Porting SpartanGold to Go

CS 168: Blockchain and Cryptocurrencies
Sihan He sihan.he@sjsu.edu
Xiang Liu xiang.liu@sjsu.edu
San Jose State University
May 15, 2022

1 Introduction

For this project, we implemented a blockchain based on SpartanGold (SG). Spartan gold is a simplified blockchain-based cryptocurrency for education and experimentation designed by Professor Thomas [1]. It simulates the basic functionality of a complete blockchain like Bitcoin. However, it is coded in javascript which is not able to offer good memory usage and raw computing performance. The goal of this project was to fully understand all the functionalities of SG and port it into Go.
Go is syntactically similar to C but has way less problems. It also has comparable performance to C++ making it one of the most popular languages for blockchain development.

1.1 Research Objective

In this project, our objective was to implement a simplified blockchain-based cryptocurrency similar to SG using Go language. By using Go language, we were able to make the blockchain software more lightweight and run faster. The miner of the blockchain, who processes the transactions, creates blocks and finds the proof of the blocks as required by the Proof of Work(PoW), needs the software to run faster as well. The PoW is the consensus system used in SG. The PoW consensus system requires miners to look for a hash value that begins with a certain number of leading zeros by tweaking and then hashing the block using cryptographic hash algorithm; this system indeed requires the miner to use a lot of computing power to find the proof. Since the Go language can provide better computing performance compared to javascript, it is beneficial to use Go language to implement the blockchain software.

Our second objective was to learn the Go language. The Go language was first introduced by google researchers Rob Pike, Robert Griesemer, and Ken Thompson back in 2009, which is quite new [1]. However, it quickly became one of the most popular programming languages. According to [2], the Go language is ranked number 13 among all the programming languages. The Go language has many advantages compared to other languages. One of the advantages is that it offers good performance and less memory usage because it is a compiled language which does not require a virtual machine like Java [1]. The other advantage is concurrent programming. The Go language is specifically good at concurrent programming thanks to Goroutine. In [3], the author describes that the goroutines can be treated as lightweight threads compared to the OS threads in Go language. They are fully managed by the Go programming runtime which programmers don't have to worry about how threads should be managed and simply call goroutine API to do concurrent processing[3].

1.2 Technical Approach

We chose Go over Rust because of its smaller learning curve. We learned the language of Go and became familiar with it. We identified possible challenges it might bring. Github was used as our main platform for version control and file sharing.
In this project, our design of blockchain was similar to SG. Same as the design of SG, Our blockchain project used the account-based model instead of the UTXO model that Bitcoin uses. There is no use of scripting language as well, our project was implemented for pure cryptocurrency transactions. The proof of work target was fixed as well to simplify the design. However, unlike SG which is restricted to performance of Javascript due to being a single threaded language, we took advantage of multithreaded programming in Go language to enable the concurrent processing on finding the proof of the block.
Same as SG, our project provided two modes: the single threaded mode and the multi-process mode. The single threaded mode is a single process that will simulate an environment with some predefined miners and non-miner clients joining the blockchain

and do some pre-defined behaviors. This mode runs some basic transactions and also simulates certain challenging situations that a distributed system needs to handle correctly in order to have the system work. The situations include network delay, peers joining the network and peers leaving the network. In the multi-process mode, however, each minor or non-minor client will be an independent process and run independently. Users are able to create as many minors or clients as they want to join the blockchain network. This mode is more realistic compared to single threaded mode. Each minor or client process provides a terminal interface for the user to interact with.
SG has eleven total JavaScript classes that we need to port into the Go language. We started with block.js and blockchain.js based on the order as they are the most essential part of a blockchain implementation. Notice block.js mainly defines a block, which is a collection of transactions with a hash connecting it to a previous block while the blockchain class tracks configuration information and settings for the blockchain. Next we converted client.js and miner.js as they define these two main roles. Then we converted the rest of the files which all serve different tasks in the original SG project.

2 Implementation

We divided our project into two phases. The first phase was to simulate the blockchain by a single process using a fake network. The second phase was to operate the blockchain by multiple independent nodes over a real network. We finished both phases.

2.1 Implementation details

Capitalized fields, methods and functions are public in Go while all the other fields are local to the package. In our implementation, all of the files are inside the blockchain package, thus it is easy for us to access them. Notice in the JavaScript implementation, we need to explicitly import/export these files. However, we do need to import related packages for external functions in Go like "fmt", "time" and "crypto/sha256".

There are no classes in Go, but Go has struct types, which are much more powerful than their C counterparts. Struct types plus their associated methods serve the same goal of a traditional class, where the struct only holds the state, not the behavior, and the methods provide them behavior, by allowing to change the state. As you can see from Fig 1, Block struct contains all declared variables that are present in block.js.

```
type Block struct {
        PrevBlockHash  string
        Target         big.Int
        Proof          uint32
        Balances       []BalanceType
        NextNonce      []NextNonceType
        Transactions   []TransactionType
        ChainLength    uint32
        Timestamp      time.Time
        RewardAddr     string
        CoinbaseReward uint32
}
```

Fig 1. Block struct

Since there are no classes in Go, we cannot create objects using constructors. Instead we use methods to achieve this task. Methods are called "func" in Go. These methods are defined outside the type definition. As an example here in Fig 2, we used a "NewClient" method to create a Client object and all initializations were done inside this method.

```
func NewClient(name string, Net *FakeNet, startingBlock *Block, keyPair *rsa.PrivateKey) *Client {
	var c Client
	c.Net = Net
	c.Name = name

	if keyPair == nil {
		c.PrivKey, c.PubKey, _ = GenerateKeypair()
	} else {
		c.PrivKey = keyPair
		c.PubKey = &keyPair.PublicKey
	}
	c.Address = GenerateAddress(c.PubKey)
	c.Nonce = 0

	c.PendingOutgoingTransactions = make(map[string]*Transaction)
	c.PendingReceivedTransactions = make(map[string]*Transaction)
	c.Blocks = make(map[string]*Block)
	c.PendingBlocks = make(map[string]*Set[*Block])

	if startingBlock != nil {
		c.SetGenesisBlock(startingBlock)
	}

	c.Emitter = emission.NewEmitter()
	c.Emitter.On(PROOF_FOUND, c.ReceiveBlockBytes)
	c.Emitter.On(MISSING_BLOCK, c.ProvideMissingBlock)
	return &c
}
```

Fig 2. NewClient method

In the SpartanGold, Miner class extends Client because miners are clients but they also mine blocks looking for "proofs". Since there is no concept of inheritance in Go, we chose to rewrite all functions from Client.go to Miner.go with some modifications. As you can see from Fig 3, inside the Miner struct, we have three additional variables that are specific to Miner.

```
type Miner struct {
        // The variables that are same as Client
        Name                        string
        Address                     string
        PrivKey                     *rsa.PrivateKey
        PubKey                      *rsa.PublicKey
        Blocks                      map[string]*Block
        PendingOutgoingTransactions map[string]*Transaction
        PendingReceivedTransactions map[string]*Transaction
        PendingBlocks               map[string]*Set[*Block]
        LastBlock                   *Block
        LastConfirmedBlock          *Block
        ReceivedBlock               *Block
        Config                      BlockchainConfig
        Nonce                       uint32
        Net                         *FakeNet
        Emitter                     *emission.Emitter
        mu                          sync.Mutex

        // Miner's specific variables
        CurrentBlock *Block
        MiningRounds uint32
        Transactions *Set[*Transaction]
}
```

Fig 3. Miner struct

We also used interfaces like NetClient in Fig 4. By declaring the NetClient as an interface, we were able to let clients and miners work together. Go interfaces are very different, and one key concept is that interfaces are satisfied implicitly. Interfaces provide polymorphism: by accepting an interface, we declare to accept any kind of object satisfying that interface.

```go
type NetClient interface {
        GetAddress() string
        GetEmitter() *emission.Emitter
}

type FakeNet struct {
        Clients map[string]NetClient
        mu      sync.Mutex
}
```

Fig 4. NetClient interface

In the JavaScript implementation, it uses a set data structure. Since go doesn't come with one, we decided to create our own set type. This is essentially a Map using an interface as shown on Fig 5. In order to guarantee its correctness, we wrote some units to test the basic add/delete operations. Notice a routine might add an item to the set while another routine is getting the list of items, or the size. So to make our data structure safe, we added a sync.Mutex inside the struct.

```go
package main

import "sync"

type SetItemInterface interface {
        GetHashStr() string
}

type Set[T SetItemInterface] struct {
        items map[string]T
        mu    sync.Mutex
}

func NewSet[T SetItemInterface]() *Set[T] {
        var newSet Set[T]
        newSet.items = make(map[string]T)
        return &newSet
}
```

Fig 5. Set

2.2 Concurrency

Unlike Javascript that doesn't support multi-threaded programming, Golang is the language that emphasizes multi-threaded programming. Golang has a built-in multi-threaded mechanism that allows the program to execute multiple methods simultaneously in different threads. The threads that Golang's built-in multi-thread mechanism uses are called goroutines. Unlike OS thread, goroutines are very low cost, simple to use, and easy to manage. The life cycle of each goroutine is automatically managed by the go runtime. As shown in Fig 6, to create a goroutine to execute a line of code, all you need is to add a "go" keyword in front of the line.

```
// Broadcasts to all clients within this.clients.
func (f *FakeNet) Broadcast(msg string, data []byte) {
    (*f).mu.Lock()
    defer (*f).mu.Unlock()
    for address := range f.Clients {
        client := f.Clients[address]
        go (client).GetEmitter().Emit(msg, data)
    }
}
```

Fig 6. Use a goroutine to emit events.

The biggest issue with multi-threaded programming is race conditions. To prevent the race condition, we need to use the mutex to do the concurrency control. The mutex can create a critical area that allows only one thread to access at one time. When a mutex is locked by a thread, all the threads that try to pass through the mutex have to wait until the mutex is unlocked again. In Fig 6, the first line of the "Broadcast" function is to lock the mutex, the second like is to unlock the mutex but with a "defer" keyword in the front. The "defer" keyword can defer executing the line of the code to the end of the function execution.

2.3 Event Emitter

Golang doesn't have a built-in event emitter mechanism like javascript has. To be able to implement a mechanism that is similar to FakeNet in the Spartan-Gold, we use a third

module called “emission”. The “emission” module is a simple event emitter implemented by Chuck Prestar [5]. In this project, each client or miner holds an emitter object that has certain events registered. Each event is associated with a specific function. To execute an event related function on a client or miner, one can simply get the emitter owned by the client or miner to call the “emit” method. The emit method takes an event message and corresponding data objects as inputs. The event message tells the emitter which function to execute, and the corresponding data objects are the objects that the function needs to use as inputs.

2.4 FakeNet

By combining goroutine and event emitter, we implement the FakeNet that simulates a network where each client or miner client runs concurrently while communicating with each other. Our FakeNet implementation is similar to the one in Spartan-Gold. The difference is that we create a new thread to emit each event. In our implementation, one client or miner sends or broadcasts an event to other clients or miners through FakeNet. The FakeNet will create a new thread to call the emit function of the emitter owned by each client or miner who needs to handle the event. Therefore, all the clients can run independently in different threads and communicate with each other through FakeNet. In the FakeNet, all the data needs to be converted to byte slice type data before being passed to other clients or miners. The purpose of this conversion is to solve the compatibility issue. In Golang, we use Marshal and Unmarshal functions in the Json module to serialize and deserialize the data. The Marshal can serialize any struct type data to byte slice data, and Unmarshal and deserialize the byte slice data back to its original type.

2.4 TcpMiner

In phase 1, all the miners and clients are running within the same process, and communication between miners and clients are through FakeNet. In the Phase 2, we implement the TcpMiner that does not rely on the FakeNet anymore, but instead each

TcpMiner will work independently as a single process. Multiple TcpMiner processes will communicate with each other through the ethernet network.

In order to communicate through ethernet, each TcpMiner needs to listen to sockets for any incoming connections. TcpMiner has a thread running a function that is specifically used to listen to a socket. As shown in Fig 7, the function uses a loop to accept incoming connections one by one. When one incoming connection is accepted, the thread will read the received data, and then create a new thread to execute a function called HandleConnection to process the received data.

```
for {
    conn, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }
    var data []byte
    for {
        chunk := make([]byte, 512)
        n, err := conn.Read(chunk)
        if err != nil {
            panic(err)
        }
        data = append(data, chunk[:n]...)

        if n < 512 {
            break
        }
    }

    go m.HandleConnection(data)
    conn.Close()
}
```

Fig. 7 Handling Incoming Connections

The received data must be able to deserialize to a TcpData Type object. As shown in Fig 8, the TcpDat Type is a struct that has two fields: Msg and Data. The Msg field is a

string object that carries the event message. The Data field is a byte slice object that carries the data that will be deserialized to other type object based on the event.

```
type TcpData struct {
    Msg  string
    Data []byte
}
```

Fig. 8 TcpData Struct Type

The HandleConnection function will deserialize the received data into TcpData Type object, and then use the emitter to emit the event based on the Msg to execute the event related function. The event related function will deserialize the data in the Data field of the TcpData object to a specific type object for further processing.

To broadcast or send messages to other processes, we cannot use FakeNet anymore, instead we implement a struct called RealNet and together with its associated functions. The difference is that the RealNet cannot not invoke other miner's event emitters directly anymore. Instead it sets up the connections to other miners through their URL and port number, and sends data, which is serialized from the TcpData object, to other miners through the real network.

3 Future Improvements

We mentioned before that since there is no concept of inheritance in Go, we chose to rewrite all the functions from Client.go to Miner.go. Another way to do this is through composition. Composition can be achieved in Go by embedding one struct type into another. In other words, we can embed the Client struct inside the Miner struct. Whenever one struct field is embedded in another, Go gives us the option to access the embedded fields as if they were part of the outer struct.
We can also use Merkle Trees to store transactions instead of maps. It will save us a lot of memory space. Merkle Trees are basically large hash trees which get saved in the

headers of a block. They are also used when the block is mined by the Proof of Work Algorithm.
We can improve our code quality by refactoring some similar functions and increasing the test coverage. Right now we used a test file as the main driver to test out the single thread mode and the multiple thread mode. Eventually, we want to write a dedicated main.go as the entering point for the program.

4 Conclusion

We were able to implement a simplified blockchain-based cryptocurrency similar to SG using Go language. While doing this project, we learnt a lot of nuances about the Go language. And most importantly, we got familiar with all the basic functionalities and design philosophies of SpartanGold. Equipped with this knowledge, we could design and build our own cryptocurrency in the future.

Reference

[1] Thomas H. Austin, “SpartanGold”, [URL] https://github.com/taustin/spartan-gold

[2] Victor Osadchiy, “Why Use the Go Language for Your Project”, [URL] https://yalantis.com/blog/why-use-go/

[3] “TIOBE Index for March 2022”, [URL] https://www.tiobe.com/tiobe-index/

[4] Naveen Ramanathan, “Goroutines”, [URL] https://golangbot.com/goroutines/

[5] Chuck Preslar, “emission”, [URL] https://github.com/chuckpreslar/emission