Implementing a Blockchain

Li Zhen	Wu Wanquan	Gou Guilin	Cui Shaobo
violetcrestfall@hotmail.com	wuwanquan@126.com	fluorinedog@gmail.com	cuishaobo@hust.edu.cn

1 Introduction

For this project, we implemented a simple blockchain. Participants of our blockchain network use a gossip-style communication protocol we implemented to reach agreement on the status of the blockchain. For simplicity, we did not implement dynamic expansion of a blockchain network. However, our blockchain does allow peers to rejoin the network after a failure. Our implementation prints the blockchain in a way that is both easy to verify the validity of the chain and rather effortless to separate the actual content of the chain. We tested our implementation by putting articles onto a blockchain which uses our own implementation, and our tests were successful.

2 Block Structure

The structure of a block is shown in Fig. 1. A block in our blockchain is 120 bytes long. We chose to use printable hex-number strings to represent control information, so that the output of our program can be easily verified as a valid blockchain. A block starts with a 32-byte field of user data, followed by an 8-byte field of block height, 4 bytes of peer identifier used to help resolve forks, 20 bytes of timestamp, a 32-byte string showing the MD5 hash of its preceding block, and a 24-byte nonce, value inserted to meet the hash threshold, in that order. The reason we chose MD5 is that it is relatively easy to compute, and using it fits our purpose of focusing on the basic mechanisms of blockchains.

Figure 1 The structure of a block

The reason we structured the block in this format is that it supports easy verification of the blockchain validity. The blockchain can be verified by a script which hashes a line and compares the hash with the characters between the 64th and the 96th on the next line if each block is printed on a separate line. We put the contents of our block at the beginning so that reading the contents will be relatively convenient.

3 Protocol Design

Our communication protocol is rather simple. Two classes of messages are transmitted between peers, namely “push messages” and “pull requests”. For each round of communication, a participant of the network randomly chooses several peers (we picked 2 peers for our implementation), and send a push message along with the length of the chain and a few blocks with the maximum heights (our decision is to include 12). Each time a machine receives a push message, it tries to update the chain with the blocks included in the push message. However, this update may not necessarily succeed. In fact, the following cases may occur:

- The length of the local chain is less that the height of the first block in the message minus one. In this case, the peer doesn’t have enough information to complete the chain, and enters a special mode called “recovery mode”.
- A mismatch happens between blocks, but it isn’t between the first block in the message and the last one on the local chain. This situation happens when there is a recent fork. In this case, the peer will try to resolve this fork in the following manner: the longest chain is always prioritized, and the chain with the smaller peer identifier is preferred when the chains are equally long. After resolving this fork, the peer notifies the source of the original push message with a fresh push message of its own if the local chain is chosen.
- A mismatch happens between the first block in the message and the last on the local chain. A fork which happened a while ago might cause this to happen. When this happens, the machine doesn’t have enough information to resolve the fork, and possibly has different blocks than those agreed upon by the other network participants. It will shrink the local chain by 12 blocks, and enter recovery mode.

In recovery mode, the peer sends pull requests to two randomly selected peers to retrieve blocks not included in a regular push message in each round. The pull request includes a sequence number which identifies the pull request and the height of the first block requested, which is the length of the local chain. When a peer not in recovery mode receives a pull request, it sends the requested blocks in a push message. The peer in recovery mode will ignore pull requests and normal push messages, and attempt to update the chain with the reply it received within a timeout window. If the update failed again because of a mismatch, the chain will be shrunk by 12 blocks, and start a new round. If the machine received no response in the timeout window, it will start the round over, sending pull requests to newly selected peers. With the chain length information included in the push messages, the peer can determine if it has already caught up with the other chains, and exits recovery mode when

it has a chain not shorter than the supplier of push messages by 6 blocks, in which situation a normal push message can bring the node fully up to date.

Note that in our implementation, the protocol is slightly modified to better suit our needs. These modifications will be explained in Section 4 of this report.

4 Implementation Details

Now we will discuss some details of our implementation. We will talk about several decisions we made in implementing the computation agent, communication agent and the top-level program, and how these decisions affect the whole program.

4.1 Computation Agent

We used OpenSSL's implementation of MD5. In each round of computation, the computation module executes a loop. In each iteration, it generates a random nonce, computes the MD5 of the block filled with that nonce, and checks if the hash is less than the threshold. If the threshold is met, the computation breaks and returns that nonce; otherwise it will continue this loop.

To make computation more efficient, we decided to block the whole program when the computation agent is trying to produce a new block. This decision reduces the thread switch overhead when computing, but also requires accommodations in other parts of the program. To make sure that communication happens effectively, however, the computation agent cannot block the execution of the program indefinitely. Our computation agent instead will indicate a failure and hand back the control after 2^{17} iterations have been executed in a round if it did not find a valid nonce.

4.2 Communication Agent

Our communication agent is mostly an implementation of the protocol described in Section 3, but changes are made to adapt to the blocking nature of our computation agent. In particular, the communication agent cannot process the message it receives during computation, and must postpone the procedure until the computation ends. We decided that it would be easier to implement the communication by providing an interface which completes several rounds of communication when called, and let the top-level program coordinate between computation and communication. This decision also makes it easier to stop the computation from running in recovery mode, as the communication agent can block the whole program until it exits recovery mode.

4.3 Top-level Program

As both the computation agent and the communication block the execution, the top-level program just keep calling the computation agent and the communication agent. However, as the computation agent may generate new blocks, the top-level program will check if a new block is produced and connect that block to the chain before calling the communication agent each time a computation round ends.

Because this top-level program is rather simple, we decided to include the top-level into our frontend. When the program starts, it reads and parses the configuration file. The configuration is then used to initialize the communication agent. The frontend then starts feeding the input to the computation agent, and successively calls the other parts of the program. When the blockchain is long enough, the frontend terminates the execution of the program, and prints out the blockchain. Each block is printed in one line.

5 Results

We tested our program by running a blockchain network of 64 homogeneous peers distributed through 4 machines in which all peers try to put the same article onto the chain. The article was successfully filled into the blockchain, but the final blocks did not agree. The reason is that the program terminated when the article is on the chain without waiting for a consensus to be reached. In a real-life blockchain, however, this is not a problem, as the peers will keep running the program without termination, and our protocol will eventually update these final blocks. Peers run on one of the machines all crashed. We could not login to that machine through SSH, so it's an issue with either the network or the machine itself. All the remaining peers printed a chain, and each chain was verified to be valid. Our test input is rather small, with the average block produced by each peer being approximately 1.23. In our first test, the results show that around half of the peers contributed exactly one block, and 87.5% of the peers contributed 0-2 blocks. However, it took around 5 minutes to complete the round. In the second test, after changing the timeout window, we were able to bring that time down to approximately 3 minutes, but the distribution is less even. This shows that our protocol doesn't have a huge bias despite the advantage given to peers with a smaller identifier when resolving forks.

6 Conclusion

We designed a fault-tolerant protocol for achieving agreement and eventually consensus with probability 1, and built up a blockchain based on that protocol in this project. Generally speaking, our system works quite well, but there are still a lot to be improved. Our blockchain network cannot expand dynamically, and all peers must have knowledge of all the other network participants. The protocol can be modified so that each peer only has partial knowledge of the network, and can learn about different parts of the network through communication. When a new machine joins the network, it can tell the peers it knows about, and let those peers learn about this new node. However, due to time constraints, we did not work out the details of this modification, or implement it.

We also did not record the forks which happened when the network is running. We can print out fork information when resolving it, but as each peer may resolve different forks, collecting the information together will be a problem.

Appendix A Usage of the Implementation

Our implementation is available online[1]. Python 3 is required to run the program. . You also need to have OpenSSL library installed, as well as a C++ compiler and cmake to compile the computation module.

To compile the computation module,create a folder called "build" in the directory where the source code is located, and change your working directory to that folder. Run "cmake .." and "make .." to compile the computation module, and go back to the parent directory.

A configuration file should be created before starting the program. The configuration file has three parts: overall-config, local-config and peer-config. Two parameters, difficulty and contentFilePath should be provided in the overall-config part. The parameter difficulty controls the threshold of the hash of a block, and must be a floating-point number between 0.0and 1.0; contentFilePath is the path of the input file. There cannot be over 31 characters between two space characters in the input file. The local-config part consists of four parameters: ipAddr, port, id and RTO. The parameters ipAddr and port are the IP address and UDP port to use by the local machine; id is the peer identifier as described in the report, and must be at most 4 characters long; RTO is the timeout window measured by seconds. In the part peer-config, a list of all the peers in the blockchain network should be provided. Numbering the peers starting with zero, the parameter for each peer should be named "node" followed by the peer number, and the parameter should be formatted as "IP Address:Port Number". An example configuration file is included in the GitHub repository. To start the program, run main.py with the path of the configuration file as a parameter.

Appendix B Work Distribution

All members of the group discussed about and contributed to the design of the project. Li Zhen continued working on protocol design and implemented the communication protocol. Wu Wanquan provided the basic utility for blockchain update, and wrote the top-level program. Gou Guilin completed the computation module, and tested the program. Cui Shaobo kept track of the project, provided ideas when debugging parts of the program, and prepared this report.

[1] See https://github.com/EAirPeter/NetProj