CMPT 383: Assignment #4

Anders Miltner
miltner@cs.sfu.ca

Due Nov 12

Introduction

In this assignment, we will be using Rust to make a simple blockchain system. A blockchain system can be thought of as a linked list. Each node in this list, or *block*, points to a prior block, and includes information of the prior block in its hash. In this way, the whole chain inductively maintains cryptographic guarantees.

In this assignment, we will be building a "proof-of-work" blockchain. Public proof-of-work blockchains, like Bitcoin, have significant negative externalities in the form of energy consumption. As such, many more modern blockchains have used to "proof-of-stake" blockchains. However, these are harder to implement, so we will be building a proof of work blockchain.

This assignment will have three major components, building a blockchain, building a work queue, and using the work queue to efficiently mine new blocks on the chain. This assignment was taken with very little modification from Greg Baker's CMPT 383, so he gets full credit for all positive aspects of this assignment.

1 `Block`

In the file `block.rs`, we have provided the `Block` struct, as well as a few function skeletons. The `Block` struct consists of 5 components.

1. `prev_hash`: The `prev_hash` field describes the hash value of the previous node in the chain.
2. `prev_hash`: The `generation` field describes the index of the current node in the chain. The node with generation 0 has no previous node in the chain.
3. `difficulty`: The `difficulty` field describes the amount of work required to add the node to the chain.
4. `data`: The `data` field describes the actual data in an individual node. In our project the data field is an arbitrary string, through it could describe financial transactions or programs.
5. `proof`: The `proof` field demonstrates that work has been done to create a block. The `proof` field is computationally expensive to generate, but is computationally cheap to find. Given a block b with difficulty d, the proof of the block would be a number which, when hashed with the block, will generate a hash code ending in d trailing 0 bits.

A simple blockchain is shown in Figure 1.

1.1 `initial`

Given a difficulty `d`, the `initial` function will create a block with: a hash of all zeros, a generation of 0, a difficulty of `d`, a data of `""`, and a proof of `None`. To create a hash of all zeroes, one can call `Hash::default()`.

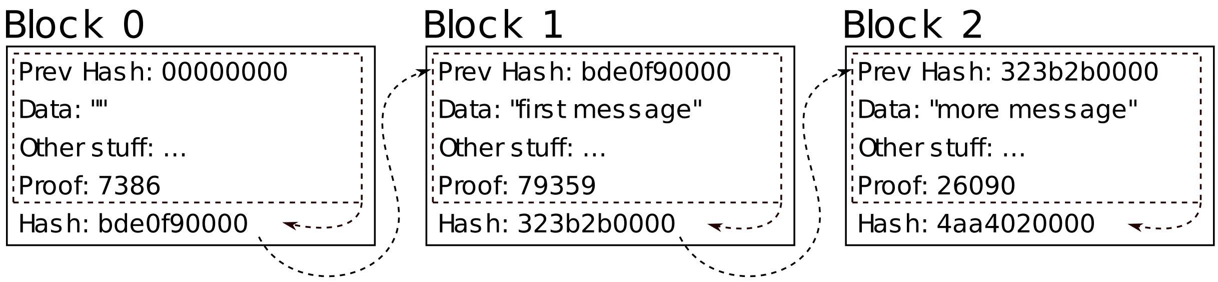

Figure 1: A simple chain of length 3. In this chain, the difficulty is 16 (so the last 4 hex characters of all hashes are 0).

1.2 `hash`

Next we must hash these nodes. In this section, we will be writing the functions `hash_string_for_proof` and `hash_for_proof`. Recall that when mining, one must find *proof* that enable the hash token to have trailing 0 bits. As such, we must build a hash function that takes arbitrary proofs as input. When calling the base hash function, it simply uses the `..._for_proof` functions, with the proof hard-coded on the node.

The hashing function we will use in this section is `Sha256`. The three methods you need are: (1) `Sha256::new()` which returns a *Digest*, (2) `d.update(s)` which updates a digest `d` with a string reference `s`, and (3) `d.finalize` which takes a digest `d` and returns a `Hash` value. Thus, we need to create a string representation of the node with the proof, then we will use that to create a hash for the node. This is the purpose of `hash_string_for_proof`.

The `hash_string_for_proof` function returns a string representation of a given node with a given proof. The `hash_string_for_proof` function should create a string formatted as follows:

```
previous_hash:generation:difficulty:data:proof
```

The previous hash should be expressed as a lowercase hexadecimal representation. To output a hash in a string representation, you can do so in the following way:

```
format!("{:02x}",hash)
```

You can build up the full string either by using the `push_str` function, or by using the `format!` macro.

With this string in hand, you should be able to extract the hash value using the built in `Sha256` functions. Create a new digest, update the digest with the generated string, and finalize the digest to get a Hash value.

1.3 `next`

Now that hash has been defined, we can finally build a chain of length longer than 1! The next function will take a reference to a block and a piece of data as input, and will create a new block of the next generation that refers to the old block.

The `next` function takes an input of a previous Block, and a String data. The created Block should have the `previous_hash` set to the hash of the previous Block, the generation should be one higher than the prior generation, the difficulty should remain the same, and there should be no proof (as that will be found later).

1.4 `hash_satisfies_difficulty`

Next, the next part is important for validation and for mining: `hash_satisfies_difficulty`. In this section, we check whether or not a provided a hash value has a sufficient number of trailing zero bits.

Hash values are represented as an array of 8 bit numbers. Thus, if the difficulty is 8, then the last element of the array must be 0u8. If the difficulty is 16, then the last two elements of the array must be 0u8. Generally, if the difficulty d is divisible by 8, then the last $d/8$ elements of the array must all be 0u8. What happens if the difficulty *is not* divisible by 8? Say the difficulty is 2. Then the last number in the hash should be divisible by 4 (as that would make it end in 2 zeros). If the difficulty is 4, then the last number should be divisible by $2^4 = 16$. If the difficulty is 10, then the last number should be 0u8, and the second to last number should be divisible by 4. (Indexing into the array may cause difficulties, you can cast a number `n` to ulong using the syntax `n as ulong`).

More generally, to check whether a proof satisfies a given difficulty:

1. Define `n_bytes` as the difficulty divided by 8
2. Define `n_bits` as the difficulty mod 8
3. Check that each of the last `n_bytes` are 0u8
4. Check that the byte one before the last `n_bytes` is divisible by `1<<n_bits` (as `1<<n_bits` $== 2^{\texttt{n\_bits}}$).

At this point, we actually have a full implementation of a blockchain! With the implementation of `hash_satisfies_difficulty`, the `mine_serial` function works! This function iterates through proofs, from 0 onwards, until one is found that creates a hash value with a sufficient number of trailing 0 bits. Once one is, the proof is set to that number.

Unfortunately, this serial mining is quite slow. We would like to speed it up by searching for it in a multi-threaded fashion. To do this, we will first implement a general work queue, then we will use that work queue to mine in a distributed fashion.

2 WorkQueue

The work queue permits enqueueing "Tasks" to send to worker threads. Fundamentally, it does this using `spmc` capabilities to distribute tasks, and `mpsc` capabilities to receive task outputs. The `WorkQueue` struct has 4 fields: `send_tasks`, `recv_tasks`, `recv_output`, and `workers`. The `send_tasks` field is a `spmc` Sender, and distributes tasks to the workers. The `recv_tasks` field is a `spmc` Receiver, and is used to drain the thread pool when the queue is being shut down. The `recv_output` is a `mpsc` Receiver, and is used to receive outputs from the workers. The `workers` field contains the JoinHandles of each of the threads doing the processing.

The WorkQueue is generic across Tasks. A task is a struct containing a output type `Output` and a `run` function that will create an output of type `Option<Output>`. If the `run` function gives a concrete `Some` output, that output should be propogated to the `mpsc` channel in the main thread. If the `run` function gives a `None` output, it should be ignored.

2.1 `new` and `run`

Creating a work queue requires doing 3 primary things: (1) create a `spmc` channel, (2) create a `mpsc` channel, and (3) create the worker threads. You can create the `spmc` and `mpsc` channels by using `spmc::channel()` and `spmc::channel()` respectively (you may need to provide the generic argument for those calls).

Then, `n_workers` threads should be created. These threads should run the `run` function, with a `spmc::Receiver` and a `mpsc::Sender`, cloned from relevant components of the generated channels.

The `run` function should be a loop (this can be introduced using `loop` syntax). Within the loop, the thread should:

1. Receive a task
2. If the task was not received successfuly: simply return – ending the thread; the analogous Sender was destroyed
3. Otherwise, run the task
4. If the task result is None, do nothing
5. If the task result is Some thing, send that result to the main thread

2.2 `enqueue`

Given a task, the `enqueue` function should send a provided task over the `spmc` channel. Unfortunately, this can be relatively involved due to the complexities of correctly borrowing the `send_tasks` field. You may be able to find some helpful built-in functions on the Option type. If the `send_tasks` field is `None`, you can simply call `panic!()` (or another function that would `panic!()` like unwrap).

2.3 `shutdown`

The `shutdown` function should: (1) destroy the `spmc::Sender` such that no more tasks are incoming, (2) drain remaining tasks from the queue using the `recv_tasks` field, (3) remove all the workers from the `workers` vector, and `.join()` them all (the vector `.drain()` method may help here).

3 Multi-threaded `mine`

Now that we have implemented a Work Queue, we can make an efficient miner. We will do this by first defining the `MiningTask` struct, and the `Task` implementation for that struct. Unfortunately, the struct and the usage of the struct are logically interleaved, so we kind of have to do them at the same time.

The `mine` function works is by splitting the space of possible proofs into 2345 individual chunks of integers. While there are actually 2^{64} possible proofs, we are likely to find a possible proof at difficulty d within the first $8 * 2^d$ candidate proofs, so we only search through that component.

You simply must implement the `mine_range` function. This function takes as input a Block reference, an integer describing how many worker threads to use, a u64 describing the lower bound on proofs to search through (when called from our mine function, this will always be zero), a u64 describing the upper bound on proofs to search through (when called from our mine function, this will always be $8 * 2^d$), and a u64 describing the number of chunks to split into (when called from our mine function, this will always be 2345).

First, a work queue should be created with `n_workers` number of workers. Next, an `Arc<Block>` should be created with a clone of the provided `self` Block reference. This `Arc<Block>` can then be cheaply cloned for each mining task.

Next, the range of integers from `start` to `end` should be broken up into `chunks` number of chunks. The `mine_range` function should then create a mining task for each chunk. A `MiningTask` then should be created for each chunk. The `run` function of each `MiningTask` should simply iterate through every number of the chunk, and check whether that number is a valid proof. If it is, return Some of that proof, if none are, return None.

In this problem, we have left the implementation of `mine_range`, the fields of `MiningTask` (besides the `Arc<Block> field`), and the run function of `MiningTask` unimplemented. You must implement these such that each chunk is added to, and processed, in the work queue.