore the error `no Go files in ...`.

This will pull in all of the imports from the current package downwards.

Lastly, you can build the `liteminer-client`, `liteminer-miner`, and `liteminer-pool` binaries from the `cmd` directory with:

```
go install ./...
```

Note that if you've made any changes to your code, you must re-install for your binaries to be up-to-date.

8 Handing In

The directory structure of your assignment should include the following:

- README
  - Please write up a simple README documenting any bugs you know of in your code, any extra features you added, an overview of your tests and anything else you think the TAs should know about your project.

Run the electronic handin script

```
/course/cs1380/bin/cs1380_handin liteminer
```

to deliver us a copy of your code.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS1380 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs138/s19/feedback.html`.