Assignment 2:

Bully Algorithm and Bitcoin Mining

Deadline 1 (Part A): Thursday, 10th October 2019 at 11:50pm
Deadline 2 (Part B): Wednesday, 23rd October 2019 at 11:50pm
Deadline 3 (Part B): Wednesday, 30th October 2019 at 11:50pm

You will be implementing the following in this assignment:

- **Part A:** Bully algorithm for leader election.
- **Part B:** A distributed system for Bitcoin mining.

There are three deadlines for this assignment and the total time is about 4 weeks. Please make sure you first carefully read the entire handout and understand the given problems. We have made this handout elaborate, and we expect most of your questions will be answered if you read the handout carefully.

Note: For deadline 1, you are required to submit only your implementation of the bully algorithm in part A of the assignment.

Note: For deadline 2 and 3, you are required to submit your implementation of the client, miner and server in part B of the assignment.

Note: The Evaluation section explains the deadlines.

Note: Process and node will be used interchangeably in this handout.

Note: Course policy about **plagiarism** is as follows:

- Students must not share actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection.
- Students cannot copy code from the Internet.
- Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Part A

Bully Algorithm Overview

Distributed algorithms are very different from those we encounter in a standard algorithms course. In a distributed setting the **same** piece of code is running on all physically **distinct** machines simultaneously, yet the task is accomplished collaboratively.

In this part, you will be coding a synchronous network algorithm:

- The Bully algorithm for leader election.

In a synchronous network model, the following are true:

- Processes are connected to each other by communication channels.
- The underlying communication subsystem is reliable. It offers the following:
  - An upper bound on the delivery time of a message.
  - In order message delivery.
- An algorithm executes in synchronous rounds. In each round:
  - A process reads arrived messages,
  - Computes a new state based on these messages, and then
  - Sends messages to other processes based on the new state.

In this model, we ignore the costs (time and space) of local computations. The focus is instead on the number of rounds of communication (time complexity), and number and size of messages (communication complexity).

Let's Get Started

You are required to implement the basic version taught in the class and also present in the book. The following assumptions relate to the bully algorithm:

System level

- The system is synchronous.
- Processes may fail at any time, including during execution of the algorithm.
- A process fails by stopping and returns from failure by restarting.
- Messages are delivered eventually and reliably.
- Each process knows its own unique process id and address, and that of every other process.

Process level

- Any process can call for an election.

- A process can call for at most one election at a time.
- Multiple processes are allowed to call for an election simultaneously.
- The result of the election should not depend on which process called for an election.

Starter code

bully/**bully.go** contains the started code. You have to complete the function `Bully`. Each process / node will be running this exact function. That is, to have n nodes, `go Bully(…)` will be executed n times.

The tests are contained in bullytests/**btests.go**. Take note of the following in **bully.go**:

- Messages: messages passed between nodes will be of type `Message` (line 13). A message should contain `Pid` of the sender process, the `Round` number and the `Type`. You should define this `Type` (line 6) according to your needs.
- Process ID: `pid int` (line 20) is the process ID of the node and `leader int` (line 20) is the process ID of the current leader node.
- Communication: `comm map[int]chan Message` (line 20) contains the channels through which a process will communicate with other processes. E.g. a process will receive messages on `comm[pid]` and can send message to a leader on `comm[leader]`.
- Election: when a process receives anything on `checkLeader chan bool` (line 20), i.e. `len(checkLeader) > 0`, only then it must check if the leader is responsive. Whenever elections conclude, each node (including the leader) must send the Process ID of the new leader on `electionResult chan<- int` (line 20). A forced takeover of a perfectly responsive leader by a previously inactive bully node (e.g. node with the highest process ID) will also be counted as an election.
- Rounds: in every round, the for-loop (lines 24-35) only executes once. `startRound <-chan int` (line 20, 32) gives the round number.

Guidelines

While implementing your algorithm, you must abide by the following guidelines:

- The setting is a clique, that is, any process can send a message to any other process.
- In every round, a process should receive at most one message from any of other process. That is, a process cannot send more than one messages to any other process in the same round.
- In every round, a process should first read all the arrived messages before sending any message to any other process.
    - Use the helper function `getMessages(…)` (line 41) to read only those messages that have arrived from the previous round (i.e. pass `roundNum - 1` as argument to `getMessages`). This is essential to ensure correctness and not run into bugs difficult to reproduce.
    - Whenever you send a `Message` object, `Round` field must be set to `roundNum`.

- A simplistic fail-stop model of failures is assumed: a node can fail at the start of any round. When it comes back up again (at the start of a round), it will function correctly according to the state you've saved. You must initialize state-related variables in your code (line 22).
- If a message (which needs a reply) was sent to a process A in round t and a reply from A has not been received when reading messages in round $t + 2$, A can be assumed to have failed. That is, the communication subsystem does not fail: ensuring reliable, in-order delivery with bounded delivery time (if receiving node is active).
- Write your code keeping in mind that there may be more than one election in a single run of a test case.

Folder placement

The folders bully/ and bullytests/ must be placed directly within your **`$GOPATH/src`** directory.

Running the test cases for Bully Algorithm (Deadline 1):

To run all the tests, use the following command:

```
$ go run [-race] btests.go
```

To run only the basic or advanced tests, use the following commands:

```
$ go run [-race] btests.go basic
$ go run [-race] btests.go advanced
```

To run a specific test (e.g. **`basic1`** and **`advanced2`**) use the following commands:

```
$ go run [-race] btests.go basic1
$ go run [-race] btests.go advanced2
```

Advice

- After every round, update the state-related variables (initialized in line 22 and below). A node can potentially use these variables to figure out if it failed at the start of a previous round and has come back up again.
- Convert the algorithm into a protocol (i.e. a state machine representation) for a robust implementation which is easier to code and debug.
- You may experiment with increasing the value of `const waitBetweenRounds` (line 11 in **btests.go**) while developing in order to get the code working correctly before optimizing for efficiency.

Requirements

As you write your code for this project, also keep in mind the following requirements:

1. The assignment **MUST** be done individually. You are not allowed to use any code that you have not written yourself. As with all assignment in this course, we will be using a software to detect plagiarism.
2. You may use any of the synchronization primitives in Go's `sync` package **only for this part of the assignment**.
3. You **MUST** format your code using `gofmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.

Submission

You are only required to submit **bully.go** (and any documentation drafted in separate files). Add your roll number to the filename before submitting it on LMS using the format **<filename>_<R>.go**, where **<R>** should be replaced by your roll number, e.g. **bully_20100010.go**.

Part B

Bitcoin Overview

Bitcoin is a decentralized digital currency. At the heart of the Bitcoin protocol is a replay prevention mechanism that prevents participants from double spending Bitcoins. The replay prevention mechanism uses a cryptographic proof-of-work function that is designed to be computationally hard to execute. Clients compete to be the first to find a solution to the proof-of-work in order to get their signature attached to a sequence of transactions. If a client wins, they are rewarded with Bitcoins. The process of finding a solution to the proof-of-work is called *mining*.

In this project, we will spare you the details of the real Bitcoin protocol. You will instead implement a mining infrastructure based upon a simplified variant: given a message M and an unsigned integer N, find the unsigned integer n which, when concatenated with M, generates the smallest hash value, for all $0 \leq n \leq N$. Throughout this document, we will refer to each of these unsigned integers as a *nonce*.

As an example, consider a request with string message `"msg"` and maximum nonce `2`. We can determine the desired result by calculating the hash value for each possible concatenation. In this case, nonce `1` generated the least hash value, and thus the final result consists of least hash value `4754799531757243342` and nonce `1`. Note that you need not worry about the details of how these hash values are computed – we have provided a `Hash` function in the starter code for you to use to calculate these hashes instead.

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

One simple approach to completing this task is to perform a brute-force search, in which we enumerate all possible scenarios across multiple machines. For tasks that require searching a large range of nonces, a distributed approach is certainly preferable over executing the compute-intensive task on a single machine. As an example, our measurements show that a typical Linux machine can compute SHA-256 hashes at a rate of around 10,000 per second. Running sequentially, a brute force approach would require around 28 hours to try all possible hashes consisting of 9 decimal digits. But, if we could harness the power of 100 machines, then we could reduce this time to around 17 minutes.

Your task for this project is to implement a simple distributed system to perform this task. We discuss our proposed system design in the next section.

System Architecture

Your distributed system will consist of the following three components:

Client: sends a user-specified request to the server, receives and prints the result, and then exits.

Miner: continually accepts requests from the server, exhaustively computes all hashes over a specified range of nonces, and then sends the server the final result.

Server: manages the entire Bitcoin cracking enterprise. At any time, the server can have any number of workers available, and can receive any number of requests from any number of clients. For each client request, it splits the request into multiple smaller jobs and distributes these jobs to its available miners. The server waits for each worker to respond before generating and sending the final result back to the client.

Type	Fields	From-To	Use
`Join`		M-S	miner's join request
`Request`	`Data Lower Upper`	C-S, S-M	send job request
`Result`	`Hash Nonce`	M-S, S-C	report job's final result

Predefined message types in the Bitcoin distributed system. In the "From-To" column,'M' denotes a miner, 'S' denotes the server, and 'C' denotes a request client.

Each of the three components communicate with one another using a set of predefined messages. Table above shows the types of messages that can be sent between the different system components. Each type of message is declared in the **message.go** file as a Go struct. Note that each message must first be marshalled into a sequence of bytes using Go's json package before being sent.

Sequence of Events

The overall operation of the system should proceed as follows:

- The server is started using the following command, specifying the port to listen on:
  `./server port`
- One or more miners are started using the following command, specifying the server's address and port number separated by a colon:
  `./miner host:port`
- When a miner starts, it sends a join request message to the server, letting the server know that it is ready to receive and execute new job requests. New miners may start up and send join requests to the server at any time.
- The user makes a request to the server by executing the following command, specifying the server's address and port number, the message to hash, and the maximum nonce to check:
  `./client host:port message maxNonce`

  The client program should generate and send a request message to the server, specifying lower nonce "`0`" and maximum nonce "`maxNonce`":
  `[Request message 0 maxNonce]`
- The server breaks this request into more manageable-sized jobs and farms them out to its available miners (it is up to you to choose a suitable maximum job size). Once the miner exhausts all possible nonces, it determines the least hash value and its corresponding nonce and sends back the final result:
  `[Result minHash nonce]`
- The server collects all final results from the workers, determines the least hash value and its corresponding nonce, and sends back the final result to the request client.

The request client should print the result of each request to standard output as follows (note that you **MUST** match this format precisely in order for our automated tests to work):

- If the server responds with a final result, it should print,
  `Result minHash nonce`

  where minHash and nonce are the values of the lowest hash and its corresponding nonce, respectively.
- If the request client loses its connection with the server, it should simply print
  `Disconnected`

Handling Failures

We will assume that the server operates all the time, but it is quite possible that a request client or some of the miners can drop out. You should take the following actions when different system components fail:

- When a miner loses contact with the server it should shut itself down.
- When the server loses contact with a miner, it should reassign any job that the worker was handling to a different worker. If there are no available miners left, the server should wait for a new miner to join before reassigning the old miner's job.
- When the server loses contact with a request client, it should cease working on any requests being done on behalf of the client (you need not forcibly terminate a job on a miner–just wait for it to complete and ignore its results).
- When a request client loses contact with the server, it should print Disconnected to standard output and exit.

Your server will need to implement a *scheduler* to efficiently assign workers to incoming client requests. You should design a scheduler that balances load across all requests, so that the number of workers assigned to each outstanding request is roughly equal. Your code should contain documentation on how your scheduler achieves this requirement.

Bitcoin Miner Starter code

The starter code for this project can be found in the bitcoin/ directory, **which needs to be placed** in your `$GOPATH/src` directory, is organized as follows:

- The **message.go** file defines the message types you will need to implement your system.
- The **hash.go** file defines a Hash function that your miners should use to compute uint64 hash values.
- The client/**client.go** file is where you will implement your request client program.
- The miner/**miner.go** file is where you will implement your miner program.
- The server/**server.go** file is where you will implement your server program.

We have provided some starter code in the three files that you need to implement as mentioned above, and feel free to make any changes to them. However, do **NOT** modify **message.go** and **hash.go**.

Using hash.go and message.go

In order to use the starter code we provide in the **hash.go** and **message.go** files (under `$GOPATH/src/bitcoin`), use the following import statement:

```
import "bitcoin"
```

Once you do this, you should be able to make use of the bitcoin package as follows:

```
hash := bitcoin.Hash("thom yorke", 19970521)
msg := bitcoin.NewRequest("jonny greenwood", 200, 71010)
```

You can also create **common.go** under `$GOPATH/src/bitcoin` to include code you would like to use in the different implementation files.

Testing on your machines

After compiling your programs using `go install`, as given right above, use the following commands to test your code yourself:

```
# Start the server, specifying the port to listen on
$GOPATH/bin/server 6060

# Start a miner, specifying the server's host:port.
$GOPATH/bin/miner localhost:6060

# Start the client, specifying the server's host:port, the message
# "bradfitz", and max nonce 9999.
$GOPATH/bin/client localhost:6060 bradfitz 9999
```

Note that you will need to use the `os.Args` variable in your code to access the user-specified command line arguments.

Running the official tests for Bitcoin Miner (Deadline 2 & 3):

After placing **ctest**, **mtest** and **stest** binaries from the **bitcoin-tests** folder in to `$GOPATH/bin`, to test your submission we will execute the following commands:

```
$ go install bitcoin/client
$ go install bitcoin/miner
$ go install bitcoin/server
$ ctest
$ mtest
$ stest
```

Note that **ctest** only tests the code in client/**client.go**. Similarly, **mtest** only tests the code in miner/**miner.go**. On the other hand, **stest** will test the code from all 3 files, i.e. client/**client.go**, miner/**miner.go** and server/**server.go**.

Debugging will be a bit **different** in the official test cases, please refer to the **Logistics** portion of the handout to learn more about it.

Requirements

As you write your code for this project, also keep in mind the following requirements:

1. The assignment **MUST** be done individually. You are not allowed to use any code that you have not written yourself. As with all assignment in this course, we will be using a software to detect plagiarism.

2. Your code **MUST NOT** use locks and mutexes for the rest of the assignment. All synchronization must be done using goroutines, channels, and Go's channel-based `select` statement. Furthermore, solutions using any locking, mutexes or implementing mutex-like behavior using Go channels will incur a grading penalty.

```
_ = <-my_mutex_channel
// some critical code
my_mutex_channel <- 1
```

   This code snippet is an example of using channels as mutexes, which isn't allowed.

3. Avoid using fixed-size buffers and arrays to store things that can grow arbitrarily in size. For example, do not use a buffered channel to store pending messages for a particular connection. Instead, use a linked list - such as the one provided by the container/list package - or some other data structure that can expand to an arbitrary size.

4. You **MUST** format your code using gofmt and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.

5. The submitted code should not have any extra print statements except the ones already there or those specifically asked for. Otherwise, the official test cases will not work and you will lose marks.

Evaluation

- You can earn up to 35 points from this assignment. There is no extra credit or bonus.

  Deadline 1 (total 10 points)

  1. **btest** - 10 points
     Test cases: basic1-5, advanced1-2

  Deadline 2 (total 10 points)

  1. **ctest** - 3 points
  2. **mtest** - 3 points
  3. **stest** - 4 points
     Test Cases: 1-4
     Test 1: 1 Miner and 1 Client, Small Tasks,

Test 2: 1 Miner and 1 Client, Huge Tasks,
Test 3: 1 Miner and 5 Clients, Small Tasks,
Test 4: 4 Miner and 4 Clients, Small Tasks

Deadline 3 (total 10 points)

1. **stest** - 10 points
   All remaining Test Cases

Deadline 1 & 2 & 3 (total 5 points)

1. **Go Formatting** - 1 point
2. **Manual Grading** - 4 points
   You can obtain full points from this if you satisfy these conditions:
   - There's a clear explanation of your implementation at the top of your submitted file. You may submit this as a separate document as well.
   - Your submission has good coding style that adheres to the formatting and naming conventions of Go.

- The test cases will be marked with the `-race` flag as well.
- The advanced test cases are worth more.
- You have a total pool of 5 days for late submissions, without deductions. Late days from this pool can be used for any of the four assignments. Once you have used up all the 5 days, no late submissions for the assignments will be accepted.

Logistics

Debugging while running official tests for Bitcoin Miner

When you run the tests, one of the first things you'll probably notice is that none of the logs you print in both the code you write for part A and part B will not appear. This is because our test binaries must capture the output of your programs in order to test that your request clients print the correct result message to standard output at the end of each test. An alternative to logging messages to standard output is to use a `log.Logger` and direct them to a file instead, as illustrated by the code below:

```
const (
    name = "log.txt"
    flag = os.O_RDWR | os.O_CREATE
    perm = os.FileMode(0666)
)

file, err := os.OpenFile(name, flag, perm)
if err != nil {
    return
```

```
}

LOGF := log.New(file, "", log.Lshortfile|log.Lmicroseconds)
LOGF.Println("Bees?!", "Beads.", "Gob's not on board.")
```

Don't forget to call `file.Close()` when you are done using it!

Submission

For **Deadline 2 & 3 (Part B)** zip together the three files (server/**server.go**, client/**client.go**, and miner/**miner.go** into **<R>.zip**, where **<R>** should be replaced by your roll number, e.g. **20100010.zip**. Upload the zip file on LMS.