Project 1: LiteMiner

Due: 11:59 PM, Feb 17, 2022

Contents

1 Introduction

Welcome to CSCI 1380! This project is a self-contained introduction to many of the concepts that you'll be using frequently. 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 Lab 0 so you can get credit for your work!

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 protocol called the *blockchain* which we will study later. Key to cryptocurrencies are peer-to-peer networking principles, proof of work, and cryptography methods that maintain security and authenticity. In this assignment, we will focus on peer-to-peer networking and proof of work.

Peer-to-peer systems coordinate a dynamic set of 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 like Bitcoin, *blocks* – which contain a collection of transactions – are constantly created and validated by network participants that perform a computationally intensive task. This is often referred to as *proof of work*. A common proof of work scheme involves finding a 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, 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 expensive and requires a lot of resources; therefore, in order to "cheat" the system, one needs to have more resources than all other participants combined.

For more information on how proof of work fits into the bigger picture in a system like Bitcoin, check out this video and the Bitcoin whitepaper.

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 achieve 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 available 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 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 pool 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`.
- `pkg/`
  - `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), a decoder (for reading messages from the connection), and a paused boolean variable (only for miners to be paused and disconnected with its pool). 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):

- `pkg/miner.go`
  - `func (m *Miner) sendHeartBeats(conn MiningConn)`
  - `func (m *Miner) Mine(data string, lower, upper uint64) (nonce uint64)`
- `pkg/pool.go`

  - `func (p *Pool) handleClientConnection(conn MiningConn)`
  - `func (p *Pool) handleMinerConnection(conn MiningConn)`
- `pkg/interval.go`
  - `func GenerateIntervals(upperBound uint64, numIntervals int) (intervals []Interval)`
- `pkg/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

We will test your code against both "good" miners and "bad" miners, where "bad" miners are intended to explore various edge cases. For example, a "bad" miner may mine extremely slowly compared to a "good" miner. A "bad" miner may also 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 correctly. The naive TA pool implementation takes about 27s to mine all the nonces. Since the naive TA implementation pool is not well optimized to cope with the slow miners, you should be able to achieve significantly better results than this. We think it is possible to get this time down to only a few seconds, but this is not required to get full credit for performance (see below).

Performance Requirement: A small portion of your grade (5%) will be reserved for how well your pool distributes work. For full credit, you should aim to be faster than **35 seconds**.

3.5 Liteminer Competition

To make this project more fun, we have created additional, "trickier" performance test cases! You can test your implementation on these tests by submitting to the Liteminer Competition assignment on Gradescope. There are three different levels of the competition, and each level will only be unlocked when you pass the previous one (ie. the testing suite will not run the next level if you do not pass the previous one). Each team's final score will be the overall runtime of all three levels – the lower the better!

Note: You **don't** need to complete the competition to get full credit, but as discussed, the winner of this competition will get donuts and a mysterious NFT!

3.6 FAQ

Should incoming transactions be queued or ignored when the pool is busy?
These transactions should be ignored.

Does a pool ever disconnect the client, or does the client do that itself?
The client disconnects itself upon receiving a BusyPool message.

Should the goroutine 'handleMinerConnection' be killed immediately if the miner is shutdown?
If there is a way for the pool to detect that the miner has been shut down other than the heartbeat mechanism, you can return from handleMinerConnection immediately.

What happens if all of the miners die?
Clients do not specify timeouts so a mining transaction should run until the end unless the transaction itself is lost.

What is the difference between p.busy and p.Client?
The pool can have a client but not be busy if it's not currently mining. In this case, the pool can still accept a new Transaction from the current client.

What should the expected behavior be if there are no miners or all miners disconnect?
If there are no miners, then the pool should continuously send error messages to the client while waiting for miners. The client will remember these error messages but continue to hold indefinitely.
If all miners disconnect, the pool should wait for new miners to come back.

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 Handing In

About Checkpoint: There is a checkpoint available on Gradescope. The number of submissions is unlimited, and you can submit any time before the final handin deadline (including late submissions). Make sure to submit in groups with "add group member" in submission page. Your code will be run against a partial version of the final grading test suite. The autograder will return a log back with your results, and you are able to make further improvements accordingly.

The directory structure of your assignment should include the following:

- Filled out stencil code from `pkg`
- Your own tests (see the next section!)
- 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, the distribution of labor in your team, and anything else you think the TAs should know about your project.

Commit and push all your changes to GitHub.

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. For full credit, we are asking for ≥ 80 % code coverage of the entire `pkg` package in your tests. Test coverage should include testing the stencil code provided by TAs.

To help you get a feel for the interactions between running clients, pools, and miners in order to think of various test cases, we have provided the following code:

- `cmd/liteminer-pool/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:

  - `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 address>`: Mining pool address 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 address>`

    - Connect to the specified pool

  - `mine <data> <upper bound on nonce>`

    - Send a mine request to connected pool

  - `pool`

    - Print the pool that the client is currently connected to

  - `debug <on|off>`

    - Toggle debug statements on or off

Go provides a tool for detecting race conditions, 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:

```
go1.17 test -race
```

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

Please put all of your tests into the `\test` directory. Your test cases will not be graded if you put them anywhere else.

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 relevant go files in this project. You will be graded for this!

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 go1.17 instead of go for all the following commands (e.g., go1.17 get or go1.17 test).

We are using GitHub Classroom for this project. Check out the Ed post for the invitation link.

GitHub Classroom will automatically create a repository for you which contains the stencil code. Each team only needs one repository, so only one group member should create the team, and subsequently invite the other group member. The team name should be `cslogin1-cslogin2`. You should not invite any other users to your team or share your repository outside your team. It might take GitHub Classroom up to 5 minutes to set up the repo, so be patient, but let us know if GitHub reports an error! For a detailed walkthrough of using teams in GitHub Classroom, please see the the Github Classroom Setup Guide.

Use `git clone github.com/brown-csci1380-s20/liteminer-<team-name>/...` to clone your repo.

Next, navigate to the project directory and run

```
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 ./...
```

then you can run these compiled binaries by `$GOPATH/bin/liteminer_miner` and so on.

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

8 Capstone & Extra Feature

There are no additional capstone features required for this project. Happy mining! :)

Feedback

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out the anonymous feedback form.