

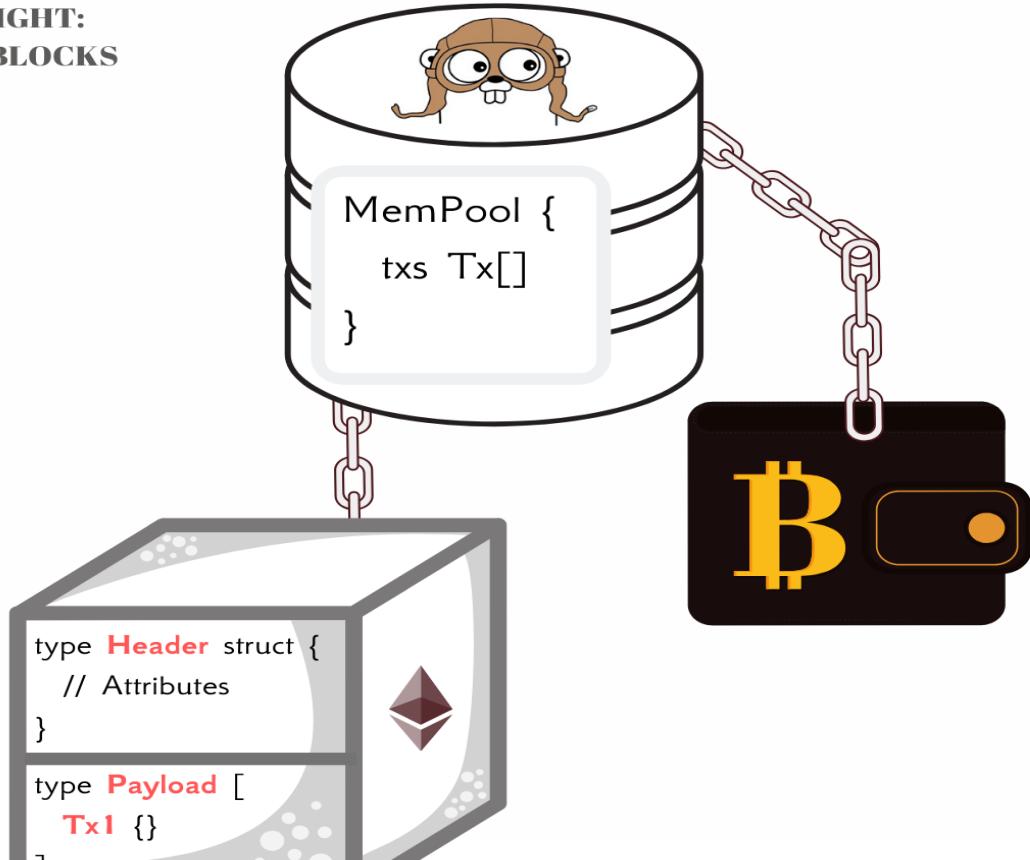
BUILD A **BLOCKCHAIN FROM SCRATCH** IN GO

Escape Crowded Java/PHP Job Market

Pioneer an Innovative Technology

Expand Your Dev Career

HEIGHT:
51 BLOCKS



Lukas Lukac

Build a Blockchain from Scratch in Go

Lukas Lukac

© 2020 - 2022 Lukas Lukac | Version: 1.21 | Last Updated:
10.09.2022 - Full Version

Contents

Foreword	1
Why Start Blockchain Development?	3
Getting Started	7
01 The MVP Database	11
User 1, Andrej	12
02 Mutating Global DB State	16
Dead Party	16
Bonus for BabaYaga	18
03 Monolithic Event vs Transaction	20
Andrej Programming	21
Building a Command-Line-Interface (CLI)	29
Mempool	40
04 Humans Are Greedy	47
Typical Business Greediness	47
05 Why We Need Blockchain	53
BabaYaga Seeks Justice	53
06 L'Hash de Immutable	57
How to Program an Immutable Database?	57
Immutability via Hash Functions	59

CONTENTS

Implementing the DB Content Hashing	61
07 The Blockchain Programming Model	69
Improving Performance of an Immutable DB	69
Batch + Hash + Linked List \Rightarrow Blocks	71
How adding a TX into a Block works	75
Migrating from TX.db to BLOCKS.db	78
08 Transparent Database	90
Flexible DB Directory	90
Centralized Public HTTP API	96
Deploying TBB Program to AWS	103
Burned-out	107
09 It Takes Two Nodes To Tango	110
Distributing the DB to Customers	110
Designing a Peer-to-Peer Sync Algorithm	112
10 Programming a Peer-to-Peer DB Sync Algorithm	117
Each Block Has a Number	117
Tell Me Your State	121
Bootstrap Nodes and Peer List	123
The Search for a Peer with New Blocks	129
Give Me Your Blocks or Life!	134
Trust but Verify	138
Use libp2p networking stack	158
11 The Autonomous Database Brain	170
The P2P Heaven: The Fastest to Rule Them All	175
How does Bitcoin Mining Works?	177
Programming Bitcoin Mining Algorithm	179
The Ethereum's Mining Algorithm: Ethash	197

CONTENTS

The Slow Elephant in the Bitcoin PoW Room	211
XRP and Proof of Authority	213
Ethereum, Proof of Stake, Sharding	214
How are Bitcoins / Ethers created?	216
Programming Bitcoin Mining Reward in 4 Steps	217
Writing Tests for Proof of Work	221
Blockchain Releases are Complicated	239
Migrating to PoW DB	242
12 Madam/Sir Your Cryptographic Signature Please	249
Hacking a User Balance	249
Current State of Web Authentication	251
Asymmetric Cryptography in Nutshell	253
Blockchain Authentication with go-Ethereum	255
Programming a Cryptocurrency Wallet	259
MySQL vs Blockchain Authorization	276
Madam/Sir, Sign this Database Change	279
Programming Signed Transactions	292
Digital Signature Replay Attack	300
Celebrating a Secure Blockchain	317
13 Bonus: Why a Transaction Costs Gas	324
What's Gas?	325
How much is 1 Gas?	325
Spamming the Network	326
Setting a “Gas Cost” and a “Gas Price”	329
Where do the Gas Fees Go?	329
Coding the Gas Fees	330
14 Bonus: Blockchain Forks - What, Why, How	338
Why a Fork Happens	341

CONTENTS

How to Plan a Consensus Fork	350
Implementing a Consensus Fork	352
New TBB Specification	355
Deploying the Fork	368
Synchronize Your Node	370
15 Bonus: IPFS - the Decentralized Storage of Web3	372
16 Bonus: Merkle Trees in Go	375
17 Bonus: Blockchain Storage (complexity level 99)	378
Official TBB Training Ledger	380
Create a New Account	381
Connect Your Node to the TBB Training Ledger	381
Query The Blockchain Bar Customer Balances	382
Request 1000 TBB Testing Tokens	383
El Fin - Congrats!	384

Foreword

Hi, you've made a great decision to expand your programming career!

Blockchain is a paradigm shift in software development and architecture. I recommend everyone to learn and master it!

In this book you learn:

- Peer-to-peer systems software architecture
- Distributed event-based architecture
- How servers can communicate autonomously (BTC, ETH, XRP)
- Go programming language ❤
- Encodings and secure hashing
- Asymmetric cryptography

To be honest, it's extremely challenging to write a technical blockchain eBook.

The junior developers want to explore the blockchain world.

The advanced developers want to grasp peer-to-peer architecture and build a working prototype but without reading academic yellow papers with hundreds of math formulas.

The hardcore experts on distributed systems need in-depth material.

This **eBook is written in a form of a story, with executable source code** you can clone for every chapter. You start your journey with basic blockchain concepts and by **programming simplified p2p components**. Every chapter will be more difficult than the previous one. **You will also find additional in-depth material in form of blogs, books and research papers referenced at the end of some chapters** - shall you wish to dive into the low level concepts such as Merkle Patricia Trie, Recursive-length prefix (RLP) serialization and other blockchain gems.

After you finish the eBook, you will have a solid understanding of core blockchain components.

I wish you a pleasant journey exploring the blockchain underworld.

Lukas,

Why Start Blockchain Development?

Two years ago, I was at a crossroads in my career. I was looking for my next programming job after five years of PHP development at trivago.

I was deciding between a Java position in NewRelic or a GoLang position at a new blockchain startup. Yep. I went with Go and blockchain. Why? Because it's better for my long-term career. And I recommend you do the same.

If I'd continued in Java development, I'd have kept programming monoliths, occasionally learning something here and there. The majority of the time, I would have been working on autopilot, fixing bugs, and maybe implementing a nice feature every second sprint. At best, it would have been a good, safe, and interesting choice, but a bit slow for my personal taste.

At its worst, I'd be working on broken micro-services :) (Just joking, they are great for scaling individual parts of the project and dividing the ownership between independent teams.)

Aside from the exciting opportunities, there is also a financial aspect to every job. We all have bills to pay. In the Java/PHP/Javascript world, I would compete for a salary raise with 10 million other Java

developers, each with 20 years of experience. Certainly doable, but it's an uphill battle that gets harder by the day.

Web 3.0 is coming. In the last ten years, we've helped build large platforms for mass communication. Life updates? Facebook. Story to tell? Medium. Picture to share? Instagram. Unfortunately, these platforms have a dark side. And this dark side powers their entire business model. Users have given up their data sovereignty and lost attention to advertisements.

But it doesn't have to be like this.

Do you remember when RSS was king, and everyone had a blog? Web3.0 aims to rebuild and improve this vision. Join me and explore the new **tokenized economies** optimizing the value exchange between participants by entirely removing an intermediary. Learn **blockchain immutability**, combine it with **asymmetric cryptography**, and **develop new transparent, open-sourced systems that users can trust** while preventing centralized data breaches.

This book will show you precisely that.

Meet the book's main character. Andrej.

Andrej is a bar owner by night and a software developer by day in a small Slovakian town called Bardejov.

Andrej is tired of:

- **Programming solid, old fashion PHP/Java applications**
- Forgetting how much money his friends and clients owe him for all the unpaid Friday nights vodka shots
- Spending time collecting and counting coins, returning change and generally touching COVID-19 bank bills
- Maintaining different plastic chips, tokens for table football, darts, billiard and poker

Andrej would love to:

- **Have a perfect auditable history of the bar's activities** and sales to make his bar compliant with tax regulations
- **Transform his bar into an autonomous, payment-efficient and safe environment his customers can trust**

“This will be a programming dream!” he tells himself. “I am going to write a simple program and keep all the balances of my clients in a virtual form.

“Every new customer will give me cash, and **I will credit them an equivalent amount of my digital tokens.** The tokens will represent a monetary unit within and outside the bar.

“The users will use the tokens for all bar functionalities from paying for drinks, borrowing and lending them to their friends, and playing table tennis, poker and kicker.

“I will call the tokens: The Blockchain Bar tokens, **TBB!**”

Getting Started

The book is written in Go, but don't worry - you don't need to have any prior Go experience to start reading the book. It's a very powerful, and beginner-friendly language - you will pick it up quickly.

Requirements

I recommend 2+ years of programming experience in Java / PHP / Javascript, or another language similar to Go.

Complete the free [17 lectures of A Tour Of Go¹](#) to get familiar with the syntax and basic concepts.

¹<https://tour.golang.org/basics/1>

Why Go?

Because like blockchain, it's a fantastic technology for your overall programming career. Go is a trendy language and better paid than an average Java/PHP position.

Go is optimized for multi-core CPU architecture. You can spawn thousands of light-weight threads (Go-routines) without problems. It's extremely practical for highly parallel and concurrent software such as blockchain networks. By writing your software in Go, you achieve nearly C++ level of performance out of the box without killing yourself for that one time you forgot to free-up memory.

Go also compiles to binary which makes it very portable.

Setup the project

Join the private Discord chat

Join The Blockchain Bar's chat server lobby <https://discord.gg/F2e2Qfz>², and send me your purchased book's **GUMROAD LICENSE KEY** by **a DM** to Web3Coach#9926 (my Discord username). Or via Twitter to [@Web3Coach](#)³.

I will add you to a private students room, called **tbb-students**, where I will be individually supporting you on your new blockchain journey.

Install Go and Clone the TBB project

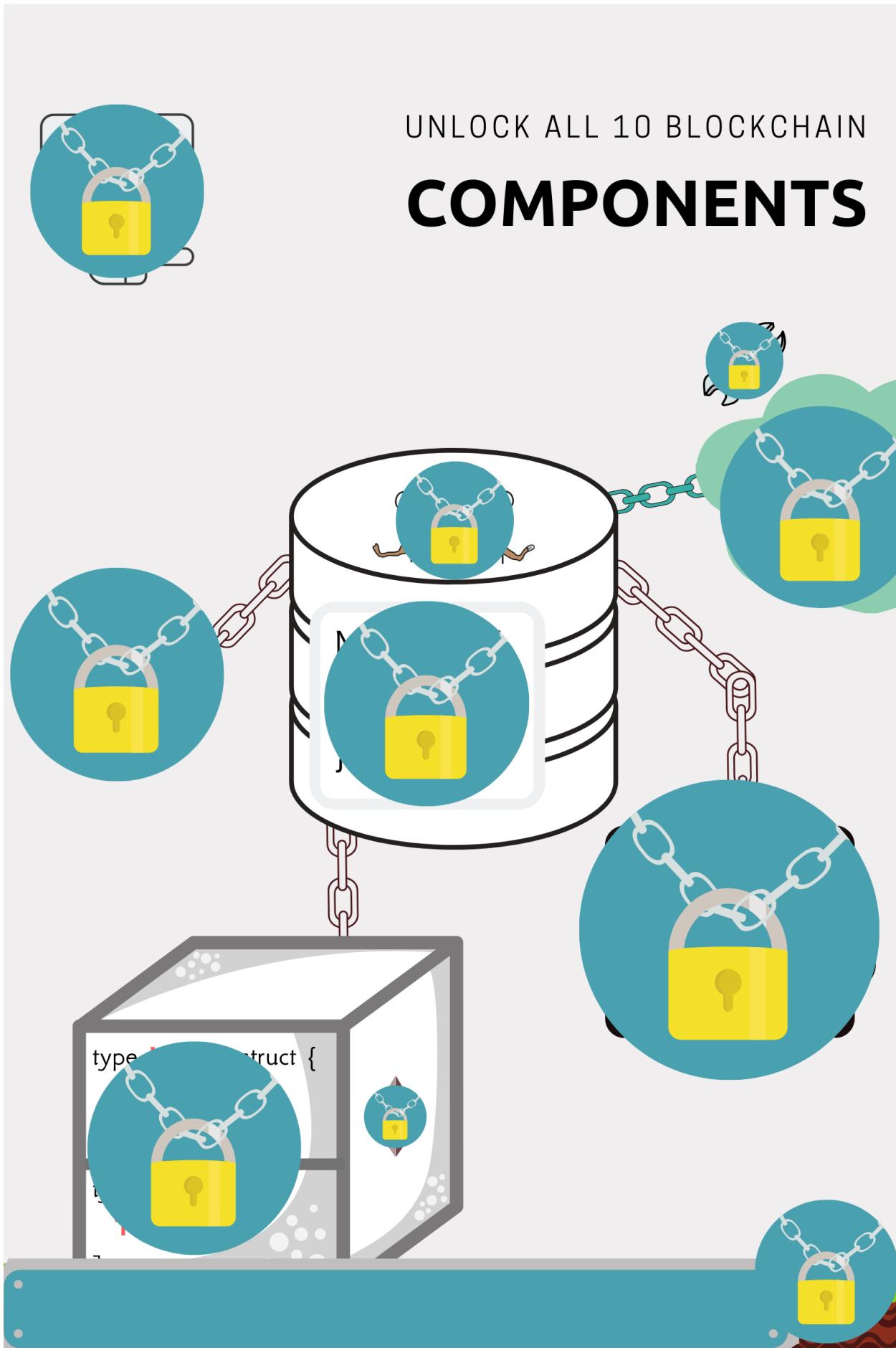
↓ Visit The Blockchain Bar Github repository and follow the installation instructions ↓

[How to install Go and this repository.](#)⁴

² <https://discord.gg/F2e2Qfz>

³ <https://twitter.com/Web3Coach>

⁴ https://github.com/web3coach/the-blockchain-bar/blob/c1_genesis_json/installation.md



01 | The MVP Database

>_ git checkout c1_genesis_json

Andrej mastered relational SQL databases in the 90s. He knows how to make advanced data models and how to optimize the SQL queries.

It's time for Andrej to catch-up with innovation and start building Web 3.0 software.

Luckily, after reading "The Lean Startup" book last week, Andrej feels like he shouldn't over-engineer the solution just yet. Hence, he chooses a simple but effective, JSON file for the bar's MVP database.

In the beginning, there was a primitive centralized database.



Blockchain is a database.

User 1, Andrej

Monday, March 18.

Andrej generates 1M utility tokens.

In the blockchain world, tokens are units inside the blockchain database. Their real value in dollars or euro fluctuates based on their demand and popularity.

Every blockchain has a “**Genesis**” file. The Genesis file is used to distribute the first tokens to early blockchain participants.

It all starts with a simple, dummy, **genesis.json**.

Andrej creates the file `./database/genesis.json` where he defines that The Blockchain Bar’s database will have 1M tokens and all of them will belong to Andrej:

```
{  
  "genesis_time": "2019-03-18T00:00:00.000000000Z",  
  "chain_id": "the-blockchain-bar-ledger",  
  "balances": {  
    "andrej": 1000000  
  }  
}
```

The tokens need to have a real “utility”, i.e., a use case. Users should be able to pay with them from day 1! Andrej must comply with law regulators (SEC). It is illegal to issue unregistered security. On the

other hand, utility tokens are fine, so he right away prints and sticks a new pricing white paper poster on the bar's door.

Andrej assigns a starting monetary value to his tokens so he can exchange them for euro, dollars or other fiat currency.

```
1 TBB token = 1€
```

Item	Price
Vodka shot	1 TBB
Orange juice	5 TBB
Burger	2 TBB
Crystal Head Vodka Bottle	950 TBB

Andrej also decides, **he should be getting 100 tokens per day** for maintaining the database and having such a brilliant disruptive idea.



Fun Facts

The first genesis Ether (ETH) on Ethereum blockchain was created and distributed to early investors and developers in the same way as Andrej's utility token.

In 2017, during an ICO (initial coin offerings) boom on the Ethereum blockchain network, project founders wrote and presented whitepapers to investors. A whitepaper is a technical document outlining a complex issue and possible solution, meant to educate and elucidate a particular matter. In the world of blockchains, a white paper serves to outline the specifications of how that particular blockchain will look and behave once it is developed.

Blockchain projects raised between €10M to €300M per **whitepaper** idea. An idea. An idea, sometimes even without any technical documentation or research supporting it! Wild times, but well, if a 49318th food delivery start-up can be valued at few billions, why not...

In exchange for money (the ICO “funding”), investor names would be included in the initial “genesis balances”, similar to how Andrej did it. Investors’ hopes through an ICO are the genesis coins go up in value and that the teams deliver the outlined blockchain.

Naturally, not all whitepaper ideas come to fruition. Massive investments lost to unclear or incomplete ideas are why blockchain received negative coverage in the media throughout these ICOs, and why some still considered it a hype. But the underlying blockchain technology is fantastic and useful, as you will learn further in this book. It’s just been abused by some bad actors.



Summary

Blockchain is a database.

The token supply, initial user balances, and global blockchain settings you define in a Genesis file.



Study Code

Commit: [c6b8eb⁵](https://github.com/web3coach/the-blockchain-bar/commit/c6b8eb3e889550f15be59e659f54970a87b3e94a)

⁵<https://github.com/web3coach/the-blockchain-bar/commit/c6b8eb3e889550f15be59e659f54970a87b3e94a>

02 | Mutating Global DB State

>_ git checkout c2_db_changes_txt

Dead Party

Monday, March 25.

After a week of work, the bar facilities are ready to accept tokens. Unfortunately, no one shows up, so Andrej orders three shots of vodka for himself and writes the database changes on a piece of paper:

```
andrei-3;    // 3 shots of vodka
andrei+3;    // technically purchasing from his own bar
andrei+700;  // Reward for a week of work (7x100 per day)
```

To avoid recalculating the latest state of each customer's balance, Andrej creates a `./database/state.json` file storing the balances in an aggregated format.

New DB state:

```
{  
  "balances": {  
    "andrey": 1000700  
  }  
}
```

Bonus for BabaYaga

Tuesday, March 26.

To bring traffic to his bar, Andrej announces an exclusive 100% bonus for everyone who purchases the TBB tokens in the next 24 hours.

Bing! He gets his first customer called **BabaYaga**. BabaYaga pre-purchases 1000€ worth of tokens, and to celebrate, she immediately spends 1 TBB for a vodka shot. She has a drinking problem.

DB transactions written on a piece of paper:

```
andrey-2000;    // transfer to BabaYaga  
babayaga+2000; // pre-purchase with 100% bonus  
babayaga-1;  
andrey+1;  
andrey+100;     // 1 day of sun coming up
```

New DB state:

```
{  
  "balances": {  
    "andrey": 998801,  
    "babayaga": 1999  
  }  
}
```



Fun Facts

Blockchain ICO (initial coin offerings based on whitepapers) projects often distribute the genesis tokens with different bonuses, depending on how many of them you buy and how early you do it. Teams offer, on average, 10-40% bonuses to early “participants”.

The word “investor” is avoided, so the law regulators won’t consider the tokens being a security. Projects would reason their main product, blockchain tokens, function as “flying, loyalty points.”

The “participants” later made even 1000% (four zeroes!) on their investment selling to the public through an exchange several months later.



Summary

Blockchain is a database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file. **The Genesis balances indicate what was the original blockchain state and are never updated afterwards.**

The database state changes are called Transactions (TX).



Study Code

Commit: [def8c1⁶](#)

⁶<https://github.com/web3coach/the-blockchain-bar/commit/def8c169292e258525e2f48a075cb26174e52809>

03 | Monolithic Event vs Transaction

>_ git checkout c3_state_blockchain_component

Developers used to event-sourcing architecture must have immediately recognized the familiar principles behind transactions. They are correct. Blockchain transactions represent a series of events, and the database is a final aggregated, calculated state after replaying all the transactions in a specific sequence.

Andrej Programming

Tuesday evening, March 26.

It's a relaxing Tuesday evening for Andrej. Celebrating his first client, he decides to play some [Starcraft⁷](#) and clean up his local development machine by removing some old pictures. Unfortunately, he prematurely pressed enter when typing a removal command path in terminal `sudo rm -rf ./`. Oops.

All his files, including the bar's `genesis.json` and `state.json` are gone.

Andrej, being a senior developer, repeatedly shouted some f* words very loudly for a few seconds, but he didn't panic! While he didn't have a backup, he had something better — a piece of paper with all the database transactions. The only thing he'd need to do is replay all the transactions one by one, and his database state would get recovered.

Impressed by the advantages of event-based architecture, he decides to extend his MVP database solution. Every bar's activity, such as individual drink purchases, MUST be recorded inside the blockchain database.

⁷ <https://www.youtube.com/watch?v=Ff4VlghrTMg&feature=youtu.be&t=516>

Each **customer** will be represented in DB using an **Account** Struct:

```
type Account string
```

Each **Transaction** (TX - a database change) will have the following four attributes: **from**, **to**, **value** and **data**.

The **data** attribute with one possible value (**reward**) captures Andrej's bonus for inventing the blockchain and increases the initial TBB tokens total supply artificially (inflation).

```
type Tx struct {
    From  Account `json:"from"`
    To    Account `json:"to"`
    Value uint    `json:"value"`
    Data  string  `json:"data"`
}

func (t Tx) IsReward() bool {
    return t.Data == "reward"
}
```

The **Genesis DB** will stay as JSON file:

```
{
  "genesis_time": "2019-03-18T00:00:00.000000000Z",
  "chain_id": "the-blockchain-bar-ledger",
  "balances": {
    "andrey": 1000000
  }
}
```

All the transactions, previously written on a piece of paper, will be stored in a local text-file database called **tx.db**, serialized in JSON format and separated by line-break character:

```
{"from": "andrey", "to": "andrey", "value": 3, "data": ""}
{"from": "andrey", "to": "andrey", "value": 700, "data": "reward"}
 {"from": "andrey", "to": "babayaga", "value": 2000, "data": ""}
 {"from": "andrey", "to": "andrey", "value": 100, "data": "reward"}
 {"from": "babayaga", "to": "andrey", "value": 1, "data": ""}
```

The most crucial database component encapsulating all the business logic will be **State**:

```
type State struct {
  Balances map[Account]uint
  txMempool []Tx

  dbFile *os.File
}
```

The State struct will know about all user balances and who transferred TBB tokens to whom, and how many were transferred.

It's constructed by reading the initial user balances from genesis.json file:

```
func NewStateFromDisk() (*State, error) {
    // get current working directory
    cwd, err := os.Getwd()
    if err != nil {
        return nil, err
    }

    genFilePath := filepath.Join(cwd, "database", "genesis.json")
    gen, err := loadGenesis(genFilePath)
    if err != nil {
        return nil, err
    }

    balances := make(map[Account]uint)
    for account, balance := range gen.Balances {
        balances[account] = balance
    }
}
```

Afterwards, the genesis State balances are updated by sequentially replaying all the database events from tx.db:

```
txDbFilePath := filepath.Join(cwd, "database", "tx.db")
f, err := os.OpenFile(txDbFilePath, os.O_APPEND|os.O_RDWR, 0600)
if err != nil {
    return nil, err
}

scanner := bufio.NewScanner(f)
state := &State{balances, make([]Tx, 0), f}

// Iterate over each the tx.db file's line
for scanner.Scan() {
    if err := scanner.Err(); err != nil {
        return nil, err
    }

    // Convert JSON encoded TX into an object (struct)
    var tx Tx
    json.Unmarshal(scanner.Bytes(), &tx)

    // Rebuild the state (user balances),
    // as a series of events
    if err := state.apply(tx); err != nil {
        return nil, err
    }
}

return state, nil
}
```

The State component is responsible for:

- **Adding** new transactions to **Mempool**
- **Validating** transactions against the current State (sufficient sender balance)
- **Changing** the state
- **Persisting** transactions to disk
- **Calculating** accounts balances by replaying all transactions since Genesis in a sequence

Adding new transactions to Mempool:

```
func (s *State) Add(tx Tx) error {
    if err := s.apply(tx); err != nil {
        return err
    }

    s.txMempool = append(s.txMempool, tx)

    return nil
}
```

Persisting the transactions to disk:

```
func (s *State) Persist() error {
    // Make a copy of mempool because the s.txMempool will be modified
    // in the loop below
    mempool := make([]Tx, len(s.txMempool))
    copy(mempool, s.txMempool)

    for i := 0; i < len(mempool); i++ {
        txJson, err := json.Marshal(mempool[i])
        if err != nil {
            return err
        }

        if _, err = s.dbFile.Write(append(txJson, '\n')); err != nil {
            return err
        }
    }

    // Remove the TX written to a file from the mempool
    s.txMempool = s.txMempool[1:]
}

return nil
}
```

Changing, Validating the state:

```
func (s *State) apply(tx Tx) error {
    if tx.IsReward() {
        s.Balances[tx.To] += tx.Value
        return nil
    }

    if tx.Value > s.Balances[tx.From] {
        return fmt.Errorf("insufficient balance")
    }

    s.Balances[tx.From] -= tx.Value
    s.Balances[tx.To] += tx.Value

    return nil
}
```

Building a Command-Line-Interface (CLI)

Tuesday evening, March 26.

Andrej wants to have a convenient way to add new transactions to his DB and list the latest balances of his customers. Because Go programs compile to binary, he builds a CLI for his program.

The easiest way to develop CLI based programs in Go is by using the third party github.com/spf13/cobra library.

Andrej initializes Go's built-in dependency manager for his project, called `go modules`:

>_ \$

```
cd $GOPATH/src/github.com/web3coach/the-blockchain-bar
```

```
go mod init github.com/web3coach/the-blockchain-bar
```

The `Go modules` command will automatically fetch any library you reference within your Go files.

Andrej creates a new directory called: `cmd` with a subdirectory `tbb`:

>_ mkdir -p ./cmd/tbb

Inside he creates a main.go file, serving as the program's CLI entry point:

```
package main

import (
    "github.com/spf13/cobra"
    "os"
    "fmt"
)

func main() {
    var tbbCmd = &cobra.Command{
        Use:   "tbb",
        Short: "The Blockchain Bar CLI",
        Run: func(cmd *cobra.Command, args []string) {
            },
    }

    err := tbbCmd.Execute()
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

The Go programs are compiled using the `install` cmd:

```
>_ go install ./cmd/tbb/...
go: finding github.com/spf13/cobra v1.0.0
go: downloading github.com/spf13/cobra v1.0.0
go: extracting github.com/spf13/cobra v1.0.0
```

Go will detect missing libraries and automatically fetch them before compiling the program. Depending on your \$GOPATH the resulting program will be saved in the \$GOPATH/bin folder.

```
>_ echo $GOPATH  
> /home/web3coach/go  
  
which tbb  
> /home/web3coach/go/bin/tbb
```

You can run tbb from your terminal now, but it will not do anything because the Run function inside the main.go file is empty.

The first thing Andrej needs is versioning support for his tbb CLI program.

Next to the `main.go` file, he creates a `version.go` command:

```
package main

import (
    "fmt"
    "github.com/spf13/cobra"
)

const Major = "0"
const Minor = "1"
const Fix = "0"
const Verbal = "TX Add && Balances List"

var versionCmd = &cobra.Command{
    Use:   "version",
    Short: "Describes version.",
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Printf("Version: %s.%s.%s-beta %s", Major, Minor, Fix, Verbal)
    },
}
```

Compiles and runs it:

```
>_ go install ./cmd/tbb/...
tbb version
> Version: 0.1.0-beta TX Add && Balances List
```

Perfect.

Identically to the version.go file, he creates a balances.go file:

```
func balancesCmd() *cobra.Command {
    var balancesCmd = &cobra.Command{
        Use:   "balances",
        Short: "Interact with balances (list...).",
        PreRunE: func(cmd *cobra.Command, args []string) error {
            return incorrectUsageErr()
        },
        Run: func(cmd *cobra.Command, args []string) {
        },
    }

    balancesCmd.AddCommand(balancesListCmd)

    return balancesCmd
}
```

The balances command will be responsible for loading the latest DB State and printing it to the standard output:

```
var balancesListCmd = &cobra.Command{
    Use:   "list",
    Short: "Lists all balances.",
    Run: func(cmd *cobra.Command, args []string) {
        state, err := database.NewStateFromDisk()
        if err != nil {
            fmt.Fprintln(os.Stderr, err)
            os.Exit(1)
        }
        defer state.Close()

        fmt.Println("Accounts balances:")
        fmt.Println("_____")
```

```
    fmt.Println("")  
    for account, balance := range state.Balances {  
        fmt.Println(fmt.Sprintf("%s: %d", account, balance))  
    }  
,  
}
```

Andrej verifies if the cmd works as expected. It should print the exact balances defined in the Genesis file because the tx.db file is still empty.

>_ go install ./cmd/tbb/...

tbb balances list

Accounts balances:

andrej: 1000000

Works well! Now he only needs a cmd for recording the bar's activity.

Andrej creates ./cmd/tbb/tx.go cmd:

```
func txCmd() *cobra.Command {
    var txsCmd = &cobra.Command{
        Use:   "tx",
        Short: "Interact with txs (add...).",
        PreRunE: func(cmd *cobra.Command, args []string) error {
            return incorrectUsageErr()
        },
        Run: func(cmd *cobra.Command, args []string) {
        },
    }

    txsCmd.AddCommand(txAddCmd())

    return txsCmd
}
```

The tbb tx add cmd uses State.Add(tx) function for persisting the bar's events into the file system:

```
func txAddCmd() *cobra.Command {
    var cmd = &cobra.Command{
        Use:   "add",
        Short: "Adds new TX to database.",
        Run: func(cmd *cobra.Command, args []string) {
            from, _ := cmd.Flags().GetString(flagFrom)
            to, _ := cmd.Flags().GetString(flagTo)
            value, _ := cmd.Flags().GetUint(flagValue)
            data, _ := cmd.Flags().GetString(flagData)

            fromAcc := database.NewAccount(from)
            toAcc := database.NewAccount(to)

            tx := database.NewTx(fromAcc, toAcc, value, data)

            state, err := database.NewStateFromDisk()
            if err != nil {
                fmt.Fprintln(os.Stderr, err)
                os.Exit(1)
            }

            // defer means, at the end of this function execution,
            // execute the following statement
            // (close DB file with all TXs)
            defer state.Close()

            // Add the TX to an in-memory array (pool)
            err = state.Add(tx)
            if err != nil {
                fmt.Fprintln(os.Stderr, err)
                os.Exit(1)
            }
        },
    }
}
```

```
// Flush the mempool TXs to disk
err = state.Persist()
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

fmt.Println("TX successfully added to the ledger.")
},
}
```

The tbb tx add cmd has 3 mandatory flags: --from, --to and --value.

```
cmd.Flags().String(flagFrom, "", "From what account to send tokens")
cmd.MarkFlagRequired(flagFrom)

cmd.Flags().String(flagTo, "", "To what account to send tokens")
cmd.MarkFlagRequired(flagTo)

cmd.Flags().Uint(flagValue, 0, "How many tokens to send")
cmd.MarkFlagRequired(flagValue)

cmd.Flags().String(flagData, "", "Possible values: 'reward'")

return cmd
```

The CLI is done!

Andrej migrates all transactions from paper to his new DB:

```
>_ tbb tx add --from=andrej --to=andrej --value=3  
      tbb tx add --from=andrej --to=andrej --value=700  
      --data=reward  
  
      tbb tx add --from=andrej --to=babayaga --value=2000  
  
      tbb tx add --from=andrej --to=andrej --value=100  
      --data=reward  
  
      tbb tx add --from=babayaga --to=andrej --value=1
```

Read all TXs from disk and calculate the latest state:

```
>_ tbb balances list
```

Accounts balances:

```
andrey: 998801  
babayaga: 1999
```

Bar data successfully restored! Phew, what a night!

About Cobra CLI library

The good thing about the Cobra lib for CLI programming is the additional features it comes with. For example, you can now run: tbb help cmd and it will print out all TBB registered sub-commands with instructions on how to use them.

```
tbb help
```

The Blockchain Bar CLI

Usage:

```
tbb [flags]  
tbb [command]
```

Available Commands:

```
balances    Interact with balances (list...).  
help        Help about any command  
tx          Interact with txs (add...).  
version     Describes version.
```

Flags:

```
-h, --help  help for tbb
```

Use "tbb [command] --help" for more information about a command.

Mempool

The TBB Mempool component is very simplified and only for educational purposes. When designing a production system you have to consider many more scenarios. How much memory should Mempool consume? What happens when you receive new Transactions, but your node's Mempool is full? Should blockchain users pay fee for getting their Transactions processed? How much is the fee (gas)? Is the fee fixed or calculated dynamically based on supply and demand? Do you process Transactions in FIFO style based on time or based on fee? Will you modify your node's codebase and favor Transactions from specific users? How do you prevent a [Distributed denial-of-service \(DDoS\)](#)⁸ attack spamming you with millions of new Transactions?

I highly recommend you to dive into the go-ethereum codebase and inspire yourself. **If you want my advice, study go-ethereum - it's the best codebase to learn backend blockchain development.** Seriously. Dive into the code and search for components: TxPool and TxPoolConfig.

TxPool config

⁸https://en.wikipedia.org/wiki/Denial-of-service_attack

```
// Configuration parameters of the transaction pool.
type TxPoolConfig struct {
    // Addresses that should be treated by default as local
    Locals      []common.Address
    // Whether local transaction handling should be disabled
    NoLocals   bool
    // Journal of local transactions to survive node restarts
    Journal    string
    // Time interval to regenerate the local transaction journal
    Rejournal time.Duration

    // Minimum gas price to enforce for acceptance into the pool
    PriceLimit uint64
    // Minimum price bump percentage to replace
    // an already existing transaction (nonce)
    PriceBump  uint64

    // Number of executable transaction slots guaranteed per account
    AccountSlots uint64
    // Maximum number of executable transaction slots for all accounts
    GlobalSlots  uint64
    // Maximum number of non-executable transaction slots
    // permitted per account
    AccountQueue uint64
    // Maximum number of non-executable transaction slots
    // for all accounts
    GlobalQueue  uint64

    // Maximum amount of time non-executable
    // transaction are queued
    Lifetime time.Duration
}

// Default configurations for the transaction pool.
var DefaultTxPoolConfig = TxPoolConfig{
    Journal: "transactions.rlp",
```

```
Rejournal: time.Hour,  
  
PriceLimit: 1,  
PriceBump: 10,  
  
AccountSlots: 16,  
GlobalSlots: 4096,  
AccountQueue: 64,  
GlobalQueue: 1024,  
  
Lifetime: 3 * time.Hour,  
}
```

TxPool

```
// TxPool contains all currently known transactions.  
//  
// Transactions enter the pool when they are received  
// from the network or submitted locally.  
//  
// They exit the pool when they are included in the blockchain.  
//  
// The pool separates processable transactions  
// (which can be applied to the current state) and future transactions.  
//  
// Transactions move between those two states over time  
// as they are received and processed.  
type TxPool struct {  
    config      TxPoolConfig  
    chainconfig *params.ChainConfig  
    chain       blockChain  
    gasPrice    *big.Int  
    txFeed     event.Feed  
    scope       event.SubscriptionScope  
    signer     types.Signer
```

```
mu sync.RWMutex

// Fork indicator whether we are in the istanbul stage.
istanbul bool

// Current state in the blockchain head
currentState *state.StateDB
// Pending state tracking virtual nonces
pendingNonces *txNonce
// Current gas limit for transaction caps
currentMaxGas uint64

// Set of local transaction to exempt from eviction rules
locals *accountSet
// Journal of local transaction to back up to disk
journal *txJournal

// All currently processable transactions
pending map[common.Address]*txList
// Queued but non-processable transactions
queue map[common.Address]*txList
// Last heartbeat from each known account
beats map[common.Address]time.Time
// All transactions to allow lookups
all *txLookup
// All transactions sorted by price
priced *txPricedList

chainHeadCh chan ChainHeadEvent
chainHeadSub event.Subscription
reqResetCh chan *txpoolResetRequest
reqPromoteCh chan *accountSet
queueTxEventCh chan *types.Transaction
reorgDoneCh chan chan struct{}
// requests shutdown of scheduleReorgLoop
reorgShutdownCh chan struct{}
```

```
// tracks loop, scheduleReorgLoop
wg sync.WaitGroup
}
```



Fun Facts

Accidentally losing customers' data is a standard Saturday in the corporate world these days. Blockchain fixes this by decentralizing the data storage.

The trick Andrej baked into the program by skipping balance verification for TXs marked as rewards. **Bitcoin and Ethereum work in the same way.** The balance of the Account who **mined a block** increases out of the blue as a subject of total tokens supply inflation affecting the whole chain. The total supply of bitcoins is capped at 21M BTC. You will learn more about “mining” and “blocks” in chapters 7 and 10.

The components **State** and **Mempool** are not unique to this program. Andrej chose the names and designs to match a simplified [go-Ethereum model⁹](#), so you have a glance inside the core Ethereum source code.

⁹ https://github.com/ethereum/go-ethereum/blob/7b32d2a47017570c44cd7f8a83612a29656c9857/core/tx_pool.go#L211



Summary

Blockchain is a database. The token supply, initial user balances, and global blockchain settings are defined in a Genesis file. The Genesis balances indicate what the original blockchain state was and are never updated afterwards.

The database state changes are called Transactions (TX).

Transactions are old fashion Events representing actions within the system.

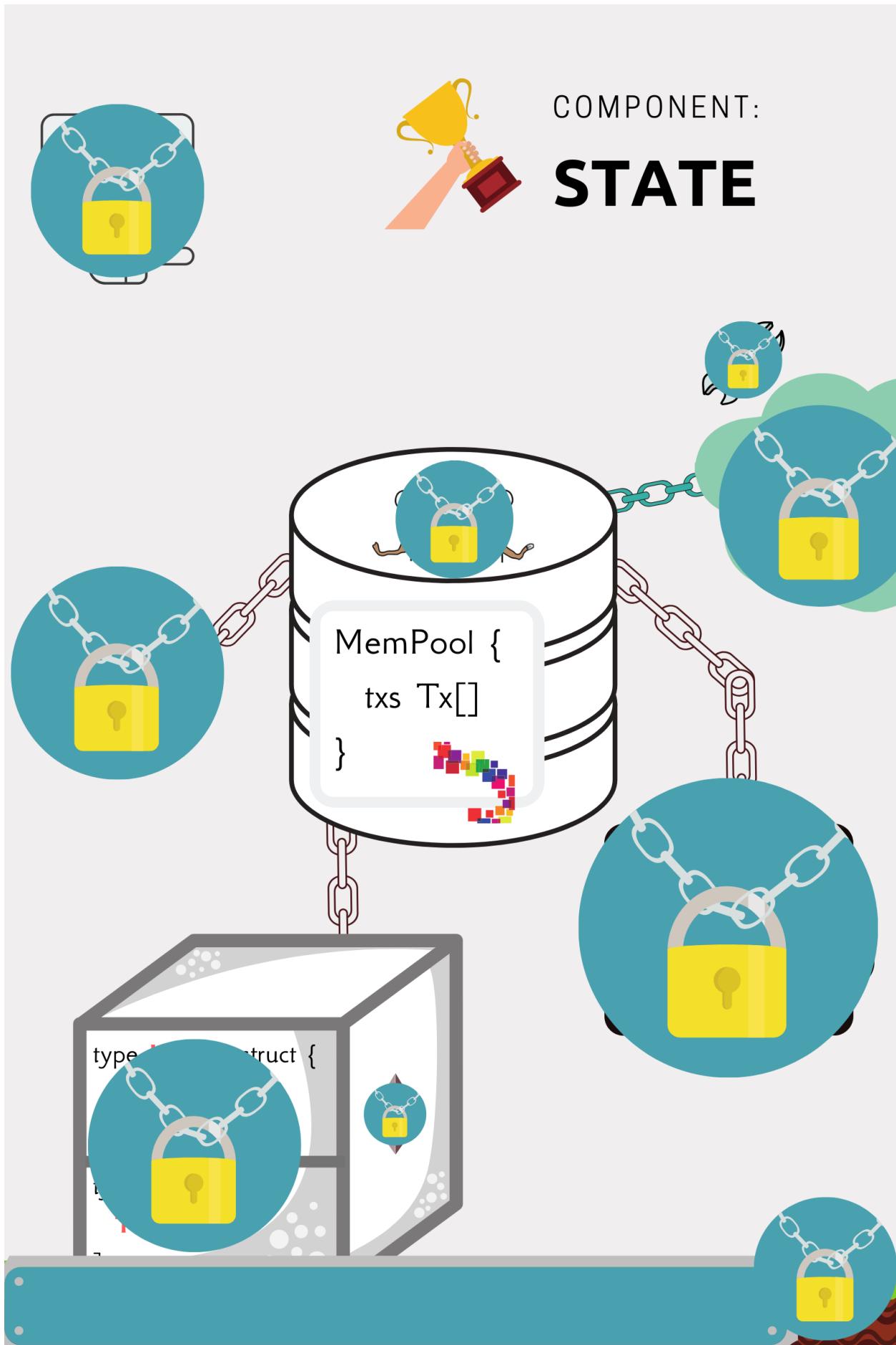


Study Code

Commit: [5d4b0b¹⁰](https://github.com/web3coach/the-blockchain-bar/commit/5d4b0b6a001e616109da732fdaf7094f1e1acf85)

Let's talk about greed.

¹⁰ <https://github.com/web3coach/the-blockchain-bar/commit/5d4b0b6a001e616109da732fdaf7094f1e1acf85>



04 | Humans Are Greedy

>_ git checkout c4_caesar_transfer

Typical Business Greediness

Wednesday, March 27.

BabaYaga invested a bit too much. She forgot her flat rent payment was around the corner, and she doesn't have the money. BabaYaga calls her flat owner, **Caesar**.

BabaYaga: Hey Caesar, I am sorry, but I don't have the cash to pay you the rent this month...

Caesar: Why not?

BabaYaga: The Blockchain Bar ICO offered a massive bonus, and I purchased 2000€ worth of tokens for just 1000€. It was a great deal!

Caesar: What the hell are you talking about? What is an ICO? What on earth are tokens? Can you pay me in some other way?

BabaYaga: Oh, not again. I can give you 1000 TBB tokens worth 1000€, and you can use them in the bar to pay for your drinks! Let me call the bar owner, Andrej, and make the transfer!

Caesar: All right... I will take it.

Andrej performs the transfer, **but decides to charge an extra 50 TBB tokens for his troubles**. He doesn't want to, BUT the bar shareholders who invested in him a few years ago are forcing him to generate profit as soon as possible.

BabaYaga won't notice this relatively small fee most likely anyway, Andrej tells himself. In the end, only he has the DB access.

```
>_ // rent payment
    tbb tx add --from=babayaga --to=caesar --value=1000

    // hidden fee charge
    tbb tx add --from=babayaga --to=andrej --value=50

    // new reward for another day of maintaining the DB
    tbb tx add --from=andrej --to=andrej --value=100
    --data=reward
```



“Fun” Facts 1/3

At the time of writing, the number one blockchain use-case is banking.

Bitcoin is positioned as a store of value.

XRP focuses on international money exchange and optimizing the currency corridors.

Ethereum is building decentralized finance via Smart Contracts.

Other projects focus on freedom and self-sovereign identity (SSI) - a digital movement that recognizes an individual should own and control their identity, assets, money without the intervening administrative authorities or other centralized intermediaries. SSI allows people to interact in the digital world with the same freedom and capacity for trust as they do in the offline world.

Cryptocurrency is a natural technological evolution. The current financial market, built on antiquated technology, has too many limitations.

Take stocks as an example. You can buy and sell only during business hours, and you MUST use a centralized broker, who takes a fee cut 1-3% from your average 7% yearly profit - can be replaced by a Smart Contract. Next, international bank transfer. It takes between 3-10 business days and can cost as much 5% of the transferred value! If you’re sending \$10,000, you may have to pay up to [\\$500.¹¹](#) Yikes. The technology behind the last 40 years of banking? FTP + CSV files.

¹¹ <https://www.ofx.com/en-au/faqs/how-much-does-it-cost-to-send-money-internationally/>



“Fun” Facts 2/3

Contrary, cryptocurrencies and ERC20 tokens (representing assets) are tradeable 24/7 from your personal computer connected to a decentralized platform without any single-point-of-failure in the middle.

To be fair, cryptocurrencies and the growing banking competition force companies to innovate. In the recent years, EU implemented [Single Euro Payments Area¹²](#) (SEPA) in 36+ countries allowing real-time transfers of funds in seconds between banks. I use it regularly and a transfer from my German N26 to a Slovakia VUB takes 2 seconds. Impressive, but relatively easy to implement in a centralized infrastructure. Remember, at any moment the bank can freeze all of your funds and suspend your account. Unlikely? Maybe. Government forcing citizens to stay at home for 2 years was also unheard of before - food for thought.

¹² <https://www.ecb.europa.eu/paym/integration/retail/sepa/html/index.en.html>



“Fun” Facts 3/3

Do you think the stock market is fair? Banks, indexes, and stocks are highly centralized and controlled by governments and private Wall Street groups. Free market? Wall Street controls how much can prices jump/fall in a single day. Federal Reserve (FED) and European Central Bank (ECB) are both PRIVATE institutions controlling the interest rates for the entire West - and failing terribly, the middle class is getting taxed with 11% inflation without any democratic voting in the process.

As an example, Wall Street halted the trading of S&P 500 Index after a 7% drop to protect their investors and hedge funds from losing money as people started selling their stocks in March 2020 following the COVID news. Afterward, the FED printed trillions of dollars to support the stock prices. Are you a developer who likes to save money and avoid debt? - Your savings just lost value overnight.

The monetary policies are decided behind closed doors. Many countries are going into negative yields, an unexplored territory with unknown consequences. What does this mean? Soon you will have to pay the bank to keep your savings. Inflation at its best. You are being forced to spend/invest your capital to support a system you don't control.



Summary

Closed software with centralized access to private data and rules puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain is a database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file. The Genesis balances indicate what was the original blockchain state and are never updated afterwards.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.



Study Code

Commit: [00d6ed¹³](#)

¹³ <https://github.com/web3coach/the-blockchain-bar/commit/00d6ede25b1e54ceb30c0a0314ef99a612db01de>

05 | Why We Need Blockchain

>_ git checkout c5_broken_trust

BabaYaga Seeks Justice

Thursday, March 28.

BabaYaga enters the bar for her birthday.

BabaYaga: Hey, Andrej! Today is my birthday! Get me your most expensive bottle!

Andrej: Happy birthday! Here you go: Crystal Head Vodka. But you need to purchase one additional TBB token. The bottle costs 950 tokens, and your balance is 949.

BabaYaga: What?! My balance is supposed to be 999 TBB!

Andrej: The funds transfer to Caesar you requested last week cost you 50 tokens.

BabaYaga: This is unacceptable! I would never agree to such a high fee! You can't do this, Andrej! I trusted your system, but you are as unreliable as every other business owner. Things must change!

Andrej: All right, look. You are my most loyal customer, and I didn't want to charge you, but my shareholders forced me. **Let me re-program my system and make it completely transparent and decentralized.** After all, if everyone were able to interact with the bar without going through me, it would significantly improve the bar's efficiency and balance the level of trust!

- Ordering drinks would take seconds instead of minutes
- The customers who forgot their wallets at home could borrow or lend tokens to each other
- I wouldn't have to worry about losing the clients data (again) as everyone would have a copy of it
- **The database would be immutable, so once everyone would agree on a specific state, no one else can change it or maliciously modify the history.** Immutability would help with yearly tax audits as well!
- If shareholders wanted to introduce new fees or raise the current ones, everyone involved in the blockchain system would notice and have to agree with it. The users and business owners would even have to engage in some decentralized governance system together, based on voting, probably. In case of a disagreement, the users walk away with all their data!

BabaYaga: Well, it certainly sounds good, but is this even possible?

Andrej: Yes, I think so. With a bit of **hashing, linked lists, immutable data structure, distributed replication, and asymmetric cryptography!**

BabaYaga: I have no idea what you have just said but go and do your geeky thing, Andrej!

Another day of the system running, another 100 TBB tokens for Andrej for maintaining and improving the blockchain:

```
>_ tbb tx add --from=andrej --to=andrej --value=100  
--data=reward
```



Fun Facts

Bitcoin and Ethereum miners also receive rewards every ~15 minutes for running the blockchain servers (nodes) and validating transactions.

Every 15 minutes, one Bitcoin miner receives 12.5 BTC (\$100k) to cover their servers cost + make some profit.

The Bitcoin network consumes as much electricity as the entire country of Austria. It accounts for 0.29% of the world's annual electricity consumption. Annually it consumes 76.84 TWh, producing 36.50 Mt CO₂ carbon footprint (New Zealand). [Source.¹⁴](#) Why? You will learn more in Chapter 11, where you will program a Bitcoin mining algorithm from scratch! Don't worry, our simplified Proof of Work algorithm in Chapter 11 will consume a bit less electricity.

¹⁴<https://digiconomist.net/bitcoin-energy-consumption>



Summary

Closed software with centralized access to private data allows for just a handful of people to have a lot of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relationship. Specifications of the blockchain system should be well-defined from the beginning and only change if its users support it.

Blockchain is a database. The token supply, initial user balances, and global blockchain settings are defined in a Genesis file. The Genesis balances indicate what was the original blockchain state and are never updated afterwards.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.



Study Code

Commit: [642045¹⁵](https://github.com/web3coach/the-blockchain-bar/commit/64204512f2173eb3f3e136e7e2674a2c456d351f)

¹⁵ <https://github.com/web3coach/the-blockchain-bar/commit/64204512f2173eb3f3e136e7e2674a2c456d351f>

06 | L'Hash de Immutable

>_ git checkout c6_immutable_hash

The book's technical difficulty starts with this chapter! The concepts will only get more challenging, but at the same time, very exciting. Buckle up.

How to Program an Immutable Database?

Friday, March 29.

If Andrej wants to figure out how to program an immutable DB, he has to understand why other database systems are mutable by design.

He decides to analyze an all-mighty MySQL DB Table:

id	name	balance
1	Andrej	998951
2	BabaYaga	949
3	Caesar	1000

In MySQL DB, anyone with access and a good enough reason can perform a table update such as:

```
UPDATE user_balance SET balance = balance + 100 WHERE id > 1
```

Updating values across different rows is possible because the table rows are independent, mutable, and the latest state is not apparent. What's the latest DB change? Last column changed? Last row inserted? If so, how can Andrej know what row was deleted recently? If the rows and table state were tightly coupled, dependent, a.k.a, updating row 1 would generate a completely new, different table, Andrej would achieve his immutability.

How to tell if any byte in a database has changed?

Immutability via Hash Functions

Saturday, March 30.

Hashing is process of taking a string input of arbitrary length and producing a hash string of fixed length. Any change in input, will result in a new, different hash.

```
package main

import (
    "crypto/sha256"
    "fmt"
)

func main() {
    balancesHash := sha256.Sum256([]byte("1 | Andrej | 99895 |"))
    fmt.Printf("%x\n", balancesHash)
    // Output: 6a04bd8e2...f70a3902374f21e089ae7cc3b200751

    // Change balance from 99895 -> 99896

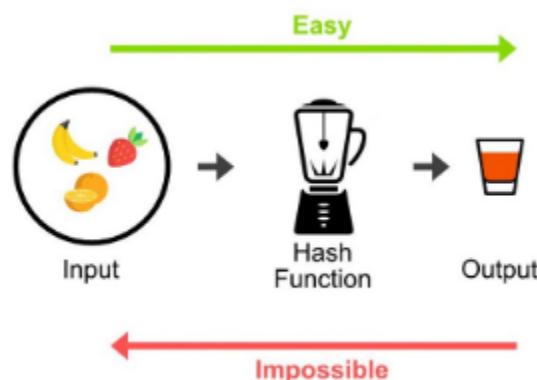
    balancesHashDiff := sha256.Sum256([]byte("1 | Andrej | 99896 |"))
    fmt.Printf("%x\n", balancesHashDiff)
    // Output: d04279207...ec6d280f6c7b3e2285758030292d5e1
}
```

Try it: <https://play.golang.org/p/FTPUa7lhOCE>¹⁶

¹⁶ <https://play.golang.org/p/FTPUa7lhOCE>

Andrej also requires some level of security for his database, so he decides for a **Cryptographic Hash Function** with the following properties:

- it is **deterministic**¹⁷ - the same message always results in the same hash
- it is quick to compute the hash value for any given message
- it is infeasible to generate a message from its hash value except by trying all possible messages
- a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
- it is **infeasible**¹⁸ to find two different messages with the same hash value



Hashing Fruits Example

img src¹⁹

¹⁷ https://en.wikipedia.org/wiki/Deterministic_algorithm

¹⁸ https://en.wikipedia.org/wiki/Computational_complexity_theory#Intractability

¹⁹ <https://twitter.com/cybergibbons/status/1203291585473110016>

Implementing the DB Content Hashing

Saturday Evening, March 30.

Andrej modifies the Persist() function to return a new content hash, Snapshot, every time a new transaction is persisted.

```
type Snapshot [32]byte
```

The Snapshot is produced by this new sha256 secure hashing function:

```
func (s *State) doSnapshot() error {
    // Re-read the whole file from the first byte
    _, err := s.dbFile.Seek(0, 0)
    if err != nil {
        return err
    }

    txsData, err := ioutil.ReadAll(s.dbFile)
    if err != nil {
        return err
    }
    s.snapshot = sha256.Sum256(txsData)

    return nil
}
```

The `doSnapshot()` is called by the modified `Persist()` function. When a new transaction is written into the `tx.db` file, the `Persist()` hashes the entire file content and returns its 32 bytes “fingerprint”, `hash`.

```
func (s *State) Persist() (Snapshot, error) {
    mempool := make([]Tx, len(s.txMempool))
    copy(mempool, s.txMempool)

    for i := 0; i < len(mempool); i++ {
        txJson, err := json.Marshal(mempool[i])
        if err != nil {
            return Snapshot{}, err
        }

        fmt.Printf("Persisting new TX to disk:\n")
        fmt.Printf("\t%s\n", txJson)
        if _, err = s.dbFile.Write(append(txJson, '\n')); err != nil {
            return Snapshot{}, err
        }
    }

    err = s.doSnapshot()
    if err != nil {
        return Snapshot{}, err
    }
    fmt.Printf("New DB Snapshot: %x\n", s.snapshot)

    s.txMempool = s.txMempool[1:]
}

return s.snapshot, nil
}
```

From this moment, everyone can 100% confidently and securely refer to any particular database state (set of data) with a specific, snapshot hash.

 **Practice time.**

1/4 Run the tbb balances list cmd and check the balances are matching.

 tbb balances list

Account balances at 7d4a360f465d...

id	name	balance
1	Andrej	999251
2	BabaYaga	949
3	Caesar	1000

2/4 Remove the last 2 rows from ./database/tx.db and check the balances again.

>_ tbb balances list

Account balances at 841770dcd3...

id	name	balance
1	Andrej	999051
2	BabaYaga	949
3	Caesar	1000

3/4 Reward Andrej for last 2 days (from 28th to 30th of March):**Reward Transaction 1:**

```
>_ tbb tx add --from=andrej --to=andrej --value=100  
--data=reward
```

Persisting new TX to disk:

```
{"from": "andrej", "to": "andrej", "value": 100, "data": "reward"}
```

New DB Snapshot: ff2470c7043f5a34169b5dd38921ba6825b03b3facb83e426

TX successfully persisted to the ledger.

Reward Transaction 2:

```
>_ tbb tx add --from=andrej --to=andrej --value=100  
--data=reward
```

Persisting new TX to disk:

```
{"from": "andrej", "to": "andrej", "value": 100, "data": "reward"}
```

New DB Snapshot: 7d4a360f468b837b662816bcd52c1869f99327d53ab4a9ca

TX successfully persisted to the ledger.

4/4 Run the tbb balances list cmd and ensure the balances and the snapshot hash is the same as at the beginning.

>_ tbb balances list

Account balances at 7d4a360f465d...

id	name	balance
1	Andrej	999251
2	BabaYaga	949
3	Caesar	1000

Done!

Because the cryptographic hash function **sha256**, given the same inputs (current tx.db and 2x tbb tx add), produces the same output, if you follow the exact steps on your own computer, you will generate the exact same database state and hashes!



Summary

Closed software with centralized access to private data puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relation. Specifications of the blockchain system should be well defined from the beginning and only change if its users support it.

Blockchain is an **immutable** database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file. The Genesis balances indicate what was the original blockchain state and are never updated afterwards.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.

The database content is hashed by a secure cryptographic hash function. The blockchain participants use the resulted hash to reference a specific database state.



Study Code

Andrej's solution is beyond inefficient, but I don't want to scare you with low-level disk and state optimization in Chapter 6. Step by step.

Ethereum uses sophisticated Patricia Merkle Trie - a cryptographically authenticated data structure that can be used to store all (key, value) bindings. **Patricia Merkle Tries are fully deterministic, meaning that a trie with the same (key, value) bindings is guaranteed to be identical—down to the last byte. This means they have the same root hash, providing the holy grail of O(log(n)) efficiency for inserts, lookups and deletes.**

The eager students can dive into the complexity by reading the following two resources:

- <https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/>²⁰
- <https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/>²¹

Andrej's commit: [b99e51](#)²²

²⁰ <https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/>

²¹ <https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/>

²² <https://github.com/web3coach/the-blockchain-bar/commit/b99e5191b19bc076b98a3869289e4788d0a4a77b>

07 | The Blockchain Programming Model

>_ git checkout c7_blockchain_programming_model

Improving Performance of an Immutable DB

Sunday, March 31.

Andrej is building not only an immutable database but also a distributed one! He will distribute, replicate all data to every client/stakeholder's computer interacting with the bar **to avoid data centralization** and one, easy to attack, source of truth. This represents a performance challenge.

The current TBB program works fine with a small database size while running on one computer but it suffers two performance bottlenecks:

- Distributing transactions to other computers one by one will be inefficient due to **network latency in distributed systems**
- Creating a snapshot hash of the database and validating it requires to **read and hash the full, potentially several gigabytes large DB from scratch for EVERY NEW TX**

Fortunately, Andrej can solve both issues by implementing a Hashed Linked List in a combination with a standard [Batch Strategy²³](#).

Batching is a common strategy when working with SQL/NoSQL/Other database systems. The batch strategy consist of “**handling multiple items at once**”. The solution is to encapsulate transactions to **linked** “chunks”, “**blocks**”.

²³ https://en.wikipedia.org/wiki/Batch_processing

Batch + Hash + Linked List \Rightarrow Blocks

Monday, April 1.

Andrej is not joking (see date). He designs the following data structure for encapsulating transactions into batches:

```
type Hash [32]byte

type Block struct {
    Header BlockHeader
    TXs    []Tx // new transactions only (payload)
}

type BlockHeader struct {
    ParentHash // parent block reference
    Time      uint64
}
```

And he renames Snapshot struct to Hash.

Block struct will have 2 attributes, **Header** and **Payload**:

- Payload stores **NEW transactions**
- Header stores block's metadata (**PARENT BLOCK HASH REFERENCE**, time)

Legacy, slow hashing of the entire DB after every new TX:

```
func (s *State) doSnapshot() error {
    // Re-read the whole file from the first byte
    _, err := s.dbFile.Seek(0, 0)
    if err != nil {
        return err
    }

    txsData, err := ioutil.ReadAll(s.dbFile)
    if err != nil {
        return err
    }

    // Give me the hash of the entire DB content
    s.snapshot = sha256.Sum256(txsData)

    return nil
}
```

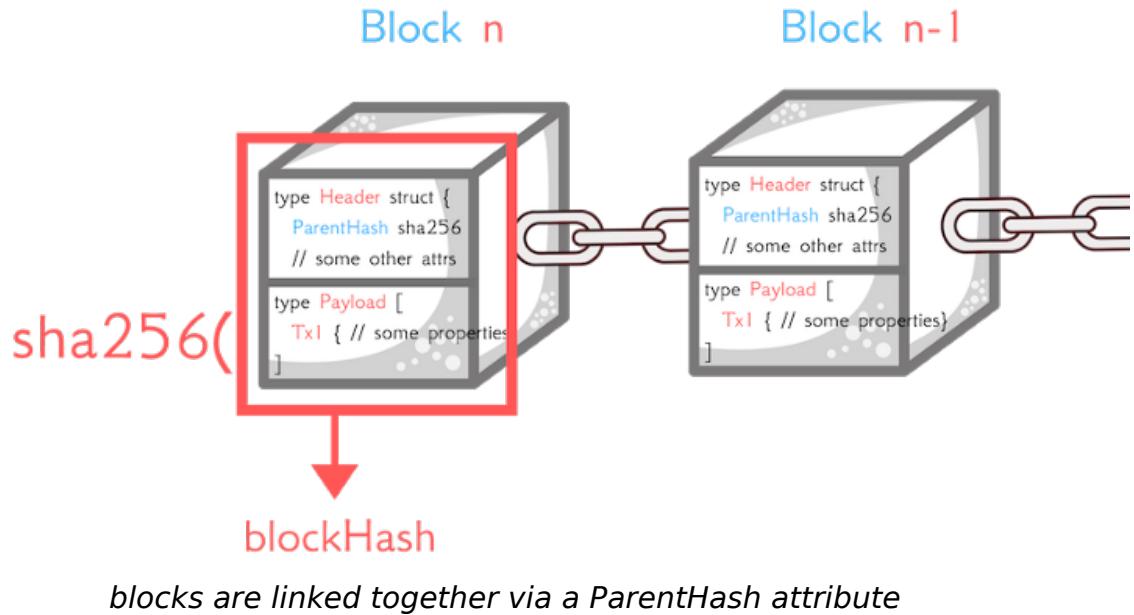
New optimized hashing - where the 32bytes Hash is calculated by only hashing the latest Block encoded as JSON:

```
type Block struct {
    Header BlockHeader // metadata (parent block hash + time)
    TXs     []Tx       // new transactions only (payload)
}

func (b Block) Hash()(Hash, error) {
    blockJson, err := json.Marshal(b)

    return sha256.Sum256(blockJson), nil
}
```

The above implementation can be visually represented as, the **blockchain programming model**:



Why is the parent block's hash reference necessary?

The ParentHash is being used as a reliable “checkpoint,” representing and referencing the previously hashed database content.

ParentHash improves performance. **Only new data + reference to previous state** needs to be hashed to achieve **immutability**.

E.g., If you attempt to modify a TX value in Block 0, it will result in a new unique Block 0 hash. Hash of Block 1, based on the parent Block 0 reference, would therefore immediately change as well. The cascade effect would affect all the blocks, making the malicious

attacker database invalid - different from the rest of the honest database stakeholders.

The attacker database would be, therefore, excluded from participating in the network. Why? You will find out in Chapter 10 - where you will program a peer-to-peer sync algorithm.

PS: If you are curious in history and in-depth theory of a Linked List data structure, I recommend to checkout this great [Wikipedia explanation](#).²⁴

²⁴https://en.wikipedia.org/wiki/Linked_list

How adding a TX into a Block works

Monday Evening, April 1.

A new TX struct is constructed from cmd params:

```
func txAddCmd() *cobra.Command {
    var cmd = &cobra.Command{
        Use:   "add",
        Short: "Adds new TX to database.",
        Run: func(cmd *cobra.Command, args []string) {
            from, _ := cmd.Flags().GetString(flagFrom)
            to, _ := cmd.Flags().GetString(flagTo)
            value, _ := cmd.Flags().GetUint(flagValue)
            data, _ := cmd.Flags().GetString(flagData)

            tx := database.NewTx(
                database.NewAccount(from),
                database.NewAccount(to),
                value,
                data,
            )
        },
    }
}
```

The TX is then stored in the **State's mempool** (a single TX, or multiple, as necessary):

```
state, err := database.NewStateFromDisk()

state.AddTx(tx)
```

Finally, the state.Persist() function to flush the mempool transactions into the ./database/block.db database file is executed:

```
state.Persist()
```

Internally, the Persist() function will create a new Block, encode it into a JSON and write it to the DB.

```
func (s *State) Persist() (Hash, error) {
    // Create a new Block with ONLY the new TXs
    block := NewBlock(
        s.latestBlockHash,
        uint64(time.Now().Unix()),
        s.txMempool,
    )

    blockHash, err := block.Hash()
    if err != nil {
        return Hash{}, err
    }

    blockFs := BlockFS{blockHash, block}

    // Encode it into a JSON string
    blockFsJson, err := json.Marshal(blockFs)
    if err != nil {
        return Hash{}, err
    }

    fmt.Printf("Persisting new Block to disk:\n")
    fmt.Printf("\t%s\n", blockFsJson)

    // Write it to the DB file on a new line
    err = s.dbFile.Write(append(blockFsJson, '\n'))
    if err != nil {
        return Hash{}, err
    }
    s.latestBlockHash = blockHash
```

```
// Reset the mempool
s.txMempool = []Tx{ }

return s.latestBlockHash, nil
}
```

Migrating from TX.db to BLOCKS.db

Monday Evening, April 1.

At the moment, all the transactions are written in the `./database/tx.db` file. To migrate them into blocks, Andrej develops a simple helper command.

He places the cmd into a `./cmd/tbbmigrate/main.go` file:

```
func main() {
    state, err := database.NewStateFromDisk()
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    defer state.Close()
```

He encapsulates the first 2 bar's TXs into a Block 0 and persists the block to disk:

```
block0 := database.NewBlock(  
    database.Hash{},  
    uint64(time.Now().Unix()),  
    []database.Tx{  
        database.NewTx("andrey", "andrey", 3, ""),  
        database.NewTx("andrey", "andrey", 700, "reward"),  
    },  
)  
  
state.AddBlock(block0)  
block0hash, _ := state.Persist()  
// Block 0 Hash: 07536e2c559b0a4566b84802...  
// Block 0 Parent Hash: 0000000...
```

Result written to disk:

The parent hash is empty because it's the first Block ever created. Andrej inserts the rest of the TX history into a Block 1. Block 1 references Block 0 as its parent by only using the Block 0' hash. It's not necessary to re-hash the entire Block 0 content! Much less CPU cycles wasted, the same level of security and immutability achieved.

```
block1 := database.NewBlock(
    block0hash,
    uint64(time.Now().Unix()),
    []database.Tx{
        database.NewTx("andrey", "babayaga", 2000, ""),
        database.NewTx("andrey", "andrey", 100, "reward"),
        database.NewTx("babayaga", "andrey", 1, ""),
        database.NewTx("babayaga", "caesar", 1000, ""),
        database.NewTx("babayaga", "andrey", 50, ""),
        database.NewTx("andrey", "andrey", 600, "reward"),
    },
)
state.AddBlock(block1)
state.Persist()
// Block 1 Hash: 2efe28463821660834cf8e3cc555...
// Block 1 Parent Hash: 07536e2c559b0a4566b84802...
```

Disk:

```
{  
  "hash": "2efe284638555aeb5aecd911b7b8653add975d0ad3f3ba0c5",  
  "block": {  
    "header": {  
      "parent": "07536e2c55ffa629a105f61c8f6c5f1c7289d139e02639436",  
      "time": 1556563065  
    },  
    "payload": [  
      {  
        "from": "andrey",  
        "to": "babayaga",  
        "value": 2000,  
        "data": ""  
      },  
      {  
        "from": "andrey",  
        "to": "andrey",  
        "value": 100,  
        "data": "reward"  
      },  
      ...  
      ...  
    ]  
  }  
}
```

Experiment with the migration cmd

 Practice time.

1/3 Remove the existing 2 blocks from ./database/block.db that Andrej committed in the current branch.

>_ cat /dev/null > ./database/block.db

2/3 Compile the code and run the new migration command.

>_ go install ./cmd/...
tbbmigrate

3/3 Observe 2 new JSON encoded blocks being written into the ./database/block.db as programmed in ./cmd/tbbmigrate/main.go.

Persisting new Block to disk:

```
{  
  "hash": "8a05167d2f...63196c740",  
  "block": {  
    "header": {  
      "parent": "0000000000...000000000",  
      "time": 1587486395  
    },  
    "payload": [  
      {  
        "from": "andrey",  
        "to": "andrey",  
        "value": 3,  
        "data": ""  
      },  
      {  
        "from": "andrey",  
        "to": "andrey",  
        "value": 700,  
        "data": "reward"  
      }  
    ]  
  }  
}
```

Persisting new Block to disk:

```
{  
  "hash": "c70775ae5e...b6a6184",  
  "block": {  
    "header": {  
      "parent": "8a05167d2...196c740",  
      "time": 1587486395  
    }  
  }  
}
```

```
},
"payload": [
{
  "from": "andrey",
  "to": "babayaga",
  "value": 2000,
  "data": ""
},
{
  "from": "andrey",
  "to": "andrey",
  "value": 100,
  "data": "reward"
},
{
  "from": "babayaga",
  "to": "andrey",
  "value": 1,
  "data": ""
},
{
  "from": "babayaga",
  "to": "caesar",
  "value": 1000,
  "data": ""
},
{
  "from": "babayaga",
  "to": "andrey",
  "value": 50,
  "data": ""
},
{
  "from": "andrey",
  "to": "andrey",
  "value": 600,
```

```
        "data": "reward"
    }
]
}
}
```

PS: Your block hashes will be different because the block.time attribute is set on runtime - when you run the tbbmigrate cmd.



Summary

Closed software with centralized access to private data puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relation. Specifications of the blockchain system should be well defined from the beginning and only change if its users support it.

Blockchain is an **immutable** database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file. The Genesis balances indicate what was the original blockchain state and are never updated afterwards.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.

The database content is hashed by a secure cryptographic hash function. The blockchain participants use the resulted hash to reference a specific database state.

Transactions are grouped into batches for performance reasons. A batch of transactions make a Block. Each block is encoded and hashed using a secure, cryptographic hash function.

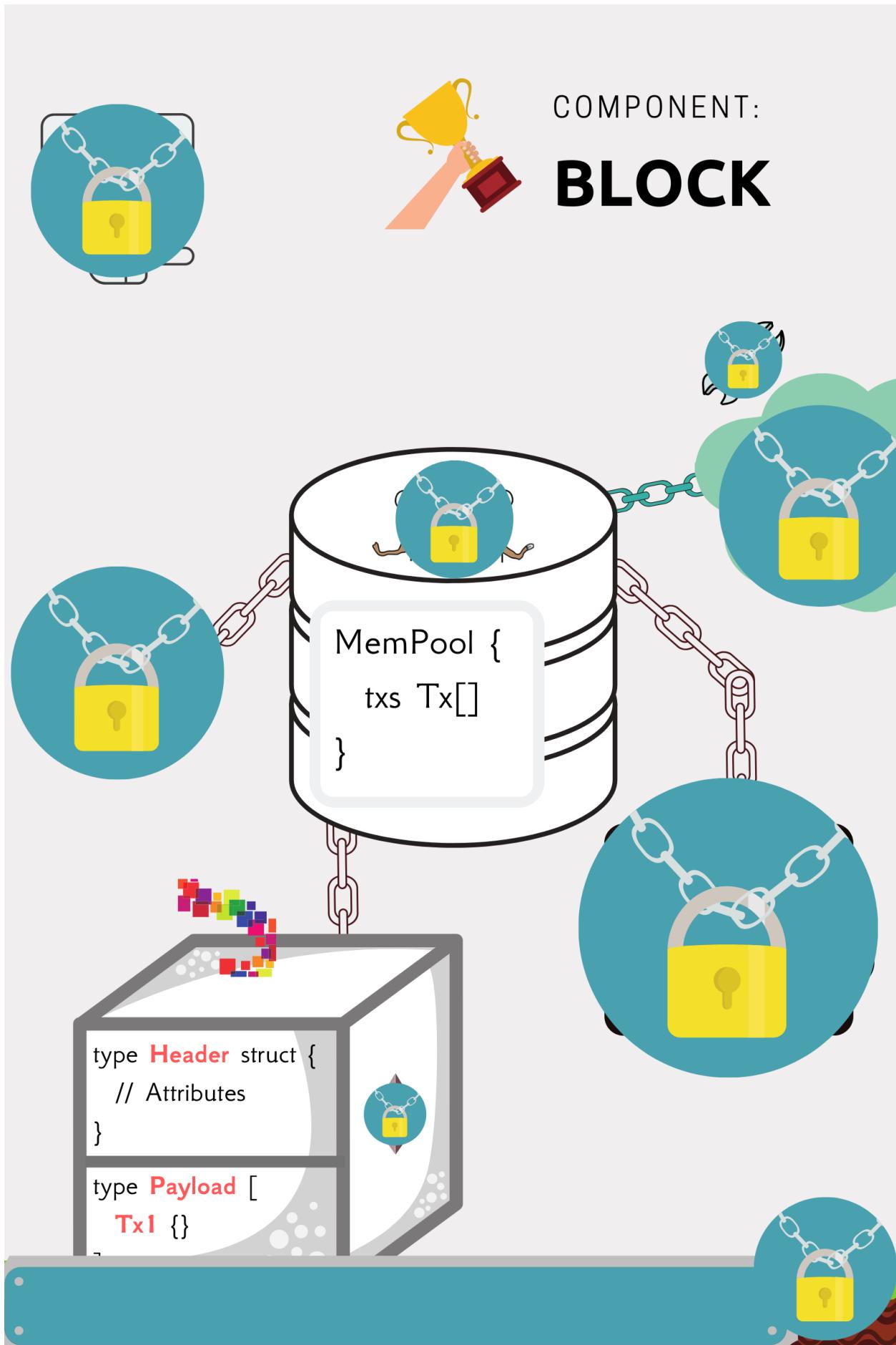
Block contains Header and Payload. The Header stores various metadata such a time and a reference to the Parent Block (the previous immutable database state). The Payload carries the new database transactions.



Study Code

Commit: [239534²⁵](#)

²⁵ <https://github.com/web3coach/the-blockchain-bar/commit/239534f5eb73480959e65557b8d9d5d14341f38b>



08 | Transparent Database

>_ git checkout c8_transparent_db

Flexible DB Directory

Tuesday, April 2.

The next step for Andrej is to make the DB fully accessible to all customers. To keep things simple, he decides to program a standard HTTP API with few core endpoints for reading customer balances and managing transactions, and to deploy its program to a dedicated server running 24/7. Don't worry, he will secure the API with a SSL certificate later, as well as migrate the HTTP API to a more powerful gRPC version (remote procedure call system) in future chapters, ye!

But to do so, he must make the application more configurable, especially where the data should be stored.

He introduces a new data directory flag required by all CLI commands.

```
func addDefaultRequiredFlags(cmd *cobra.Command) {
    cmd.Flags().String(
        flagDataDir,
        "",
        "Absolute path where all data will/is stored",
    )
    cmd.MarkFlagRequired(flagDataDir)
}
```

Andrej will need a new file `fs.go` inside the database pkg where he groups together all the file-system related business logic for creating a new `datadir` to store inside the database files.

The `fs.go` has 3 helper functions to determine the DB paths:

```
func getDatabaseDirPath(dataDir string) string {
    return filepath.Join(dataDir, "database")
}

func getGenesisJsonFilePath(dataDir string) string {
    return filepath.Join(getDatabaseDirPath(dataDir), "genesis.json")
}

func getBlocksDbFilePath(dataDir string) string {
    return filepath.Join(getDatabaseDirPath(dataDir), "block.db")
}
```

The first ever `tbb run` program execution should also create a fresh new database with an empty `blocks.db` and a default `genesis.json`.

```
func initDataDirIfNotExists(dataDir string) error {
    if fileExist(getGenesisJsonFilePath(dataDir)) {
        return nil
    }

    dbDir := getDatabaseDirPath(dataDir)
    if err := os.MkdirAll(dbDir, os.ModePerm); err != nil {
        return err
    }

    gen := getGenesisJsonFilePath(dataDir)
    if err := writeGenesisToDisk(gen); err != nil {
        return err
    }

    blocks := getBlocksDbFilePath(dataDir)
    if err := writeEmptyBlocksDbToDisk(blocks); err != nil {
        return err
    }

    return nil
}
```

 **Practice time.**

1/3 Initialize a new blockchain dir in a path of your choice. The convention is to use a “hidden” directory in your \$HOME path.

>_ tbb balances list --datadir=/home/web3coach/.tbb

Accounts balances at 0000000000...0000000:

andrej: 1000000

2/3 Explore the directory and ensure the blocks.db and genesis.json are present.

>_ ls -la /home/web3coach/.tbb/database

```
drwxrwxr-x 2 .
drwxrwxr-x 3 ..
-rwxrwxr-x 1 block.db
-rw-r--r-- 1 genesis.json
```

>_ vim /home/web3coach/.tbb/database/genesis.json

```
{  
  "genesis_time": "2019-03-18T00:00:00.000000000Z",  
  "chain_id": "the-blockchain-bar-ledger",  
  "balances": {  
    "andrey": 1000000  
  }  
}  
new datadir genesis
```

3/3 Copy the blocks.db from previous chapter into the new dir.

```
>_ export TBBS="$GOPATH/src/github.com/web3coach/the-blockchain-bar"  
      export TBB="/home/web3coach/.tbb"  
  
      cp $TBBS/database/block.db $TBB/database/
```

```
>_ tbb balances list --datadir=/home/web3coach/.tbb
```

Accounts balances at 2efe284638...ad3f3ba0c5:

babayaga: 949

caesar: 1000

andrej: 999451

Done!

The DB data directory is now configurable, but Andrej is missing another essential part, a public interface to communicate with other blockchain peers.



Study Code

Commit: [c6be95²⁶](https://github.com/web3coach/the-blockchain-bar/commit/c6be95ae7bb3da1bd947e92e41cba4cb23af5b4) | Full source code: [v0.5.0²⁷](https://github.com/web3coach/the-blockchain-bar/tree/0.5.0)

²⁶ <https://github.com/web3coach/the-blockchain-bar/commit/c6be95ae7bb3da1bd947e92e41cba4cb23af5b4>

²⁷ <https://github.com/web3coach/the-blockchain-bar/tree/0.5.0>

Centralized Public HTTP API

Wednesday, April 3.

Andrej needs to convert his CLI program into a HTTP API so he can communicate with other peers and to make his DB public and transparent.

He starts by removing the tx.go command because the HTTP Server will be a long running process and will be responsible for maintaining the State. He doesn't want to be updating the DB from 2 different input sources and run into collisions and unpredictable behaviour.

Programming an HTTP server in Go is very easy. You don't need any Apache or Nginx.

Andrej programs a new tbb run cmd:

```
package main

func runCmd() *cobra.Command {
    var runCmd = &cobra.Command{
        Use:   "run",
        Short: "Launches the TBB node and its HTTP API.",
        Run: func(cmd *cobra.Command, args []string) {
            dataDir, _ := cmd.Flags().GetString(flagDataDir)

            fmt.Println("Launching TBB node and its HTTP API...")

            err := node.Run(dataDir)
            if err != nil {
                fmt.Println(err)
                os.Exit(1)
            }
        },
    }

    addDefaultRequiredFlags(runCmd)

    return runCmd
}
```

He places the HTTP API code inside a new node/node.go package'.

Why a node pkg? In the blockchain world, the terminology “client” and “server” are avoided in favor of the “node” concept. Every peer (computer) is a server (backend), a completely valid, independent database.

The Run() method loads the state from disk and starts listening on a hardcoded (for now) port 8080.

```
const httpPort = 8080

func Run(dataDir string) error {
    state, err := database.NewStateFromDisk(dataDir)
    if err != nil {
        return err
    }
    defer state.Close()

    http.ListenAndServe(fmt.Sprintf(":%d", httpPort), nil)
}
```

This is all you really have to do to launch a basic HTTP server in Go. Impressive right?

Now that he has the server, he attaches 2 new HTTP endpoints to it.

A `http://localhost:8080/balances/list` HTTP GET route for querying account balances:

```
http.HandleFunc("/balances/list", listBalancesHandler)
func listBalancesHandler(w ResponseWriter, r Req, state *state) {
    writeRes(w, BalancesRes{state.LatestBlockHash(), state.Balances})
}
```

Returning a JSON encoded struct with the latest block hash and all bar customers balances:

```
type BalancesRes struct {
    Hash      database.Hash           `json:"block_hash"`
    Balances map[database.Account]uint `json:"balances"`
}
```

And a `http://localhost:8080/tx/add` HTTP POST route for recording new transactions on the ledger:

```
type TxAddReq struct {
    From  string `json:"from"`
    To    string `json:"to"`
    Value uint   `json:"value"`
    Data  string `json:"data"`
}

http.HandleFunc("/tx/add", txAddHandler)
func txAddHandler(w ResWriter, r Req, state *database.state) {
    req := TxAddReq{}

    // Parse the POST request body
    err := readReq(r, &req)
    if err != nil {
        writeErrRes(w, err)
        return
    }

    tx := database.NewTx(
        database.NewAccount(req.From),
        database.NewAccount(req.To),
        req.Value,
        req.Data,
    )

    // Exactly like from the CLI command.
    // Add a new TX into the Mempool
```

```
err = state.AddTx(tx)
if err != nil {
    writeErrRes(w, err)
    return
}

// Flush the Mempool TX to the disk
hash, err := state.Persist()
if err != nil {
    writeErrRes(w, err)
    return
}

writeRes(w, TxAddRes{hash})
}
```

The HTTP support feature added in few minutes using just **one node.go file!!!**

Andrej must love Go 😊

 Practice time.

1/3 Boot your new, shinny blockchain node!

>_ tbb run --datadir=\$HOME/.tbb

```
Launching TBB node and its HTTP API...
Listening on: 127.0.0.1:8080
```

2/3 Query the customers balances.

>_ curl -X GET http://localhost:8080/balances/list

```
{
  "block_hash": "2efe284638...d3f3ba0c5",
  "balances": {
    "andrej": 999451,
    "babayaga": 949,
    "caesar": 1000
  }
}
```

3/3 Add a new TX via cURL, or a Postman.com with a nice GUI.

>_ \$

```
curl --location --request POST 'http://localhost:8080/tx/add' \
--header 'Content-Type: application/json' \
--data-raw '{
    "from": "andrey",
    "to": "babayaga",
    "value": 100
}'
```

> {"block_hash": "9efa5f844...7e20a187"}

Perfect. The blockchain still works like a charm.

Deploying TBB Program to AWS

Thursday, April 4.

Andrej can now deploy the program to a dedicated server where a **Supervisor** service will manage the program to keep it running nonstop.

He uploads his local database and the compiled binary to a AWS server:

```
ssh tbb
mkdir -p /home/ec2-user/.tbb/database
exit

scp -i ~/.ssh/tbb_aws.pem $TBB/database/genesis.json ec2-user@ec2-3-12\
7-248-10.eu-central-1.compute.amazonaws.com:/home/ec2-user/.tbb/database\
se/

scp -i ~/.ssh/tbb_aws.pem $TBB/database/block.db ec2-user@ec2-3-127-24\
8-10.eu-central-1.compute.amazonaws.com:/home/ec2-user/.tbb/database/\
se

scp -i ~/.ssh/tbb_aws.pem $GOPATH/bin/tbb ec2-user@ec2-3-127-248-10.eu\
-central-1.compute.amazonaws.com:/home/ec2-user/\
se

ssh tbb
sudo ln -s /home/ec2-user/tbb /usr/local/bin/tbb

tbb version
> Version: 0.6.1-beta HTTP API
```

Installs and sets up the Supervisor process ([full instructions](#))²⁸:

```
sudo amazon-linux-extras install epel
sudo yum install -y supervisor

sudo vim /etc/supervisord.d/tbb.conf
[program:tbb]
command=/usr/local/bin/tbb run --datadir=/home/ec2-user/.tbb
autostart=true
autorestart=true
stderr_logfile=/home/ec2-user/.tbb/tbb.err.log
stdout_logfile=/home/ec2-user/.tbb/tbb.out.log
:wq
// he even figures how to save and exit Vim after several months :)
```

```
sudo vim /etc/supervisord.conf
// Configure loading of custom config files
[include]
files = supervisord.d/*.conf
```

sudo service supervisord start

```
cat /home/ec2-user/.tbb/tbb.out.log
> Launching TBB node and its HTTP API...
> Listening on HTTP port: 8080
```

²⁸<https://tn710617.github.io/supervisor/>

Opens the HTTP port in AWS network settings:

The screenshot shows the AWS EC2 console with the 'Security Groups' section selected. On the left, there's a sidebar with various services like Reserved Instances, Dedicated Hosts, Capacity Reservations, IMAGES, AMIs, Bundle Tasks, ELASTIC BLOCK STORE, Volumes, Snapshots, Lifecycle Manager, NETWORK & SECURITY, Security Groups (which is highlighted), Elastic IPs, Placement Groups, Key Pairs, Network Interfaces, LOAD BALANCING, Load Balancers, Target Groups, and AUTO SCALING.

The main area displays a table of security groups:

Name	Group ID	Group Name	VPC ID	Owner	Description
sg-0737eab2002bd3a	launch-wizard-1	launch-wizard-1	vpc-cf0c50e5	528128798615	launch-wizard-1 created 2019-12-07T11:45:20Z
sg-e7503a82		default	vpc-cf0c50e5	528128798615	default VPC security group

Below the table, there's an 'Edit' button and another table showing inbound rules:

Type	Protocol	Port Range	Source	Description
Custom TCP Rule	TCP	8080	0.0.0.0/0	
Custom TCP Rule	TCP	8090	::/0	
SSH	TCP	22	0.0.0.0/0	

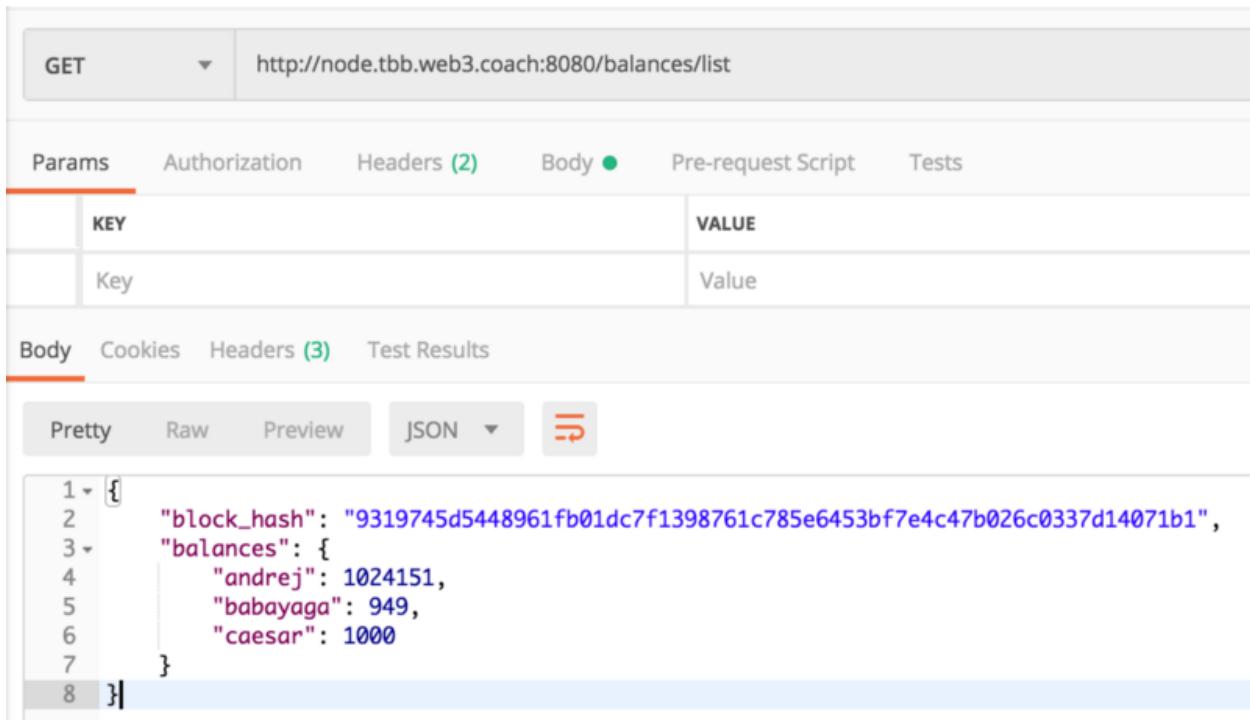
At the bottom, there are links for Feedback, English (US), and AWS terms, along with a note about copyright and privacy.

open AWS HTTP port

DONE. Andrej's blockchain is now publicly accessible, even though in a very centralized manner, for the time being!

Go ahead, dear reader, query the current blockchain state using cURL.

> curl -X GET http://node.tbb.web3.coach/balances/list



The screenshot shows the Postman application interface. A GET request is being made to `http://node.tbb.web3.coach:8080/balances/list`. In the 'Params' tab, there is one parameter named 'Key' with the value 'Value'. The 'Body' tab shows the response in JSON format:

```
1 {  
2   "block_hash": "9319745d5448961fb01dc7f1398761c785e6453bf7e4c47b026c0337d14071b1",  
3   "balances": {  
4     "andrey": 1024151,  
5     "babayaga": 949,  
6     "caesar": 1000  
7   }  
8 }
```

curl TBB balances

Burned-out

Friday, 6th of December 2019.

Andrej was trying to balance a 9-5 start-up life, teaching blockchain development in his free time and running a blockchain bar for his customers on weekends.

Surprise, surprise, he got burned out.

No such a thing as 10x developer, endless motivation, or 24/7 entrepreneur hustler.

We are animals with complicated emotions and the ability to reason. We are subject to the basic principles of nature and physics.

Take care of your mental and physical health, my friend!

Now, Andrej being back and stronger than ever and because blockchain never burns-out, he collects his substantial recovery bonus for the last 247 days.

```
>_ tbb tx add --from=andrej --to=andrej --value=24700  
--data=reward
```



Summary

Closed software with centralized access to private data puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relation. Specifications of the blockchain system should be well defined from the beginning and only change if its users support it.

Blockchain is an immutable, **transparent** database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file. The Genesis balances indicate what was the original blockchain state and are never updated afterwards.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.

The database content is hashed by a secure cryptographic hash function. The blockchain participants use the resulted hash to reference a specific database state.

Transactions are grouped into batches for performance reasons. A batch of transactions make a Block. Each block is encoded and hashed using a secure, cryptographic hash function.

Block contains Header and Payload. The Header stores various metadata such a time and a reference to the Parent Block (the previous immutable database state). The Payload carries the new database transactions.



Study Code

Commit: [4f89e4²⁹](https://github.com/web3coach/the-blockchain-bar/commit/4f89e45627b67d2ea3393e2949c8f76939033962) | Full source code: [v0.6.0³⁰](https://github.com/web3coach/the-blockchain-bar/releases/tag/v0.6.0)

²⁹ <https://github.com/web3coach/the-blockchain-bar/commit/4f89e45627b67d2ea3393e2949c8f76939033962>

³⁰ <https://github.com/web3coach/the-blockchain-bar/releases/tag/v0.6.0>

09 | It Takes Two Nodes To Tango

Distributing the DB to Customers

Saturday, December 7.

Andrej, excited about the new version of his immutable, transparent database, calls BabaYaga to share his progress, hoping to score some points after his betrayal.

Andrej: Hey BabaYaga, still mad? I have some significant progress to share!

BabaYaga: Depends. Did you manage to develop a transparent, immutable, and decentralized database for storing my assets in a way I can interact with the database personally without you in the middle, intercepting all the requests and charging me hidden fees?

Andrej: Almost done! (oh, the devs estimations). The immutable DB is ready. You can already query all the data and see the exact state of your balance at any moment using a centralized HTTP API. The next step is to program a **decentralized peer-to-peer sync algorithm** so you can have the same database as me but on your personal computer, **automatically and bidirectionally** updated

with every new bar activity. Vice versa, if you add new data into your database, your node database will automatically replicate the change to mine. Full transparency and decentralization.

Designing a Peer-to-Peer Sync Algorithm

Saturday Evening, December 7.

As Andrej, the currently only blockchain database node,
I want to share my database with everyone,
So I am not the only source of truth,
So there are many copies of my bar's database in the world,
So the bar will work (any client can order a vodka shot and pay for it,
transfer the funds between each other, etc.) even if I go on vacation
and turn off my node;

As Babayaga,

I want to get an automatically updated copy of Andrej's database,
So I can verify how much TBB tokens I have at any point,
So I can test the program's business logic and ensure there are no
hidden fees when I transfer my tokens to someone;

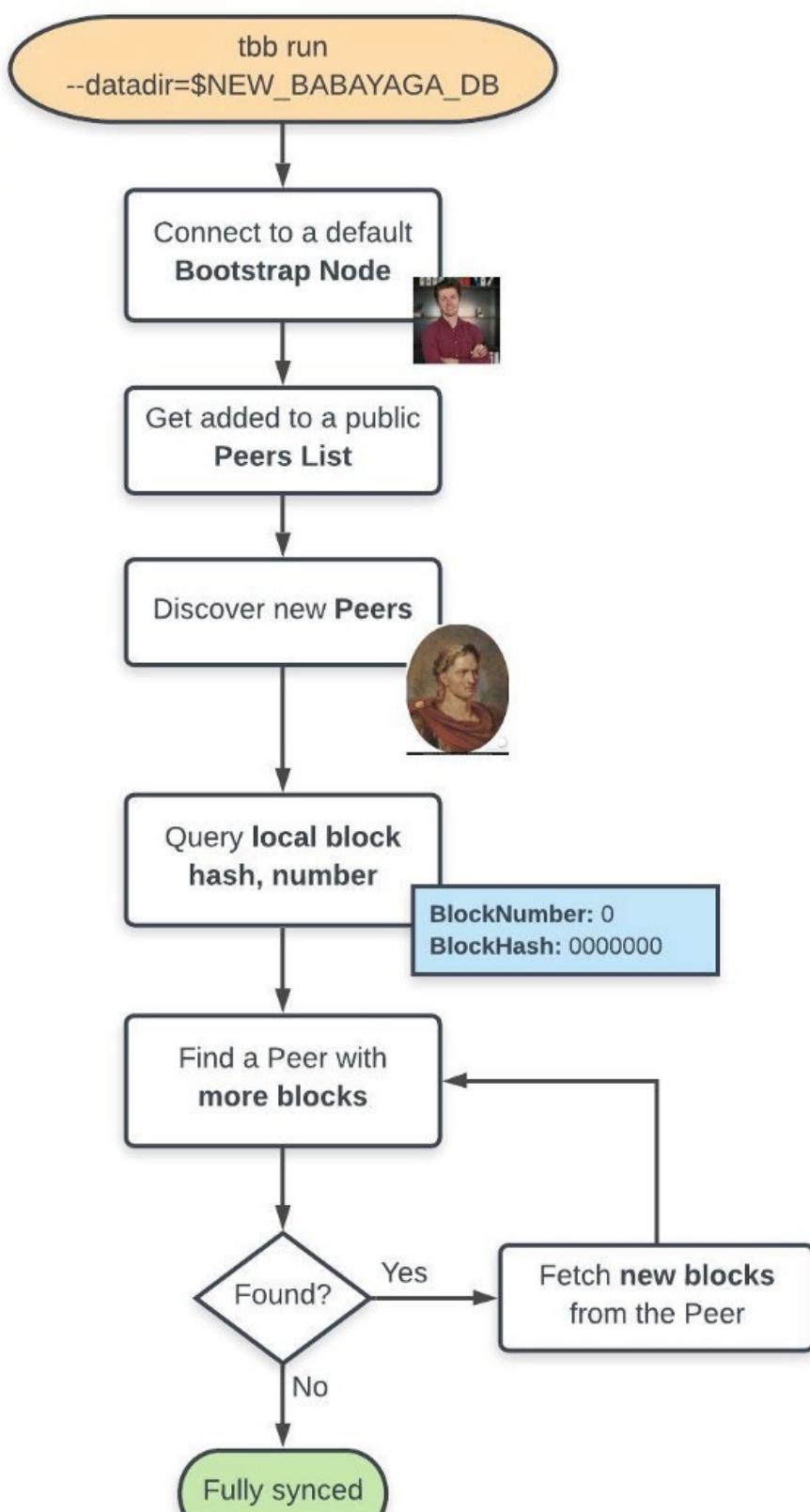
As Babayaga,

I also want to be "the master" database,
So I can update the database even if Andrej's node is offline;

As Julius Caesar,

I want to be able to bidirectionally synchronize my database with
anyone currently online,
So I have always an up-to-date overview of bar situation and
activity;

MVP Design:



Why is the Bootstrap Node necessary?

Because as there is no centralized database, without the bootstrap node, the network participants wouldn't know how to find each other. **The Bootstrap Node is used to initiate the peer discovery.**

E.g.:

1. Andrej is running an AndrejNode1 with current, existing DB
2. BabaYaga runs its BabaYagaNode2 with empty DB. BabaYaga needs to find other peers to sync its DB. How? By connecting to the default bootstrap node, AndrejNode1, BabaYaga will discover other network participants
3. If Caesar decides to run a CaesarNode3, he wouldn't know about BabaYagaNode2, but by connecting to a default bootstrap AndrejNode1, Caesar discovers the existence of the BabaYagaNode2 giving the possibility to synchronize its DB from either AndrejNode1 or BabaYagaNode2
4. If Daenerys running DaenerysNode4, would want to synchronize its DB, it would connect to AndrejNode1. AndrejNode1 would notify her about the existence of BabaYagaNode2 and CaesarNode3, giving the possibility to synchronize from anyone in the network

Blockchain synchronization in a nutshell!



Fun Facts

Bootstrap node solution sounds centralized at the beginning, but well, nothing is perfect + every node always has the possibility to overwrite or skip the default pre-configured bootstrap node.

Bitcoin, Ethereum, XRP ledger, IPFS, Torrent, all use the bootstrap node concept as well! Of course, with much higher level of sophistication, more optimized peer discovery, sync streaming process, data transport layer, etc. In Q3 and Q4 of 2020, I will most likely add new bonus chapters where I will analyse and document the different optimization techniques used within those protocols!

The **Node** component and the **peer discovery** are not unique to this program. Andrej, once again, chose the names and designs to match a simplified model of go-Ethereum ([peers³¹](#), [node³²](#)). Congrats, now you also have an idea of how Ethereum peer discovery works!

³¹<https://github.com/ethereum/go-ethereum/blob/v1.9.9/p2p/server.go#L104>

³²<https://github.com/ethereum/go-ethereum/blob/v1.9.9/p2p/server.go#L104>

10 | Programming a Peer-to-Peer DB Sync Algorithm

>_ git checkout c10_peer_sync

Each Block Has a Number

Sunday, December 8.

A Node must know how many blocks it has. Therefore every block header needs a **number** representing the sequence in which the block was added into the blockchain. The blockchain industry commonly refers to the block number as “block height.” The “height” (size) of the database ledger.

Andrej adds a new Number attribute into the BlockHeader:

```
type Block struct {
    Header BlockHeader `json:"header"`
    TXs     []Tx        `json:"payload"`
}

type BlockHeader struct {
    Parent Hash      `json:"parent"`
    Number uint64   `json:"number" // new attribute`
    Time   uint64   `json:"time"`
}
```

And modifies the Persist() function to construct every consecutive block with an incremental block height:

```
func (s *State) Persist() (Hash, error) {
    block := NewBlock(
        latestBlockHash,
        s.latestBlock.Header.Number + 1,
        uint64(time.Now().Unix()),
        s.txMempool,
    )
}
```

It seems like a trivial change, right?

Not in blockchain! Because blockchain is an immutable database, modifying any previous block value will result in a series of new hashes. The above change in production would break the whole system!

Fortunately, Andrej is the only blockchain node at the moment, and he can migrate the database without any consequences. He re-

moves the content (not the file) from \$HOME/.tbb/database/block.db and re-generates the DB using his new tbb migrate cmd.

>_ cat /dev/null > \$HOME/.tbb/database/block.db

>_ tbb migrate --datadir=\$HOME/.tbb

New blocks.db with the number attribute:

```
{"hash": "1e85bc", "block": { "header": { "parent": "000000", "number": 0
{"hash": "b3d057", "block": { "header": { "parent": "1e85bc", "number": 1
{"hash": "357d11", "block": { "header": { "parent": "b3d057", "number": 2
```



Fun Facts

Blockchain and immutability has many drawbacks that are a piece of cake to overcome when working with traditional database systems such as MySQL. Adding a new column? Please, one ALTER TABLE query with a default or a dynamically calculated value. Blockchain? Let me re-write the entire database from scratch.



Study Code

Adds support for expanding a relative --datadir [b1e6ba³³](#).

Updates to version 0.6.1 [7bc9ad³⁴](#)

Adds a new attribute to Block Header: "number" [c362c3³⁵](#)

³³ <https://github.com/web3coach/the-blockchain-bar/commit/b1e6ba67c045141ec4617f6dfd47c2c561e6a650>

³⁴ <https://github.com/web3coach/the-blockchain-bar/commit/7bc9adba9be4b4ae02cb7b8e6ac5ef8630985870>

³⁵ <https://github.com/web3coach/the-blockchain-bar/commit/c362c3b341aa297ddc1ae0bba7ad195d7357e9dd>

Tell Me Your State

Monday, December 9.

For a Node to synchronize with the rest of the blockchain network, each Node needs to expose its current state. Why? So you know who has new database content to synchronize from.

Andrej adds a new HTTP endpoint /node/status returning the information about the node's latest block:

```
type StatusRes struct {
    Hash    database.Hash `json:"block_hash"`
    Number uint64          `json:"block_number"`
}
```

Status HTTP handler, handling the new endpoint:

```
http.HandleFunc("/node/status", func(w ResponseWriter, r *Req) {
    statusHandler(w, r, state)
})
```

```
func statusHandler(w ResponseWriter, r Req, state *database.State) {  
  
    res := StatusRes{  
        Hash: state.LatestBlockHash(),  
        Number: state.LatestBlock().Header.Number,  
    }  
  
    writeRes(w, res)  
}
```

Ten lines of code! Go ❤️

➤ curl -X GET http://localhost:8080/node/status

```
{  
    "block_hash": "357d11...",  
    "block_number": 2  
}
```



Study Code

Adds /node/status HTTP endpoint 379cea³⁶

³⁶ <https://github.com/web3coach/the-blockchain-bar/commit/379ceaf9718bee5c723f5bf31e7e7e16f0a24e60>

Bootstrap Nodes and Peer List

Tuesday, December 10.

A running TBB node needs to have pre-configured at least one bootstrap node to discover other peers connected to the TBB blockchain network. As this is not State component business logic, Andrej creates two new domain objects.

A Node struct for controlling all blockchain services such as HTTP API and the Sync algorithm.

```
type Node struct {
    dataDir string
    port     uint64

    // To inject the State into HTTP handlers
    state *database.State

    knownPeers []PeerNode
}
```

Constructed as:

```
func New(dataDir string, port uint64, bootstrap PeerNode) *Node {
    return &Node{
        dataDir: dataDir,
        port: port,
        knownPeers: []PeerNode{bootstrap},
    }
}
```

Responsible mainly for running the HTTP API on a hardcoded (for now) port 8080:

```
const DefaultHTTPPort = 8080

func (n *Node) Run() error {
    fmt.Println(fmt.Sprintf("Listening on HTTP port: %d", n.port))

    state, err := database.NewStateFromDisk(n.dataDir)
    if err != nil {
        return err
    }
    defer state.Close()

    n.state = state

    http.HandleFunc("/balances/list", func(w ResponseWriter, r *Req) {
        listBalancesHandler(w, r, state)
    })

    http.HandleFunc("/tx/add", func(w ResponseWriter, r *Req) {
        txAddHandler(w, r, state)
    })

    http.HandleFunc("/node/status", func(w ResponseWriter, r *Req) {
        statusHandler(w, r, n)
    })

    return http.ListenAndServe(fmt.Sprintf(":%d", n.port), nil)
}
```

The second new object is a `PeerNode` struct to encapsulate the node's network details: IP, PORT:

```
type PeerNode struct {
    IP      string `json:"ip"`
    Port    uint64 `json:"port"`
    IsBootstrap bool `json:"is_bootstrap"`
    IsActive   bool `json:"is_active"`
}
```

He modifies the `tbb run` command and configures the node's default bootstrap node to point to the Andrej's AWS bootstrap node:

```
bootstrap := node.NewPeerNode(
    "3.127.248.10",
    8080,
    true,
    true,
)

n := node.New(dataDir, port, bootstrap)
err := n.Run()
```

In the spirit of decentralization and recursive peer discovery, each node must also expose a **list of known peers** as an additional attribute in the previously exposed `/node/status` endpoint:

```
type StatusRes struct {
    Hash         database.Hash `json:"block_hash"`
    Number       uint64        `json:"block_number"`
    KnownPeers  []PeerNode   `json:"peers_known" // tell me your peers`
}
```

Andrej compiles the code, restarts the TBB node and queries the status endpoint again and formats the output using his favorite CLI JSON processor. [jq installation source](#).³⁷

> curl -X GET http://localhost:8080/node/status | jq

```
{
  "block_hash": "357d11...",
  "block_number": 2,
  "peers_known": [
    {
      "ip": "3.127.248.10", // Andrej's AWS bootstrap node
      "port": 8080,
      "is_bootstrap": true,
      "is_active": true
    }
  ]
}
```

PS: Andrej also added a new --port flag and re-organized the node functionalities by moving the HTTP specific business logic to separate files: /node/http_routes.go and /node/http_req_res_utils.go. See commit below for the exact changes.

³⁷ <https://github.com/stedolan/jq>



Study Code

Adds Node struct with new concepts of peerNodes [7b2ce1³⁸](#)

³⁸<https://github.com/web3coach/the-blockchain-bar/commit/7b2ce1a5647abb59b8bd2822b0a5b1ecf12c4684>

The Search for a Peer with New Blocks

Wednesday, December 11.

Andrej wants to make things right with Babayaga ASAP. Therefore, he won't over-engineer the sync algorithm just yet. Periodically iterating over the known peer list and querying the HTTP status endpoint of every peer to find new peers and new blocks will do the trick.

This `sync()` function will run asynchronously, in a separate thread (go-routine):

```
func (n *Node) Run() error {
    fmt.Println(fmt.Sprintf("Listening on HTTP port: %d", n.port))

    state, err := database.NewStateFromDisk(n.dataDir)

    // Run sync() in a separate thread
    go n.sync(ctx)

    http.HandleFunc("/balances/list", func(w ResponseWriter, r *Req) {
        listBalancesHandler(w, r, state)
    })

    // ...
}
```

What the `sync()` function does exactly?

Every 45 seconds, it searches for new network's peers and their new blocks:

```
func (n *Node) sync(ctx context.Context) error {
    ticker := time.NewTicker(45 * time.Second)

    for {
        select {
        case <- ticker.C:
            fmt.Println("Searching for new Peers and Blocks...")
            n.fetchNewBlocksAndPeers()

        case <- ctx.Done():
            ticker.Stop()
        }
    }
}
```

fetchNewBlocksAndPeers() queries the status of every peer using previously defined /node/status endpoint until it finds a peer with more blocks and then it fetches the blocks. The same with peers.

Oh, the beauty of distributed systems.

```
func (n *Node) fetchNewBlocksAndPeers() {
    for _, peer := range n.knownPeers {
        status, err := queryPeerStatus(peer)
        if err != nil {
            fmt.Printf("ERROR: %s\n", err)
            continue
        }

        localBlockNumber := n.state.LatestBlock().Header.Number
        if localBlockNumber < status.Number {
            newBlocksCount := status.Number - localBlockNumber
        }

        for _, statusPeer := range status.KnownPeers {
            newPeer, isKnownPeer := n.knownPeers[statusPeer.TcpAddress()]
            if !isKnownPeer {
                fmt.Printf("Found new Peer %s\n", peer.TcpAddress())

                n.knownPeers[statusPeer.TcpAddress()] = newPeer
            }
        }
    }
}
```

If you noticed that Andrej also refactored the knownPeers from an Array to a Map for more efficient lookups, and introduced a new TcpAddress() function on PeerNode struct then congratulation, you have a fantastic Eagle Eye!

```
type Node struct {
    // knownPeers []PeerNode
    knownPeers map[string]PeerNode
}

func (pn PeerNode) TcpAddress() string {
    return fmt.Sprintf("%s:%d", pn.IP, pn.Port)
}

func New(dataDir string, port uint64, bootstrap PeerNode) *Node {
    // Initialize a new map with only one known peer,
    // the bootstrap node
    knownPeers := make(map[string]PeerNode)
    knownPeers[bootstrap.TcpAddress()] = bootstrap

    return &Node{
        dataDir: dataDir,
        port: port,
        // knownPeers: []PeerNode{bootstrap},
        knownPeers: knownPeers,
    }
}
```



Fun Facts

Go language has a convention of controlling the processes and threads via a context. The context can have a time deadline or be manually closed. You pass the ctx variable always as the first function's argument.

You can pause the program execution, or the current thread (go-routine) by waiting until something happens. E.g., until the context is closed: <- ctx.Done() and then execute some additional logic. In this case, when Andrej closes the tbb run command, the ctx.Done() triggers, and the periodic 45 seconds Sync() timer stops.

The <- is Go's channel messaging system and what makes the language unique and special. Checkout [this tutorial³⁹](#) because you will use channels a lot in the next chapters.



Study Code

Adds sync algorithm searching for new Peers and Blocks
[8d308e⁴⁰](#)

Fixes a bug in previous commit of syncing new peers [c0ce5e⁴¹](#)

³⁹ <https://gobyexample.com/channels>

⁴⁰ <https://github.com/web3coach/the-blockchain-bar/commit/8d308e70e5cb37954629c12015770a718594f476>

⁴¹ <https://github.com/web3coach/the-blockchain-bar/commit/c0ce5e8a573d53e4b0c108d1a293877e1002c957>

Give Me Your Blocks or Life!

Thursday, December 12.

Great. Now the algorithm knows which peer has new blocks. Fetching them is a matter of adding a new HTTP endpoint to every node:

/node sync?fromBlock=7d9248064...

```
func syncHandler(w ResponseWriter, r *Req, dataDir string) {
    // What's your latest block?
    // I will check my state, if I have newer blocks
    reqHash := r.URL.Query().Get(endpointSyncQueryKeyFromBlock)

    hash := database.Hash{}
    err := hash.UnmarshalText([]byte(reqHash))
    if err != nil {
        writeErrRes(w, err)
        return
    }

    // Read newer blocks from the DB
    blocks, err := database.GetBlocksAfter(hash, dataDir)
    if err != nil {
        writeErrRes(w, err)
        return
    }

    // JSON encode the blocks and return them in the response
    writeRes(w, SyncRes{Blocks: blocks})
}
```

Internally, the program will iterate over the database state until it finds the requested block hash, collecting all blocks afterwards and sending them back in the HTTP response.

The `GetBlocksAfter()` function in `database/database.go` file with the following, READ-ONLY algorithm:

```
func GetBlocksAfter(blockHash Hash, dataDir string) ([]Block, error) {
    // open the DB
    f, err := os.OpenFile(
        getBlocksDbFilePath(dataDir),
        os.O_RDONLY,
        0600)
    if err != nil {
        return nil, err
    }

    blocks := make([]Block, 0)
    shouldStartCollecting := false

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        // read the database file one line at the time
        if err := scanner.Err(); err != nil {
            return nil, err
        }

        var blockFs BlockFS
        err = json.Unmarshal(scanner.Bytes(), &blockFs)
        if err != nil {
            return nil, err
        }

        if shouldStartCollecting {
            blocks = append(blocks, blockFs.Value)
        }
    }
}
```

```

        continue
    }

    // found the block hash, collect new blocks
    if blockHash == blockFs.Key {
        shouldStartCollecting = true
    }
}

return blocks, nil
}

```

Currently Andrej's DB has the following 3 blocks inside the \$HOME/.tbb/database (your will vary):

```
{ "hash": "7d9248064edc575e8d..." }
```

```
{ "hash": "d4aad206a8a0cad2e0..." }
```

```
{ "hash": "b775bcd0b90e944bf3..." }
```

BabaYaga latest block hash in her DB is **7d9248**, after she boots her node, she automatically synchronizes and fetches the remaining two missing blocks from Andrej (not because he is the bootstrap node but because he is the only active node at the moment):

```
{ "hash": "d4aad206a8a0cad2e0..." }
```

```
{ "hash": "b775bcd0b90e944bf3..." }
```

```
the-blockchain-bar $curl -X GET http://127.0.0.1:8080/node-sync?fromBlock=7d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123
{
  "blocks": [
    {
      "header": {
        "parent": "7d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123652884",
        "number": 2,
        "time": 1575820141
      },
      "payload": [
        {
          "from": "andrey",
          "to": "babayaga",
          "value": 2000,
          "data": ""
        }
      ]
    },
    {
      "header": {
        "parent": "d4aad206a8a0cad2ef5d71272fd02451ed73a5207cd3f65ef8d878eba742de2",
        "number": 3,
        "time": 1575820141
      },
      "payload": [
        {
          "from": "andrey",
          "to": "andrey",
          "value": 24700,
          "data": "reward"
        }
      ]
    }
  ]
}
```

1

2

new synced blocks

Study Code

Adds a /node-sync endpoint for fetching blocks [d0cf2a⁴²](#)

⁴²<https://github.com/web3coach/the-blockchain-bar/commit/d0cf2a4aaaf4c73ea21e1ea0857ee4ac5d0341d96>

Trust but Verify

Friday, December 13.

What's the last missing part of sync's business logic? Correct. It is persisting the newly fetched blocks into the node's local database. Andrej reviews how the current State component works.

The State component contains an in-memory txMempool of not-yet-persisted transactions as well as an aggregated state of all user's balances:

```
type State struct {
    Balances map[Account]uint
    txMempool []Tx
```

The State has the following public API:

```
func (s *State) AddBlock(b Block) error
func (s *State) AddTx(tx Tx) error
func (s *State) Persist() (Hash, error)
```

But there are several significant problems with the [current implementation!](#)⁴³ Pause reading. Open the [state.go source code](#)⁴⁴ and try to find out as many bugs and edge cases as you can.

⁴³ https://github.com/web3coach/the-blockchain-bar/blob/c9_tango/database/state.go#L91

⁴⁴ https://github.com/web3coach/the-blockchain-bar/blob/c9_tango/database/state.go#L3

Andrej: How many did you find?

You: XXX

Congratulations! You are excellent at finding bugs. For the developers reading this book tired after work, who want to enjoy the show, here goes Andrej complaining about his code:

```

func (s *State) AddBlock(b Block) error {
    for _, tx := range b.TXs {
        if err := s.AddTx(tx); err != nil {
            return err
        }
    }
    return nil
}

func (s *State) AddTx(tx Tx) error {
    if err := s.apply(tx); err != nil {
        return err
    }
    s.txMempool = append(s.txMempool, tx)
    return nil
}

```

Only extracting the Payload, forgetting the Header

No concurrency support. If I sync a new Block with TXs from BabaYaga and in the same time I add a local TX from CLI and call Persist(). The State breaks!!!

```

func (s *State) Persist() (Hash, error) {
    latestBlockHash, err := s.latestBlock.Hash()
    if err != nil {
        return Hash{}, err
    }
    block := NewBlock(latestBlockHash, s.latestBlock.Header.Number+1, uint64(time.Now().Unix()))
    blockHash, err := block.Hash()
    if err != nil {
        return Hash{}, err
    }
    blockFs := BlockFS{Key: blockHash, Value: block}
    blockFsJson, err := json.Marshal(blockFs)
    if err != nil {
        return Hash{}, err
    }
}

```

Only works with locally created TX, Blocks!

Forcing current time.
No support for Blocks from past

finding bugs in current state impl

Andrej goes on a refactoring rampage and re-designs the State component given new business requirements, including a strict validation given the blocks could originate from anyone.

First, he replaces the AddTx(), Persist() functions with a more encapsulated AddBlock() function, internally handling the validation and persistence:

```

func (s *State) AddBlock(b Block) (Hash, error) {
    pendingState := s.copy()

    err := applyBlock(b, pendingState)
    if err != nil {
        return Hash{}, err
    }

    blockHash, err := b.Hash()
    if err != nil {
        return Hash{}, err
    }

    blockFs := BlockFS{ Key: blockHash, Value: b}

    blockFsJson, err := json.Marshal(blockFs)
    if err != nil {
        return Hash{}, err
    }

    fmt.Printf( format: "Persisting new Block to disk:\n")
    fmt.Printf( format: "\t%s\n", blockFsJson)

    _, err = s.dbFile.Write(append(blockFsJson, elems: '\n'))
    if err != nil {
        return Hash{}, err
    }

    s.Balances = pendingState.Balances
    s.latestBlockHash = blockHash
    s.latestBlock = b

    return blockHash, nil
}

```

add block

Validate block meta + payload.

Replays transactions to verify balances.

Write to disk.

All TXs are valid and no error writing to disk?

Update the main state!

- All DB, State changes will go through the AddBlock()
- Strict validation on an isolated copy of the State. Why a copy? Because in case someone would sneak a malicious transaction, it's crucial to not corrupt the main balance sheet (state.Balances

Map) of all accounts. An example of a malicious transaction would be a transfer of 1M TBB tokens from an account with a balance of only 500 TBB tokens

- If the block is valid, hash it and immediately persist it to disk

The `state.copy()` is a matter of creating an identical struct but without any pointers to the original State. It's necessary to re-create the Maps and Arrays from scratch. Attempt to re-assign them to a new variable would only create a new pointer in Go.

`stateCopy := state` wouldn't work for internal Maps, Arrays (slices).

A manual, deep copy is required:

```
func (s *State) copy() State {
    c := State{}
    c.latestBlock = s.latestBlock
    c.latestBlockHash = s.latestBlockHash
    c.txMempool = make([]Tx, len(s.txMempool))
    c.Balances = make(map[Account]uint)

    for acc, balance := range s.Balances {
        c.Balances[acc] = balance
    }

    for _, tx := range s.txMempool {
        c.txMempool = append(c.txMempool, tx)
    }

    return c
}
```

AddBlock() calls internally the applyBlock() function:

```
// applyBlock verifies if block can be added to the blockchain.
//
// Block metadata are verified as well as transactions within -
// sufficient balances, etc.
func applyBlock(b Block, s State) error {
    nextExpectedBlock := s.latestBlock.Header.Number + 1

    // validate the next block number increases by 1
    if s.hasGenesisBlock && b.Header.Number != nextExpectedBlock {
        return fmt.Errorf(
            "next expected block must be '%d' not '%d'",
            nextExpectedBlockNumber,
            b.Header.Number,
        )
    }

    // validate the incoming block parent hash equals
    // the current (latest known) hash
    if s.hasGenesisBlock && s.latestBlock.Header.Number > 0
        && !reflect.DeepEqual(b.Header.Parent, s.latestBlockHash) {
        return fmt.Errorf(
            "next block parent hash must be '%x' not '%x'",
            s.latestBlockHash,
            b.Header.Parent,
        )
    }

    return applyTXs(b.TXs, &s)
}
```

And the applyTXs verifies the account balances:

```
func applyTx(tx Tx, s *State) error {
    if tx.IsReward() {
        s.Balances[tx.To] += tx.Value
        return nil
    }

    if tx.Value > s.Balances[tx.From] {
        return fmt.Errorf(
            "wrong TX. Sender '%s' balance is %d TBB. Tx cost is %d",
            tx.From,
            s.Balances[tx.From],
            tx.Value,
        )
    }

    s.Balances[tx.From] -= tx.Value
    s.Balances[tx.To] += tx.Value

    return nil
}
```

Fantastic!

Andrej, happy with the new State component, updates the sync() logic to call doSync():

```
func (n *Node) sync(ctx context.Context) error {
    ticker := time.NewTicker(10 * time.Second)

    for {
        select {
        case <- ticker.C:
            n.doSync()

        case <- ctx.Done():
            ticker.Stop()
        }
    }
}
```

doSync imports all new fetched blocks from peers as well as the network details (TCP IP, PORT) of peers of other peers. The beauty of distributed, decentralized systems. A recursive net crawler:

```
func (n *Node) doSync() {
    for _, peer := range n.knownPeers {
        status, err := queryPeerStatus(peer)
        err = n.joinKnownPeers(peer)
        err = n.syncBlocks(peer, status)
        err = n.syncKnownPeers(peer, status)
    }
}
```

Blocks synchronization

Fetch new blocks from a peer and import them into the refactored, shinny State component:

```
func (n *Node) syncBlocks(peer PeerNode, status StatusRes) error {
    localBlockNumber := n.state.LatestBlock().Header.Number
    if localBlockNumber < status.Number {
        newBlocksCount := status.Number - localBlockNumber

        fmt.Printf(
            "Found %d new blocks from Peer %s\n",
            newBlocksCount,
            peer.TcpAddress())

        // Call the node's /node-sync endpoint and read new blocks
        blocks, err := fetchBlocksFromPeer(
            peer,
            n.state.LatestBlockHash())
        if err != nil {
            return err
        }

        // Write the newly downloaded blocks to this node's local DB
        err = n.state.AddBlocks(blocks)
        if err != nil {
            return err
        }
    }

    return nil
}
```

Peer synchronization

Ask each peer what peers he knows, and in case an unknown peer is encountered, add the peer to the node's list of known peers.

```
func (n *Node) syncKnownPeers(peer PeerNode, status StatusRes) error {
    for _, statusPeer := range status.KnownPeers {
        if !n.IsKnownPeer(statusPeer) {
            fmt.Printf("Found new Peer %s\n", statusPeer.TcpAddress())

            n.AddPeer(statusPeer)
        }
    }

    return nil
}

func (n *Node) AddPeer(peer PeerNode) {
    n.knownPeers[peer.TcpAddress()] = peer
}
```

Andrej is DONE.

He tests the algorithm on his local machine, simulating a peer-to-peer (p2p) communication between 3 nodes.

The screenshot shows three terminal windows on a Linux system (Ubuntu 12.04 LTS) demonstrating a peer-to-peer sync test between three nodes: Andrej, Babayaga, and Caesar. The nodes are running a TBB (The Blockchain Bar) application.

- Node 1: Andrej runs his bootstrap node** (top window):


```
the-blockchain-bar $tbb balances list --datadir=~/tbb
Accounts balances at block:
  - hash: b775bcd090e944bf3e9aeb68d6b1c3bd5f2bd53c33175391be78
  - number: 3
```
- Node 2: Babayaga syncs all Andrej blocks** (middle window):


```
the-blockchain-bar $tbb run --datadir=~/tbb_babayaga -ip=127.0.0.1 --port=8081
Launching TBB node and its HTTP API...
Listening on: 127.0.0.1:8081
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8080'
Found 3 new blocks from Peer 127.0.0.1:8080
Importing blocks from Peer 127.0.0.1:8080...
Persisting new Block to disk:
  {"hash": "7d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123652884", "block": {"header": {"parent": "46438b2675171b3e40b013218805de961e8d40af7af252fe160b5eb2089d027", "number": 1, "time": 1575820141}, "payload": [{"from": "andrej", "to": "andrej", "value": 3, "data": ""}, {"from": "andrej", "to": "andrej", "value": 700, "data": "reward"}]}}
Persisting new Block to disk:
  {"hash": "d4aaad206a80cad2e055d71272fd02451ed73a5207cd3f65ef8d878eba742de2", "block": {"header": {"parent": "7d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123652884", "number": 2, "time": 1575820141}, "payload": [{"from": "andrej", "to": "babayaga", "value": 2000, "data": ""}, {"from": "andrej", "to": "babayaga", "value": 1, "data": ""}, {"from": "babayaga", "to": "caesar", "value": 1000, "data": ""}, {"from": "babayaga", "to": "andrej", "value": 50, "data": ""}, {"from": "andrej", "to": "andrej", "value": 600, "data": "reward"}]}}
Persisting new Block to disk:
  {"hash": "b775bcd090e944bf3e9aeb68d6b1c3bd5f2bd53c33175391be78a0662e17bac", "block": {"header": {"parent": "d4aaad206a80cad2e055d71272fd02451ed73a5207cd3f65ef8d878eba742de2", "number": 3, "time": 1575820141}, "payload": [{"from": "andrej", "to": "babayaga", "value": 2000, "data": ""}, {"from": "andrej", "to": "babayaga", "value": 1, "data": ""}, {"from": "babayaga", "to": "caesar", "value": 1000, "data": ""}, {"from": "babayaga", "to": "andrej", "value": 50, "data": ""}, {"from": "andrej", "to": "andrej", "value": 600, "data": "reward"}]}}
```
- Node 3: Caesar syncs all Andrej blocks** (bottom window):


```
$tbb run --datadir=~/tbb_caesar -ip=127.0.0.1 --port=8082
Launching TBB node and its HTTP API...
Listening on: 127.0.0.1:8082
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8080'
Found 3 new blocks from Peer 127.0.0.1:8080
Importing blocks from Peer 127.0.0.1:8080...
Persisting new Block to disk:
  {"hash": "7d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123652884", "block": {"header": {"parent": "46438b2675171b3e40b013218805de961e8d40af7af252fe160b5eb2089d027", "number": 1, "time": 1575820141}, "payload": [{"from": "andrej", "to": "andrej", "value": 3, "data": ""}, {"from": "andrej", "to": "andrej", "value": 700, "data": "reward"}]}}
Persisting new Block to disk:
  {"hash": "d4aaad206a80cad2e055d71272fd02451ed73a5207cd3f65ef8d878eba742de2", "block": {"header": {"parent": "7d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123652884", "number": 2, "time": 1575820141}, "payload": [{"from": "andrej", "to": "babayaga", "value": 2000, "data": ""}, {"from": "andrej", "to": "babayaga", "value": 1, "data": ""}, {"from": "babayaga", "to": "caesar", "value": 1000, "data": ""}, {"from": "babayaga", "to": "andrej", "value": 50, "data": ""}, {"from": "andrej", "to": "andrej", "value": 600, "data": "reward"}]}}
Persisting new Block to disk:
  {"hash": "d4aaad206a80cad2e055d71272fd02451ed73a5207cd3f65ef8d878eba742de2", "block": {"header": {"parent": "d4aaad206a80cad2e055d71272fd02451ed73a5207cd3f65ef8d878eba742de2", "number": 3, "time": 1575820141}, "payload": [{"from": "andrej", "to": "babayaga", "value": 2000, "data": ""}, {"from": "andrej", "to": "babayaga", "value": 1, "data": ""}, {"from": "babayaga", "to": "caesar", "value": 1000, "data": ""}, {"from": "babayaga", "to": "andrej", "value": 50, "data": ""}, {"from": "andrej", "to": "andrej", "value": 600, "data": "reward"}]}}
```

p2p sync test

He must also test a scenario where BabaYaga adds a TX through her node and observes that both Andrej and Caesar nodes get automatically synced.

The screenshot shows three terminal windows illustrating a bidirectional sync test:

- Top Terminal (BabaYaga's node):** Shows BabaYaga adding a new transaction (tx) via a curl POST request to localhost:8081. The transaction details are shown in the terminal output.
- Middle Terminal (Andrej's node):** Shows Andrej syncing from BabaYaga's node. It displays logs of peers being searched for and blocks being imported from Peer 127.0.0.1:8081.
- Bottom Terminal (Caesar's node):** Shows Caesar syncing from Andrej's node. It displays logs of peers being searched for and blocks being imported from Peer 127.0.0.1:8081.

Annotations with arrows indicate the flow of data: a blue arrow points from the BabaYaga tx add command to the Andrej sync logs; an orange arrow points from the Andrej sync logs to the Caesar sync logs.

1) BabaYaga adds a new TX

Peer-to-Peer blockchain

p2p sync bidirectional test

```

the-blockchain-bar $curl -X POST http://localhost:8081/tx/add \
> -d '{
  >   "from": "babayaga",
  >   "to": "andrej",
  >   "value": 50
  > }'
  > {"block_hash": "f53e82424e52325731fc0a5847f41ff1bafe69a257910db34bc0a8b5b09d934c4"}
the-blockchain-bar $
  
```

 **Practice time.**

Uff! This was a lot of information and changes.

But don't worry. The following practice steps will help you solidify the understanding of how the sync() algorithm in blockchain networks works.

You will reproduce the same activity/environment as on the Andrej's screenshots above.

1/4 Ensure you have the latest branch's TBB version.

>_ go install ./cmd/...

tbb version

Version: 0.7.2-beta Sync

2/4 Verify your program's bootstrap node.

Open `./cmd/tbb/run.go` and check line 20. You should see the bootstrap node to be configured as "127.0.0.1". This is necessary because you will now setup the entire test blockchain network consisting of 3 nodes on your own machine. How awesome is that?!

```
bootstrap := node.NewPeerNode(  
    "127.0.0.1",  
    8080,  
    true,  
    false,  
)
```

3/4 Boot 3 new nodes in 3 separate terminals.

Boot your main node in a data directory `$HOME/.andrey_sync` on port 8080.

Let's call it Andrej's node for the sake of having the same language. Don't worry if the `$HOME/.andrey_sync` directory doesn't exist, the directory will be initialized the first time you run the `tbb run` cmd with the balances generated from the `genesis.json` file.

➤ `tbb run --datadir=$HOME/.andrey_sync --ip=127.0.0.1
--port=8080`

Boot your second node in a data directory `$HOME/.babayaga_sync` on port 8081 .

➤ `tbb run --datadir=$HOME/.babayaga_sync --ip=127.0.0.1
--port=8081`

Boot your third node in a data directory `$HOME/.caesar_sync` on port 8082 .

➤ `tbb run --datadir=$HOME/.caesar_sync --ip=127.0.0.1
--port=8082`

Every 45 seconds you will see 6 new message in the standard output as the nodes will communicate together in true, decentralized peer-to-peer fashion.

Andrej's node will be searching for new blocks and peers from BabaYaga and Caesar nodes.

```
Launching TBB node and its HTTP API...
```

```
Listening on: 127.0.0.1:8080
```

```
Peer '127.0.0.1:8081' was added into KnownPeers
```

```
Peer '127.0.0.1:8082' was added into KnownPeers
```

```
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8081'
```

```
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8082'
```

BabaYaga's node will be searching for new blocks and peers from Andrej and Caesar nodes.

```
Launching TBB node and its HTTP API...
```

```
Listening on: 127.0.0.1:8081
```

```
Peer '127.0.0.1:8082' was added into KnownPeers
```

```
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8080'
```

```
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8082'
```

Caesar's node will be searching for new blocks and peers from Andrej and BabaYaga nodes.

```
Launching TBB node and its HTTP API...
```

```
Listening on: 127.0.0.1:8082
```

```
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8080'
```

```
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8081'
```

4/4 Send a TX to Andrej's node - transferring 100 TBB tokens from Andrej to BabaYaga.

```
curl --request GET 'localhost:8080/tx/add' \
--header 'Content-Type: application/json' \
--data-raw '{
    "from": "andrey",
    "to": "babayaga",
    "value": 100
}'

> {"block_hash": "a0a44d..."}
```

PS: Your hash will be different because of the block's time attribute.

5/5 Observe how all nodes will synchronize the new block.

Andrej's node:

Persisting new Block to disk:

```
{"hash": "a0a44d", "block": {"header": {"parent": "00000", "number": 0, "time": \n:1588844231}, "payload": [{"from": "andrey", "to": "babayaga", "value": 100, "data": ""}]}}
```

BabaYaga's node:

```
Found 1 new blocks from Peer 127.0.0.1:8080\nImporting blocks from Peer 127.0.0.1:8080...\nPersisting new Block to disk:
```

```
{"hash": "a0a44d", "block": {"header": {"parent": "000000", "number": 0, "time": \n":1588844231}, "payload": [{"from": "andrey", "to": "babayaga", "value": 100, "data": ""}]}}
```

Caesar's node:

```
Found 1 new blocks from Peer 127.0.0.1:8080\nImporting blocks from Peer 127.0.0.1:8080...\nPersisting new Block to disk:
```

```
{"hash": "a0a44d", "block": {"header": {"parent": "000000", "number": 0, "time": \n":1588844231}, "payload": [{"from": "andrey", "to": "babayaga", "value": 100, "data": ""}]}}
```

All the peers successfully interconnected by connecting first to the Andrej's bootstrap's node, and later exchanging the known_peers

list provided by the public /node/status endpoint running on every node.

DONE! Good job.

Use libp2p networking stack

Andrej's syncing solution works, but it's rather simplified for educational purposes. What about Peer identity? Trusted connections? Streams? Handshakes?

How could Andrej improve his p2p syncing solution? By using a **libp2p - a framework and suite of protocols for building peer-to-peer network applications.**

libp2p is the product of a long, and arduous quest of understanding – a deep dive into the internet’s network stack, and plentiful peer-to-peer protocols from the past. Building large-scale peer-to-peer systems has been complex and difficult in the last 15 years, and libp2p is a way to fix that. It is a “network stack” – a protocol suite – that cleanly separates concerns, and enables sophisticated applications to only use the protocols they absolutely need, without giving up interoperability and upgradeability. libp2p grew out of IPFS, but it is built so that lots of people can use it, for lots of different projects.

Use libp2p for connectivity, security, multiplexing, and other purposes.

Projects⁴⁵ built on top of the libp2p:

- identify - Exchange keys and addresses with other peers
- kademlia - The Kademlia Distributed Hash Table (DHT) subsystem
- mplex - The friendly stream multiplexer
- plaintext - An insecure transport for non-production usage
- pnet - Private networking in libp2p using pre-shared keys
- pubsub - PubSub interface for libp2p
- gossipsub - An extensible baseline PubSub protocol
- episub - Proximity Aware Epidemic PubSub for libp2p
- relay - Circuit Switching for libp2p (similar to TURN)
- rendezvous - Rendezvous Protocol for generalized peer discovery
- secio - SECIO, a transport security protocol for libp2p
- tls - The libp2p TLS Handshake (TLS 1.3+)

Libp2p offers to developers two important concepts out of the box: **Transport** and **Peer Identity**.

Transport

When you make a connection from your computer to a machine on the internet, chances are pretty good you're sending your bits and bytes using TCP/IP, the wildly successful combination of the Internet Protocol, which handles addressing and delivery of data packets, and the Transmission Control Protocol, which ensures that the data

⁴⁵ [<https://github.com/libp2p/specs#protocols>](https://github.com/libp2p/specs#protocols)

that gets sent over the wire is received completely and in the right order.

While TCP and UDP (together with IP) are the most common protocols in use today, they are by no means the only options. Alternatives exist at lower levels (e.g. sending raw ethernet packets or bluetooth frames), and higher levels (e.g. QUIC, which is layered over UDP).

In libp2p, we call these foundational protocols that move bits around transports, and one of libp2p's core requirements is to be transport agnostic. This means that the decision of what transport protocol to use is up to the developer, and in fact one application can support many different transports at the same time.

Here's an example of a multiaddr for a TCP/IP transport:

```
/ip4/7.7.7.7/tcp/6543
```

Source: <https://docs.libp2p.io/concepts/transport/>⁴⁶

⁴⁶ <https://docs.libp2p.io/concepts/transport/>

Peer Identity

A Peer Identity (often written PeerId) is a unique reference to a specific peer within the overall peer-to-peer network.

As well as serving as a unique identifier for each peer, a PeerId is a verifiable link between a peer and its public cryptographic key. Each libp2p peer controls a private key, which it keeps secret from all other peers. Every private key has a corresponding public key, which is shared with other peers. **Together, the public and private key (or “key pair”) allow peers to establish secure communication channels with each other.**

Conceptually, a PeerId is a cryptographic hash of a peer's public key. When peers establish a secure channel, the hash can be used to verify that the public key used to secure the channel is the same one used to identify the peer.

Here's an example of a PeerId represented as a base58-encoded multihash:

QmYyQSo1c1Ym7orWxLYvCrM2EmxFtANf8wXmmE7Dwjhx5N

As with other multiaddrs, a /p2p address can be encapsulated into another multiaddr to compose into a new multiaddr.

For example, you can combine the above with a transport address /ip4/7.7.7.7/tcp/4242 to produce this very useful address:

/ip4/7.7.7.7/tcp/4242/p2p/QmYyQSo1c...FTANf8wXmmE7DWjhx5N

This provides enough information to dial a specific peer over a TCP/IP transport. If some other peer has taken over that IP address or port, it will be immediately obvious, since they will not have control over the key pair used to produce the PeerId embedded in the address.

Source: <https://docs.libp2p.io/concepts/peer-id/>⁴⁷

⁴⁷ <https://docs.libp2p.io/concepts/peer-id/>

Setting up libp2p host

Andrej could set-up his node's p2p host with the following code snippet:

```
// Set your own keypair
priv, _, err := crypto.GenerateKeyPair(
    crypto.Ed25519, // Select your key type. Ed25519 are nice short
    -1,             // Select key length when possible (i.e. RSA).
)
if err != nil {
    panic(err)
}

var idht *dht.IpfsDHT

host, err := libp2p.New(
    // Use the keypair we generated
    libp2p.Identity(priv),
    // Multiple listen addresses
    libp2p.ListenAddrStrings(
        "/ip4/0.0.0.0/tcp/9000",      // regular tcp connections
        "/ip4/0.0.0.0/udp/9000/quic", // QUIC transport
    ),
    // support TLS connections
    libp2p.Security(libp2ptls.ID, libp2ptls.New),
    // support Noise connections
    libp2p.Security(noise.ID, noise.New),
    // support QUIC
    libp2p.Transport(libp2pquic.NewTransport),
    // support any other default transports (TCP)
    libp2p.DefaultTransports,
    // Let's prevent our peer from having too many
    // connections by attaching a connection manager.
```

```
libp2p.ConnectionManager(connmgr.NewConnManager(
    100,           // Lowwater
    400,           // HighWater,
    time.Minute,  // GracePeriod
)),
// Attempt to open ports using uPNP for NATed hosts.
libp2p.NATPortMap(),
// Let this host use the DHT to find other hosts
libp2p.Routing(func(h host.Host) (routing.PeerRouting, error) {
    idht, err = dht.New(ctx, h)
    return idht, err
}),
// Let this host use relays and advertise itself on relays if
// it finds it is behind NAT. Use libp2p.Relay(options...) to
// enable active relays and more.
libp2p.EnableAutoRelay(),
)
if err != nil {
    panic(err)
}
```

Learn libp2p from the official docs: <https://docs.libp2p.io>⁴⁸ and start using it in your decentralized projects.

⁴⁸ <https://docs.libp2p.io>



Summary

Closed software with centralized access to private data puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relation. Specifications of the blockchain system should be well defined from the beginning and only change if its users support it.

Blockchain is an immutable, transparent database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.

The database content is hashed by a secure cryptographic hash function. The blockchain participants use the resulted hash to reference a specific database state.

Transactions are grouped into batches for performance reasons. A batch of transactions make a Block. Each block is encoded and hashed using a secure, cryptographic hash function.

Block contains Header and Payload. The Header stores various metadata such a time and a reference to the Parent Block (the previous immutable database state). The Payload carries the new database transactions.

Each block has a number indicating the blockchain size (height).

The blockchain network consists of Nodes (Peers). Every Full Node is an independent computer storing the entire, real-time blockchain database.

All new nodes connect to a default Bootstrap Node(s) to discover and retrieve the current database state as well as the full transaction history.

Nodes continually and recursively communicate with each other using a sync algorithm and exchange information about each others new blocks and new network's peers.



Study Code

Adds support for expanding a relative --datadir `b1e6ba49`.

Adds a new attribute to Block Header: "number" `c362c350`

Adds /node/status HTTP endpoint `379cea51`

Adds Node struct with new concepts of peerNodes `7b2ce152`

Adds sync algorithm searching for new Peers and Blocks
`8d308e53`

Fixes a bug in previous commit of syncing new peers `c0ce5e54`

Re-adds back the CLI balances command `7b92dc55`

Adds a /node/sync endpoint for fetching blocks `d0cf2a56`

Adds sync algorithm with periodic fetching of new blocks and peers `95717357`

Fixes block number validation breaking the migrate cmd
`cdb4cc58`

Changes sync ticket to 45s and makes bootstrap node a localhost `0585ef59`

Fixes sync() function on genesis block 0 `98b38560`

⁴⁹ <https://github.com/web3coach/the-blockchain-bar/commit/b1e6ba67c045141ec4617f6dfd47c2c561e6a650>

⁵⁰ <https://github.com/web3coach/the-blockchain-bar/commit/c362c3b341aa297ddc1ae0bba7ad195d7357e9dd>

⁵¹ <https://github.com/web3coach/the-blockchain-bar/commit/379ceaf9718bee5c723f5bf31e7e7e16f0a24e60>

⁵² <https://github.com/web3coach/the-blockchain-bar/commit/7b2ce1a5647abb59b8bd2822b0a5b1ecf12c4684>

⁵³ <https://github.com/web3coach/the-blockchain-bar/commit/8d308e70e5cb37954629c12015770a718594f476>

⁵⁴ <https://github.com/web3coach/the-blockchain-bar/commit/c0ce5e8a573d53e4b0c108d1a293877e1002c957>

⁵⁵ <https://github.com/web3coach/the-blockchain-bar/commit/7b92dc81882cbdc8adf38a5de8e33a31d28d4e1a>

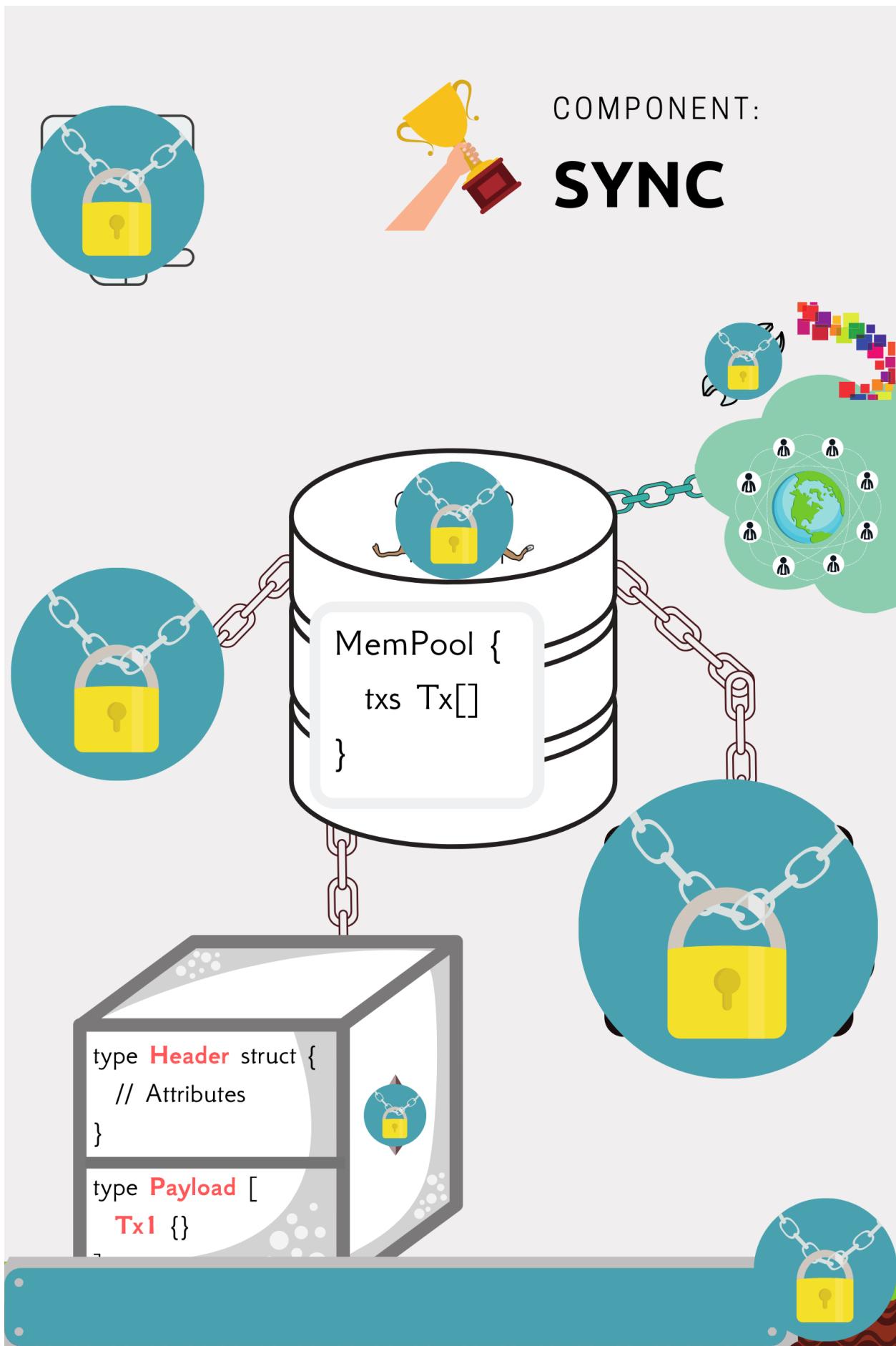
56 <https://github.com/web3coach/the-blockchain-bar/commit/d0cf2a4aaf4c73ea21e1ea0857ee4ac5d0341d96>

57 <https://github.com/web3coach/the-blockchain-bar/commit/9571733c6bc5316332296dbecb64779d0f9981e0>

58 <https://github.com/web3coach/the-blockchain-bar/commit/cdb4cc44a80d0b94f0e5775510149deeb02b2f58>

59 <https://github.com/web3coach/the-blockchain-bar/commit/0585ef9199951f21a7aa461729b1bd3a70973c90>

60 <https://github.com/web3coach/the-blockchain-bar/commit/98b3851455cf8d55326f2bce290f559b094163bc>



11 | The Autonomous Database Brain

>_ git checkout c11_consensus

To make the next examples more clear, Andrej modifies the /etc/hosts file on his machine to assign more user-friendly domain names to his local testing IPs.

```
# /etc/hosts
127.0.0.1      andrej.tbb
127.0.0.1      babayaga.tbb
```

BabaYaga setup her node during Christmas and performed a full sync. Perfect. She and Andrej now have an identical blockchain history, forming a strong network.

```
the-blockchain-bar $tbb run --datadir=~/tbb_andrej --ip=andrej.tbb --port=8080
Launching TBB node and its HTTP API...
Listening on: andrej.tbb:8080
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8080'
Peer 'andrej.tbb:8080' was added into KnownPeers
Peer 'babayaga.tbb:8081' was added into KnownPeers
Peer 'babayaga.tbb:8081' was added into KnownPeers
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8080'
Searching for new Peers and their Blocks and Peers: 'babayaga.tbb:8081'
1'
```

```
the-blockchain-bar $tbb run --datadir=~/tbb_babayaga --ip=babayaga.tbb --port=8081
Launching TBB node and its HTTP API...
Listening on: babayaga.tbb:8081
Searching for new Peers and their Blocks and Peers: '127.0.0.1:8080'
Found 3 new blocks from Peer 127.0.0.1:8080
Importing blocks from Peer 127.0.0.1:8080...
Persisting new Block to disk:
  {"hash": "d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123652884", "block": {"header": {"parent": "46438b2675171b3e40b013218805de961e8d40af7af252fe166b5eb22089d027", "number": 1, "time": 1575820141}, "payload": [{"from": "andrej", "to": "andrej", "value": 3, "data": ""}, {"from": "andrej", "to": "andrej", "value": 700, "data": "reward"}]}}
Persisting new Block to disk:
  {"hash": "d4aad206a800cad2e055d71272fd02451ed73a5207cd3f65ef8d878eba742de2", "block": {"header": {"parent": "7d9248064edc575e8de8a720e1616e50f5278a17182ea1cd9c56645123652884", "number": 2, "time": 1575820141}, "payload": [{"from": "andrej", "to": "babayaga", "value": 2000, "data": ""}, {"from": "andrej", "to": "andrej", "value": 100, "data": "reward"}, {"from": "babayaga", "to": "andrej", "value": 1, "data": ""}, {"from": "babayaga", "to": "caesar", "value": 1000, "data": ""}, {"from": "babayaga", "to": "andrej", "value": 50, "data": ""}, {"from": "andrej", "to": "andrej", "value": 600, "data": "reward"}]}}
Persisting new Block to disk:
  {"hash": "b775bcd0b90e944bf3e9aeb68d6b1c3bd5f2bd53c33175391be780a6e17", "block": {"header": {"parent": "d4aad206a800cad2e055d71272fd02451ed73a5207cd3f65ef8d878eba742de2", "number": 3, "time": 1575820141}, "payload": [{"from": "andrej", "to": "babayaga", "value": 100, "data": ""}]}}



- Andrey
- BabaYaga



| Body                                                                                                                                                                                                                                                                                                                                                                                                                          | Cookies | Headers (3) | Test Results                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>"block_hash": "b775bcd0b90e944bf3e9aeb68d6b1c3bd5f2bd53c33175391be780a6e17", "block_number": 3, "peers_known": {   "127.0.0.1:8080": {     "ip": "127.0.0.1",     "port": 8080,     "is_bootstrap": true   },   "andrej.tbb:8080": {     "ip": "andrej.tbb",     "port": 8080,     "is_bootstrap": false   },   "babayaga.tbb:8081": {     "ip": "babayaga.tbb",     "port": 8081,     "is_bootstrap": false   } }</pre> |         |             | <pre>"block_hash": "b775bcd0b90e944bf3e9aeb68d6b1c3bd5f2bd53c33175391be780a6e17", "block_number": 3, "peers_known": {   "127.0.0.1:8080": {     "ip": "127.0.0.1",     "port": 8080,     "is_bootstrap": true   } }</pre> |


```

fully synced two-peers blockchain network

Open image in full screen.⁶¹

No such a thing as Christmas without alcohol in Slovakia, and where BabaYaga is from. Andrej orders a celebration shot for finishing the sync algorithm. BabaYaga orders some drinks for finally understanding how blockchain nodes synchronize. They both record the transactions in the ledger, creating new blocks in their retrospective nodes.

⁶¹ https://web3coach-public.s3.eu-central-1.amazonaws.com/img/andrej_baba_sync.jpeg

Andrej goes for two vodka shots and an Orange juice. A classic [Screwdriver](#).⁶²

He records the TX through his local node.

>— At **15:00:00**

```
curl -X POST http://andrej.tbb:8080/tx/add \
-d '{
  "from": "andrej",
  "to": "andrej",
  "value": 7
}'
```

```
{"block_hash": "874222c7..."}
```

⁶² [https://en.wikipedia.org/wiki/Screwdriver_\(cocktail\)](https://en.wikipedia.org/wiki/Screwdriver_(cocktail))

BabaYaga, a heavier drinker, goes for five vodka shots in two orders, blocks. **As it's common and encouraged in the blockchain ecosystem for everyone to run and use their own nodes**, she records the TX via her node running on port 8081.

>_ First block at 15:00:05

```
curl -X POST http://babayaga.tbb:8081/tx/add \
-d '{
  "from": "babayaga",
  "to": "babayaga",
  "value": 2
}'

{"block_hash": "ec400ea3..."}
```

>_ Second block at 15:00:10

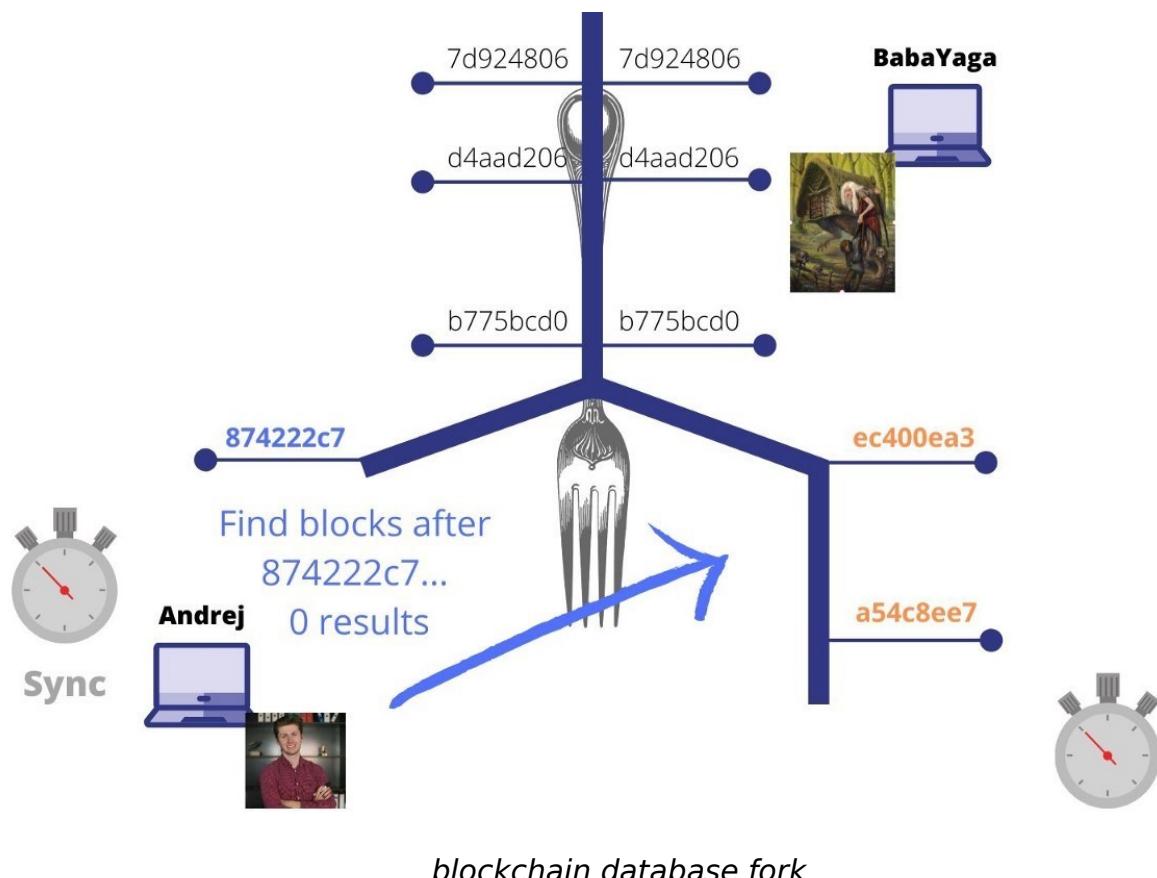
```
curl -X POST http://babayaga.tbb:8081/tx/add \
-d '{
  "from": "babayaga",
  "to": "babayaga",
  "value": 3
}'

{"block_hash": "a54c8ee7..."}
```

Unfortunately, they both did it around the same time in a 10 seconds time window. **The sync algorithm running every 45s**

didn't have enough time to coordinate the changes between Andrej's and BabaYaga's nodes. When such a thing happens in a distributed, decentralized p2p system like blockchain, without any coordination, you end up in an inconsistent blockchain state called **fork**.

After fork happens, nodes will not be able ever to synchronise again because the last block's hash on one's node doesn't exist in the other node's history. **When fork happens, the network splits** and blockchain Christmas gets ruined.



“Network latency is a bummer in distributed systems”, Andrej tells himself.

The P2P Heaven: The Fastest to Rule Them All

Wednesday, December 25.

Once a fork happens, there is no way to resolve it. Andrej must find a design pattern to **prevent the fork** from happening in the first place. He must find a way for nodes to come to a **consensus**.

The primary responsibility of a consensus algorithm is to decide which **particular node will be responsible for creating the next valid block**.

Consensus decides and coordinates:

- the p2p syncing rules
- what transactions, blocks are valid
- what peers are trustworthy
- **who can create the next block**

The most famous consensus algorithm, used in Bitcoin and Ethereum, is called **Proof of Work** (PoW).



Fun Facts

Ethereum is migrating from Proof of Work to Proof of Stake due to the PoW performance issue with low transactions throughput and slow block confirmation, mining time.

The PoW is the slowest but is considered the safest, and most decentralized consensus algorithm in 2020 - arguable, but definitely most tested one.

The Bitcoin PoW currently consumes as much electricity as the entire Austria. It accounts for 0.29% of the world's annual electricity consumption.

Annually it consumes 76.84 TWh (Chile), producing 36.50 Mt CO₂ carbon footprint (New Zealand). [Source.⁶³](#)

⁶³<https://digiconomist.net/bitcoin-energy-consumption>

How does Bitcoin Mining Works?

Thursday, December 26.

Mining is the activity Proof of Work consensus performs.

Miners are blockchain Nodes, Peers who receive transactions from other Nodes and batch them together into a valid PoW block.

Miners create PoW blocks by solving a computational, cryptographic puzzle.

What puzzle?

Bitcoin puzzle consists of generating a block hash with X amount of leading zeroes, e.g., **000000dfs5d4f58sd4f3fs4fda6**.

The fastest miner who finds a valid block hash starting with a pre-defined amount of zeroes has “mined the block” and receives a **block reward**.

The higher the amount of zeroes the block hash must start with, the higher the difficulty.

How can Miners find a block hash starting with X zeroes?

Wait, Andrej, hash of a block is deterministic and always returns the same output given the same input. You are right. The solution is to

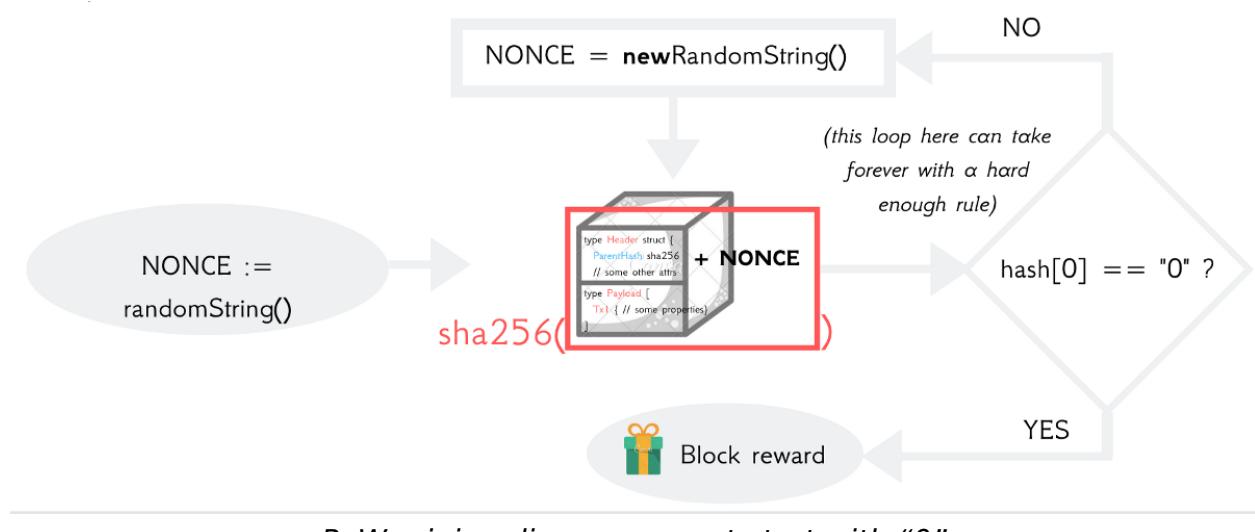
keep modifying the block in a way, the required data (Header, TXs Payload) are untouched, yet, the block is different in bytes. The trick overlies in a concept called: “**Nonce.**”

What is a Nonce?

- new Block Header property
- a random string, it can be anything, letters, numbers, special characters
- it’s appended to each block and keeps changing till the hash of the block content (Header + Payload + Nonce) satisfies the block hash requirements

Can you visualize it for me, Andrej?

PoW block mining diagram where the block hash must start with a zero:



Programming Bitcoin Mining Algorithm

Friday, December 27.

A real programming challenge incoming, Andrej tells himself. He goes to his local cafeteria and orders a double espresso; this was back in the time when cafeterias weren't closed for 5 weeks because of a pandemic. "Glog," "Glog." He is ready to convert the coffee into code.

Andrej currently generates a block hash by secure hashing the JSON encoded block:

```
func (b Block) Hash() (Hash, error) {
    blockJson, err := json.Marshal(b)
    if err != nil {
        return Hash{}, err
    }

    return sha256.Sum256(blockJson), nil
}
```

This hashing logic will stay the same, but as the previous diagram explains, Andrej needs to add a new attribute called **Nonce** to the Block Header. He must also define the criteria for a valid block hash. After some testing, he decides that from now on, **every hex-encoded block hash must start with six leading zeroes**; otherwise, it's not valid.

Andrej adds the new Nonce attribute into the BlockHeader, as explained in the diagram above.

```
type BlockHeader struct {
    Parent Hash      `json:"parent"`
    Number uint64   `json:"number"`
    Nonce  uint32    `json:"nonce"` // new attribute
    Time   uint64    `json:"time"`
}
```

And adds a function verifying the hex hash starts with six zeroes. In Go, you can use %x format to convert a string into a hex format.

```
func IsBlockHashValid(hash Hash) bool {
    return fmt.Sprintf("%x", hash[0]) == "0" &&
        fmt.Sprintf("%x", hash[1]) == "0" &&
        fmt.Sprintf("%x", hash[2]) == "0" &&
        fmt.Sprintf("%x", hash[3]) != "0"
}
```

```
type BlockHeader struct {
    Parent Hash `json:"parent"`
    Number uint64 `json:"number"`
    Nonce uint32 `json:"nonce"`
    Time uint64 `json:"time"`
}

type BlockFS struct {
    Key Hash `json:"hash"`
    Value Block `json:"block"`
}

func NewBlock(parent Hash, number uint64, nonce uint32, time uint64, txs []Tx) Block {
    return Block{BlockHeader{parent, number, nonce, time}, txs}
}

func (b Block) Hash() (Hash, error) {
    blockJson, err := json.Marshal(b)
    if err != nil {
        return Hash{}, err
    }

    return sha256.Sum256(blockJson), nil
}

func IsBlockHashValid(hash Hash) bool {
    return fmt.Sprintf("%x", hash[0]) == "0" &&
        fmt.Sprintf("%x", hash[1]) == "0" &&
        fmt.Sprintf("%x", hash[2]) == "0" &&
        fmt.Sprintf("%x", hash[3]) != "0"
}
```

valid block hash

As a solid developer, he decides to test the logic with a simple unit test verifying the `isBlockHashValid()` works as expected. Andrej creates a new file `./node/miner_test.go` and programs the following function in it.

```
// If Go, you suffix your files with `_test` and prefix the
// testing functions with "Test".
//
// The first argument is the testing helper `t *testing.T`,
// automatically injected by the Go test compiler.
func TestValidBlockHash(t *testing.T) {
    // Create a random hex string starting with 6 zeroes
    hexHash := "000000fa04f816039...a4db586086168edfa"
    var hash = database.Hash{}

    // Convert it to raw bytes
    hex.Decode(hash[:], []byte(hexHash))

    // Validate the hash
    isValid := database.IsBlockHashValid(hash)
    if !isValid {
        t.Fatalf("hash '%s' with 6 zeroes should be valid", hexHash)
    }
}
```

Go ahead, run it.

>_ go test ./node -test.v -test.run ^TestValidBlockHash\$

```
==== RUN    TestValidBlockHash
--- PASS: TestValidBlockHash (0.00s)
PASS
```

GO CODING CHALLENGE

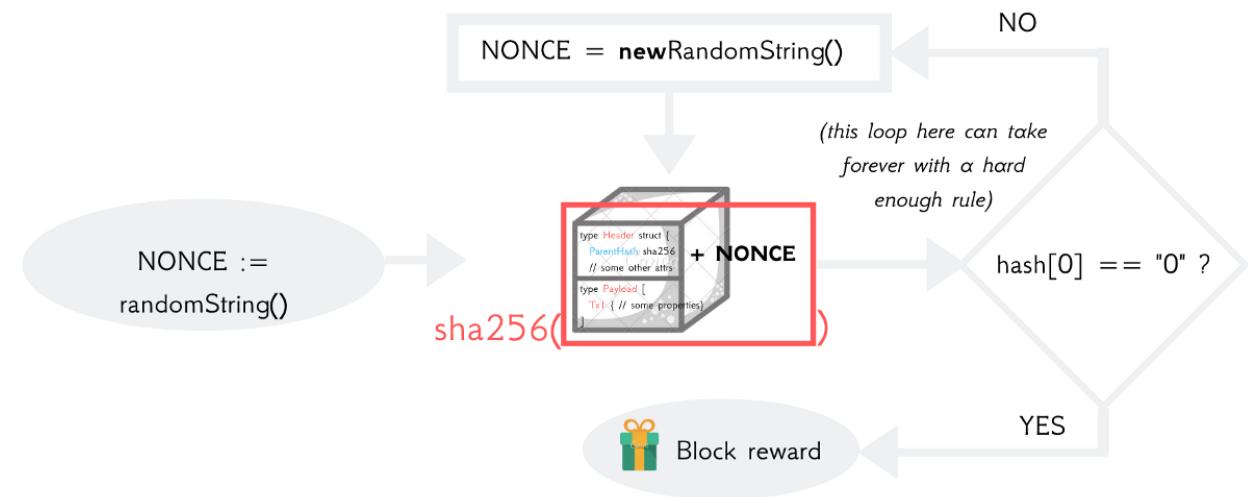
Perfect. Now, time to think.

Given the following PendingBlock structure:

```
type PendingBlock struct {
    parent database.Hash
    number uint64
    time   uint64
    txs    []database.Tx
}

func createRandomPendingBlock() PendingBlock {
    return NewPendingBlock(
        database.Hash{},
        0,
        []database.Tx{
            database.NewTx("andrey", "andrey", 3, ""),
            database.NewTx("andrey", "andrey", 700, "reward"),
        },
    )
}
```

And the following PoW design diagram:



How would you program a PoW algorithm, passing this mining test `TestMine()`?

```
func TestMine(t *testing.T) {
    pendingBlock := createRandomPendingBlock()

    // Context is used to carry deadlines, cancellation signals,
    // and other request-scoped values between processes.
    ctx := context.Background()

    minedBlock, err := Mine(ctx, pendingBlock)
    if err != nil {
        t.Fatal(err)
    }

    minedBlockHash, err := minedBlock.Hash()
    if err != nil {
        t.Fatal(err)
    }

    if !database.IsBlockHashValid(minedBlockHash) {
        t.Fatal()
    }
}
```

Do you have an idea?

Congratulation! You are becoming an excellent blockchain developer. For the developers reading this book tired after work, who want to enjoy the show, here is Andrej's version: [./node/miner.go#L24](#)⁶⁴

⁶⁴ https://github.com/web3coach/the-blockchain-bar/blob/c11_consensus/node/miner.go#L24

```
func Mine(ctx context.Context, pb PendingBlock) (Block, error) {
    // skip empty blocks
    if len(pb.txs) == 0 {
        err := fmt.Errorf("mining empty blocks is not allowed")
        return database.Block{}, err
    }

    // prepare necessary variables
    start := time.Now()
    attempt := 0
    var block database.Block
    var hash database.Hash
    var nonce uint32

    // start a loop, repeating the hash generation until
    // a valid block hash is found
    for !database.IsBlockHashValid(hash) {
        select {
        // in case someone stops the mining process
        // (e.g.: closes the program)
        case <-ctx.Done():
            fmt.Println("Mining cancelled!")

            err := fmt.Errorf("mining cancelled. %s", ctx.Err())
            return database.Block{}, err
        default:
        }
    }

    // generate a random big number
    attempt++
    nonce = generateNonce()

    // print every 1M attempt to don't spam the standard output
    if attempt%1000000 == 0 || attempt == 1 {
        fmt.Printf("Mining %d Pending TXs. Attempt: %d\n", len(pb.\n
txs), attempt)
```

```
}

// try to construct a block with this random nonce
block = database.NewBlock(
    pb.parent,
    pb.number,
    nonce,
    pb.time,
    pb.miner,
    pb.txs,
)
// hash it and literally hope for the best
blockHash, err := block.Hash()
if err != nil {
    err = fmt.Errorf("couldn't mine block. %s", err.Error())
    return database.Block{}, err
}

// fortunate to find a valid hash? stop the loop
hash = blockHash
}

fmt.Printf("\nMined new Block '%x' using Pow%:n", hash))
fmt.Printf("\tHeight: '%v'\n", block.Header.Number)
fmt.Printf("\tNonce: '%v'\n", block.Header.Nonce)
fmt.Printf("\tCreated: '%v'\n", block.Header.Time)
fmt.Printf("\tMiner: '%v'\n", block.Header.Miner)
fmt.Printf("\tParent: '%v'\n\n", block.Header.Parent.Hex())

fmt.Printf("\tAttempt: '%v'\n", attempt)
fmt.Printf("\tTime: %s\n\n", time.Since(start))

return block, nil
}
```

```
func generateNonce() uint32 {
    rand.Seed(time.Now().UTC().UnixNano())

    return rand.Uint32()
}
```

```

func Mine(ctx context.Context, pb PendingBlock) (database.Block, error) {
    if len(pb.txs) == 0 {
        return database.Block{}, fmt.Errorf("mining empty blocks is not allowed")
    }

    start := time.Now()
    attempt := 0
    var block database.Block
    var hash database.Hash
    var nonce uint32

    for !database.IsBlockHashValid(hash) {
        select {
        case <-ctx.Done():
            fmt.Println("Mining cancelled!")
            return database.Block{}, fmt.Errorf("mining cancelled. #{ctx.Err()}")
        default:
        }

        attempt++
        nonce = generateNonce()          Construct a potentially
                                         valid block

        if attempt%1000000 == 0 || attempt == 1 {
            fmt.Printf("Mining #{len(pb.txs)} Pending Tx's, Attempt: #{attempt}\n")
        }

        block = database.NewBlock(pb.parent, pb.number, nonce, pb.time, pb.txs)
        blockHash, err := block.Hash()
        if err != nil {
            return database.Block{}, fmt.Errorf("couldn't mine block. #{err.Error()}")
        }

        hash = blockHash
    }

    fmt.Printf("\nMined new Block '#{hash}' using PoW#{fs.Unicode("\U1F389")}\n")
    fmt.Printf("\tHeight: '#{pb.number}'\n")
    fmt.Printf("\tNonce: '#{nonce}'\n")
    fmt.Printf("\tCreated: '#{pb.time}'\n")
    fmt.Printf("\tParent: '#{pb.parent.Hex()}'\n\n")

    fmt.Printf("\tAttempt: '#{attempt}'\n")
    fmt.Printf("\tTime: #{time.Since(start)}\n\n")

    return block, nil
}

func generateNonce() uint32 {
    rand.Seed(time.Now().UTC().UnixNano())
    return rand.Uint32()
}

```

The code is annotated with several orange arrows and text labels:

- A large arrow points from the `for` loop back up to the `attempt++` line, labeled "Repeat mining until we find a valid hash".
- An arrow points from the `case <-ctx.Done():` block to the `attempt++` line, labeled "Context to be able to stop the loop".
- An arrow points from the `if attempt%1000000 == 0 || attempt == 1 {` block down to the `block = database.NewBlock(pb.parent, pb.number, nonce, pb.time, pb.txs)` line, labeled "Construct a potentially valid block".
- An arrow points from the `Generate a random integer` label down to the `rand.Seed` call in the `generateNonce` function.

Proof Of Work mining algorithm

Andrej crosses his fingers and runs the TDD test:

```
Mining 2 Pending TXs. Attempt: 1000000
Mining 2 Pending TXs. Attempt: 2000000
...
Mining 2 Pending TXs. Attempt: 94000000

Mined new Block '000000d92fa8...1da3' using PoW:
  Height: '0',
  Nonce: '3703266860'
  Created: '15797234404'
  Parent: '000000...00000'

  Attempt: '94157252'
  Time: 18m21s

--- PASS: TestMine (1101.35s)
```

Voila! It took 94.1 million different nonces to find a valid hash for this block! The total mining time of this block was 18 mins 21 secs on an AMD Ryzen 5 2600 3.4GHz personal computer.

The number of CPU cycles required can vary substantially, depending literally on how lucky you are.

Test the performance of your mining machine!

```
>_ go test -timeout=0 ./node -test.v -test.run ^TestMine$  
  
==== RUN TestMine  
Mining 1 Pending TXs. Attempt: 1  
Mining 1 Pending TXs. Attempt: 1000000  
...  
Mining 1 Pending TXs. Attempt: 21000000  
Mining 1 Pending TXs. Attempt: 22000000  
  
Mined new Block '00000044cf8...044dc7' using PoW|||||:  
  Height: '0'  
  Nonce: '2880469942'  
  Created: '1589880548'  
  Miner: 'andrey'  
  Parent: '000000...000000'  
  
  Attempt: '22577398'  
  Time: 4m13.429294962s  
  
--- PASS: TestMine (253.43s)  
PASS  
ok      github.com/web3coach/the-blockchain-bar/node    253.433s
```

This difficulty will be sufficient to secure the bar's blockchain network, overcome latency sync issues, and prevent database forks.

Two tasks left:

1. All nodes must propagate every new transaction across the network, so every node has the opportunity to validate it and mine the next block to create a truly decentralized blockchain.
2. If another node mines the next block faster, all other nodes solving the same cryptographic puzzle should immediately stop mining and wasting energy. They should verify the new block hash, congratulate the winner and synchronize the new block.

In order to resolve Task 1, Andrej reuses the sync-status behavior between nodes and adds a new attribute "Pending TXs" containing the current's node transactions that are yet to be mined and included in the next blocks.

```
type TxAddRes struct {
    // return confirmation not block hash because
    // the mining takes sometimes several minutes
    // and the TX should be distributed to all nodes
    // so everyone has equal chance of mining the block
    Success bool `json:"success"`
}
```

```
type StatusRes struct {
    Hash         database.Hash      `json:"block_hash"`
    Number       uint64            `json:"block_number"`
    KnownPeers  map[string]PeerNode `json:"peers_known"`

    // exchange pending TXs as part of the periodic Sync() interval
    PendingTXs []database.Tx      `json:"pending_txs"`
}
```

Andrej also tweaks several things around the codebase and implements the following node's functionality responsible for **stopping the mining activity in case a different blockchain node mines the next block faster**:

```
func (n *Node) mine(ctx context.Context) error {
    var miningCtx context.Context
    var stopCurrentMining context.CancelFunc

    ticker := time.NewTicker(time.Second * miningIntervalSeconds)

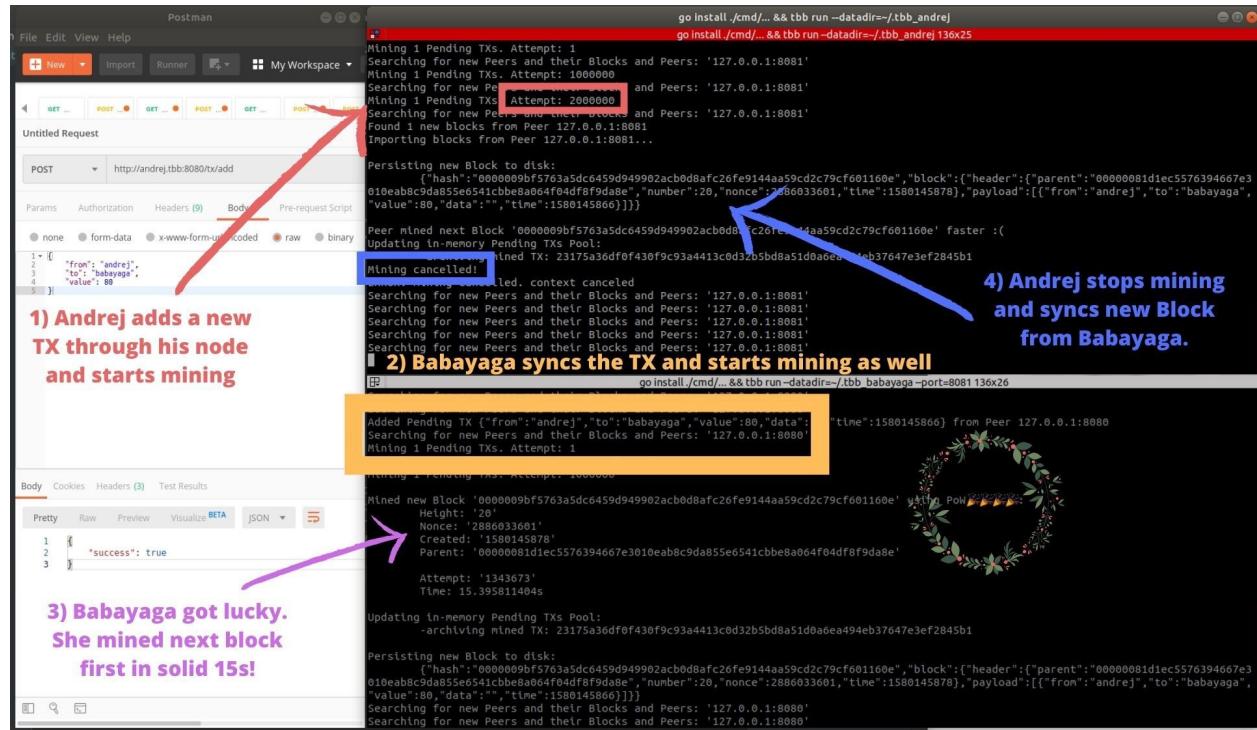
    for {
        select {
        case <-ticker.C:
            go func() {
                if len(n.pendingTXs) > 0 && !n.isMining {
                    n.isMining = true

                    miningCtx, stopCurrentMining = context.WithCancel(ctx)
                    err := n.minePendingTXs(miningCtx)
                    if err != nil {
                        fmt.Printf("ERROR: %s\n", err)
                    }
                }
            }()
        case block, _ := <-n.newSyncedBlocks:
            if n.isMining {
                blockHash, _ := block.Hash()
                fmt.Printf("\nPeer mined next Block '%s' faster :(\n", blockHash.Hex())

                n.removeMinedPendingTXs(block)
                stopCurrentMining()
            }
        case <-ctx.Done():
            ticker.Stop()
            return nil
        }
    }
}
```

stop mining on synced block

Blockchain consensus beauty in action. Full screen to see the details⁶⁵



mining + synced test

65 https://web3coach-public.s3.eu-central-1.amazonaws.com/img/mining_p2p.png

The Ethereum's Mining Algorithm: Ethash

The `Mine()` function Andrej implemented is extremely simplistic and for educational purposes only. The mining algorithm in Bitcoin and Ethereum is much more complex and designed to defend against different technological and economical attacks. Let's briefly explore Ethash, so you have an idea how complex is to program production level blockchain components.

Quoting from Ethereum's YellowPaper⁶⁶, Page 16:

Mining Proof-of-Work. The mining proof-of-work (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the determination of some token value n . It is utilised to enforce the blockchain security by giving meaning and credence to the notion of difficulty (and, by extension, total difficulty). However, since mining new blocks comes with an attached reward, the proof-of-work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism.

For both reasons, there are two important goals of the proof-of-work function; firstly, it should be as accessible as possible to as many people as possible. The requirement of, or reward from, specialised and uncommon hardware should be minimised. This makes the distribution model as open as

⁶⁶ <https://ethereum.github.io/yellowpaper/paper.pdf>

possible, and, ideally, makes the act of mining a simple swap from electricity to Ether at roughly the same rate for anyone around the world.

Secondly, it should not be possible to make super-linear profits, and especially not so with a high initial barrier. Such a mechanism allows a well-funded adversary to gain a troublesome amount of the network's total mining power and as such gives them a super-linear reward (thus skewing distribution in their favour) as well as reducing the network security.

One plague of the Bitcoin world is ASICs. These are specialised pieces of **compute hardware that exist only to do a single task.** In Bitcoin's case the task is the **SHA256 hash function.** While ASICs exist for a proof-of-work function, both goals are placed in jeopardy. Because of this, a proof-of-work function that is ASIC-resistant (i.e. difficult or economically inefficient to implement in specialised compute hardware) has been identified as the proverbial silver bullet.

Two directions exist for ASIC resistance; firstly make it sequential memory-hard, i.e. engineer the function such that the determination of the nonce requires a lot of memory and bandwidth such that the memory cannot be used in parallel to discover multiple nonces simultaneously. The second is to make the type of computation it would need to do general-purpose; the meaning of "specialised hardware" for a general-purpose task set is, naturally, general purpose hardware and as such commodity desktop computers are likely to be pretty close to "specialised hardware" for

the task. For Ethereum 1.0 we have chosen the first path.

Damn, the YellowPapers are heavy. That was a lot of information, Andrej tells himself.

To put things into a perspective and cut to the chase, you can do crypto mining using:

1. CPU (central processing unit) - the slowest option. CPU was designed to perform software's business logic. Follow if, else instructions and hardcore context switching.
2. GPU (graphics processing unit) - much faster option. GPU can follow instructions and switch between tasks, but GPUs are excellent in doing single, repetitive work like rendering pixels, OR **calling the same hash() function until a valid hash is generated satisfying the block's difficulty level.**
3. ASIC (application-specific integrated circuit) - the most powerful option. Dedicated hardware to perform a specific instructions. In the crypto world, call a hash() function millions of times.

The Ethereum community wanted to prevent miners from using ASIC machines for the sake of decentralization. A small group of people with access to specialized hardware gain an unfair mining advantage. A noble goal, and I know what you are thinking. Seems like a complicated problem. Yes. However, they came up with an interesting solution.

Ethereum designed an ASIC resistant Proof of Work algorithm called **Ethash** by **bounding the hashing performance to memory.**

Therefore, attempting to find a nonce using parallel processing is capped by the memory bandwidth, not the processing units count.

Memory bound refers to a situation in which the time to complete a given computational problem is decided primarily by the amount of memory required to hold the working data. This is in contrast to algorithms that are compute-bound, where the number of elementary computation steps is the deciding factor.

Ethash requirements

Source: <https://eth.wiki/concepts/ethash/design-rationale>⁶⁷.

1. **IO saturation:** The algorithm should consume nearly the entire available memory access bandwidth (this is a strategy toward achieving ASIC resistance, the argument being that commodity RAM, especially in GPUs, is much closer to the theoretical optimum than commodity computing capacity)
2. **GPU friendliness:** We try to make it as easy as possible to mine with GPUs. Targeting CPUs is almost certainly impossible, as potential specialization gains are too great, and there do exist criticisms of CPU-friendly algorithms that they are vulnerable to botnets, so we target GPUs as a compromise.
3. **Light client verifiability:** a light client should be able to verify a round of mining in under 0.01 seconds on a desktop in C, and under 0.1 seconds in Python or Javascript, with at most 1 MB of memory (but exponentially increasing)
4. **Light client slowdown:** the process of running the algorithm with a light client should be much slower than the process with a full client, to the point that the light client algorithm is not an economically viable route toward making a mining implementation, including via specialized hardware.
5. **Light client fast startup:** a light client should be able to become fully operational and able to verify blocks within 40 seconds in Javascript.

⁶⁷ <https://eth.wiki/concepts/ethash/design-rationale>

Ethash parameters rationale

1. A 32 MB cache was chosen because a smaller cache would allow for an ASIC to be produced far too easily using the light-evaluation method. The 32 MB cache still requires a very high bandwidth of cache reading, whereas a smaller cache could be much more easily optimized. A larger cache would lead to the algorithm being too hard to evaluate with a light client.
2. 256 parents per DAG item was chosen in order to ensure that time-memory tradeoffs can only be made at a worse-than-1:1 ratio.
3. The ~4.5 GB DAG size was chosen in order to require a level of memory larger than the size at which most specialized memories and caches are built, but still small enough for ordinary computers to be able to mine with it.
4. The ~0.73x per year growth level was chosen to roughly be balanced with Moore's law increases at least initially (exponential growth has a risk of overshooting Moore's law, leading to a situation where mining requires very large amounts of memory and ordinary GPUs are no longer usable for mining).
5. 64 accesses was chosen because a larger number of accesses would lead to light verification taking too long, and a smaller number would mean that the bulk of the time consumption is the SHA3 at the end, not the memory reads, making the algorithm not so strongly IO-bound.
6. The epoch length cannot be infinite (ie. constant dataset) because then the algorithm could be optimized via ROM, and very long epoch lengths make it easier to create memory which is designed to be updated very infrequently and only read

often. Excessively short epochs would increase barriers to entry as weak machines would need to spend much of their time on a fixed cost of updating the dataset. The epoch length can probably be reduced or increased substantially if design considerations require it.

7. The cache size and dataset size is prime in order to help mitigate the risk of cycles appearing in dataset item generation or mining.

Ethash process execution

1. There exists a seed which can be computed for each block by scanning through the block headers up until that point.
2. From the seed, one can compute a 32 MB pseudorandom cache. Light clients store the cache.
3. From the cache, we can generate a ~4.5 GB dataset, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time.
4. Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache.
5. The large dataset is updated once every 30000 blocks, so the vast majority of a miner's effort will be reading the dataset, not making changes to it.

Now you understand why the latest Nvidia and AMD graphic cards with high GPU VRAM are so expensive. Second sad news is that ASIC resistance was broken in 2018. Fortunately, Ethereum is moving to Proof of Stake in 2022, and the ASIC mining will be a thing of the past. Together with Bitcoin. A rather cheeky comment, I know.

Ethash codebase

Clone the go-ethereum codebase from Github and explore the consensus/ethash package. Warning: prepare a bucket of coffee, the Ethash programming logic is HEAVY to read.

The Ethash implementation is extensive and complicated. Andrej could write an entire book just about the concept itself, therefore here are at least 3 functions (generateCache, generateDatasetItem, hashimoto) copied from the go-ethereum codebase to paint you a picture on how the core Ethereum Geth team implemented the ASIC resistance mining algorithm in production:

[github.com/ethereum/go-ethereum/tree/master/consensus/ethash⁶⁸](https://github.com/ethereum/go-ethereum/tree/master/consensus/ethash)

```
// generateCache creates a verification cache of a given size for
// an input seed.
//
// The cache production process involves first sequentially
// filling up 32 MB of memory, then performing two passes
// of Sergio Demian Lerner's RandMemoHash algorithm from
// Strict Memory Hard Hashing Functions (2014). The output is a
// set of 524288 64-byte values.
//
// This method places the result into dest in machine byte order.
func generateCache(dest []uint32, epoch uint64, seed []byte) {
    // Print some debug logs to allow analysis on low end devices
    logger := log.New("epoch", epoch)

    start := time.Now()
    defer func() {
```

⁶⁸<https://github.com/ethereum/go-ethereum/tree/master/consensus/ethash>

```
elapsed := time.Since(start)

logFn := logger.Debug
if elapsed > 3*time.Second {
    logFn = logger.Info
}
logFn("Generated ethash verification cache", "elapsed", common.Pre\
ttyDuration(elapsed))
}()

// Convert our destination slice to a byte buffer
var cache []byte
cacheHdr := (*reflect.SliceHeader)(unsafe.Pointer(&cache))
dstHdr := (*reflect.SliceHeader)(unsafe.Pointer(&dest))
cacheHdr.Data = dstHdr.Data
cacheHdr.Len = dstHdr.Len * 4
cacheHdr.Cap = dstHdr.Cap * 4

// Calculate the number of theoretical rows
// (we'll store in one buffer nonetheless)
size := uint64(len(cache))
rows := int(size) / hashBytes

// Start a monitoring goroutine to report progress on low end devices
var progress uint32

done := make(chan struct{})
defer close(done)

go func() {
    for {
        select {
        case <-done:
            return
        case <-time.After(3 * time.Second):
            logger.Info("Generating ethash verification cache", "percentag\
e", atomic.LoadUint32(&progress)*100/uint32(rows)/(cacheRounds+1), "el\
e")
        }
    }
}
```

```
apsed", common.PrettyDuration(time.Since(start)))
    }
}
}()

// Create a hasher to reuse between invocations
keccak512 := makeHasher(sha3.NewLegacyKeccak512())

// Sequentially produce the initial dataset
keccak512(cache, seed)
for offset := uint64(hashBytes); offset < size; offset += hashBytes {
    keccak512(cache[offset:], cache[offset-hashBytes:offset])
    atomic.AddUint32(&progress, 1)
}

// Use a low-round version of randmemohash
temp := make([]byte, hashBytes)

for i := 0; i < cacheRounds; i++ {
    for j := 0; j < rows; j++ {
        var (
            srcOff = ((j - 1 + rows) % rows) * hashBytes
            dstOff = j * hashBytes
            xorOff = (binary.LittleEndian.Uint32(cache[dstOff:])) % uint32(\nrows)) * hashBytes
        )
        bitutil.XORBytes(temp, cache[srcOff:srcOff+hashBytes], cache[xor\\
Off:xorOff+hashBytes])
        keccak512(cache[dstOff:], temp)

        atomic.AddUint32(&progress, 1)
    }
}

// Swap the byte order on big endian systems and return
if !isLittleEndian() {
    swap(cache)
}
}
```

```
// generateDatasetItem combines data from
// 256 pseudorandomly selected cache nodes,
// and hashes that to compute a single dataset node.
func generateDatasetItem(cache []uint32, index uint32, keccak512 hash\er) []byte {
    // Calculate the number of theoretical rows
    // (we use one buffer nonetheless)
    rows := uint32(len(cache) / hashWords)

    mix := make([]byte, hashBytes)

    binary.LittleEndian.PutUint32(mix, cache[(index%rows)*hashWords]^ind\ex)
    for i := 1; i < hashWords; i++ {
        binary.LittleEndian.PutUint32(mix[i*4:], cache[(index%rows)*hashWo\rds+uint32(i)])
    }
    keccak512(mix, mix)

    intMix := make([]uint32, hashWords)
    for i := 0; i < len(intMix); i++ {
        intMix[i] = binary.LittleEndian.Uint32(mix[i*4:])
    }
    // fnv it with a lot of random cache nodes based on index
    for i := uint32(0); i < datasetParents; i++ {
        parent := fnv(index^i, intMix[i%16]) % rows
        fnvHash(intMix, cache[parent*hashWords:])
    }
    // Flatten the uint32 mix into a binary one and return
    for i, val := range intMix {
        binary.LittleEndian.PutUint32(mix[i*4:], val)
    }
    keccak512(mix, mix)
    return mix
}
```

```
// hashimoto aggregates data from the full dataset in order to
// produce our final value for a particular header hash and nonce.
func hashimoto(hash []byte, nonce uint64, size uint64, lookup func(ind\
ex uint32) []uint32) ([]byte, []byte) {
    // Calculate the number of theoretical rows (we use one buffer nonet\
heless)
    rows := uint32(size / mixBytes)

    // Combine header+nonce into a 64 byte seed
    seed := make([]byte, 40)
    copy(seed, hash)
    binary.LittleEndian.PutUint64(seed[32:], nonce)

    seed = crypto.Keccak512(seed)
    seedHead := binary.LittleEndian.Uint32(seed)

    // Start the mix with replicated seed
    mix := make([]uint32, mixBytes/4)
    for i := 0; i < len(mix); i++ {
        mix[i] = binary.LittleEndian.Uint32(seed[i%16*4:])
    }
    // Mix in random dataset nodes
    temp := make([]uint32, len(mix))

    for i := 0; i < loopAccesses; i++ {
        parent := fnv(uint32(i)^seedHead, mix[i%len(mix)]) % rows
        for j := uint32(0); j < mixBytes/hashBytes; j++ {
            copy(temp[j*hashWords:], lookup(2*parent+j))
        }
        fnvHash(mix, temp)
    }
    // Compress mix
    for i := 0; i < len(mix); i += 4 {
        mix[i/4] = fnv(fnv(fnv(mix[i], mix[i+1]), mix[i+2]), mix[i+3])
    }
    mix = mix[:len(mix)/4]
```

```
digest := make([]byte, common.HashLength)
for i, val := range mix {
    binary.LittleEndian.PutUint32(digest[i*4:], val)
}
return digest, crypto.Keccak256(append(seed, digest...))
}
```

The Slow Elephant in the Bitcoin PoW Room

Saturday, December 28.

All the mining magic debunked!

What are the consequences of Proof of Work design? Proof of Work is slow by design and can't scale. Period.

The on-chain transaction processing capacity of the bitcoin network is limited by the average block creation time of 10 minutes and the original block size limit of 1 megabyte. These jointly constrain the network's throughput. The transaction processing capacity maximum estimated using an average or median transaction size is between 3.3 and 7 transactions per second.

Inserting a transaction into a database takes up to 15 mins; therefore, it's not suitable as a real-time payment system. Imagine if your MySQL write query would take 15 to 60 minutes, and there is nothing you can do about it. Hence, why Bitcoin pivoted from "real-time electronic cash" to "store of value." Secure, as you need a massive amount of power, and anyone can join the network and contribute to the security model with even more hash power but also slow for the same reason.

Fortunately between Andrej's bar customers are names such as BabaYaga and Caesar. They drink all night, no rush to pay, security first.

[https://en.wikipedia.org/wiki/Bitcoin_scalability_problem⁶⁹](https://en.wikipedia.org/wiki/Bitcoin_scalability_problem)

⁶⁹ https://en.wikipedia.org/wiki/Bitcoin_scalability_problem

XRP and Proof of Authority

Saturday, December 28.

XRP ledger is based on Proof of Authority (PoA) consensus and can scale to thousands of transactions per second. A new block is sealed every 4 seconds. PoA is criticized because you have to trust a predefined set of validators and sacrifice few levels of decentralization. The trade-off for electricity consumption of a standard server and real-time transaction performance.

Another important consensus design is to decide what happens in case the consensus can't be reached. In the Bitcoin, Ethereum, and now TBB case, the database forks into different histories. In the XRP case, the network stops operating (halts) until consensus is resolved. This is an intended behavior to prevent financial entities from acting based on an invalid state.

You can't have a safe, live, and fault-tolerant asynchronous system. You must choose only 2 out of 3 properties. Read this excellent paper: “[Impossibility of Distributed Consensus with One Faulty Process](https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf)”⁷⁰ to learn why.

⁷⁰ <https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf>

Ethereum, Proof of Stake, Sharding

Saturday, December 28.

Ethereum is moving to a consensus mechanism called Proof of Stake (PoS) from Proof of Work (PoW) after 6 years to scale and become energy efficient.

It requires users to stake their ETH to become a validator in the network. Validators are responsible for the same thing as miners in Proof of Work: ordering transactions and creating new blocks so that all nodes can agree on the state of the network.

Proof of Stake comes with a number of improvements:

- significant energy efficiency - consensus security is not based on electricity consumption
- lower barriers to entry - reduced hardware requirements, you don't need elite hardware to stand a chance of creating new blocks (you need 32 ETH or participate in a staking pool)
- stronger immunity to centralization - more nodes in the network
- **stronger support for shard chains - a key upgrade in scaling the Ethereum network**

How is the network security achieved? By making it expensive to act maliciously. If validators act maliciously or aren't reliable (crash/go offline), they get slashed and lose a part of their stake.

The Merge

There is a functional PoS chain called the Beacon Chain that has been running since December 2020 that is demonstrating the viability of protocol. The merge refers to the point in time when Ethereum leaves PoW behind and fully adopts PoS. The merge is expected to happen ~Q2 2022.

The merge to PoS could result in a 99.95% reduction in total energy use, being ~2000x more efficient. The energy expenditure of Ethereum will be roughly equal to the cost of running a home computer for each node on the network.

Ethereum's current transaction rate (15 transactions per second) will be increased by at least 64x (the number of shards), not accounting for additional optimization from rollups. A realistic estimate for post-merge, sharded Ethereum with rollups is 25,000 - 100,000 transactions per second.

<https://ethereum.org/en/eth2/beacon-chain/>⁷¹

<https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/>⁷²

⁷¹<https://ethereum.org/en/eth2/beacon-chain/>

⁷²<https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/>

How are Bitcoins / Ethers created?

Sunday, December 29.

Out of thin binary air. Given this book is written in Go, check out this part of Ethereum consensus production [source code⁷³](#):

```
// Block reward of 3 ETH (400$ in December 2019) for a mined block  
  
reward = big.NewInt(3e+18)  
state.AddBalance(miner, reward)
```

Magic!

Andrej gave it a thought and decided he will change the way rewards on the TBB network are handled. At the moment, Andrej receives 100 TBB tokens every day. Regardless of the network usage. He decides to adopt the decentralized industry mentality and reward only the miner who created a new block.

⁷³<https://github.com/ethereum/go-ethereum/blob/v1.9.10/consensus/ethash/consensus.go#L42>

Programming Bitcoin Mining Reward in 4 Steps

Sunday Evening, December 29.

Step 1

Andrej adds a new block's header attribute “**miner**” to define who mined the block:

```
type BlockHeader struct {
    Parent Hash      `json:"parent"`
    Number uint64   `json:"number"`
    Nonce  uint32    `json:"nonce"`
    Time   uint64    `json:"time"`

    // new attribute -> who mined this block and gets reward
    Miner Account `json:"miner"`
}
```

Step 2

He adjusts the State logic and increase the miner's balance every time the mined block is persisted into the database ledger:

```
func applyBlock(b Block, s *State) error {
    // ...
    if !IsBlockHashValid(hash) {
        return fmt.Errorf("invalid block hash %x", hash)
    }

    err = applyTXs(b.TXs, s)
    if err != nil {
        return err
    }

    // if the block and its TXs are valid, reward the miner
    s.Balances[b.Header.Miner] += BlockReward

    return nil
}
```

Step 3

Andrej also adds a new command flag `--miner` to configure what's the account of the miner running the node:

```
func runCmd() *cobra.Command {
    var runCmd = &cobra.Command{
        Use:      "run",
        Short:    "Launches the TBB node and its HTTP API.",
        Run: func(cmd *cobra.Command, args []string) {
            // make miner configurable via a CLI flag on boot-up
            miner, _ := cmd.Flags().GetString(flagMiner)

            fmt.Println("Launching TBB node and its HTTP API...")
            // ...

            n := node.New(
                getDataDirFromCmd(cmd),
                ip,
                port,
                database.NewAccount(miner), // configure node's miner
                bootstrap,
            )
        },
    }
}
```

Step 4

And configures every new pending block on this node with the miner's account - from previously defined --miner flag:

```
func (n *Node) minePendingTXs(ctx context.Context) error {
    blockToMine := NewPendingBlock(
        n.state.LatestBlockHash(),
        n.state.NextBlockNumber(),
        n.info.Account, // configure the potential block miner
        n.getPendingTXsAsArray(),
    )
}
```

Writing Tests for Proof of Work

Friday Evening, December 27.

You know the rule. The Jira task is not done without tests. Although, there will be a dedicated chapter to writing tests for distributed systems, Andrej is determined to ensure the TBB blockchain works as expected right now.

The `TestMine()` unit test did its job but this codebase is growing substantially and Andrej needs something more extensive. He needs an integration test - testing all pieces together.

Andrej creates a new file `./node/node_integration_test.go` and starts with the first, simple, warm-up test - testing whenever the Node can start and stop without any issue.

TestNode_Run

```
func getTestDataDirPath() string {
    return filepath.Join(os.TempDir(), ".tbb_test")
}

func TestNode_Run(t *testing.T) {
    // Remove the test directory if it already exists
    datadir := getTestDataDirPath()
    err := fs.RemoveDir(datadir)
    if err != nil {
        t.Fatal(err)
    }

    // Construct a new Node instance
    n := New(
        datadir,
        "127.0.0.1",
        8085,
        database.NewAccount("andrey"),
        PeerNode{},
    )

    // Define a context with timeout for this test - meaning that
    // the Node.Run() will only be running for 5s
    ctx, _ := context.WithTimeout(context.Background(), time.Second*5)
    err = n.Run(ctx)
    if err.Error() != "http: Server closed" {
        // assert expected behaviour
        t.Fatal("node server was suppose to close after 5s")
    }
}
```

```
>_ go test -timeout=0 -count=1 ./node -test.v -test.run
^TestNode_Run$  
  
==== RUN TestNode_Run  
Listening on: 127.0.0.1:8085  
Blockchain state:  
  - height: 0  
  - hash: 0000000...000000  
--- PASS: TestNode_Run (5.00s)  
PASS  
ok      github.com/web3coach/the-blockchain-bar/node      5.004s
```

What's the `-count=1`? A trick to invalid test cache. Andrej already suggested adding the `-no-cache` flag instead on [Go's Github Issue #24573](#).⁷⁴

⁷⁴ <https://github.com/golang/go/issues/24573#issuecomment-574627439>

TestNode_Mining

This test will be a bit more complicated. Andrej wants to test that the TB node is able to mine transactions into different blocks and that Mempool, State, Miner and the Node components play nicely together.

Andrej attended an Agile workshop the other day so he starts with the following user story.

Mining test requirements

As a Node,

I store all pending TXs in memory (Mempool),
So I can seal them into blocks, mine them, periodically.

As a Node,

If I am already mining a block and a new TX comes,
I will keep this TX in memory (Mempool),
So I can include it in the next block when the mining interval
restarts.

As a Node,

If I mine a TX from Mempool into a block,
I will remove the TX from the Mempool,
So it's not mined twice.

Great. Let's write the test.

```
func TestNode_Mining(t *testing.T) {
    // Remove the test directory if it already exists
    datadir := getTestDataDirPath()
    err := fs.RemoveDir(datadir)
    if err != nil {
        t.Fatal(err)
    }

    // Required for AddPendingTX() to describe
    // from what node the TX came from (local node in this case)
    nInfo := NewPeerNode(
        "127.0.0.1",
        8085,
        false,
        database.NewAccount(""),
        true,
    )

    // Construct a new Node instance and configure
    // Andrej as a miner
    n := New(
        datadir,
        nInfo.IP,
        nInfo.Port,
        database.NewAccount("andrey"),
        nInfo,
    )

    // Allow the mining to run for 30 mins, in the worst case
    ctx, closeNode := context.WithTimeout(
        context.Background(),
        time.Minute * 30,
    )

    // Schedule a new TX in 3 seconds from now, in a separate thread
    // because the n.Run() few lines below is a blocking call
```

```
go func() {
    time.Sleep(time.Second * miningIntervalSeconds / 3)
    tx := database.NewTx("andrey", "babayaga", 1, "")

    // Add it to the Mempool
    _ = n.AddPendingTX(tx, nInfo)
}()

// Schedule a new TX in 12 seconds from now simulating
// that it came in while the first TX is being mined
go func() {
    time.Sleep(time.Second * miningIntervalSeconds + 2)
    tx := database.NewTx("andrey", "babayaga", 2, "")

    _ = n.AddPendingTX(tx, nInfo)
}()

go func() {
    // Periodically check if we mined the 2 blocks
    ticker := time.NewTicker(10 * time.Second)

    for {
        select {
        case <-ticker.C:
            // 2 blocks mined as expected? (height: 0 and 1)
            if n.state.LatestBlock().Header.Number == 1 {
                closeNode()
                return
            }
        }
    }
}()

// Run the node, mining and everything
// in a blocking call (hence the go-routines before)
_ = n.Run(ctx)
```

```
// Assert the test result
if n.state.LatestBlock().Header.Number != 1 {
    t.Fatal("2 pending TX not mined into 2 under 30m")
}
}
```

>_ go test -timeout=0 -count=1 ./node -test.v -test.run
^TestNode_Mining\$

```
==== RUN TestNode_Mining
Listening on: 127.0.0.1:8085
Blockchain state:
- height: 0
- hash: 000000...000000
```

// FIRST TX IS ADDED TO THE MEMPOOL

```
Added Pending TX {"from":"andrey","to":"babayaga","value":1,"data":"","\n"time":1589968805} from Peer 127.0.0.1:8085
```

// MINING STARTS

Mining 1 Pending TXs. Attempt: 1

// SECOND TX IS ADDED TO THE MEMPOOL

```
Added Pending TX {"from":"andrey","to":"babayaga","value":2,"data":"","\n"time":1589968821} from Peer 127.0.0.1:8085
```

// ATTEMPTING TO SOLVE THE CRYPTO PUZZLE
// FOR THE FIRST TX

```
Mining 1 Pending TXs. Attempt: 1000000
Mining 1 Pending TXs. Attempt: 2000000
Mining 1 Pending TXs. Attempt: 3000000
```

```
Mined new Block '00000090fcc9...135e4' using PoW::SHA256:
```

```
Height: '0'
Nonce: '3038181196'
Created: '1589968811'
Miner: 'andrey'
Parent: '000000...00000'
```

```
Attempt: '3622536'
Time: 41.695038118s
```

```
// TX MINDED -> REMOVE IT FROM MEMPOOL
```

```
Updating in-memory Pending TXs Pool:
-archiving mined TX: bc3f5a...a13bc4
```

```
Persisting new Block to disk:
```

```
{"hash": "00000090fcc9...135e4", "block": {"header": {"parent": "000000\000000", "number": 0, "nonce": 3038181196, "time": 1589968811, "miner": "andrey"}, "payload": [{"from": "andrey", "to": "babayaga", "value": 1, "data": "", "time": 1589968805}]}}
```

```
// START MINING SECOND TX
```

```
Mining 1 Pending TXs. Attempt: 1
Mining 1 Pending TXs. Attempt: 1000000
Mining 1 Pending TXs. Attempt: 2000000
...
Mining 1 Pending TXs. Attempt: 14000000
```

```
Mined new Block '000000c6e9aa...a7317' using PoW::SHA256:
```

```
Height: '1'
Nonce: '826980728'
```

```
Created: '1589968861'  
Miner: 'andrey'  
Parent: '00000090fcc9...135e4'
```

```
Attempt: '14962379'  
Time: 2m57.521435463s
```

```
Updating in-memory Pending TXs Pool:  
-archiving mined TX: 6eb7ea...17bf3c
```

```
Persisting new Block to disk:
```

```
{"hash": "000000c6e9aa...a7317", "block": {"header": {"parent": "000000\\  
90fcc9...135e4", "number": 1, "nonce": 826980728, "time": 1589968861, "miner": "\\  
:andrey"}, "payload": [{"from": "andrey", "to": "babayaga", "value": 2, "data": "\\  
:", "time": 1589968821}]}}
```

```
--- PASS: TestNode_Mining (240.00s)
```

```
PASS
```

```
ok      github.com/web3coach/the-blockchain-bar/node    240.005s
```

TestNode_MiningStopsOnNewSyncedBlock

If you thought the previous test was problematic, wait until this one. This particular test nicely demonstrates how tricky can blockchain be. Also keep in mind, these first 12 Chapters are skipping a lot of complexity and details to be suitable for beginners - developers new to the blockchain ecosystem.

Synced blocks test requirements

As a BabaYaga Node,

I am mining a TX1 and a TX2 into a new block,
If the sync() algorithm finds a new block containing TX1 from Andrej,
I will stop mining my block,
I will verify and append Andrej's block into my database,
I will archive the TX1,
I will re-start the mining with attempt to create the next block,
I will include only the un-mined, pending TX2 in it.

As Andrej

I mined the TX1 faster than BabaYaga,
So I will get a block reward.

As BabaYaga

I mined the TX2 faster than Andrej,
So I will get a block reward.

```
func TestNode_MiningStopsOnNewSyncedBlock(t *testing.T) {
    // Remove the test directory if it already exists
    datadir := getTestDataDirPath()
    err := fs.RemoveDir(datadir)
    if err != nil {
        t.Fatal(err)
    }

    // Required for AddPendingTX() to describe
    // from what node the TX came from (local node in this case)
    nInfo := NewPeerNode(
        "127.0.0.1",
        8085,
        false,
        database.NewAccount(""),
        true,
    )

    andrejAcc := database.NewAccount("andrej")
    babayagaAcc := database.NewAccount("babayaga")

    n := New(datadir, nInfo.IP, nInfo.Port, babayagaAcc, nInfo)

    // Allow the test to run for 30 mins, in the worst case
    ctx, closeNode := context.WithTimeout(
        context.Background(),
        time.Minute * 30,
    )

    tx1 := database.NewTx("andrej", "babayaga", 1, "")
    tx2 := database.NewTx("andrej", "babayaga", 2, "")
    tx2Hash, _ := tx2.Hash()

    // Pre-mine a valid block without running the `n.Run()``
    // with Andrej as a miner who will receive the block reward,
    // to simulate the block came on the fly from another peer
```

```
validPreMinedPb := NewPendingBlock(
    database.Hash{},
    0,
    andrejAcc,
    []database.Tx{tx1},
)

validSyncedBlock, err := Mine(ctx, validPreMinedPb)
if err != nil {
    t.Fatal(err)
}

// Add 2 new TXs into the BabaYaga's node
go func() {
    time.Sleep(time.Second * (miningIntervalSeconds - 2))

    err := n.AddPendingTX(tx1, nInfo)
    if err != nil {
        t.Fatal(err)
    }

    err = n.AddPendingTX(tx2, nInfo)
    if err != nil {
        t.Fatal(err)
    }
}()

// Once BabaYaga is mining the block, simulate that
// Andrej mines the block with TX1 in it faster
go func() {
    time.Sleep(time.Second * (miningIntervalSeconds + 2))
    if !n.isMining {
        t.Fatal("should be mining")
    }

    _, err := n.state.AddBlock(validSyncedBlock)
```

```
if err != nil {
    t.Fatal(err)
}
// Mock the Andrej's block came from a network
n.newSyncedBlocks <- validSyncedBlock

time.Sleep(time.Second * 2)
if n.isMining {
    t.Fatal("synced block should have canceled mining")
}

// Mined TX1 by Andrej should be removed from the Mempool
_, onlyTX2IsPending := n.pendingTXs[tx2Hash.Hex()]

if len(n.pendingTXs) != 1 && !onlyTX2IsPending {
    t.Fatal("TX1 should be still pending")
}

time.Sleep(time.Second * (miningIntervalSeconds + 2))
if !n.isMining {
    t.Fatal("should attempt to mine TX1 again")
}
}()

go func() {
    // Regularly check whenever both TXs are now mined
ticker := time.NewTicker(time.Second * 10)

    for {
        select {
        case <-ticker.C:
            if n.state.LatestBlock().Header.Number == 1 {
                closeNode()
                return
            }
        }
    }
}
```

```
        }

    }()

go func() {
    time.Sleep(time.Second * 2)

    // Take a snapshot of the DB balances
    // before the mining is finished and the 2 blocks
    // are created.
    startingAndrejBalance := n.state.Balances[andrejAcc]
    startingBabaYagaBalance := n.state.Balances[babayagaAcc]

    // Wait until the 30 mins timeout is reached or
    // the 2 blocks get mined and
    // the closeNode() is triggered
    <-ctx.Done()

    // Query balances again
    endAndrejBalance := n.state.Balances[andrejAcc]
    endBabaYagaBalance := n.state.Balances[babayagaAcc]

    // In TX1 Andrej transferred 1 TBB token to BabaYaga
    // In TX2 Andrej transferred 2 TBB tokens to BabaYaga
    expectedEndAndrejBalance :=
        startingAndrejBalance -
        tx1.Value
        - tx2.Value
        + database.BlockReward

    expectedEndBabaYagaBalance :=
        startingBabaYagaBalance +
        tx1.Value +
        tx2.Value +
        database.BlockReward

    if endAndrejBalance != expectedEndAndrejBalance {
```

```
t.Fatalf(
    "Andrej expected end balance is %d not %d",
    expectedEndAndrejBalance,
    endAndrejBalance,
)
}

if endBabaYagaBalance != expectedEndBabaYagaBalance {
    t.Fatalf(
        "BabaYaga expected end balance is %d not %d",
        expectedEndBabaYagaBalance,
        endBabaYagaBalance,
    )
}

t.Logf("Before Andrej: %d TBB", startingAndrejBalance)
t.Logf("Before BabaYaga: %d TBB", startingBabaYagaBalance)
t.Logf("After Andrej: %d TBB", endAndrejBalance)
t.Logf("After BabaYaga: %d TBB", endBabaYagaBalance)
}()

_ = n.Run(ctx)

if n.state.LatestBlock().Header.Number != 1 {
    t.Fatal("2 pending TX not mined into 2 blocks under 30m")
}

if len(n.pendingTXs) != 0 {
    t.Fatal("no pending TXs should be left to mine")
}
}
```

```
>_ go test -timeout=0 -count=1 ./node -test.v -test.run
^TestNode_MiningStopsOnNewSyncedBlock$  
  
==== RUN TestNode_MiningStopsOnNewSyncedBlock  
// PRE-MINED BLOCK TO SIMULATE, MOCK A BLOCK  
// WAS RECEIVED FROM ANOTHER PEER FROM THE NETWORK  
  
Mining 1 Pending TXs. Attempt: 1  
Mining 1 Pending TXs. Attempt: 15000000  
  
Mined new Block '0000000223930...99224' using PoW|||||:  
  Height: '0'  
  Nonce: '3707911869'  
  Created: '1589993912'  
  Miner: 'andrey' // Andrey will get block reward for this one  
  Parent: '000000...0000'  
  
  Attempt: '15672760'  
  Time: 2m58.74059411s  
  
// RUNNING THE BABAYAGA NODE  
  
Listening on: 127.0.0.1:8085  
Blockchain state:  
  - height: 0  
  - hash: 000000...00000  
  
Added Pending TX {"from":"andrey","to":"babayaga","value":1,"data":"","\n"time":1589993912} from Peer 127.0.0.1:8085  
Added Pending TX {"from":"andrey","to":"babayaga","value":2,"data":"","\n"time":1589993912} from Peer 127.0.0.1:8085  
  
// BABAYAGA IS MINING THE ABOVE 2 TXs
```

```
Mining 2 Pending TXs. Attempt: 1
```

```
// BUT ANDREJ MINED IT FASTER (MOCKED BLOCK)
```

```
Persisting new Block to disk:
```

```
{ "hash": "000000022393099224", "block": { "header": { "parent": "000000..\\00000", "number": 0, "nonce": 3707911869, "time": 1589993912, "miner": "andrej"}, "payload": [ { "from": "andrey", "to": "babayaga", "value": 1, "data": "", "time": 1589993912} ] } }
```

```
Peer mined next Block '0000000223930...99224' faster :(
```

```
// BABAYAGA REMOVES THE MINED TX FROM MEMPOOL
```

```
Updating in-memory Pending TXs Pool:
```

```
-archiving mined TX: 6b11a...9b84c
```

```
Mining cancelled!
```

```
ERROR: mining cancelled. context canceled
```

```
// BABAYAGA RESTARTING THE MINING WITH TX1 ONLY
```

```
Mining 1 Pending TXs. Attempt: 1
```

```
...
```

```
Mining 1 Pending TXs. Attempt: 13000000
```

```
Mined new Block '0000008d042ae...02826' using PoW::SHA256:
```

```
Height: '1'
```

```
Nonce: '2444425187'
```

```
Created: '1589994111'
```

```
Miner: 'babayaga'
```

```
Parent: '0000000223930...99224'
```

```
Attempt: '13676729'
```

```
Time: 2m38.70435543s
```

```
Updating in-memory Pending TXs Pool:
```

```
-archiving mined TX: c3344c...b81a2
```

Persisting new Block to disk:

```
{"hash": "0000008d042a...02826", "block": {"header": {"parent": "000000\\0223930...99224", "number": 1, "nonce": 2444425187, "time": 1589994111, "mine\\r": "babayaga"}, "payload": [{"from": "andrey", "to": "babayaga", "value": 2, "\\data": "", "time": 1589993912}]}}
```

```
// ASSERT TRANSFER + BLOCK REWARDS
```

```
--- PASS: TestNode_MiningStopsOnNewSyncedBlock (358.74s)
    node_integration_test.go:224: Starting Andrej balance: 1000000
    node_integration_test.go:225: Starting BabaYaga balance: 0
    node_integration_test.go:226: Ending Andrej balance: 1000097
    node_integration_test.go:227: Ending BabaYaga balance: 103
```

PASS

ok github.com/web3coach/the-blockchain-bar/node 358.745s



Study Code

Implements PoW draft [fa1b1d⁷⁵](#)

Implements PoW Reward [1a34e0⁷⁶](#)

Fixes sync() function on genesis block 0 [fc5ae1⁷⁷](#)

Improves PoW tests [e2d5cd⁷⁸](#)

Updates migration cmd to PoW and sorts TXs by time [b194d4⁷⁹](#)

⁷⁵ <https://github.com/web3coach/the-blockchain-bar/commit/fa1b1dc237ee15fcb51a0857e55ec8fd02191492>

⁷⁶ <https://github.com/web3coach/the-blockchain-bar/commit/1a34e0db405c359220c1ceelab75e5318f04b2d4>

⁷⁷ <https://github.com/web3coach/the-blockchain-bar/commit/fc5ae1ff923f697d9c8990d7da7fdc3ee0024366>

⁷⁸ <https://github.com/web3coach/the-blockchain-bar/commit/e2d5cd8b0f02a693c12d9641854c4f84af79401a>

⁷⁹ <https://github.com/web3coach/the-blockchain-bar/commit/b194d4a3468f60b578ccb4658352b04761ccf468>

Blockchain Releases are Complicated

Monday, December 30.

Do you remember the chapter about a database fork? Well, Andrej broke TBB blockchain once again:

```
tbb run --datadir=$HOME/.tbb_andrej_fork --miner=andrej

> Launching TBB node and its HTTP API...
> ...
> Searching for new peers and their blocks and peers...
>
> Found 3 new blocks from peer `node.tbb.web3.coach`
> Importing blocks from peer `node.tbb.web3.coach`...
>
> ERROR: invalid block hash 05370a480956...
```

Why did the block import fail?

Pause and think.

Do you know? Congratulation! You are excellent in debugging a broken blockchain. For the developers reading this book tired after work, who want to enjoy the show, here is why:

Because every block now has a new “miner” attribute, causing all block hashes to mismatch. In a centralized world, you would release a new version of the software to all servers and call the job done. But! Andrej doesn’t control all network nodes. Only his. He can’t force BabaYaga neither Caesar to upgrade. He must, therefore, get in touch with every network participant and explain to them the advantages of the latest code changes and PoW mining algorithm.

BabaYaga, excited to drink 100 vodka shots for every block she mines gives Andrej the green light.

Caesar, a big supporter of a sustainable environment movement, gets infuriated but doesn't have a choice. If Caesar doesn't want to fork away, where he is the only participant in his network, he must upgrade as well. "Decentralize releases" in action.

Andrej executes the `tbb migrate` cmd and tells BabaYaga and Caesar to reconfigure their bootstrap node and re-sync their blockchains from scratch.

Migrating to PoW DB

Andrej modifies the `./cmd/tbb/migrate.go` cmd by taking all the known DB TX, encapsulates them to one block and runs the mining algorithm.

```
var migrateCmd = func() *cobra.Command {
    var migrateCmd = &cobra.Command{
        Use:   "migrate",
        // ...
        // ...
        // ...

    n := node.New(
        getDataDirFromCmd(cmd),
        ip,
        port,
        database.NewAccount(miner),
        peer,
    )

    n.AddPendingTX(NewTx("andrey", "andrey", 3))
    n.AddPendingTX(NewTx("andrey", "babayaga", 2000))
    n.AddPendingTX(NewTx("babayaga", "andrey", 1, ""), peer)
    n.AddPendingTX(database.NewTx("babayaga", "caesar", 1000))
    n.AddPendingTX(database.NewTx("babayaga", "andrey", 50))

    // Run the node -> and its PoW mining algorithm
    err := n.Run(ctx)
```

 **Practice time.**

1/3 Re-create your local DB using Proof of Work.

>_ tbb migrate --datadir=\$HOME/.c11_pow --miner=andrej

2/3 Re-boot the node!

>_ tbb run --datadir=\$HOME/.c11_pow --miner=andrej



Fun Facts

In production blockchains like Ethereum, users pay fees for their transactions. The higher the fee, the faster will be their transaction mined. The regular fee is a few dollars, but sometimes you can find a user in a real hurry, or with a fat finger, paying 83 000\$ to a miner to process his [single transaction⁸⁰](#) ASAP!

Transaction details

0x5f86480294076256406755e62bf7d02a43f3b03579fe4a22f46fc3004428b1ee

Hash	0x5f86480294076256406755e62bf7d02a43f3b03579fe4a22f46fc3004428b1ee
Block	9660661 3302 Confirmations
Time:	13/03/2020 04:04:52 (13 hours ago)
From	0x167A9333BF582556f35Bd4d16A7E80E191aa6476
To	0x8454190C164e52664Af2c9C24ab58c4e14D6bbE4
Value	0 ETH 0 USD
Fee	656.0468 ETH \$83,842.78

massive transaction fee

⁸⁰ <https://www.etherchain.org/tx/5f86480294076256406755e62bf7d02a43f3b03579fe4a22f46fc3004428b1ee>



Study Code

Explore The Byzantine General's Problem

[https://www.preethikasireddy.com/post/lets-take-a-crack-at-understanding-distributed-consensus⁸¹](https://www.preethikasireddy.com/post/lets-take-a-crack-at-understanding-distributed-consensus)

Learn about the notion of time in a distributed system, and its effects on ordering [https://distsys.substack.com/p/time-clocks-and-order⁸²](https://distsys.substack.com/p/time-clocks-and-order)

Study Ethash algorithm⁸³.

⁸¹ <https://www.preethikasireddy.com/post/lets-take-a-crack-at-understanding-distributed-consensus>

⁸² <https://distsys.substack.com/p/time-clocks-and-order>

⁸³ <https://github.com/ethereum/go-ethereum/tree/master/consensus/ethash>



Summary

Closed software with centralized access to private data puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relation. Specifications of the blockchain system should be well defined from the beginning and only change if its users support it.

Blockchain is an immutable, transparent database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.

The database content is hashed by a secure cryptographic hash function. The blockchain participants use the resulted hash to reference a specific database state.

Transactions are grouped into batches for performance reasons. A batch of transactions make a Block. Each block is encoded and hashed using a secure, cryptographic hash function.

Block contains Header and Payload. The Header stores various metadata such a time and a reference to the Parent Block (the previous immutable database state). The Payload carries the new database transactions.

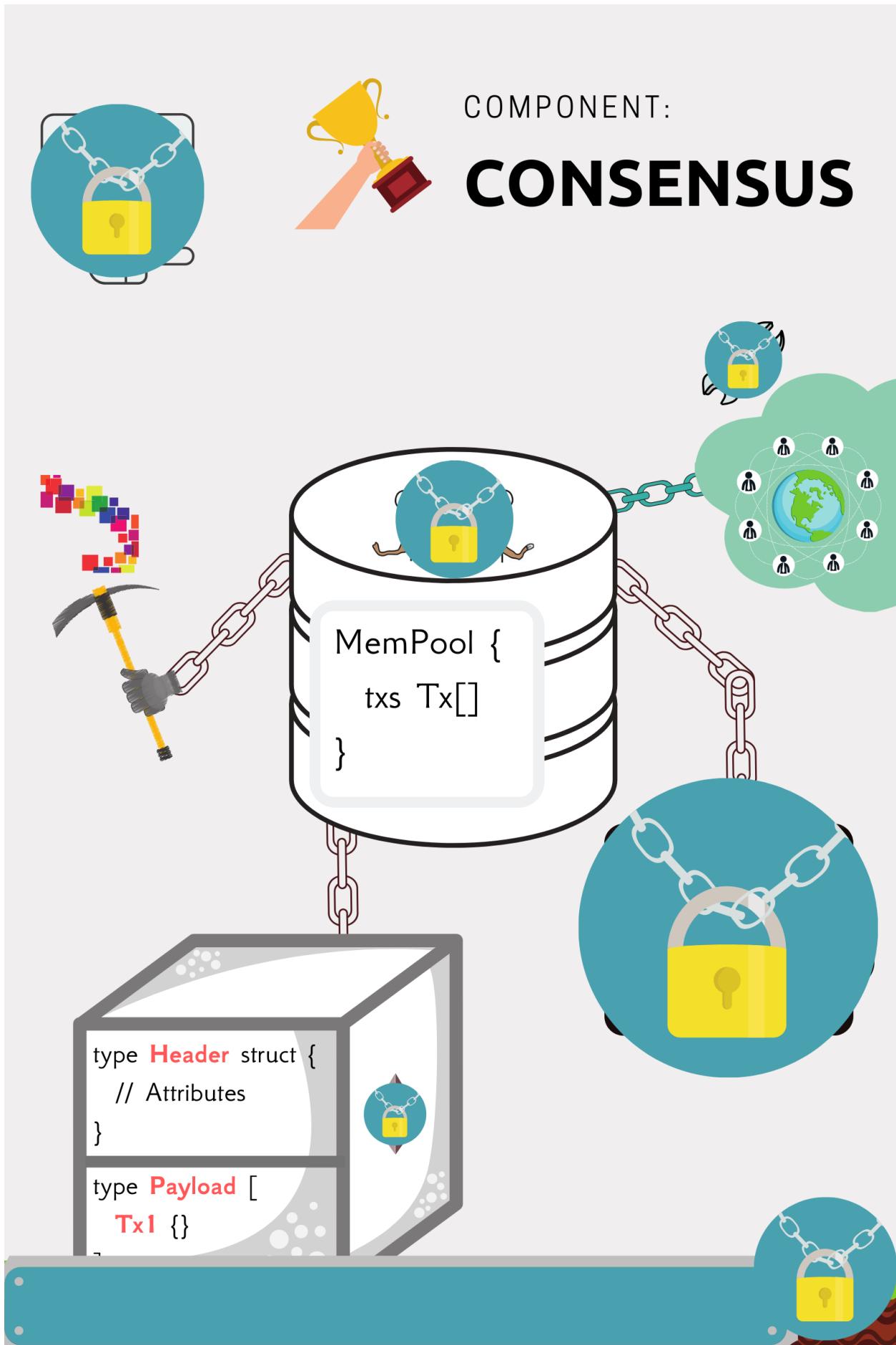
The blockchain network consists of Nodes and Peers. All new peers connect to a default Bootstrap Peer(s) to discover and retrieve the current database state as well as the full transaction history. Peers continually communicate with each other using a programmed sync algorithm. Peers exchange information about new blocks and new network's peers.

Blockchain peers coordinate by following a pre-defined set of rules called “consensus.” The primary responsibility of a consensus algorithm is to decide who and how can produce the next valid block.

If consensus can't happen, the database history splits. A database split is called “hard fork.” Once hard fork occurs, peers with different blockchain history can't communicate and synchronize because everyone will have a different source of truth.

The most famous consensus algorithm implemented in Bitcoin is Proof of Work. PoW security consists of solving a cryptographic puzzle. Peers call this activity: “mining.” Miners repeatedly generate a random number, “nonce,” insert the nonce into a block header, and hash the block until the resulted hash matches the pre-defined consensus criteria.

Blockchain releases are complicated and require asynchronous communication between independent parties—the participants with the broadest influence, lobbying, and momentum win.



12 | Madam/Sir Your Cryptographic Signature Please

>_ git checkout c12_crypto

Hacking a User Balance

Tuesday, December 31.

The current version of TBB blockchain has a fatal flaw. Can you spot it?

>_ \$

```
curl --location --request POST 'localhost:8080/tx/add' \
--header 'Content-Type: application/json' \
--data-raw '{
    "from": "babayaga",
    "to": "andrey",
    "value": 100
}'
```

Congratulation! You are excellent at noticing broken blockchains.
Anyone can create a new transaction on behalf of someone else.

If Andrej wants more tokens, he can create the following transfer transaction on behalf of BabaYaga:

As long as BabaYaga has sufficient balance (100 TBB) and a miner finds a valid hash for the block containing this transaction, the hack will succeed.

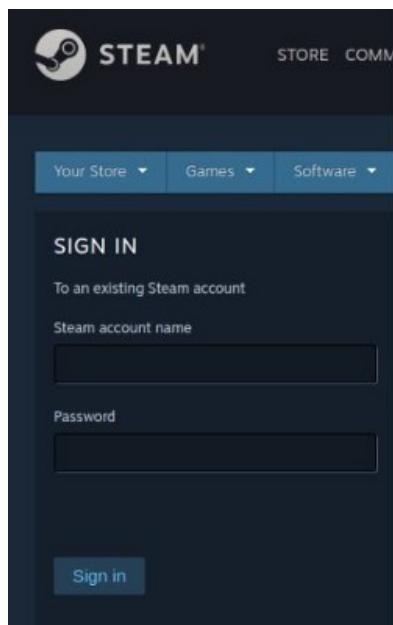
How can Andrej fix it?

Current State of Web Authentication

Wednesday, January 1.

Happy new year 2020!!! 200 OK... back to work.

Andrej has an idea! He registers each bar customer in a MySQL DB:



MySQL authentication

Pretty straight forward with a secure, behind comfortable firewall, centralized database. Two SQL queries. One source of truth. Some extra sessions, cookies, and Andrej would have a secure working authentication in place.

The problem is... blockchain decentralization philosophy doesn't favor MySQL solutions. All data are stored on every node. Therefore,

Andrej definitely can't store all clients' credentials (usernames, emails, passwords) on every network node.

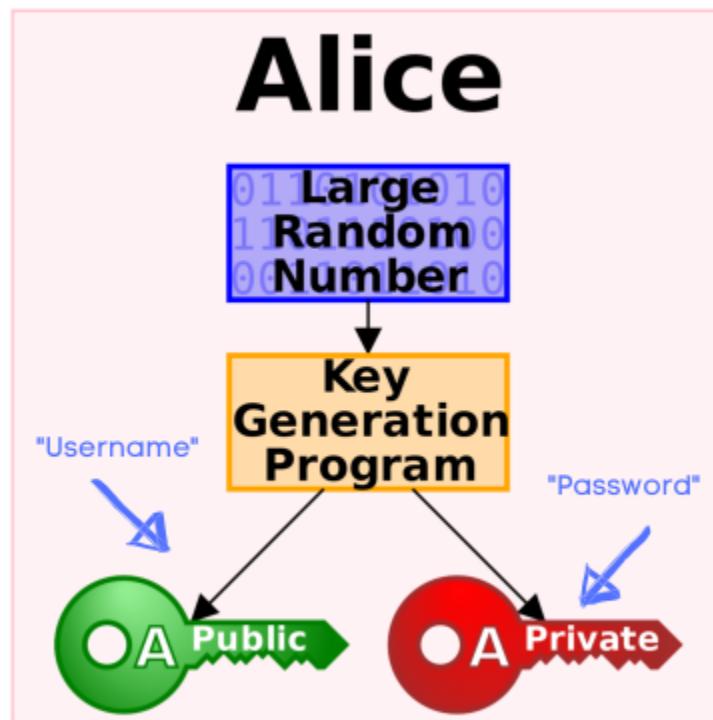
What's the alternative?

Cryptography. Your new year starts with a cumbersome topic, dear reader!

Asymmetric Cryptography in Nutshell

Thursday, January 2.

Asymmetric? **Public-key cryptography**.



asymmetric keys set

Public Key == Your Username

Private Key == Your Password

Key Generation Program == A secure algorithm that generates a big random number (e.g., 78 digits long **Private Key**) and mathematically derives a **Public Key** from it. An example would be your RSA 2048-bit SSH key.

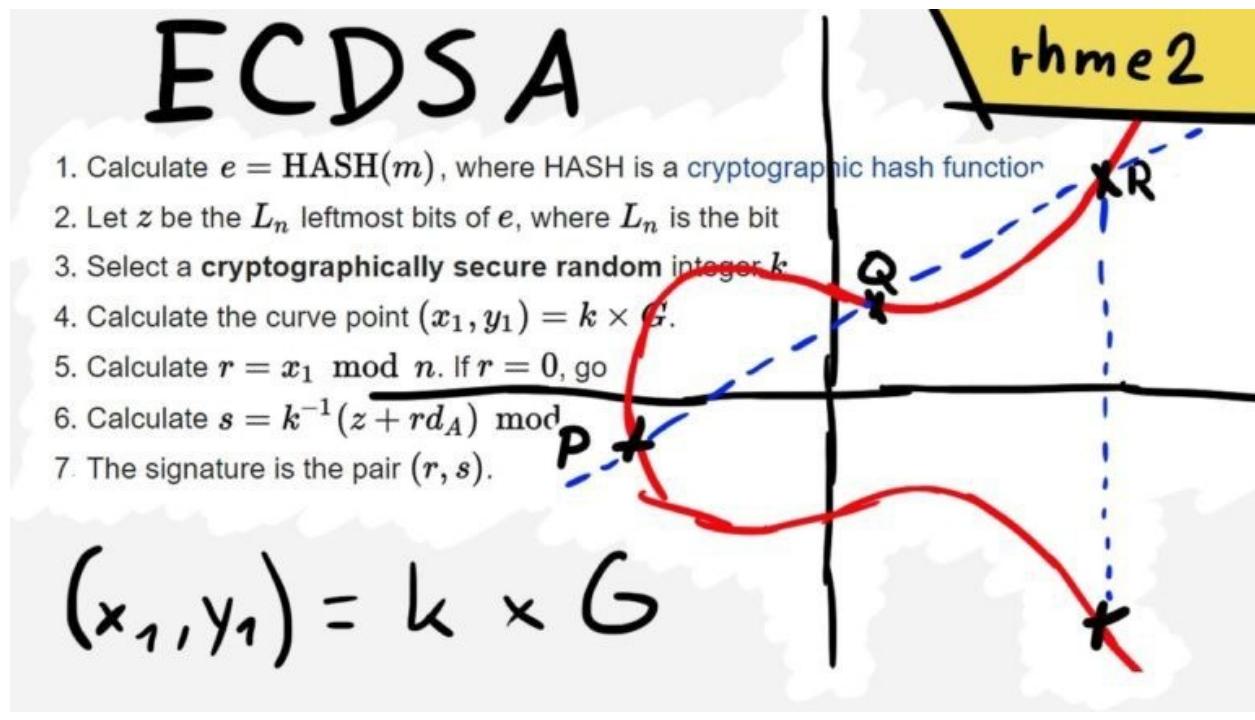
Cryptography is all about proofs. Prove me; it's you. Prove me; you did it. Bitcoin, Ethereum, XRP are all based on asymmetric proofs.

But Andrej, how are the asymmetric keys used in practice?

Blockchain Authentication with go-Ethereum

Friday, January 3.

Ethereum's key generation algorithm generates the pair of keys from an [elliptic curve⁸⁴](#) (ECDSA), specifically **secp256k1 curve**. Bitcoin does the same.



[image src⁸⁵](#)

Looks bloody complex, Andrej tells himself. Fortunately, he is very

⁸⁴ https://en.wikipedia.org/wiki/Elliptic-curve_cryptography

⁸⁵ <https://www.cryptanalyst.net/2017/05/19/breaking-ecdsa-elliptic-curve-cryptography-rhme2-secure-filesystem-v1-92r1-crypto-150/>

much familiar with the quote: “[Never roll your own cryptography](#)”⁸⁶, therefore he will import the official go-Ethereum package into his codebase doing all the math magic behind the scenes. It will also speed up his development, given he doesn’t have to do all the tedious non-blockchain work of persisting the keys (strings, large numbers) to disk, loading them, etc. You already mastered that process at the beginning of the book on the `genesis.json` file.

As a bonus, he will learn how to use Bitcoin and Ethereum libraries; so will you! This will help transitioning from this training blockchain into the real world where you will explore Ethereum 2.0, Bitcoin and others production blockchains in the next upcoming, to-be-released chapters.

⁸⁶ <https://security.stackexchange.com/questions/18197/why-shouldnt-we-roll-our-own>

Andrej analyses the **go-ethereum** packages related to cryptography. Open image in full screen.⁸⁷

The screenshot shows a code editor with the following details:

- Project Structure:** The left sidebar shows the project structure under "go-ethereum/accounts/keystore". It includes files like `account_cache.go`, `key.go`, `keystore.go`, `passphrase.go`, `plain.go`, `presale.go`, `wallet.go`, `watch.go`, and `watch_fallback.go`. A yellow box highlights this section.
- keystore.go Code:** The main pane shows the `keystore.go` file. It defines a `Wallet` type and a `KeyStore` struct. A yellow box highlights the `KeyStore` struct definition. An annotation box states: "Wallet - a software, object responsible for storing your Accounts(Private, Public Keys)".
- crypto Package:** A separate sidebar shows the contents of the `crypto` package, which contains sub-directories like `blake2b`, `bn256`, `ecies`, and `secp256k1`, along with files such as `curve.go`, `ext.h`, and `secp256.go`. A yellow box highlights this section. An annotation box states: "Crypto - math algorithms written in C for generating elliptic curve keys for accounts".
- keystore.go Callout:** An arrow points from the `KeyStore` struct in `keystore.go` to the `crypto` package sidebar.
- Annotation Boxes:** Three annotation boxes provide descriptions of the highlighted components:
 - Wallet:** "a software, object responsible for storing your Accounts(Private, Public Keys)"
 - KeyStore:** "Ethereum file-system based \"wallet manager\""
 - Crypto:** "math algorithms written in C for generating elliptic curve keys for accounts"
- Footer:** The text "go-ethereum crypto packages" is centered at the bottom of the editor window.

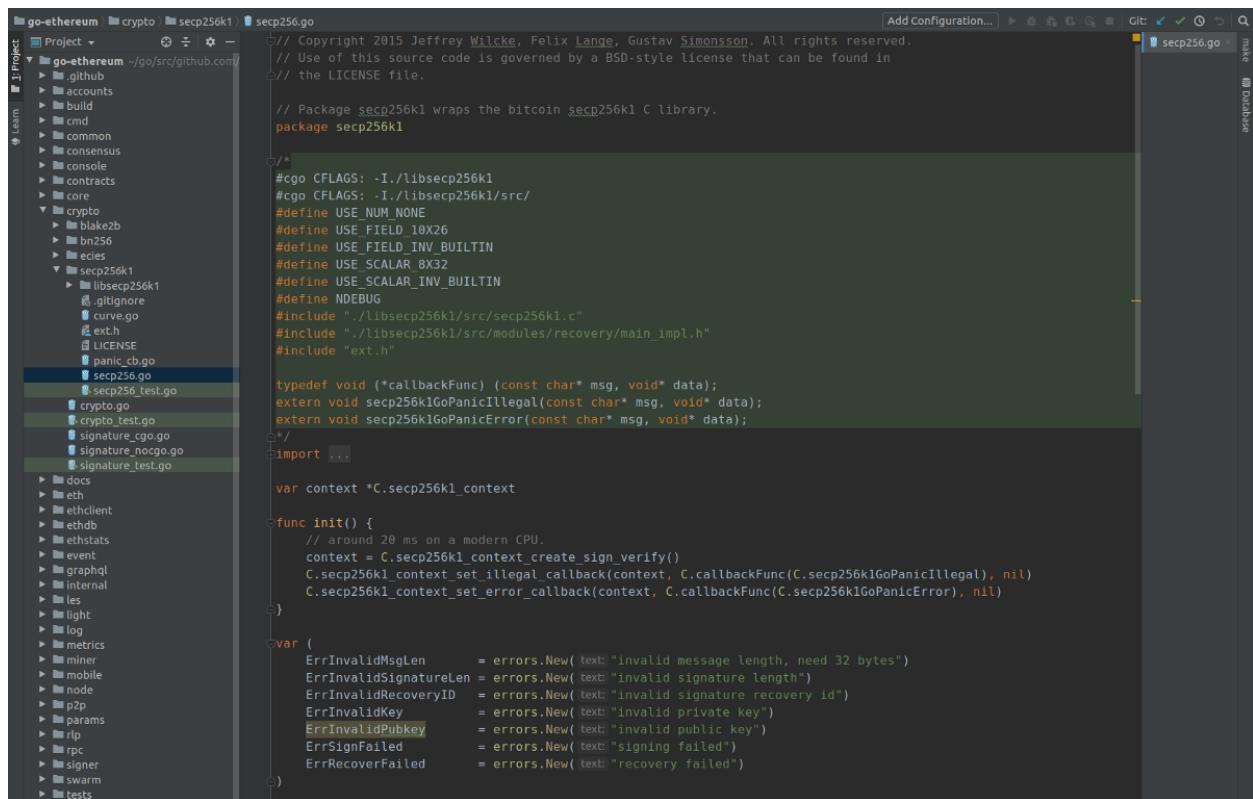
- **Wallet** — is a general blockchain concept. The main purpose of a wallet is to store and protect the Private Key. The Private Key is always persisted in disk in a password-encrypted form.
- **Keystore** — is Ethereum specific concept. It's a wrapper around

⁸⁷ https://web3coach-public.s3.eu-central-1.amazonaws.com/img/go_ethereum_crypto.png

the Wallet responsible for generating new keys, persisting them to disk in an encrypted format, loading them from disk, etc.

Awesome. The **Keystore**, **Wallet**, **Crypto** packages are precisely what Andrej was looking for.

Are you curious how the secp256k1 C library looks like? [Open image in full screen.](#)⁸⁸



The screenshot shows the Go code for the `secp256k1` package within the `go-ethereum` project. The `secp256.go` file contains the following code:

```

// Copyright 2015 Jeffrey Wilcke, Felix Lange, Gustav Simonsson. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be found in
// the LICENSE file.

// Package secp256k1 wraps the bitcoin secp256k1 C library.
package secp256k1

/*
#cgo CFLAGS: -I./libsecp256k1
#cgo CFLAGS: -I./libsecp256k1/src
#define USE_NUM_NONE
#define USE_FIELD_10X26
#define USE_FIELD_INV_BUILTIN
#define USE_SCALAR_8X32
#define USE_SCALAR_INV_BUILTIN
#define NDEBUG
#include "./libsecp256k1/src/secp256k1.c"
#include "./libsecp256k1/src/modules/recovery/main_impl.h"
#include "ext.h"

typedef void (*callbackFunc) (const char* msg, void* data);
extern void secp256k1GoPanicIllegal(const char* msg, void* data);
extern void secp256k1GoPanicError(const char* msg, void* data);

import ...

var context *C.secp256k1_context

func init() {
    // around 20 ms on a modern CPU.
    context = C.secp256k1_context_create_sign_verify()
    C.secp256k1_context_set_illegal_callback(context, C.callbackFunc(C.secp256k1GoPanicIllegal), nil)
    C.secp256k1_context_set_error_callback(context, C.callbackFunc(C.secp256k1GoPanicError), nil)
}

var (
    ErrInvalidMsgLen      = errors.New("invalid message length, need 32 bytes")
    ErrInvalidSignatureLen = errors.New("invalid signature length")
    ErrInvalidRecoveryID   = errors.New("invalid signature recovery id")
    ErrInvalidKey          = errors.New("invalid private key")
    ErrInvalidPubkey        = errors.New("invalid public key")
    ErrSignFailed           = errors.New("signing failed")
    ErrRecoverFailed        = errors.New("recovery failed")
)

```

go-ethereum crypto package wrapping secp256k1 C library

PS: Crypto stands for cryptography not cryptocurrencies!

⁸⁸ <https://web3coach-public.s3.eu-central-1.amazonaws.com/img/secp.png>

Programming a Cryptocurrency Wallet

Friday Evening, January 3.

Andrej goes ahead and programs a new tbb wallet command to manage the node's blockchain accounts:

```
// cmd/tbb/wallet.go
package main

import (
    "fmt"
    "github.com/ethereum/go-ethereum/cmd/utils"
    "github.com/ethereum/go-ethereum/console"
    "github.com/spf13/cobra"
    "github.com/web3coach/the-blockchain-bar/wallet"
    "os"
)

func walletCmd() *cobra.Command {
    var walletCmd = &cobra.Command{
        Use:   "wallet",
        Short: "Manages blockchain accounts and keys.",
        PreRunE: func(cmd *cobra.Command, args []string) error {
            return incorrectUsageErr()
        },
        Run: func(cmd *cobra.Command, args []string) {
        },
    }

    return walletCmd
}
```

He also needs a `getPassphrase()` function to be used later for requesting a password from the user. This password will be used to encrypt the asymmetric key set on disk symmetrically:

```
func getPassPhrase(prompt string, confirmation bool) string {
    password, err := console.Stdin.PromptPassword(prompt)
    if err != nil {
        utils.Fatalf("Failed to read password: %v", err)
    }

    if confirmation {
        confirm, err := console.Stdin.PromptPassword("Repeat")
        if err != nil {
            utils.Fatalf("Failed to read password confirmation: %v", e\
rr)
        }
        if password != confirm {
            utils.Fatalf("Passwords do not match")
        }
    }

    return password
}
```

The last new command is a tbb wallet new-account, accessing the following Keystore interface and generating new “Ethereum” (TBB) account:

```
func walletNewAccountCmd() *cobra.Command {
    var cmd = &cobra.Command{
        Use:   "new-account",
        Short: "Creates a new account with a new set of a elliptic-cur\\
ve Private + Public keys.",
        Run: func(cmd *cobra.Command, args []string) {
            password := getPassPhrase("Please enter a password to encr\\
ypt the new wallet:", true)

            dataDir := getDataDirFromCmd(cmd)

            ks := keystore.NewKeyStore(
                wallet.GetKeystoreDirPath(dataDir),
                keystore.StandardScryptN,
                keystore.StandardScryptP,
            )
            acc, err := ks.NewAccount(password)
            if err != nil {
                fmt.Println(err)
                os.Exit(1)
            }

            fmt.Printf("New account created: %s\n", acc.Address.Hex())
        },
    }

    addDefaultRequiredFlags(cmd)

    return cmd
}
```

If you are confused about the `keystore.StandardScryptN`, it's a crypto strength setting:

```
// StandardScryptN is the N parameter of Scrypt encryption algorithm,  
// using 256MB memory and  
// taking approximately 1s CPU time on a modern processor.
```

```
StandardScryptN = 1 << 18
```

Practice time.

>_ tbb wallet new-account --datadir=\$HOME/.c12_crypto

```
Please enter a password to encrypt the new wallet:  
Repeat password:
```

```
New account created: `0x22ba1F80452E6220c7cc6ea2D1e3EEDDaC5F694A`  
Saved in: /home/web3coach/.c12_crypto/keystore
```

The `tbb wallet new-account` run successfully but returned quite a suspicious account name:

```
0x22ba1F80452E6220c7cc6ea2D1e3EEDDaC5F694A
```

What on earth is this monstrosity?

Ethereum cryptography magic:

1. An **account wallet is an asymmetric Private Key** (a very long integer)
2. A **Public Key is a mathematically derived from the Private Key** (also a very long integer)
3. The account's address (nick, username) is a **32 bytes secure hash of the Public Key**

Therefore:

The **22ba1F80452E6220c7cc6ea2D1e3EEDDaC5F694A** is from now on, the Andrej's blockchain username!

The “0x” is just a prefix defining the string is in a hex format and has nothing to do with the hash of the public key. Given it's only a hash of the public key, Andrej can freely share this value publicly as he would do with any username in Web 2.0.

Anyway, for the sake of sanity, in this book, let's keep calling **22ba1F80452E6220c7cc6ea2D1e3EEDDaC5F694A**, Andrej.

Out of curiosity, how does the encrypted Keystore file look persisted in disk?

```
cat $HOME/.c12_crypto/keystore/UTC--2020-02-16T11-55-05.804446086Z--22\  
ba1f80452e6220c7cc6ea2d1e3eeddac5f694a
```

```
{
  "address": "22ba1f80452e6220c7cc6ea2d1e3eeddac5f694a",
  "crypto": {
    "cipher": "aes-128-ctr",
    "ciphertext": "3cca5f8ef1d7e997b3a852d93eaea6f08030fe4acd6379b8c39\  
6feccf41bcfee",
    "cipherparams": {
      "iv": "15cf839a8b7e4ae9cab961065c3f6a6b"
    },
    "kdf": "scrypt",
    "kdfparams": {
      "dklen": 32,
      "n": 262144,
      "p": 1,
      "r": 8,
      "salt": "5d1e52f317038c9e82cf736c47011cc72b20936d93f646dce89447\  
da9e6df96"
    },
    "mac": "9726ae3ee2f64bd1dd9c3897d08609afb2dff084897bb1e078545bc38d\  
49c347"
  },
  "id": "ae6f8802-2232-4909-bca2-b6718cdb067b",
  "version": 3
}
```

- Attribute **address** →the public account's username. **32 bytes secure hash of the Public Key**

- Attribute **crypto** →the **symmetric aes-128-ctr encryption** settings for encrypting the user's **asymmetric elliptic-curve ECDSA Private, Public Key on disk**

Don't you believe **22ba1F...5F694A?**

I mean, don't you believe Andrej how the Ethereum cryptography and address generation works? Fair enough. Trust but verify.

All you have to do are 3 steps.

Step 1) Load the encrypted Keystore file to memory

Step 2) Decrypt the Keystore file with the password

Step 3) Examine the Private Key struct

To make this simple for you, Andrej created a helper command so you can experiment with the keystore logic and the ecdsa.PrivateKey directly from your terminal.

The helper cmd tbb wallet pk-print works in the following way:

```
func walletPrintPrivKeyCmd() *cobra.Command {
    var cmd = &cobra.Command{
        Use:   "pk-print",
        Short: "Unlocks keystore file and prints the Private + Public \
keys.",
        Run: func(cmd *cobra.Command, args []string) {
            ksFile, _ := cmd.Flags().GetString(flagKeystoreFile)
            password := getPassPhrase("Please enter a password to decr\
ypt the keystore file:", true)

            // Load the symmetrically encrypted key from disk
            keyJson, err := ioutil.ReadFile(ksFile)
            if err != nil {
                fmt.Println(err.Error())
                os.Exit(1)
            }

            // Decrypt the Private Key file using your password
            key, err := keystore.DecryptKey(keyJson, password)
            if err != nil {
                fmt.Println(err.Error())
                os.Exit(1)
            }

            // Print it to terminal
            spew.Dump(key)
        },
    }

    addKeystoreFlag(cmd)

    return cmd
}
```

Practice time.**Step 0)** Create a new account.

```
>_ tbb wallet new-account --datadir=$HOME/.c12_crypto
```

Step 1) Copy the path to your account's keystore file. It's located in the keystore dir.

```
>_ ls -la $HOME/.c12_crypto/keystore/
```

Step 2 and 3) Decrypt the key and print it.

```
>_ $
```

```
tbb wallet pk-print --keystore=$HOME/.c12_crypto/keystore/UTC--2020-05\  
-24T14-40-39.113139054Z--492aebc3fca19573177c20f309cd956b080e86e9
```

This is how it will look like. Open image in full screen.⁸⁹

⁸⁹ https://web3coach-public.s3.eu-central-1.amazonaws.com/img/ethereum_crypto_action.png

The screenshot shows a terminal window with a Go program running. The program reads a keystore file, decrypts it with a password, and then prints the new account's address. It also includes a block of code for generating math parameters to derive a public key from a private key.

```

keyjson, err := ioutil.ReadFile(filename: "/home/web3coach/.tbb_andrej_crypto/keystore/UTC--2020-02-16T11-55-05.804446086Z--22ba1f80452e6220c7cc6ea2d1e3eedda5f69")
if err != nil {
    fmt.Println(err)
    os.Exit(code: 1)
}

key, err := keystore.DecryptKey(keyjson, password)
if err != nil {
    fmt.Println(err)
    os.Exit(code: 1)
}

spew.Dump(key)

//fmt.Printf("New account created: %s\n", acc.Address.Hex())
}
}

walletNewAccountCmd0 *cobra.Command = func(cmd *cobra.Command, args []string) {
Terminal: Local + Repeat password:
(*keystore.Key)(0xc000032100)({
Id: (uuid.UUID) (len=16 cap=16) ae6f88072232-4909-bca2-b6/18cd006/b,
Address: (common.Address) (len=20 cap=20) 0x22ba1f80452e6220c7cc6ea2d1e3eedda5f694A,
PrivateKey: (*ecdsa.PrivateKey)(0xc000000000000000),
PublicKey: (*ecdsa.PublicKey) {
Curve: (*secp256k1.BitCurve)(0xc000000000000000)({P: (*big.Int)(0xc000000e5e0)(115792899237316195423570985008687907853269984665640564039457584067908834671663), N: (*big.Int)(0xc000000e620)(115792899237316195423570985008687907852837564279074904382605163141518161494337), B: (*big.Int)(0xc000000e660)(7), Gx: (*big.Int)(0xc00000e6a0)(550662302227734366957871889516853432625060345377759417550018736038911672940), Gy: (*big.Int)(0xc00000e6e0)(326705100207580169780083085130507043184471273380659243275938904335753737482424), BitSize: (int) 256}),
X: (*big.Int)(0xc00000e080)(5878639908531916970782739043735172912065137369288611019755305365362058463494),
Y: (*big.Int)(0xc00000e280)(96394471370872363584164655508480344569687326885018631912019354187646721481997}),
D: (*big.Int)(0xc00000e020)(1109669032243223642934318247741818336385676217400754022493247430001945933610)
})
}
+ The-blockchain-bar git:(master) ✘

```

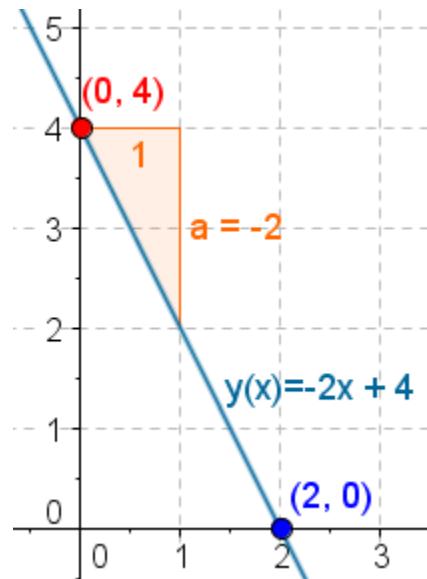
Andrej's account address

**Math params to derive Pub Key
from Priv Key**

**The famous,
Almighty,
Elliptic Curve Private Key...
i.e just a very long integer**

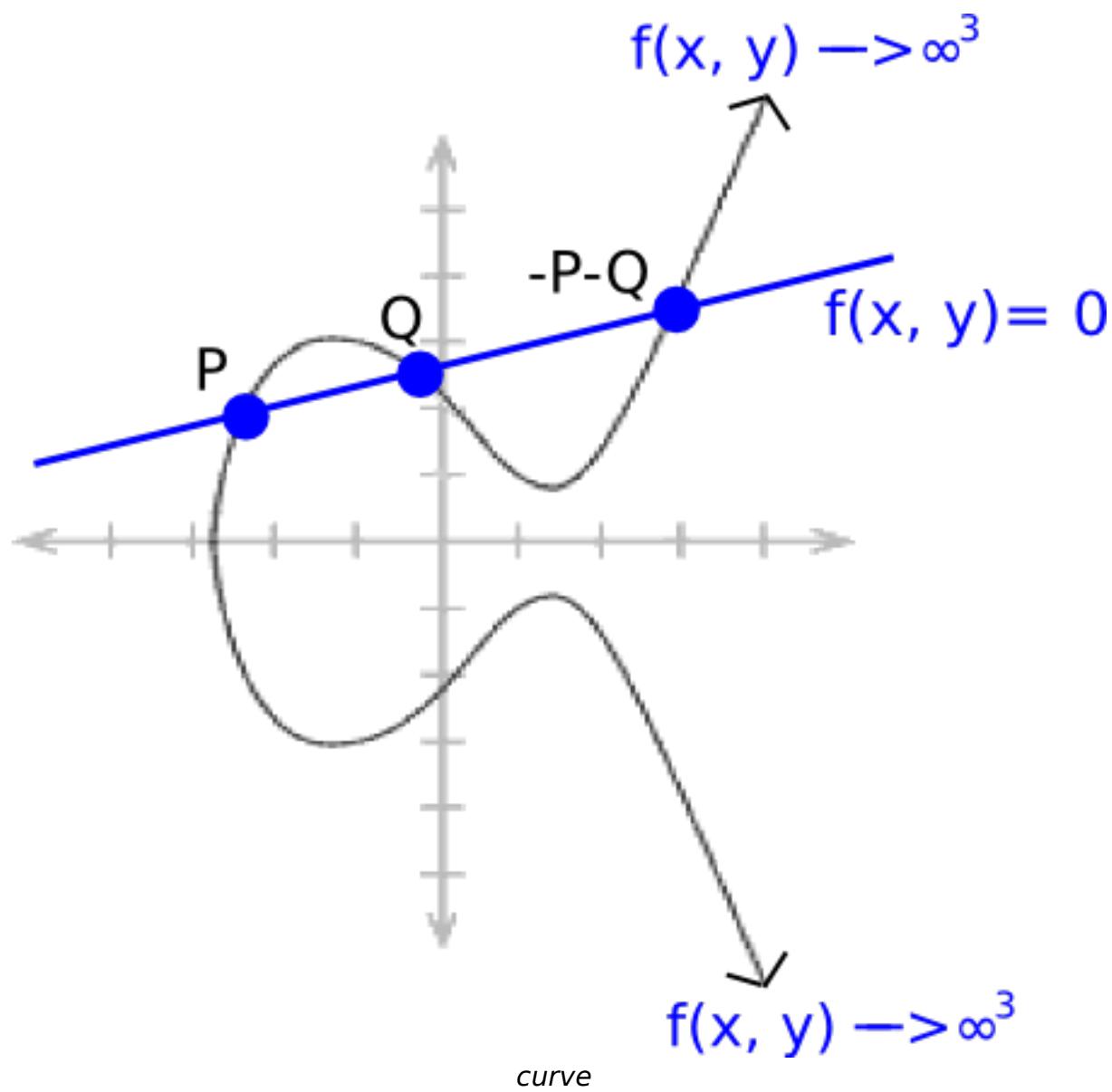
Private Key + Public Key + Account are related

The printed **Private Key Curve attributes** look a bit intimidating, but you can think about the specific values as typical X and Y points on a graph, like from the high school/university math classes. Andrej was used to values and equations such as: $y = -2x + 4$.



https://commons.wikimedia.org/wiki/File:Wiki_linearna_funkcija_eks1.png

The main (extremely, extremely simplified) difference here is the values are exponentially bigger because they are positioned on a much, much larger curve (graph) for security reasons, so an attacker can't guess them. **The points are calculated with a secure, irreversible math function** powering all the asymmetric, cryptographic math magic.



So how are the curve attributes represented as a **Private Key** Go struct?

```
// PublicKey represents an ECDSA public key.
type PublicKey struct {
    elliptic.Curve
    X, Y *big.Int
}

// PrivateKey represents an ECDSA private key.
type PrivateKey struct {
    PublicKey
    D *big.Int
}

type ecdsaSignature struct {
    R, S *big.Int
}

// Public returns the public key corresponding to priv.
func (priv *PrivateKey) Public() crypto.PublicKey {
    return &priv.PublicKey
}
```

The Andrej's new tbb wallet new-account command is powered by the following three go-ethereum functions, generating the above Private Key struct, located in \$HOME/go/pkg/mod/github.com/ethereum/go-ethereum@v1.9.10/accounts/keystore/key.go:

Generate key:

```
func GenerateKey() (*ecdsa.PrivateKey, error) {
    return ecdsa.GenerateKey(S256(), rand.Reader)
}
```

Convert the Private Key to a Key wrapper:

```
func newKeyFromECDSA(privateKeyECDSA *ecdsa.PrivateKey) *Key {
    id := uuid.NewRandom()
    key := &Key{
        Id:         id,
        Address:   crypto.PubkeyToAddress(privateKeyECDSA.PublicKey),
        PrivateKey: privateKeyECDSA,
    }
    return key
}
```

Generate the address from a Public Key:

```
func PubkeyToAddress(p *ecdsa.PublicKey) common.Address {
    pubBytes := FromECDSAPub(&p)
    return common.BytesToAddress(Keccak256(pubBytes[1:])[12:])
}
```

You will explore exactly what's happening behind the scenes in the next chapter. For now, Andrej just prepares the codebase to support the Ethereum's account address format: **22ba1F...5F694A**.

Andrej refactors all the accounts from string format to the new common.Address{} Ethereum format. See commit at the end of the chapter.

```
func NewAccount(value string) common.Address {
    return common.HexToAddress(value)
}
```

And of course, he also fixes all the tests! If you run the `TestMine()` again, you will see the blocks' `miner` attribute changed from `andrei` to `0x22ba1F...5F694A`.

```
>_ go test -timeout=0 ./node -test.v -test.run ^TestMine$
```

Mined new Block '0000003b69c3...e04a9a5' using PoW:

```
Height: '1'  
Nonce: '938304572'  
Created: '1581962772'
```

// NEW ACCOUNT FORMAT, YOUR MINER ADDRESS WILL BE DIFFERENT
// BECAUSE THE PRIVATE KEY IS GENERATED ON THE FLY.

Miner: '0x22ba1F80452E6220c7cc6ea2D1e3EEDDaC5F694A'

```
Time: 2m15.041460717s
--- PASS: TestMine (135.04s)
PASS
```

Blockchain cryptography-powered authentication in action.



Fun Facts

The go-ethereum's **crypto** package is a wrapper around a secp256k1 library written in **C** for performance reasons.

The default, empty `0x000...000090` Ethereum address where people accidentally send money if they mess up has currently 2M \$\$\$ balance. Whoops. Always verify the recipient's address! No rollbacks in blockchain.

The biggest problems of the cryptocurrency/blockchain space is in the weak tooling and horrible UX. **A great opportunity for someone like you to develop the tools for the next generation of users and developers!**

Reading other people's code is one of the best ways to progress as a developer. Check out the below commit of this chapter release as it contains all the account refactoring and adjusted tests.



Study Code

Implements go-ethereum crypto for accounts authentication
a8979a⁹¹

91 <https://github.com/web3coach/the-blockchain-bar/commit/a8979a2a26defddba90070e3701f098ea14d4b6f>

MySQL vs Blockchain Authorization

Saturday, January 4.

Using 32 bytes secure hash of a Public Key for users authentication seems fancy but is not enough. Andrej can still create an unauthorized TX, stealing TBB tokens from BabaYaga, given all account addresses are public.

```
curl --location --request POST 'localhost:8080/tx/add' \
--header 'Content-Type: application/json' \
--data-raw '{
    "from": "0x21973d33e048f5ce006fd7b41f51725c30e4b76b",
    "to": "0x22ba1f80452e6220c7cc6ea2d1e3eeddac5f694a",
    "value": 100
}'
```

How can Andrej prevent this?

Andrej analyses his current web 2.0 authorization tactics to see if he could reuse the same solution in a distributed, decentralized system. In a traditional PHP/Java HTTP API, the endpoint is protected by comparing against a secure session:

```
func TxAddHandler(w ResponseWriter, r *Req) {
    // oh Andrej misses the web 2.0 simplicity...
    if session->getUser() != r.Body().GetByKey("from") {
        w.WriteHeader(http.StatusForbidden)
        w.Write("You are not who you claim to be!")
        return
    }

    // All good!
    // Perform the balance transfer and save it in a MySQL DB.
}
```

The problem is... This solution works because there is only one **centralized company server doing the authorization**, behind a firewall.

No one can access the server except the one trustworthy, underpaid company's sysadmin (or the new rich \$\$\$ DevOps version of him). When you say that loud... it doesn't sound that secure, anyway.

In the blockchain peer-to-peer world, this won't work because every peer is its own server, node. A malicious developer, sysadmin could modify the source code, remove the verification code, and other peers would have no way to verify **if BabaYaga was the one who authorized the TX** so they would synchronize this DB change.

Take a pause and think. The solution is in the last sentence. How could Andrej implement the authorization in a distributed and decentralized system?

Hint: BabaYaga must be the one who is performing the authorization. No particular node, server, or another peer.

Do you know the answer?

Congratulation! You are a great blockchain architect! For the developers who are reading this book to learn this, praise as well! You are now going to learn a fundamental, core blockchain architecture skill: **“Digital signatures”!**

Madam/Sir, Sign this Database Change

Sunday, January 5.

Andrej gets a cafe. This will be a difficult core blockchain chapter.

From Wikipedia:⁹²

Digital signatures are a standard element of most cryptographic protocol suites. They are commonly used for software distribution, financial transactions, [contract management software](#),⁹³ and in other cases where it is important to detect forgery or [tampering](#).⁹⁴

“For financial transactions... where it is important to detect forgery or tampering”.

Signing is a technique to produce a **verifiable proof** that someone did exactly what they say they did, and only they could have done it. No other entity could have forged it. In the blockchain world, you use signing to generate a mathematical proof that you, as a specific user, did authorize the transaction yourself. It seems like a perfect cryptographic technique to solve Andrej's design flaw.

⁹² https://en.wikipedia.org/wiki/Digital_signature

⁹³ https://en.wikipedia.org/wiki/Contract_management_software

⁹⁴ [https://en.wikipedia.org/wiki/Tampering_\(crime\)](https://en.wikipedia.org/wiki/Tampering_(crime))

1) In theory

The ECDSA secp256k1 crypto library that generated Andrej's Private Key also provides a signing function expecting two arguments:

1. The first argument is the **user's message** (claim/action) to sign (authorize) in a 32 bytes format produced by a secure hash function
2. The second argument is the **Private Key** itself (representing a unique user)

```
signatureProof = ECDSA_sign_function(hash(message), privatekey)
```

This theory is implemented in the previously explored secp256k1 C library in the following manner. [Open image in full screen.](#)⁹⁵

⁹⁵ https://web3coach-public.s3.eu-central-1.amazonaws.com/img/sign_secp256k1.png

```

// Sign creates a recoverable ECDSA signature.
// The produced signature is in the 65-byte [R || S || V] format where V is 0 or 1.
//
// The caller is responsible for ensuring that msg cannot be chosen
// directly by an attacker. It is usually preferable to use a cryptographic
// hash function on any input before handing it to this function.
func Sign(msg []byte, seckey []byte) ([]byte, error) {
    if len(msg) != 32 {
        return nil, ErrInvalidMsgLen
    }
    if len(seckey) != 32 {
        return nil, ErrInvalidKey
    }
    seckeydata := (*C.uchar)(unsafe.Pointer(&seckey[0]))
    if C.secp256k1_ec_seckey_verify(context, seckeydata) != 1 {
        return nil, ErrInvalidKey
    }

    var (
        msgdata    = (*C.uchar)(unsafe.Pointer(&msg[0]))
        noncefunc = C.secp256k1_nonce_function_rfc6979
        sigstruct C.struct_secp256k1_ecdsa_recoverable_signature
    )
    if C.secp256k1_ecdsa_sign_recoverable(context, &sigstruct, msgdata, seckeydata, noncefunc, nil) {
        return nil, ErrSignFailed
    }

    var (
        sig     = make([]byte, 65)
        sigdata = (*C.uchar)(unsafe.Pointer(&sig[0]))
        recid   C.int
    )
    C.secp256k1_ecdsa_recoverable_signature_serialize_compact(context, sigdata, &recid, &sigstruct)
    sig[64] = byte(recid) // add back recid to get 65 bytes sig
    return sig, nil
}

```

sign secp256k1

[Open image in full screen.](#)⁹⁶

The signature output consists of 3 math variables: **v**, **r**, **s** necessary for the Elliptic curve cryptography math to work. You don't need to know how the math works to be a productive blockchain developer but if you are curious and would like to explore it in detail, Andrej recommends an article⁹⁷ on Elliptic cryptography written by the creator of Ethereum himself: **Vitalik Buterin**.

⁹⁶ https://web3coach-public.s3.eu-central-1.amazonaws.com/img/sign_secp256k1.png

⁹⁷ <https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627>

2) In day to day practice

Based on the ECDSA sign theory, Andrej programs a wrapper function inside his wallet package:

```
func Sign(msg []byte, privKey *ecdsa.PrivateKey) (sig []byte, error) {
    // Hash the message to 32 bytes
    msgHash := crypto.Keccak256(msg)

    // Sign the message using the Private Key
    sig, err = crypto.Sign(msgHash, privKey)
    if err != nil {
        return nil, err
    }

    // Verify the length
    if len(sig) != crypto.SignatureLength {
        return nil, fmt.Errorf(
            "wrong size for signature: got %d, want %d",
            len(sig),
            crypto.SignatureLength,
        )
    }

    return sig, nil
}
```

And of course, he also programs a test proving the theory:

```
func TestSignCryptoParams(t *testing.T) {
    // Generate the key on the fly
    privKey, err := ecdsa.GenerateKey(crypto.S256(), rand.Reader)
    if err != nil {
        t.Fatal(err)
    }
    spew.Dump(privKey)

    // Prepare a message to digitally sign
    msg := []byte("the Web3Coach students are awesome")

    // Sign it
    sig, err := Sign(msg, privKey)
    if err != nil {
        t.Fatal(err)
    }

    // Verify the length is 65 bytes
    if len(sig) != crypto.SignatureLength {
        t.Fatal(fmt.Errorf("wrong size for signature: got %d, want %d",
            len(sig), crypto.SignatureLength))
    }

    // Print the 3 required Ethereum signature crypto values
    r := new(big.Int).SetBytes(sig[:32])
    s := new(big.Int).SetBytes(sig[32:64])
    v := new(big.Int).SetBytes([]byte{sig[64]})

    spew.Dump(r, s, v)
}
```

Practice time.

```
>_ go test ./wallet -test.v -test.run
^TestSignCryptoParams$
```

R, S, V curve attributes:

```
// (*ecdsa.PrivateKey)(0xc000099980)({
// PublicKey: (ecdsa.PublicKey) {
//   Curve: (*secp256k1.BitCurve)(0xc0000982d0)({
//     BitSize: (int) 256
//   }),
//   X: (*big.Int)(134416086130162441...1047269353924650464711),
//   Y: (*big.Int)(735240064515671280...3711113049610951453654)
// }

(*big.Int)(0xc0000b1b20)(881812922...74656)
(*big.Int)(0xc0000b1b40)(234767225...97049)
(*big.Int)(0xc0000b1b60)(1)
```

So far, so good!

BUT, signing is useless on its own. We must be able to also **verify the signature**. Together, **Signature Generation + Signature Verification** business logic is what completes this core blockchain authorization design pattern.

The **signature verification** works in the opposite direction. It's a function expecting the **original message + signed signature**, and the secp256k1 lib will attempt to recover the Public Key from this combination.

```
func Verify(msg, sig []byte) (*ecdsa.PublicKey, error)
```

The Verify function will internally call the secp256k1.RecoverPubKey():

```
// Ecrecover returns the public key that created the signature
func Ecrecover(hash, sig []byte) ([]byte, error) {
    return secp256k1.RecoverPubkey(hash, sig)
}
```

If you provide a correct **msg** and **signature**, you get back a **Public Key**, matching exactly the signer's Public Key. And as you already know, by knowing someone's Public Key, you can calculate his account address, his blockchain identity.

Andrej extends his signature generation test and attempts to recover the original account address from the signed signature, proving what blockchain account signed the message. [Open image in full screen.](#)⁹⁸

```

func TestSign(t *testing.T) {
    privKey, err := ecdsa.GenerateKey(crypto.S256(), rand.Reader)
    if err != nil {
        t.Fatal(err)
    }

    pubKey := privKey.PublicKey
    pubKeyBytes := elliptic.Marshal(crypto.S256(), pubKey.X, pubKey.Y)
    pubKeyBytesHash := crypto.Keccak256(pubKeyBytes[1:])

    account := common.BytesToAddress(pubKeyBytesHash[12:])

    msg := []byte("the Web3Coach students are awesome")

    sig, err := Sign(msg, privKey)
    if err != nil {
        t.Fatal(err)
    }

    recoveredPubKey, err := Verify(msg, sig)
    if err != nil {
        t.Fatal(err)
    }

    recoveredPubKeyBytes := elliptic.Marshal(crypto.S256(), recoveredPubKey.X, recoveredPubKey.Y)
    recoveredPubKeyBytesHash := crypto.Keccak256(recoveredPubKeyBytes[1:])
    recoveredAccount := common.BytesToAddress(recoveredPubKeyBytesHash[1:])

    if account.Hex() != recoveredAccount.Hex() {
        t.Fatal("msg was signed by account #1")
    }
}

```

Annotations on the code:

- New in-memory key**: Points to the line `privKey, err := ecdsa.GenerateKey(crypto.S256(), rand.Reader)`.
- 1) Convert graph points to 65 bytes**: Points to the line `pubKeyBytes := elliptic.Marshal(crypto.S256(), pubKey.X, pubKey.Y)`.
- 2) Hash first 64 bytes into a [32]byte hash**: Points to the line `pubKeyBytesHash := crypto.Keccak256(pubKeyBytes[1:])`.
- 3) Sign, Verify, Recover the Account and compare the results. Will match.**: Points to the final comparison line `if account.Hex() != recoveredAccount.Hex() { t.Fatal("msg was signed by account #1") }`.

Hex dump of the recovered public key bytes (highlighted in blue box):

```
([]uint8) (len=32 cap=32) {
00000000 04 3b 35 c0 5b 04
00000010 a4 8d 13 43 30 04
00000020 f5 bb 01 d0 0d 23
00000030 db 46 d7 d9 d7 13
00000040 6f
```

Hex dump of the recovered account address (highlighted in green box):

```
([]uint8) (len=20 cap=20) 0xc79aC444899C7Ecf26c81edf2c4670356c9E69D1
```

sign explained

⁹⁸ https://web3coach-public.s3.eu-central-1.amazonaws.com/img/test_sign_secp256k1.png

```
func TestSign(t *testing.T) {
    // Generate Private Key on the fly
    privKey, err := ecdsa.GenerateKey(crypto.S256(), rand.Reader)
    if err != nil {
        t.Fatal(err)
    }

    // Convert the Public Key to bytes with elliptic curve settings
    pubKey := privKey.PublicKey
    pubKeyBytes := elliptic.Marshal(crypto.S256(), pubKey.X, pubKey.Y)

    // Hash the Public Key to 32 bytes
    pubKeyBytesHash := crypto.Keccak256(pubKeyBytes[1:])

    // The last 20 bytes of the Public Key hash will
    // be it's public username
    account := common.BytesToAddress(pubKeyBytesHash[12:])

    msg := []byte("the Web3Coach students are awesome")

    // Sign a message -> generate message's signature
    sig, err := Sign(msg, privKey)
    if err != nil {
        t.Fatal(err)
    }

    // Recover a Public Key from the signature
    recoveredPubKey, err := Verify(msg, sig)
    if err != nil {
        t.Fatal(err)
    }

    // Convert the Public Key to username again
    recoveredPubKeyBytes := elliptic.Marshal(
        crypto.S256(),
        recoveredPubKey.X,
```

```

        recoveredPubKey.Y,
    )
    recoveredPubKeyBytesHash := crypto.Keccak256(
        recoveredPubKeyBytes[1:],
    )
    recoveredAccount := common.BytesToAddress(
        recoveredPubKeyBytesHash[12:],
    )

// Compare the username matches meaning,
// The signature generation and account verification
// by extracting the Pub Key from signature works.
if account.Hex() != recoveredAccount.Hex() {
    t.Fatalf(
        "msg was signed by account %s but signature recovery produ\\
ced an account %s",
        account.Hex(),
        recoveredAccount.Hex())
}
}
```

Practice time.

>_ go test ./wallet -test.v -test.run ^TestSign\$

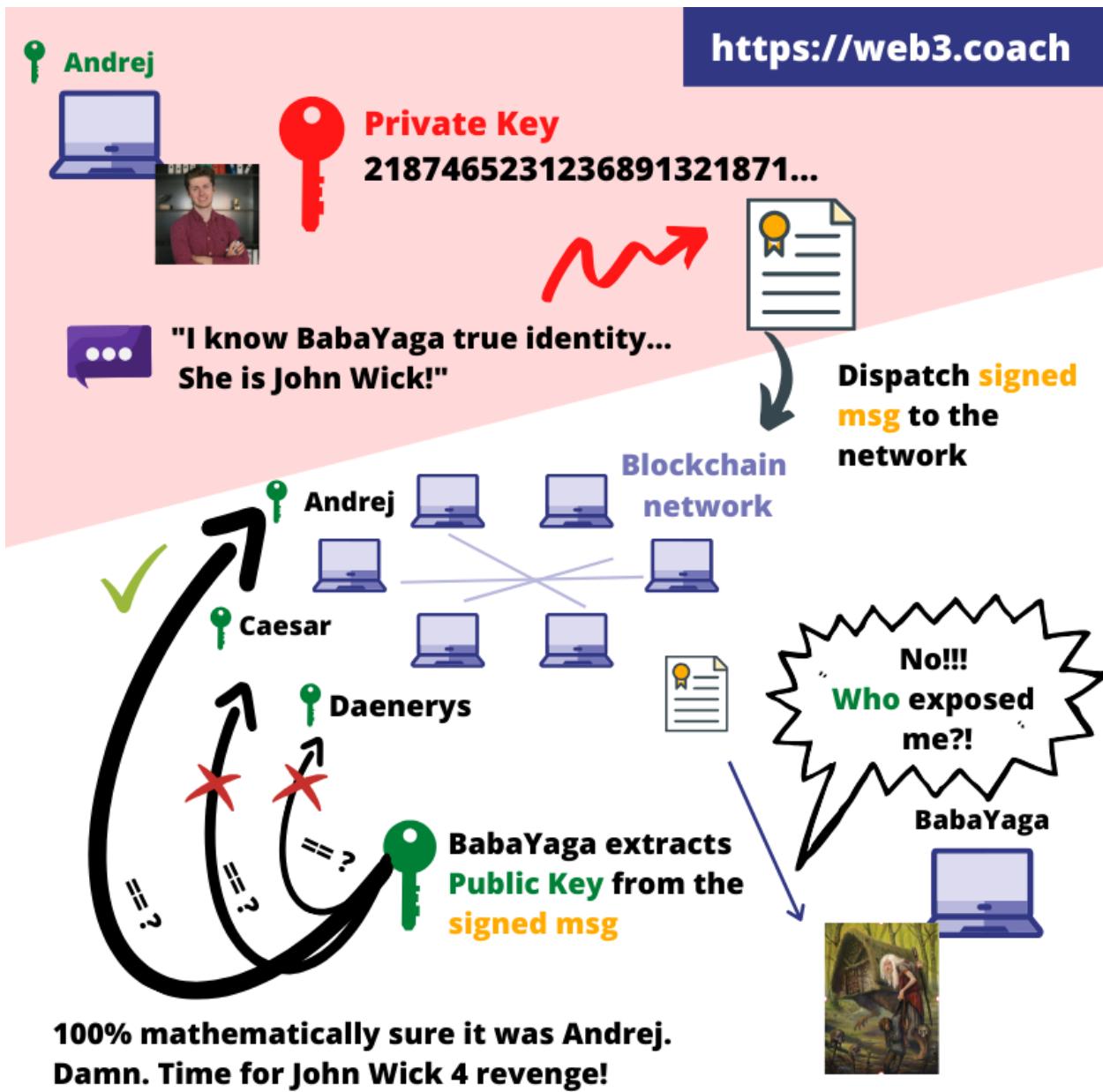
```

== RUN TestSign
--- PASS: TestSign (0.00s)
PASS
```

Done! Test passed! Signer account matches the recovered account from signature.

Last words. Notice one particular curiosity. Andrej previously said the account's address is calculated as a secure keccak256 hash of the Public Key. Well, that is not fully true. It's actually calculated from the **last 20 bytes of the Public Key hash**. You can see it above common.BytesToAddress(**pubKeyBytesHash[12:]**) . Given the hash is always of a fixed length, 32 bytes, the first 12 bytes are ignored. Why? It's just an Ethereum convention. It helps to generate unique accounts, so other blockchains such as Bitcoin relying on the same cryptography don't share the same account addresses to prevent human mistakes.

Cryptographic Blockchain Authorization Summary Diagram:





Fun Facts

Mathematical signatures are much more secure than the physical ones in a bank that we draw on a piece of paper. Not to even mention the practicality of digital signatures in the era of an online world. Ah, the legacy systems “securing” all our money and houses.

This chapter highlights why you made a good choice buying this book and learning about programming new, secure applications of Web 3.0.



Study Code

Adds Sign() and Verify() functions into wallet package
[7890a4⁹⁹](#)

⁹⁹ <https://github.com/web3coach/the-blockchain-bar/commit/686d9233d6bcf63e701b28f3e0d1bd86a366cb63>

Programming Signed Transactions

Monday, January 6.

Back to the original problem. Andrej knows how digital signatures work now; **it's time to start signing the TBB transactions** and secure the network, without any firewall or centralized server, in a fully decentralized manner.

Andrej creates a new struct containing the transaction and its signature:

```
type Tx struct {
    From  common.Address `json:"from"`
    To    common.Address `json:"to"`
    Value uint          `json:"value"`
    Data  string         `json:"data"`
    Time  uint64         `json:"time"`
}

// New struct encapsulating TX + Signature
type SignedTx struct {
    Tx
    Sig []byte `json:"signature"`
}
```

Signed the transaction itself will be identical to the previous chapter. The only difference will be the content. Previously, Andrej was testing the signing on a dummy message:

```
msg := []byte("the Web3Coach students are awesome")
```

Now he will sign a message containing a JSON encoded TX. Andrej also creates a new `SignTx()` method to encode the TX into a slice of bytes and internally call the `Sign(msg []byte)` function from the previous chapter.

```
func SignTx(tx Tx, privKey *ecdsa.PrivateKey) (SignedTx, error) {
    rawTx, err := tx.Encode()
    if err != nil {
        return database.SignedTx{}, err
    }

    sig, err := Sign(rawTx, privKey)
    if err != nil {
        return database.SignedTx{}, err
    }

    return database.NewSignedTx(tx, sig), nil
}
```

The output is an almighty: `SignedTx`. Andrej programs one more helper function in the `wallet` package: `SignTxWithKeystoreAccount()` to make it easier for the Node user to sign his transactions directly through the already existing `/tx/add` HTTP endpoint. Of course, to sign the TX using the user's Private Key, the user must provide his password to decrypt the Keystore file containing the key.

Technically, the `Tx.From` attribute is not necessary. Because every Transaction is signed, you can reconstruct this attribute from a Public Key, which is extractable from the Signature, as Andrej did few

lines above: recoveredPubKey, err := Verify(msg, sig) and recoveredAccount := common.BytesToAddress(recoveredPubKeyBytesHash[12:]). Removing the from attribute would save you some network usage as every Transaction would have fewer bytes, but for the sake of learning and simplicity, let's stick to having it explicitly included.

Andrej adds new password attribute into the TxAddReq:

```
type TxAddReq struct {
    From      string `json:"from"`
    FromPwd  string `json:"from_pwd" // to decrypt the keystore`
    To        string `json:"to"`
    Value     uint   `json:"value"`
    Data      string `json:"data"`
}
```

And he modifies the txAddHandler() to sign the transaction before its added to the node's pending transactions list:

```
func txAddHandler(w ResWriter, r *Request, node *Node) {
    // ...
    // Decrypt the Private key stored in Keystore file
    // and Sign the TX
    signedTx, err := wallet.SignTxWithKeystoreAccount(
        tx,
        from,
        req.FromPwd,
        wallet.GetKeystoreDirPath(node.dataDir),
    )
    if err != nil {
        writeErrRes(w, err)
        return
    }

    // Add TX to the Mempool, ready to be mined
    err = node.AddPendingTX(signedTx, node.info)
    if err != nil {
        writeErrRes(w, err)
        return
    }
}
```

```
    writeRes(w, TxAddRes{Success: true})
}
```

Then he goes ahead and replaces all the usages of **raw Tx** with **SignedTx** across the codebase. See commit at the end of the chapter.

The next step is crucial. As you know, signing itself is not sufficient without the verification - business logic. Andrej assigns a new function, `IsAuthentic`, to the `SignedTx` struct. The function recovers the account address who has signed the signature and compares it with the `tx.from` attribute. If the signature matches the sender, Andrej can be 100% sure the `tx.From` account authorized the balance transfer, and no hacker could have forged it.

```
func (t Tx) Hash() (Hash, error) {
    // Encode all TX content to JSON
    txJson, err := t.Encode()
    if err != nil {
        return Hash{}, err
    }

    // Hash it to 32 bytes compatible with signing function
    return sha256.Sum256(txJson), nil
}

func (t SignedTx) IsAuthentic() (bool, error) {
    // Convert to 32 bytes hash
    txHash, err := t.Tx.Hash()
    if err != nil {
        return false, err
    }
```

```

// Verify if the signature is compatible with this msg (TX)
recoveredPubKey, err := crypto.SigToPub(txHash[:], t.Sig)
if err != nil {
    return false, err
}

// Convert the recovered Pub Key to an Account
recoveredPubKeyBytes := elliptic.Marshal(
    crypto.S256(),
    recoveredPubKey.X,
    recoveredPubKey.Y,
)
recoveredPubKeyBytesHash := crypto.Keccak256(
    recoveredPubKeyBytes[1:],
)
recoveredAccount := common.BytesToAddress(
    recoveredPubKeyBytesHash[12:],
)

// Compare the signature owner with TX owner
return recoveredAccount.Hex() == t.From.Hex(), nil
}

```

Where should you call the IsAuthentic function?

Directly, in the heart of blockchain database business logic.
In the State component.

Before you add the transaction into the blockchain, the State component checks whenever the sender has sufficient balance, etc. Now, it also checks whenever the sender authorized the transaction himself by signing it, and ONLY then the balance transfer happens:

```
func applyTx(tx SignedTx, s *State) error {
    // Verify the TX was not forged
    ok, err := tx.IsAuthentic()
    if err != nil {
        return err
    }

    if !ok {
        return fmt.Errorf(
            "wrong TX. Sender '%s' is forged", tx.From.String(),
        )
    }
}

// rest of the validation...

s.Balances[tx.From] -= tx.Value
s.Balances[tx.To] += tx.Value

return nil
}
```

The last thing to do is, of course, adjust all the integration tests to work with the new SignedTx business logic. For this, Andrej generates two Keystore files representing Andrej and BabaYaga in his test suite and makes the generation of `Genesis.json` more flexible so he can configure enough test tokens, balances for the pre-generated Keystore test accounts before the individual integration tests are run.

>_ ls -la ./node

```
-rw-rw-r-- http_req_res_utils.go  
-rw-rw-r-- http_routes.go  
-rw-rw-r-- miner.go  
-rw-rw-r-- miner_test.go  
-rw-rw-r-- node.go  
-rw-rw-r-- node_integration_test.go  
-rw-rw-r-- sync.go  
  
// pre-generated keystore files for testing purposes  
-rw-rw-r-- test_andrej--3eb92807f1f91a8d4d85bc908c7f86dcddb1df57  
-rw-rw-r-- test_babayaga--6fdc0d8d15ae6b4ebf45c52fd2aafbcbb19a65c8
```

Phew. A lot of information. But the blockchain is now officially secured using asymmetric cryptography! Andrej encourages you to check out the full commit and review all the adjusted and new tests.

Congratulation! You learned the **Digital Signatures** design pattern!



Study Code

Requires TX to be Signed before added to blockchain
[958752¹⁰⁰](#)

¹⁰⁰ <https://github.com/web3coach/the-blockchain-bar/commit/7e2369f0cbab0ffa16e8d5ceba7e76dcab88027d>

Digital Signature Replay Attack

Tuesday, January 7.

Andrej fixed the fundamental security flaw but accidentally created another one. Can you spot it?

How can be the TX signature abused in the current implementation?

Did you find the security flaw?

Congratulation! You are excellent at breaking Digital Signatures design pattern! For the developers new to cryptography hell, what happens if:

1. Andrej creates a TX transferring 5 TBB tokens to BabaYaga
2. Andrej signs the TX and submits it to the network
3. The TX gets mined by one of the miners and gets successfully added to the blockchain
4. Greedy BabaYaga creates a new, almost identically looking TX (**with only different TX.time**) and attaches Andrej's signature from previous TX to it
5. BabaYaga submits the TX to the network
6. What happens?
7. Well, because the TX hash will be different given the TX.time attribute changed, the TX will not be authentic, and it won't be possible to recover the Andrej's Public Key from the signature. The TX will be marked as forged and rejected by the network
8. Great!

Here is a test doing exactly this and proving it:

```
// Expect:
//     ERROR: wrong TX. Sender '0x3EB9....' is forged
func TestNode_ForgedTx(t *testing.T) {
    dataDir, andrej, babaYaga, err := setupTestNodeDir()
    if err != nil {
        t.Error(err)
    }
    defer fs.RemoveDir(dataDir)

    n := New(dataDir, "127.0.0.1", 8085, andrej, PeerNode{})
    ctx, closeNode := context.WithTimeout(
        context.Background(),
        time.Minute*30,
    )

    andrejPeerNode := NewPeerNode(
        "127.0.0.1", 8085, false, andrej, true,
    )

    txValue := uint(5)
    tx := database.NewTx(andrej, babaYaga, txValue, "")

    // Create a valid TX transferring 5 TBB tokens
    // from Andrej to BabaYaga
    validSignedTx, err := wallet.SignTxWithKeystoreAccount(
        tx,
        andrej,
        testKsAccountsPwd,
        wallet.GetKeystoreDirPath(dataDir),
    )
    if err != nil {
        t.Error(err)
        return
    }

    if _, err := database.CommitTx(tx, validSignedTx); err != nil {
        t.Error(err)
    }
}
```

```

}

// Trigger mining
_ = n.AddPendingTX(validSignedTx, andrejPeerNode)

go func() {
    ticker := time.NewTicker(
        time.Second * (miningIntervalSeconds - 3),
    )
    wasForgedTxAdded := false

    for {
        select {
        case <-ticker.C:
            if !n.state.LatestBlockHash().IsEmpty() {
                if wasForgedTxAdded && !n.isMining {
                    closeNode()
                    return
                }

                if !wasForgedTxAdded {
                    // Attempt to forge the same TX but with
                    // modified time.
                    //
                    // Because the TX.time changed, the
                    // TX.signature will be considered forged.
                    //
                    // database.NewTx() changes the TX time
                    forgedTx := database.NewTx(
                        andrej,
                        babaYaga,
                        txValue,
                    )

                    // Use the signature from a valid TX
                    forgedSignedTx := database.NewSignedTx(

```


Because, `NewTx()` generates a new time, every time:

```
func NewTx(from, to common.Address, value uint, data string) Tx {
    return Tx{from, to, value, data, uint64(time.Now().Unix())}
}
```

Notice the test output. Identical TX but with a different time.

>_ go test ./node -timeout=0 -test.v -test.run ^TestNode_ForgedTx

```
== RUN TestNode_ForgedTx
Listening on: 127.0.0.1:8085
Blockchain state:
- height: 0
- hash: 000000...000000

// Original TX made by Andrej
Added Pending TX {
    "from": "0x3eb92807f1f91a8d4d85bc908c7f86dcddb1df57",
    "to": "0x6fdc0d8d15ae6b4ebf45c52fd2aafbcbb19a65c8",
    "value": 5,
    "data": "",
    "time": 1583337929,
    "signature": "346iQ81poGFAx8jcx4YKkFUpAdKYULo9zXYhGLzkDh8Cmy1WJJZOZ0\HoF+URuZi7kleIhfLT9WFYPpNooD1MqXwE="} from Peer 127.0.0.1:8085
```

Mining 1 Pending TXs. Attempt: 1

```
// Forged Andrej's TX by BabaYaga
Added Pending TX {
    "from": "0x3eb92807f1f91a8d4d85bc908c7f86dcddb1df57",
    "to": "0x6fdc0d8d15ae6b4ebf45c52fd2aafbcbb19a65c8",
```

```
"value":5,  
"data":"",
"time":1583337941, // MODIFIED TIME  
"signature":"346iQ81poGFAx8jcx4YKkFUpAdKYULo9zXYhGLzkDh8Cmy1WJJZO\HoF+URuZi7kleIhfLT9WFYPpNooD1MqXwE="} from Peer 127.0.0.1:8085  
...  
...  
ERROR: wrong TX. Sender '0x3EB928...B1Df57' is forged  
--- PASS: TestNode_ForgedTx (900.00s)
```

Okay BabaYaga was not able to forge the TX. Great. But! What happens if BabaYaga repeats this exact attack but doesn't modify the TX time?

What happens if BabaYaga replays, the same, valid, authentic TX from Andrej, AGAIN. Given the TX wasn't forged, Andrej did sign the TX at some point in the past in an attempt to transfer 5 TBB tokens to BabaYaga, the TX will pass the blockchain validation, **and BabaYaga will end up with a double balance.** She can repeat the process until she drains all of Andrej's funds, his entire tokens balance. Ooops.

Andrej writes a new TestNode_ForgedTx proving the attack before he attempts to fix it in a true TDD fashion. Same test, the only difference is that BabaYaga won't modify anything - she will simply re-submit the same valid TX again.

```
func TestNode_ForgedTx(t *testing.T) {
    dataDir, andrej, babaYaga, err := setupTestNodeDir()
    if err != nil {
        t.Error(err)
    }
    defer fs.RemoveDir(dataDir)

    n := New(dataDir, "127.0.0.1", 8085, andrej, PeerNode{})
    ctx, closeNode := context.WithTimeout(
        context.Background(),
        time.Minute*30,
    )

    andrejPeerNode := NewPeerNode(
        "127.0.0.1", 8085, false, andrej, true,
    )

    txValue := uint(5)
    tx := database.NewTx(andrej, babaYaga, txValue, "")

    // Create a valid TX transferring 5 TBB tokens
    // from Andrej to BabaYaga
    validSignedTx, err := wallet.SignTxWithKeystoreAccount(
        tx,
        andrej,
        testKsAccountsPwd,
        wallet.GetKeystoreDirPath(dataDir),
    )
    if err != nil {
        t.Error(err)
```

```

    return
}

// Trigger mining
_ = n.AddPendingTX(validSignedTx, andrejPeerNode)

go func() {
    ticker := time.NewTicker(
        time.Second * (miningIntervalSeconds - 3),
    )
    wasForgedTxAdded := false

    for {
        select {
        case <-ticker.C:
            if !n.state.LatestBlockHash().IsEmpty() {
                if wasForgedTxAdded && !n.isMining {
                    closeNode()
                    return
                }

                if !wasForgedTxAdded {
                    // Attempt to forge the same TX but with
                    // modified time.
                    //
                    // Because the TX.time changed, the
                    // TX.signature will be considered forged.

                    // Re-submit the TX
_ = n.AddPendingTX(
    validSignedTx,
    andrejPeerNode,
)
                wasForgedTxAdded = true
            }
        }
    }
}

```

```
        time.Sleep(time.Second * (miningIntervalSecond\
s + 3))
    }
}
}
}
}()

_ = n.Run(ctx)

if n.state.LatestBlock().Header.Number != 0 {
    t.Fatal("should mine only one TX. The second TX was forged")
}

if n.state.Balances[babaYaga] != txValue {
    t.Fatal("forged tx succeeded")
}
}
```

The block containing the repeated malicious TX gets mined, proving the attack worked.

>_ go test ./node -timeout=0 -test.v -test.run ^TestNode_-ReplayedTx

```
==== RUN    TestNode_ReplayedTx
Added Pending TX {
  "from": "0x3eb9280...",
  "to": "0x6fdc0d8",
  "value": 5,
  "data": "",
  "time": 1590584906,
  "signature": "xSLLEyrPXBmpbhGjLkpfTReecM475c30TkTN/x8ZoXReGIyA8JANk\
6HLiZXBrnFqqb1L0hCVDTgXKMs52+Co+wA="
}
from Peer 127.0.0.1:8085
```

Persisting new Block to disk:

```
{"hash": "000000208cbc...ad219",
"block": {
  "header": {"parent": "0000001ae5df...28d44",
  "number": 1,
  "nonce": 1329297332,
  "time": 1583341503,
  "miner": "0x3eb928...b1df57"},
  "payload": [
    {"from": "0x3eb92807f1f91a8d4d85bc908c7f86dcddb1df57",
     "to": "0x6fdc0d8d15ae6b4ebf45c52fd2aaafbcb19a65c8",
     "value": 5,
     "data": "",
     "time": 1583341082,
     "signature": "pRYFUf/r+xpmMyCW0ZkwGCzA+vRw81ndrK1NqLwFAcg05Tgb1\
jqLo0e8+neySIH7i2++k08r0LrvqCMEqc0wKAA="}]}}
```

```
--- FAIL: TestNode_ReplayedTx (480.65s)
  node_integration_test.go:204: replayed attack was successful
```

```
FAIL    github.com/web3coach/the-blockchain-bar/node    480.693s
```

How can Andrej defend against this attack?

He reverse-engineers the go-Ethereum implementation and discovers a new TX attribute: "**AccountNonce**," often referred via shorten version: "**nonce**."

Do not confuse this attribute with the block's nonce used in the mining process!

The tx.Nonce is an incremental number representing the order, count of transactions sent from a particular account. Each time an account signs and sends a transaction, the next **TX nonce** must be incremented by 1.

If Andrej increases the nonce of every TX he signs, together with the rest of the attributes, he ensures all of his transactions are unique, and no JohnWick (BabaYaga) can replay them. Why not? Because he also modifies the **blockchain' state component to keep track of the current nonce value of every account** in the network.

Identically how every:

```
new block number = current block number + 1
```

so every:

```
new account tx nonce = current account tx nonce + 1
```

Andrej pulls up his sleeves and programs this defensive mechanism.
New tx.nonce attribute:

```
type Tx struct {
    From  common.Address `json:"from"`
    To    common.Address `json:"to"`
    Value uint          `json:"value"`
    Nonce uint          `json:"nonce" // new attribute`
    Data  string         `json:"data"`
    Time  uint64         `json:"time"`
}
```

New state.Account2Nonce map attribute to keep a track of current nonce for every account:

```
type State struct {
    Balances      map[common.Address]uint
    Account2Nonce map[common.Address]uint // new map
}
```

Before adding a new TX via the /tx/add HTTP endpoint, increment its nonce:

```

func (s *State) GetNextAccountNonce(account common.Address) uint {
    return s.Account2Nonce[account] + 1
}

func txAddHandler(w ResWriter, r *Request, node *Node) {
    // ...
    nonce := node.state.GetNextAccountNonce(from)

    tx := database.NewTx(
        from,
        database.NewAccount(req.To),
        req.Value,
        nonce, // TX order count
        req.Data
    )

    // ...
}

```

The State component now also validates the next expected account nonce for every TX:

```

func applyTx(tx SignedTx, s *State) error {
    ok, err := tx.IsValid()
    if err != nil {
        return err
    }

    if !ok {
        return fmt.Errorf("wrong TX. Sender '%s' is forged", tx.From.String())
    }

    expectedNonce := s.GetNextAccountNonce(tx.From)
    if tx.Nonce != expectedNonce {
        return fmt.Errorf(

```

```
"wrong TX. Sender '%s' next nonce must be '%d', not '%d'",  
    tx.From.String(),  
    expectedNonce,  
    tx.Nonce,  
)  
}  
  
// rest of the validation, logic  
// ...  
  
s.Account2Nonce[tx.From] = tx.Nonce  
  
return nil  
}
```

Perfect. Every new submitted TX now has its tx.nonce incremented.

Andrej runs the TestNode_ReplayedTx again and crosses his fingers for the test to succeed; stopping the malicious BabaYaga's attack.

```
>_ go test -timeout=0 -test.v -test.run ^TestNode_-  
ReplayedTx$
```

...While waiting forever until the test finishes, starting to regret, he didn't mock the mining process in tests...

After two Vampire Diaries series and one Lord of The Rings movie, the **test passed!**

```
==== RUN    TestNode_ReplayedTx
Added Pending TX {
  "from": "0x3eb9280...",
  "to": "0x6fdc0d8",
  "value": 5,
  "nonce": 1, // NEW NONCE ATTRIBUTE
  "data": "",
  "time": 1590584906,
  "signature": "xSLLEyrPXBmpbhGjLkpfTReecM475c30TkTN/x8ZoXReGIyA8JANk\
6HLiZXBrnFqqb1L0hCVDTgXKM52+Co+wA="}
from Peer 127.0.0.1:8085
```

```
Mining 1 Pending TXs. Attempt: 10000000
```

```
Mining 1 Pending TXs. Attempt: 13000000
```

```
...
```

```
Mined new Block '000000cbc97aeb63d82aaf9ec...8dd017b99b4' using Pow:
```

```
Height: '1'
```

```
Nonce: '4120480127'  
Created: '1583428820'  
Miner: '0x3EB92807F1f91a8d4d85BC908c7f86dcDB1Df57'  
Parent: '000000f09f6ee260e7fa96d8...c9af75bd157fe'  
  
Attempt: '13788778'  
Time: 3m23.042607037s  
  
// BUT failed to persist it, as expected,  
// due to the new Nonce validation  
  
ERROR: wrong TX. Sender '0x3EB9280...' next nonce must be '2', not '1'  
--- PASS: TestNode_ReplayedTx (357.65s)  
PASS  
ok      github.com/web3coach/the-blockchain-bar/node      357.692s
```

Celebrating a Secure Blockchain

Wednesday, January 8.

Andrej's base blockchain version is ready.

He (and now you) developed a fully decentralized peer-to-peer blockchain.

You secured the database using Bitcoin-like Proof of Work consensus.

You made the database fully transparent and fair, thanks to your sync algorithm.

You used Ethereum's cryptographic library to implement user authentication and authorization without any centralized database storing all users' confidential credentials.

You ensured an uncheatable tax audit is possible thanks to secure hashing functions making your database immutable.

You leveraged a Linked List to optimize the database performance.

You created two interfaces to control the blockchain nodes - CLI and HTTP.

You created a new economy leveraging a cryptocurrency and made the bar's payment system fully autonomous.

Congratulations! Great job!

What's next?

Andrej, BabaYaga, Caesar, and Daenerys are not finished, just yet!

Having a bar powered by blockchain tokens can generate tons of value for the bar's customers. Contrary to the competition and other bars on this street, where the customers only spend money and get a hangover in exchange, TBB customers holding bar's tokens could have shareholders' rights!

Similarly to owning a large portion of stocks in a company like Apple or Microsoft, the customers holding the TBB bar tokens could be able to decide how the bar will operate by voting and deciding on:

- drinks prices
- opening hours
- new features (TV, Jukebox...)
- interior and exterior design
- profits allocation
- etc.

BabaYaga: Wait, Andrej! Decentralized voting? How would this even work?

Andrej: That's correct. All you need is an Ethereum based virtual machine supporting Turing complete Smart Contracts!

Thanks for reading.

Until the next chapter is released ;)

Your Web3Coach, Lukáš



Summary

Closed software with centralized access to private data puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relation. Specifications of the blockchain system should be well defined from the beginning and only change if its users support it.

Blockchain is an immutable, transparent database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.

The database content is hashed by a secure cryptographic hash function. The blockchain participants use the resulted hash to reference a specific database state.

Transactions are grouped into batches for performance reasons. A batch of transactions make a Block. Each block is encoded and hashed using a secure, cryptographic hash function.

Block contains Header and Payload. The Header stores various metadata such a time and a reference to the Parent Block (the previous immutable database state). The Payload carries the new database transactions.

The blockchain network consists of Nodes and Peers. All new peers connect to a default Bootstrap Peer(s) to discover and retrieve the current database state as well as the full transaction history. Peers continually communicate with each other using a programmed sync algorithm. Peers exchange information about new blocks and new network's peers.

Blockchain peers coordinate by following a pre-defined set of rules called "consensus." The primary responsibility of a consensus algorithm is to decide who and how can produce the next valid block.

If consensus can't happen, the database history splits. A database split is called "hard fork." Once hard fork occurs, peers with different blockchain history can't communicate and synchronize because everyone will have a different source of truth.

The most famous consensus algorithm implemented in Bitcoin is Proof of Work. PoW security consists of solving a cryptographic puzzle. Peers call this activity: "mining." Miners repeatedly generate a random number, "nonce," insert the nonce into a block header, and hash the block until the resulted hash matches the pre-defined consensus criteria.

Blockchain releases are complicated and require asynchronous communication between independent parties—the participants with the broadest influence, lobbying, and momentum win.

Blockchain security (authentication and authorization) are implemented with asymmetric cryptography. Asymmetric cryptography is possible thanks to mathematical elliptic curves.

Asymmetric cryptography works with a set of keys: Public key and Private key.

The Public key is derived from the Private Key.

The Ethereum account address is the last 20 bytes of the secure hash of the Public key.

The Private Key is used as a decentralized password for generating Digital Signatures.

Digital Signatures are cryptographically generated proofs of action or identity. Signing is a technique to produce a verifiable proof: Signature.

A user (author) signs a message with his Private key and produces a Signature. Another user (reader) can verify the message authenticity by restoring the author's Public key from the Signature. If the restored Public key matches the author's Public key, the message can be trusted and couldn't be forged.

All blockchain transactions must be signed and verified.



Study Code

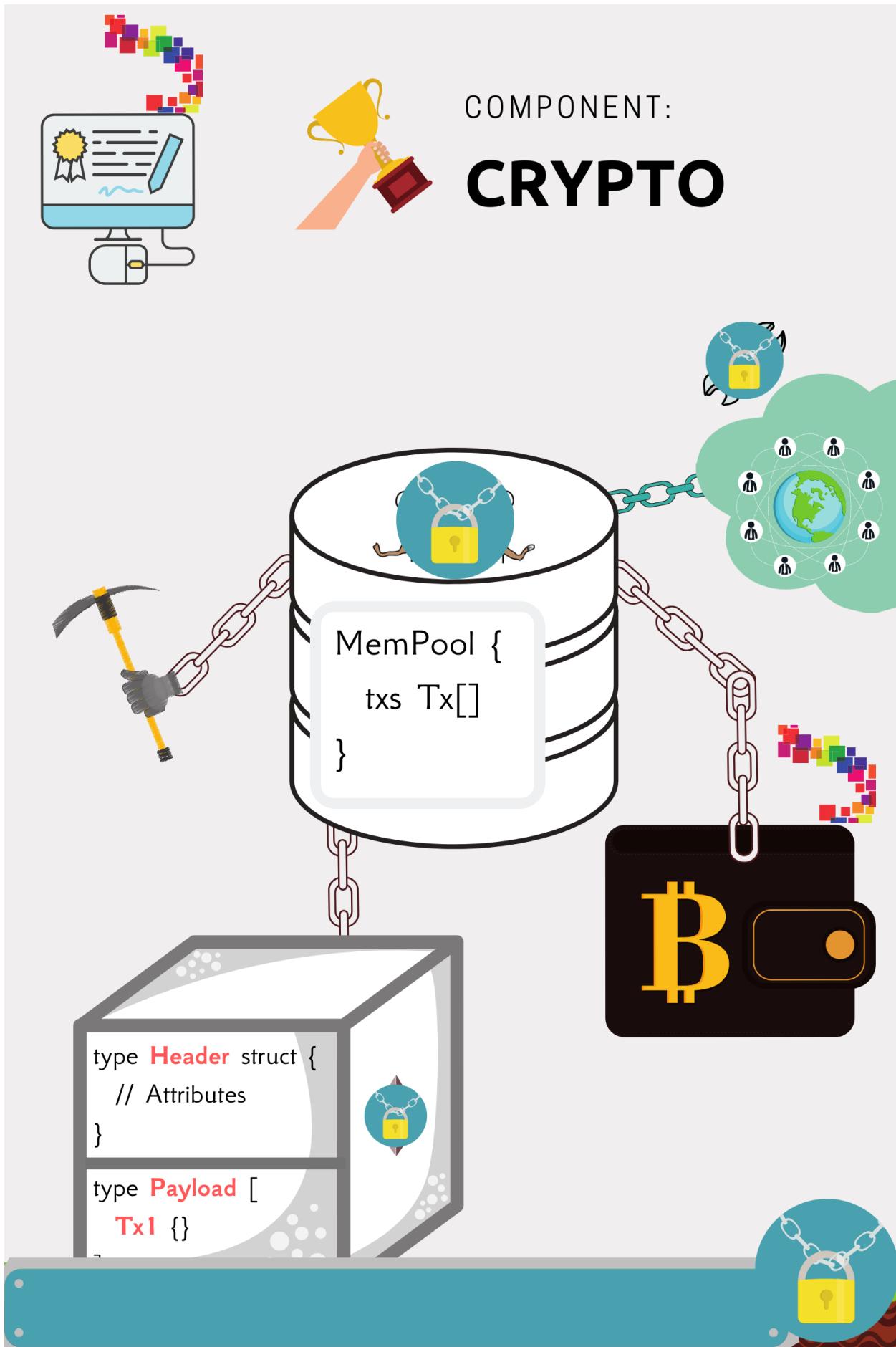
Adds TX.Nonce to prevent signature replay attacks [cc9b5e¹⁰¹](#))

Read an in-depth article on digital signatures:

[https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7¹⁰²](https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7)

¹⁰¹ <https://github.com/web3coach/the-blockchain-bar/commit/cc9b5e0ce93f71918f4b0195a29f1e73ffbd28ed>

¹⁰² <https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7>



13 | Bonus: Why a Transaction Costs Gas

```
>_ git checkout c14_why_transaction_costs_gas
```

The TBB transaction follows the same encapsulation as a transaction on the Ethereum network.

```
type Tx struct {
    From  common.Address `json:"from"`
    To    common.Address `json:"to"`
    Value uint          `json:"value"`
    Nonce uint          `json:"nonce"`
    Data  string         `json:"data"`
    Time  uint64         `json:"time"`
}

type SignedTx struct {
    Tx
    Sig []byte `json:"signature"`
}
```

But, there is one crucial attribute missing. **Gas**.

What's Gas?

Gas is an Ethereum's blockchain term for "fee". Gas/fee defines how much execution of a transaction will cost.

Every TX costs a fee to prevent malicious wallets/nodes from spamming the network.

Each operation has a pre-defined gas cost. To make your transition from the TBB training ledger to a production blockchain like Ethereum the smoothest, I am using the same terminology and values. An Ethereum transaction to transfer funds from Andrej to Babayaga would cost 21 000 Gas.

How much is 1 Gas?

The gas fees are paid in the native currency, TBB tokens on this blockchain, or ETH on Ethereum. The price per Gas is usually dynamically calculated based on the current network activity. If the activity is low, making transfers is cheap. As the activity grows and nodes are getting busy mining all the new transactions, the price for doing operations on the blockchain network grows.

Spamming the Network

Andrej writes an integration test that proves he can make as many transactions as he wants in the current implementation without paying any fee.

He extends the `./node/node_integration_test.go`. The test starts with allocating 1000 TBB tokens to Andrej at Genesis and setting up his mining node.

```
func TestNode_MiningSpamTransactions(t *testing.T) {
    andrejBalance := uint(1000)
    babaYagaBalance := uint(0)
    dataDir, andrej, babaYaga, err := setupTestNodeDir(andrejBalance)
    if err != nil {
        t.Error(err)
    }
    defer fs.RemoveDir(dataDir)

    n := New(dataDir, "127.0.0.1", 8085, andrej, PeerNode{})
    ctx, closeNode := context.WithCancel(context.Background())
    andrejPeerNode := NewPeerNode("127.0.0.1", 8085, false, andrej, true)
```

The test will broadcast 4 TXs transferring 200 TBB tokens each to the network from Andrej to BabaYaga. This is problematic because instead of broadcasting 4 TXs to the network, he could broadcast 4M tiny TXs, and the nodes would have to process it. He intentionally spammed the network as he could have sent 1 TX of 800 TBB tokens (4x 200).

```

txValue := uint(200)

// Schedule 4 transfers from Andrej -> BabaYaga
txCount := uint(4)
for i := uint(1); i <= txCount; i++ {
    txNonce := i
    tx := database.NewTx(andrej, babaYaga, txValue, txNonce, "")
}

signedTx, err := wallet.SignTxWithKeystoreAccount(
    tx,
    andrej,
    testKsAccountsPwd,
    wallet.GetKeystoreDirPath(dataDir))
if err != nil {
    t.Fatal(err)
}

_ = n.AddPendingTX(signedTx, andrejPeerNode)
}

```

He finishes the integration test by asserting the final state:

```

_ = n.Run(ctx, true, "")

expAndrejBalance :=
    andrejBalance - (txCount * txValue) + database.BlockReward
expBabaYagaBalance :=
    babaYagaBalance + (txCount * txValue)

if n.state.Balances[andrej] != expAndrejBalance {
    t.Errorf(
        "Andrej balance is incorrect. Expected: %d. Got: %d",
        expAndrejBalance,
        n.state.Balances[andrej])
}

```

```
}

if n.state.Balances[babaYaga] != expBabaYagaBalance {
    t.Errorf(
        "BabaYaga balance is incorrect. Expected: %d. Got: %d",
        expBabaYagaBalance,
        n.state.Balances[babaYaga])
    return
}

t.Logf("Andrej final balance: %d TBB", n.state.Balances[andrey])
t.Logf("BabaYaga final balance: %d TBB", n.state.Balances[babaYaga])
```

The result is clear. All 4 TXs got processed, and Andrej didn't have to pay any fee. The network is insecure to Denial-of-Service (DoS) attack.

Persisting new Block to disk:

```
{"hash": "0000000c81765b...5f151a",
"block": {"header": {"parent": "0000000", "number": 0, "nonce": 1466040767.\
...
node_integration_test.go:547: Andrej final balance: 300 TBB
node_integration_test.go:548: BabaYaga final balance: 800 TBB
--- PASS: TestNode_MiningSpamTransactions (592.87s)
PASS
```

Andrej decides to fix this.

Setting a “Gas Cost” and a “Gas Price”

To keep things simple, Andrej omits the dynamic pricing model and will hardcode that each TBB TX transfer will cost 21 000 Gas, priced at 50 TBB tokens.

```
const TxFee = uint(50)
```

Where do the Gas Fees Go?

Similarly, as Block Rewards, the Gas Fees go to the miners as an additional incentive to **process and include the next block transactions**.

Coding the Gas Fees

Andrej defines a new `TxFee` constant and modifies the `applyBlock` and `applyTx` functions. On `applyTx` he subtracts the `TxFee` from the sender's balance and adds it to the Miner's balance.

```
func applyTx(tx SignedTx, s *State) error {
    ...

    txCost := tx.Value + TxFee

    if txCost > s.Balances[tx.From] {
        return fmt.Errorf("wrong TX. Sender '%s' balance is %d TBB. Tx cost is...")
    }

    s.Balances[tx.From] -= txCost
    s.Balances[tx.To] += tx.Value

    s.Account2Nonce[tx.From] = tx.Nonce

    return nil
}
```

Give the fees from all block's TXs to the Miner:

```
func applyBlock(b Block, s *State) error {
    ...

    err = applyTXs(b.TXs, s)
    if err != nil {
        return err
    }

    s.Balances[b.Header.Miner] += BlockReward
    s.Balances[b.Header.Miner] += uint(len(b.TXs)) * TxFee
```

Andrej adds a new test assertion, validating the fees are correctly calculated:

```
    ...

expAndrejBalance :=
    andrejBalance - (txCount * txValue) + database.BlockReward - (txCo\
unt * database.TxFee)
expBabaYagaBalance :=
    babaYagaBalance + (txCount * txValue)
expectedMinerBalance :=
    minerBalance + database.BlockReward + (txCount * database.TxFee)

if n.state.Balances[andrej] != expAndrejBalance {
    t.Errorf(...
}

if n.state.Balances[babaYaga] != expBabaYagaBalance {
    t.Errorf(...
}
```

```
if n.state.Balances[miner] != expectedMinerBalance {  
    t.Errorf(...  
}  
  
t.Logf("Andrej final balance: %d TBB", n.state.Balances[andrey])  
t.Logf("BabaYaga final balance: %d TBB", n.state.Balances[babaYaga])  
t.Logf("Miner final balance: %d TBB", n.state.Balances[miner])
```

Time to run the new `TestNode_MiningSpamTransactions()` and prove the users can no longer spam the network with their transactions because each blockchain interaction cost them the fee that miners will happily collect.

➤— `go test ./node -test.v -test.run ^TestNode_-
MiningSpamTransactions`

```
==== RUN TestNode_MiningSpamTransactions
Added Pending TX {"from":"0x3eb92...","value":200,"nonce":1,...}
Added Pending TX {"from":"0x3eb92...","value":200,"nonce":2,...}
Added Pending TX {"from":"0x3eb92...","value":200,"nonce":3,...}
Added Pending TX {"from":"0x3eb92...","value":200,"nonce":4,...}
Listening on: 127.0.0.1:8085
Blockchain state:
  - height: 0
  - hash: 000000...
Mining 4 Pending TXs. Attempt: 1
Mining 4 Pending TXs. Attempt: 1000000
Mining 4 Pending TXs. Attempt: 2000000
...
Mining 4 Pending TXs. Attempt: 17000000
Persisting new Block to disk:
  { "hash":"37d73a8ab9bbfd7206d99b6...
node_integration_test.go:559: Andrej final balance: 0 TBB
node_integration_test.go:560: BabaYaga final balance: 800 TBB
node_integration_test.go:561: Miner final balance: 300 TBB
--- PASS: TestNode_MiningSpamTransactions (316.72s)
PASS
```



Summary

Closed software with centralized access to private data puts only a few people to the position of power. Users don't have a choice, and shareholders are in business to make money.

Blockchain developers aim to develop protocols where applications' entrepreneurs and users synergize in a transparent, auditable relation. Specifications of the blockchain system should be well defined from the beginning and only change if its users support it.

Blockchain is an immutable, transparent database. The token supply, initial user balances, and global blockchain settings you define in a Genesis file.

The database state changes are called Transactions (TX). Transactions are old fashion Events representing actions within the system.

The database content is hashed by a secure cryptographic hash function. The blockchain participants use the resulted hash to reference a specific database state.

Transactions are grouped into batches for performance reasons. A batch of transactions make a Block. Each block is encoded and hashed using a secure, cryptographic hash function.

Block contains Header and Payload. The Header stores various metadata such a time and a reference to the Parent Block (the previous immutable database state). The Payload carries the new database transactions.

The blockchain network consists of Nodes and Peers. All new peers connect to a default Bootstrap Peer(s) to discover and retrieve the current database state as well as the full transaction history. Peers continually communicate with each other using a programmed sync algorithm. Peers exchange information about new blocks and new network's peers.

Blockchain peers coordinate by following a pre-defined set of rules called “consensus.” The primary responsibility of a consensus algorithm is to decide who and how can produce the next valid block.

If consensus can't happen, the database history splits. A database split is called “hard fork.” Once hard fork occurs, peers with different blockchain history can't communicate and synchronize because everyone will have a different source of truth.

The most famous consensus algorithm implemented in Bitcoin is Proof of Work. PoW security consists of solving a cryptographic puzzle. Peers call this activity: “mining.” Miners repeatedly generate a random number, “nonce,” insert the nonce into a block header, and hash the block until the resulted hash matches the pre-defined consensus criteria.

Blockchain releases are complicated and require asynchronous communication between independent parties—the participants with the broadest influence, lobbying, and momentum win.

Blockchain security (authentication and authorization) are implemented with asymmetric cryptography. Asymmetric cryptography is possible thanks to mathematical elliptic curves.

Asymmetric cryptography works with a set of keys: Public key and

Private key.

The Public key is derived from the Private Key.

The Ethereum account address is the last 20 bytes of the secure hash of the Public key.

The Private Key is used as a decentralized password for generating Digital Signatures.

Digital Signatures are cryptographically generated proofs of action or identity. Signing is a technique to produce a verifiable proof: Signature.

A user (author) signs a message with his Private key and produces a Signature. Another user (reader) can verify the message authenticity by restoring the author's Public key from the Signature. If the restored Public key matches the author's Public key, the message can be trusted and couldn't be forged.

All blockchain transactions must be signed and verified.

Gas is an Ethereum's blockchain term for “fee”. Every TX operation costs Gas to prevent spam. The final TX cost is calculated as Gas * Gas Price. The TX fee goes to Miners as an incentive to include and process the transaction in the next block.



Study Code

Adds a test validating TXs are free and spam-able. [580594¹⁰³](#)

Adds Gas cost mechanism. [eed396¹⁰⁴](#)

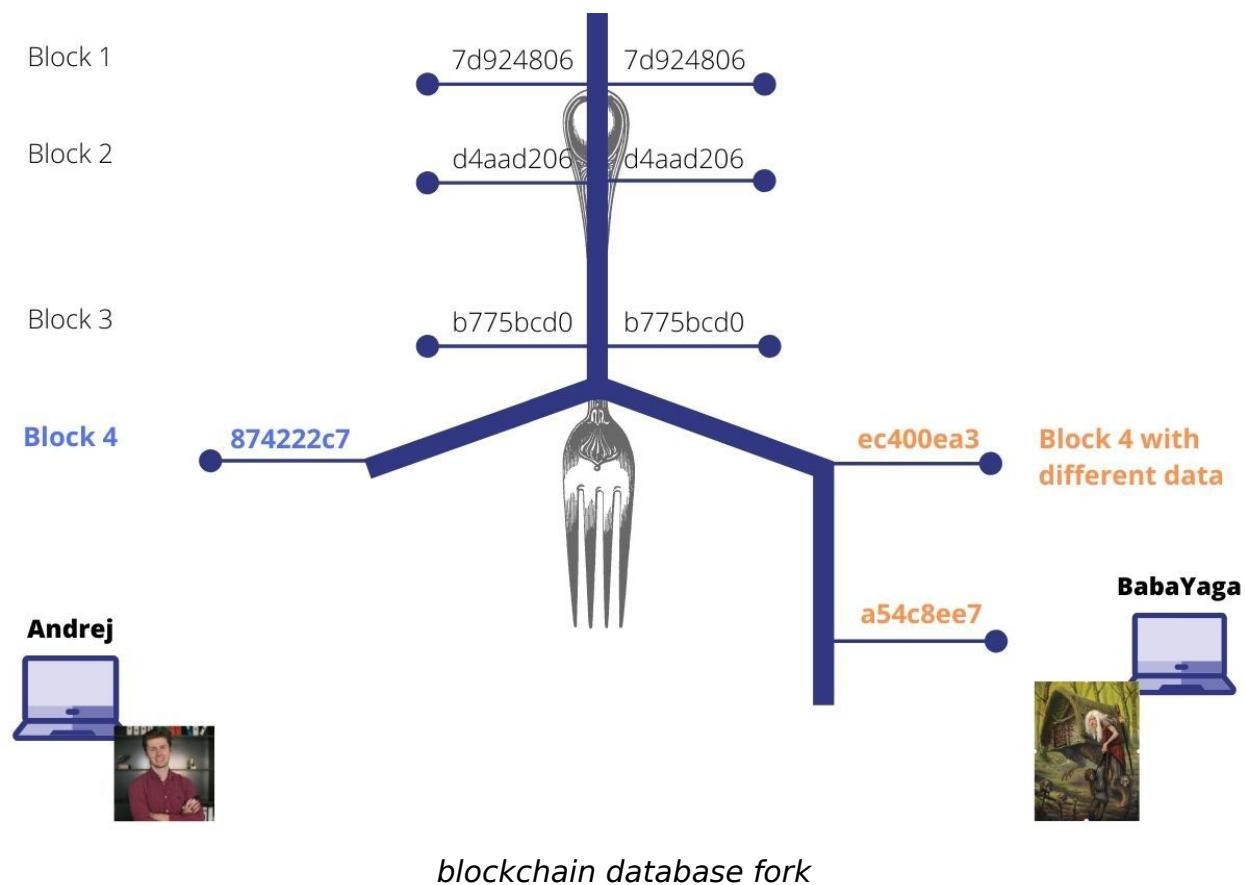
¹⁰³ <https://github.com/web3coach/the-blockchain-bar/commit/5805940b41a23ffa908e040af534d68cc3a7c6f4>

¹⁰⁴ <https://github.com/web3coach/the-blockchain-bar/commit/eed396960d2b331d74628ab7a87c807dd28e4b64>

14 | Bonus: Blockchain Forks – What, Why, How

>_ git checkout c15_blockchain_forks_what_why_how

Blockchain fork is a database split. The forked databases often have an identical history (past blocks), but will have different new blocks and transactions. It will become incompatible. When blockchain database forks, the network users decide what fork/version/blocks path they want to follow. The users must form a new social consensus.



The image illustrates a database split where BabaYaga forked from the original Andrej's blockchain for reasons you will explore soon. The account balances on the first three blocks are identical, but if an account made a new transaction on BabaYaga's fork, this transaction would only exist there. The forked chain often changes its token symbol and name.

Example: Bitcoin (BTC), Bitcoin Cash (BCH), Bitcoin SV (BSV), Ethereum (ETH), Ethereum Classic (ETC).

There is no “correct fork”; it all depends on the context. We can divide the reasons why a blockchain fork happens into several categories:

- Different Miner in Proof of Work
- P2P Latency, Different Miner, Transactions
- Mempool Economics
- Politics
- Consensus Change

Why a Fork Happens

Reason 1 – Different Miner in Proof of Work

Scenario: The TBB blockchain has 2 miners (Andrej, BabaYaga) and 10 users.

All 10 users broadcast 1 TX to both miner nodes. The mining process begins.

Andrej is trying to mine a block:

```
block := database.NewBlock(  
    pb.parent,  
    pb.number,  
    nonce,  
    pb.time,  
    andrejWalletAddress,  
    pb.txs, // TX 1...TX 10  
)
```

While BabaYaga wants to mine a block with the same 10 TXs but with the `miner` attribute set to her wallet address to win the block reward for herself.

```
block := database.NewBlock(  
    pb.parent,  
    pb.number,  
    nonce,  
    pb.time,  
    babaYagaWalletAddress,  
    pb.txs, // TX 1...TX 10  
)
```

After 10 minutes, they both mine a block, but even though the block number is the same, the block hashes differ as time, nonce, miner attributes coming to the hash function as inputs result in a unique block hash output.

Andrej and BabaYaga forked. Both miners insist their database version is the “correct” one. Funny enough, this is expected behavior, and these small forks happen several times a day. Proof of Work has a build-in consensus behavior for resolving these forks by making the miners continue mining new blocks on top of their temporary fork. The miner with the most hashing power, or luck, that builds **the longest chain over time** gets his blocks “merged” (will be considered the new main), and the rest of the miners throw away their forked work and reset their chain to the winner’s fork. All the miners then start mining new blocks on top of the new longest chain again, and the process repeats.

Keep in mind, the longest chain doesn’t necessarily refer to the chain with most blocks, but the chain that was the most difficult to build - required more hashing power and energy. In production blockchains, the **difficulty to produce a block changes** dynamically. Bitcoin aims to achieve an average of 1 block per 10 minutes

and Ethereum 1 block per 15 seconds. The reason nodes adopt the most **difficult chain** to produce is to secure historical blocks - as maliciously modifying the history and re-hashing, replacing all the blocks would be beyond expensive.

Because there is only one winner and all the shorter forks go to the trash, the Proof of Work is inefficient, wasteful. Advantage? It's secure and decentralized, optimized for liveliness. In other protocols, such as the XRP ledger based on Proof of Authority, the consensus halts for safety; the network stops until the majority of validators agree on a new state.

Halting for safety has a significant advantage: once validators agree on a block in 4 seconds, the decision is FINAL! NOT the case in Bitcoin or Ethereum.

As you learned in Chapter 11, you can't have a safe, live, and fault-tolerant asynchronous system. You must choose 2 out of 3 properties. Source: “[Impossibility of Distributed Consensus with One Faulty Process](#)”¹⁰⁵.

¹⁰⁵ <https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf>

How does block finality affect the UX?

When you transfer your BTC or ETH to an exchange wallet, the exchange makes you wait for 6 Bitcoin blocks and 15 Ethereum blocks; a 30 minutes safety window.

The safety window prevents the exchange from acting on an invalid fork that may end up in the trash, rewriting the history, and causing nasty bugs such as **double-spending**; I will explain soon. The lack of the block's finality makes the developer's life difficult. Imagine every time you insert a new row into a MySQL database, and you have to wait for 30 minutes to ensure the server doesn't roll back your transaction.

When you deposit your XRP to exchange, in contrast, your XRP tokens are available immediately because every block is final, and there are no possible "rollbacks."

The different strategies have different pros and cons, as you will see in the following Reason 2.

Reason 2 – P2P Latency, Different Miner, Transactions

Scenario: 2 miners, 10 users situated around the globe.

All 10 users broadcast 1 TX, but because of the network latencies or slow node discovery process, one miner receives 4 TXs and the other 6 TXs.

Andrej is mining TX 1 ... TX 4:

```
block := database.NewBlock(  
    pb.parent,  
    pb.number,  
    nonce,  
    pb.time,  
    andrejWalletAddress,  
    pb.txs, // TX 1...TX 4  
)
```

While BabaYaga TX 5 ... TX 10:

```
block := database.NewBlock(  
    pb.parent,  
    pb.number,  
    nonce,  
    pb.time,  
    babaYagaWalletAddress,  
    pb.txs, // TX 5...TX 10  
)
```

Same block number, but now even the transactions payload differs, chains forked again. This brings us to a tricky topic, unique to

cryptocurrency systems where there isn't a single source of truth (temporarily, until nodes resolve the fork): **double-spending**.

The danger of double-spending

The fork resolution strategy significantly impacts the users and other network participants. You must be aware of this “detail”; otherwise, you may lose money. **How could you lose money?**

Your name is Caesar, and you want to buy a Lamborghini for 50 TBB from Daenerys. You make a TX that gets included in Block 4, mined by Andrej:

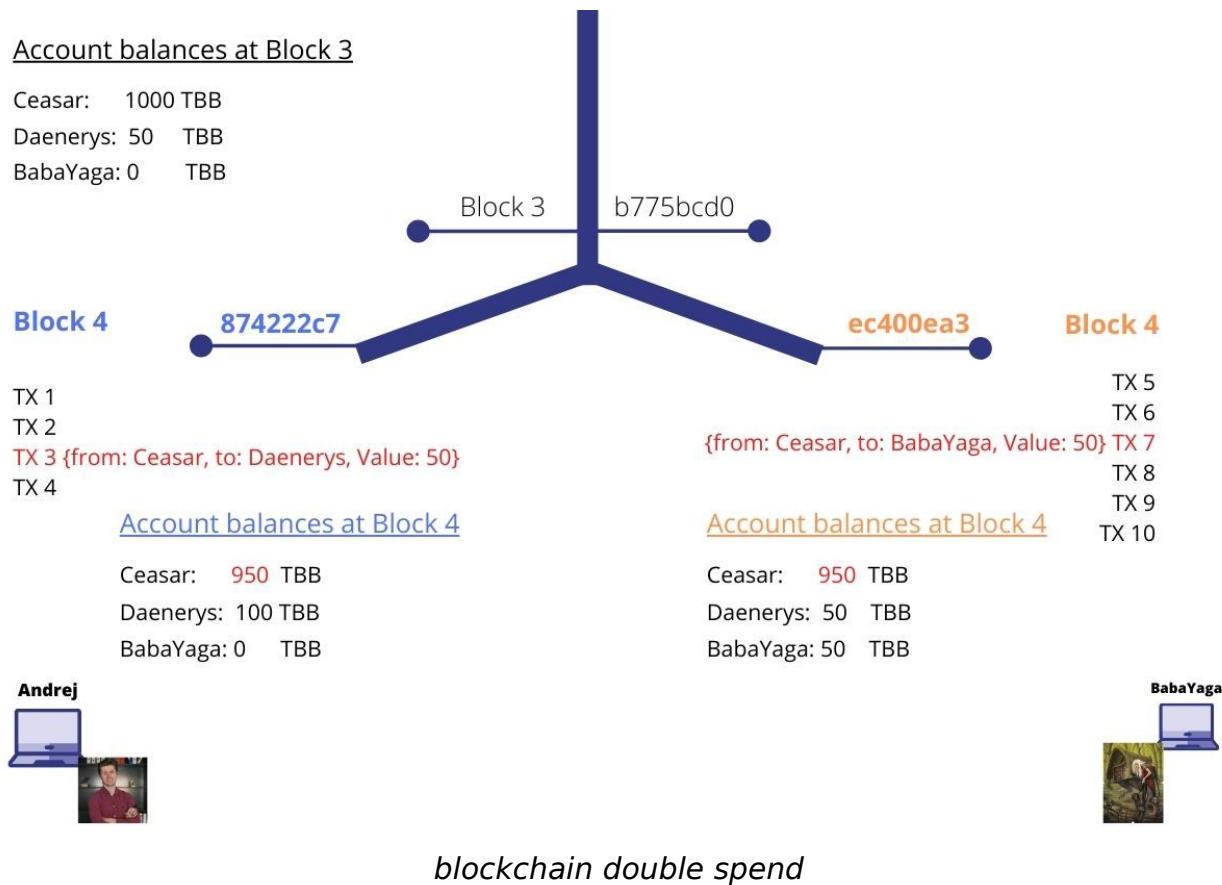
```
{from: Ceasar, to: Daenerys, Value: 50}
```

Daenerys wallet displays a received 50 TBB tokens TX, and she gives you the car keys. She made a giant mistake.

You were sneaky and did another 50 TBB tokens TX paying for a Ferrari from BabaYaga. You broadcasted this TX directly to the BabaYaga node, and she included it in Block 4. The network didn't have time to propagate to other nodes because of latency:

```
{from: Ceasar, to: BabaYaga, Value: 50}
```

Because neither of the two miners is aware of your double spend, once they resolve the fork, only Daenerys or BabaYaga will end up with 50 TBB tokens, not both of them, and you are already on your way to Mallorca with two car keys in your pocket. Hence, the X-blocks long safety window when depositing funds to an exchange.



Reason 3 – Mempool Economics

Some protocols let miners decide what transactions to include.

A miner can pick only the most expensive, gas-rewarding transactions, including the cheap transactions only if there is still some block space left. A miner wants to find a valid block hash to collect the reward without waiting for all the users' transactions. Usually, users "bid" for the space in the next block.

Reason 4 – Consensus Change

The code your node is running decides the consensus, the compatibility with other node's software.

Ethereum released a new client software, including a breaking change documented under [EIP-1559¹⁰⁶](#) (Ethereum Improvement Proposal) after two years of hard work last month on July 2021.

"EIP-1559 is a transaction pricing mechanism that includes fixed-per-block network fee that is burned and dynamically expands/contracts block sizes to deal with transient congestion."

The core developers added a new block attribute: `BaseFee *big.Int json: "baseFeePerGas"` and changed a bunch of business logic, as you can read in the official linked specification. Modifying protocol

¹⁰⁶<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md>

block specification affects the consensus. All network nodes had to coordinate a software update. A topic easier said than done.

Different departments within the same company have trouble communicating and coordinating.

Imagine coordinating a blockchain update between tens/hundreds/thousands of independent parties - each with distinct goals, motivations, and opinions!

[go-ethereum/blob/master/core/types/block.go#L87](https://github.com/ethereum/go-ethereum/blob/master/core/types/block.go#L87)¹⁰⁷

How did they do it?

¹⁰⁷ <https://github.com/ethereum/go-ethereum/blob/0a68558e7e025afebf67b81bf48ecb8b0fa7c06d/core/types/block.go#L87>

How to Plan a Consensus Fork

The consensus changes are included in the node's software, and the code path is "enabled", executed after an in-advance scheduled block number.

For example, the EIP-1559 was scheduled for block **12 965 000**, expected to be mined on Aug-05-2021 13:00 PM +UTC. The Ethereum developers added this block milestone into a [config.go¹⁰⁸](#) to decide from what exact block should the new consensus rules apply.

```
MainnetChainConfig = &ChainConfig{
    ChainID:                  big.NewInt(1),
    HomesteadBlock:            big.NewInt(1_150_000),
    DAOForkBlock:              big.NewInt(1_920_000),
    DAOForkSupport:            true,
    EIP150Block:               big.NewInt(2_463_000),
    EIP150Hash:                common.HexToHash("0x208679...6514f0"),
    EIP155Block:               big.NewInt(2_675_000),
    EIP158Block:               big.NewInt(2_675_000),
    ByzantiumBlock:            big.NewInt(4_370_000),
    ConstantinopleBlock:       big.NewInt(7_280_000),
    PetersburgBlock:           big.NewInt(7_280_000),
    IstanbulBlock:              big.NewInt(9_069_000),
    MuirGlacierBlock:           big.NewInt(9_200_000),
    BerlinBlock:                big.NewInt(12_244_000),
    LondonBlock:                 big.NewInt(12_965_000),
    Ethash:                     new(EthashConfig),
}
```

¹⁰⁸ <https://github.com/holiman/go-ethereum/blob/8601f139481587d55fe8830cc72ec3809ccaf102/params/config.go#L72>

```
// IsLondon returns whether num is either equal to the London fork blo\
ck or greater.
func (c *ChainConfig) IsLondon(num *big.Int) bool {
    return isForked(c.LondonBlock, num)
}
```

Implementing a Consensus Fork

What Ethereum does

The official [go-ethereum/core/types/transaction.go¹⁰⁹](#) has several fee/gas related methods:

```
// Cost returns gas * gasPrice + value.
func (tx *Transaction) Cost() *big.Int {
    total := new(big.Int).Mul(tx.GasPrice(), new(big.Int).SetUint64(tx\
.Gas()))
    total.Add(total, tx.Value())
    return total
}

// Gas returns the gas limit of the transaction.
func (tx *Transaction) Gas() uint64 {
    return tx.inner.gas()
}

// GasPrice returns the gas price of the transaction.
func (tx *Transaction) GasPrice() *big.Int {
    return new(big.Int).Set(tx.inner.gasPrice())
}

// GasTipCap returns the gasTipCap per gas of the transaction.
func (tx *Transaction) GasTipCap() *big.Int {
    return new(big.Int).Set(tx.inner.gasTipCap())
}
```

¹⁰⁹ <https://github.com/ethereum/go-ethereum/blob/57feabea663496109e59df669238398239438fb1/core/types/transaction.go#L296>

```
// GasFeeCap returns the fee cap per gas of the transaction.  
func (tx *Transaction) GasFeeCap() *big.Int {  
    return new(big.Int).Set(tx.inner.gasFeeCap())  
}
```

The final Ethereum tx.Cost() is calculated as **gas * gasPrice + value**.

Andrej decides he doesn't need the latest EIP-1559 miner tipping feature for now, but he definitely wants to copy the cost calculation to learn a programming technique from the PROs and grow as a developer:

```
// Cost returns gas * gasPrice + value.  
func (tx *Transaction) Cost() *big.Int {  
    total := new(big.Int).Mul(tx.GasPrice(), new(big.Int).SetUint64(tx.Gas()))  
    total.Add(total, tx.Value())  
    return total  
}
```

Example

The legacy Ethereum TX transfer from account A to account B costs 21 000 gas. The Price of Gas depends on the network usage. If the Price is 2 GWei per 1 Gas, the Gas Price would be $2 * 21\ 000 = 42\ 000$ Gwei or 0.000042 ETH.

The Ether currency structure with **Wei** being the smallest unit:

```
const (
    Wei     = 1
    GWei   = 1e9
    Ether  = 1e18
)
```

New TBB Specification

The TBB protocol will define how much **Gas** each action will require, and the user and network economics will decide the **Gas Price**.

Action	Gas Required
Transfer	21

Each Transaction will require two new attributes:

- **Gas**: user must set this value to 21
- **GasPrice**: user decides how much he wants to spend, miner chooses whenever it's enough to include this transaction into a block and cover the mining costs

For simplicity, the default **GasPrice value will be 1 TBB** token (1 TBB token is the smallest protocol unit).

Andrej formulates this new specification as [TheBlockchainBar Improvement Proposal \(TIP\)](#)¹¹⁰ in the Github repository. TIPs describe standards for the TheBlockchainBar platform, including core protocol specifications, client APIs, and contract standards.

The first one will be **TIP-1: Dynamic Transaction Cost like in Ethereum**. Miners can create a new TheBlockchainBar Improvement Proposal and define what criteria the **GasPrice** will be sufficient to pay for a transaction to get included in a block.

¹¹⁰ https://github.com/web3coach/the-blockchain-bar/tree/c15_blockchain_forks_what_why_how/TIPs

PS: This eBook serves as a learning playfield. A testing, experimental blockchain where you can practice peer-to-peer development and programming crypto applications. If you want to improve The Blockchain Bar with a new feature available in Bitcoin, Ethereum, XRP, or any other blockchain protocol, create a new TIP-X proposal and describe the feature specifications. It will be a valuable exercise for you to grow as a developer.

Let's start coding.

Andrej adds the two new attributes to the TX struct. As this change affects a lot of places in the codebase, so he adjusts all the involved parts and unit/integration tests.

```
type Tx struct {
    From      common.Address `json:"from"`
    To        common.Address `json:"to"`

    Gas       uint          `json:"gas"`
    GasPrice uint          `json:"gasPrice"`

    Value     uint          `json:"value"`
    Nonce    uint          `json:"nonce"`
    Data      string        `json:"data"`
    Time     uint64        `json:"time"`
}
```

HTTP API for sending transactions:

```
type TxAddReq struct {
    From      string `json:"from"`
    FromPwd   string `json:"from_pwd"`
    To        string `json:"to"`

    Gas       uint   `json:"gas"`
    GasPrice uint   `json:"gasPrice"`

    Value     uint   `json:"value"`
    Data      string `json:"data"`
}
```

New gas constants:

```
const TxGas = 21
const TxGasPriceDefault = 1
```

A helper function for convenience to avoid passing the gas constants everywhere:

```
func NewBaseTx(from, to common.Address, value, nonce uint, data string\
) Tx {
    return NewTx(from, to, TxGas, TxGasPriceDefault, value, nonce, data)
}
```

There is a problem with this approach though. Processing past blocks must be backward-compatible according to old consensus rules for hashes to match on the database/file-system level. Andrej adds a new configurable fork number into the Genesis file to coordinate the business logic execution.

The fork TIP-1 is scheduled on block 35!

```
var genesisJson = `{
  "genesis_time": "2020-06-01T00:00:00.000000000Z",
  "chain_id": "the-blockchain-bar-ledger",
  "symbol": "TBB",
  "balances": {
    "0x09eE50f2F37FcBA1845dE6FE5C762E83E65E755c": 1000000
  },
  "fork_tip_1": 35
}`

type Genesis struct {
  Balances map[common.Address]uint `json:"balances"`
  Symbol   string                  `json:"symbol"`

  ForkTIP1 uint64 `json:"fork_tip_1"`
}
```

Andrej starts with defining a helper state function for controlling the code flow depending on the block number:

```
func (s *State) IsTIP1Fork() bool {
    return s.NextBlockNumber() >= s.forkTIP1
}
```

And changes the tx.Cost() calculation but preserves the backwards compatibility:

```
// old:
func (t Tx) Cost() uint {
    return t.Value + TxFee
}

// new:
func (t Tx) Cost(isTip1Fork bool) uint {
    if isTip1Fork {
        return t.Value + t.GasCost()
    }

    return t.Value + TxFee
}

func (t Tx) GasCost() uint {
    return t.Gas * t.GasPrice
}
```

The miner will then collect all the gas costs as an additional reward for including the transaction in a block:

```
func (b Block) GasReward() uint {
    reward := uint(0)

    for _, tx := range b.TXs {
        reward += tx.GasCost()
    }

    return reward
}

// on applying a new block in state.go:
s.Balances[b.Header.Miner] += BlockReward
if s.IsTIP1Fork() {
    s.Balances[b.Header.Miner] += b.GasReward()
} else {
    s.Balances[b.Header.Miner] += uint(len(b.TXs)) * TxFee
}
```

State must equivalently process the historical as well as new transactions according to the current block number and what fork is active at a given time in history:

```
func ApplyTx(tx SignedTx, s *State) error {
    err := ValidateTx(tx, s)
    if err != nil {
        return err
    }

    s.Balances[tx.From] -= tx.Cost(s.IsTIP1Fork())
    s.Balances[tx.To] += tx.Value

    s.Account2Nonce[tx.From] = tx.Nonce

    return nil
}

func ValidateTx(tx SignedTx, s *State) error {
    ok, err := tx.isAuthenticated()
    if err != nil {
        return err
    }

    if !ok {
        return fmt.Errorf(
            "wrong TX. Sender '%s' is forged", tx.From.String()
        )
    }
}

expectedNonce := s.GetNextAccountNonce(tx.From)
if tx.Nonce != expectedNonce {
    return fmt.Errorf(
        "wrong TX. Sender '%s' next nonce must be '%d', not '%d'",
        tx.From.String(),
    )
}
```

```
    expectedNonce,
    tx.Nonce
)
}

if s.IsTIP1Fork() {
    // For now we only have one type, transfer TXs,
    // so all TXs must pay 21 gas like on Ethereum (21 000)
    if tx.Gas != TxGas {
        return fmt.Errorf(
            "insufficient TX gas %v. required: %v",
            tx.Gas,
            TxGas
        )
    }
}

if tx.GasPrice < TxGasPriceDefault {
    return fmt.Errorf(
        "insufficient TX gasPrice %v. required at least: %v",
        tx.GasPrice,
        TxGasPriceDefault
    )
}

} else {
    // Prior to TIP1, a signed TX must NOT populate
    // the Gas fields to prevent consensus from crashing.
    //
    // It's not enough to add this validation to http_routes.go
    // because a TX could come from another node
    // that could modify its software and broadcast
    // such a TX, it must be validated here too.
    if tx.Gas != 0 || tx.GasPrice != 0 {
        return fmt.Errorf(
            "invalid TX. `Gas` and `GasPrice` can't be populated before TIP\
1 fork is active"
```

```

        )
    }

if tx.Cost(s.IsTIP1Fork()) > s.Balances[tx.From] {
    return fmt.Errorf(
        "wrong TX. Sender '%s' balance is %d TBB. Tx cost is %d TBB",
        tx.From.String(),
        s.Balances[tx.From],
        tx.Cost(s.IsTIP1Fork())
    )
}

return nil
}

```

If a node broadcasts a TX constructed with TIP1 format before block 35, they will get an error:

```
{"error":"invalid TX. `Gas` and `GasPrice` can't be populated before T\IP1 fork is active"}
```

Any wallet, client, or node that don't update their software will be excluded from consensus and get the following error when submitting transactions:

```
{"error":"insufficient TX gas 0. required: 21"}
```

Finally, Andrej adjusts all the integration tests affecting the state, balances, TX costs, and rewards. All the tests will now run for all fork versions. The most elegant way to do this is by implementing an array of test cases.

```
func TestNode_MiningSpamTransactions(t *testing.T) {
    tc := []struct {
        name      string
        ForkTIP1 uint64
    }{
        {"Legacy", 35}, // Prior ForkTIP1 was activated on number 35
        {"ForkTIP1", 0}, // To test new blocks when the ForkTIP1 is active
    }

    for _, tc := range tc {
        t.Run(tc.name, func(t *testing.T) {
            dataDir, andrej, babaYaga, err := setupTestNodeDir(andrejBalance\
, tc.ForkTIP1)

            // run the code as from previous chapters

            // in nutshell: sender occurs tx.Cost(),
            // receiver gains tx.Value() and miner collects tx.GasCost()

            if n.state.IsTIP1Fork() {
                expectedAndrejBalance = andrejBalance
                expectedMinerBalance = minerBalance + database.BlockReward

                for _, tx := range spamTXs {
                    expectedAndrejBalance -= tx.Cost(true)
                    expectedMinerBalance += tx.GasCost()
                }

                expectedBabaYagaBalance = babaYagaBalance + (txCount * txValue)
            } else {
                expectedAndrejBalance = andrejBalance - (txCount * txValue) - \
(txCount * database.TxFee)
                expectedBabaYagaBalance = babaYagaBalance + (txCount * txValue)
                expectedMinerBalance = minerBalance + database.BlockReward + (\
txCount * database.TxFee)
            }
        })
    }
}
```

```
    })  
}
```

Done? Seems like. Except there is a consequential backward-incompatible change on the datastore in `blocks.db`. Because the transactions now have two new attributes: `Gas` and `GasPrice`, running the TBB node with existing legacy blocks already saved on disk will crash as hashing these two attributes even with empty values will change the block hash:

```
> tbb run --datadir=$HOME/.tbb --ip=127.0.0.1 --port=8081 --disable-ssl  
  
Launching TBB node and its HTTP API...  
Listening on: 127.0.0.1:8081  
  
invalid block hash bd0347...969118
```

In a standard Web2 software, Andrej would write a database migration, but in this case, he can't; other apps, explorers, links may rely on these existing hashes.

Hashing a TX goes as: TX → Encoded JSON → Hash.

If your Struct implements the `MarshalJSON` interface, it will automatically be called on JSON serialization by the Go compiler; Andrej takes advantage of this and excludes the new attributes for old blocks.

```
// MarshalJSON is the main source of truth for encoding
// a TX for hash calculation (backward-compatible for TIPs).
//
// The logic is bit ugly and hacky but prevents infinite marshaling
// loops of embedded objects and allows the structure
// to change with new TIPs.

func (t SignedTx) MarshalJSON() ([]byte, error) {
    // Prior TIP1
    if t.Gas == 0 {
        type legacyTx struct {
            From    common.Address `json:"from"`
            To      common.Address `json:"to"`
            Value   uint           `json:"value"`
            Nonce   uint           `json:"nonce"`
            Data    string         `json:"data"`
            Time    uint64         `json:"time"`
            Sig     []byte         `json:"signature"`
        }
        return json.Marshal(legacyTx{
            From:  t.From,
            To:    t.To,
            Value: t.Value,
            Nonce: t.Nonce,
            Data:  t.Data,
            Time:  t.Time,
            Sig:   t.Sig,
        })
    }
}

type tip1Tx struct {
    From    common.Address `json:"from"`
    To      common.Address `json:"to"`
    Gas     uint           `json:"gas"`
    GasPrice uint          `json:"gasPrice"`
    Value   uint           `json:"value"`
    Nonce   uint           `json:"nonce"`
}
```

```
    Data      string          `json:"data"`
    Time     uint64          `json:"time"`
    Sig      []byte          `json:"signature"`

}

return json.Marshal(tip1Tx{
    From:      t.From,
    To:        t.To,
    Gas:       t.Gas,
    GasPrice: t.GasPrice,
    Value:    t.Value,
    Nonce:    t.Nonce,
    Data:     t.Data,
    Time:     t.Time,
    Sig:      t.Sig,
})
}
```

Deploying the Fork

Andrej restarts the node with the latest TBB Version: 1.9.0-alpha 2aeeef4 TX Gas, and observes the fork happening:

```
Launching TBB node and its HTTP API...
Listening on: node.tbb.web3.coach:443
```

```
Blockchain state:
- height: 34
- hash: 00000055e944d6cca5c8591816151842629e1f16382c15a019f3b2f21ce8\
ecd9
```

```
Added Pending TX {
  "from": "0x09e...",
  "to": "0x09e",
  "value": 1,
  "nonce": 15,
  "data": "",
  "time": 1631087644,
  "signature": "hxhDI...D0wE="
}
```

```
Mining 1 Pending TXs. Attempt: 1
Mining 1 Pending TXs. Attempt: 1000000
Mining 1 Pending TXs. Attempt: 2000000
```

```
// Fork with new TX format at block 35
```

```
Added Pending TX  {
  "from": "0x09e...",
  "to": "0x09e",
  "value": 1,
```

```
"nonce":16,  
  
"gas":21,  
"gasPrice":1,  
  
"data":"",
"time":1631097644,  
"signature":"cH..."  
}
```

The Blockchain Bar network successfully forked at block number 35!

Synchronize Your Node



Compile a new binary from the latest c15_blockchain_forks_-what_why_how branch and update your `genesis.json` file.

Modify the `.tbb/database/genesis.json` file and define the same block number when the fork should activate:

```
{  
  "genesis_time": "2020-06-01T00:00:00.000000000Z",  
  "chain_id": "the-blockchain-bar-ledger",  
  "symbol": "TBB",  
  "balances": {  
    "0x09eE50f2F37FcBA1845dE6FE5C762E83E65E755c": 1000000  
  },  
  "fork_tip_1": 35  
}
```

>— `tbb run -datadir=$HOME/.tbb -ip=127.0.0.1 -port=8081 - disable-ssl`



Study Code

Drafts C15 with fork TIP-1 for dynamic gas cost. abac37¹¹¹

Adds ForkTIP1 with new GasCost calculation. 394b7c¹¹²

Adds backwards compatibility for serializing previous blocks.
aca82e¹¹³

Adds Gas and GasPrice validation to consensus. 2aeef4¹¹⁴

¹¹¹ <https://github.com/web3coach/the-blockchain-bar/commit/abac37c0318937277b0552b698b68956bae64352>

¹¹² <https://github.com/web3coach/the-blockchain-bar/commit/394b7ccc817b0fcf60a6469489f3a700a36c896d>

¹¹³ <https://github.com/web3coach/the-blockchain-bar/commit/aca82e3ad93858116d676e81bd9d67506179fbaa>

¹¹⁴ <https://github.com/web3coach/the-blockchain-bar/commit/2aeef469925134e4f16d95e5db0e7420de48a876>

15 | Bonus: IPFS – the Decentralized Storage of Web3



<https://www.freecodecamp.org/news/technical-guide-to-ipfs-decentralized-storage-of-web3/>¹¹⁵

Blockchain is fantastic for managing state, automating processes via Smart Contracts, and exchanging economic value.

But where do you store your application's content? Images? Videos? The application's website front-end composed of all the HTML, CSS, and JS files? Are your application and your users' content loaded from a centralized AWS server?

Storing the content on the blockchain would be expensive and inefficient. Your blockchain application needs decentralized storage!

If you want to continue learning Web3, I wrote [a deep technical guide to IPFS](#)¹¹⁶ for you where I will introduce you to the InterPlanetary File System, or IPFS.

¹¹⁵ <https://www.freecodecamp.org/news/technical-guide-to-ipfs-decentralized-storage-of-web3/>

¹¹⁶ <https://www.freecodecamp.org/news/technical-guide-to-ipfs-decentralized-storage-of-web3/>

You will learn:

- How to store and retrieve content from a decentralized storage
- How to run your IPFS node
- All about the low-level internals of the IPFS protocol
- And we'll read a Wikipedia website stored on IPFS

The InterPlanetary File System, or IPFS for short, is a peer-to-peer hypermedia protocol designed to make the web faster, safer, and more open.

- What is the IPFS?
- How to setup an IPFS node
- How to store and retrieve IPFS content using the CLI and HTTP
- What is CID - the IPFS content-based identifier
- How to reverse engineer the IPFS datastore
- How to connect an IPFS node to a decentralized network
- How to exchange data using the peer-to-peer Bitswap protocol
- How to persist content from the peer-to-peer network

What is unique about IPFS?

The IPFS is decentralized because it loads the content from thousands of peers instead of one centralized server. Every piece of data is cryptographically hashed, resulting in a safe, unique content identifier: CID.

Store your website in IPFS to avoid censorship and a single point of failure. Your personal IPFS node goes offline? Don't worry, the website will still load from other nodes across the globe serving it.

For example, suppose your government bans Wikipedia. In that case, you can still access a decentralized version of Wikipedia indexed on April 17th by loading it from the IPFS peer-to-peer network persisted under CID:

“QmT5NvUtoM5nWFfrQdVrFtvGfKFmG7AHE8P34isapyhCxX”

Second, the integrity of IPFS content can be cryptographically verified.

And finally, the IPFS content is de-duplicated. If you tried storing two identical 1MB files in the same IPFS node, they would be stored only once, eliminating the duplication, because their hash would produce an identical CID.

Continue reading this chapter in the form of a blog post:

<https://www.freecodecamp.org/news/technical-guide-to-ipfs-decentralized-storage-of-web3/>¹¹⁷

¹¹⁷ <https://www.freecodecamp.org/news/technical-guide-to-ipfs-decentralized-storage-of-web3/>

16 | Bonus: Merkle Trees in Go



<https://web3.coach/golang-blockchain-filesystem-and-merkle-trees>¹¹⁸

In a peer-to-peer blockchain network, you can receive data from anyone in the world. This is great because there is no central entity, but also a nightmare. After all, you could receive tampered data from a malicious anonymous peer.

If you signed and executed a TX from your mobile phone, how can you verify that your TX was successful and included in the next block?

An AWS server with large bandwidth and enough CPU can handle to perform a full validation, but what about a laptop? Phone? Remember, in the blockchain network; every user should run their own software (node) to be able to verify the blockchain integrity independently. That's a lot of data blockchain peers must exchange with each other over the network. A problematic performance challenge to tackle!

Solution?

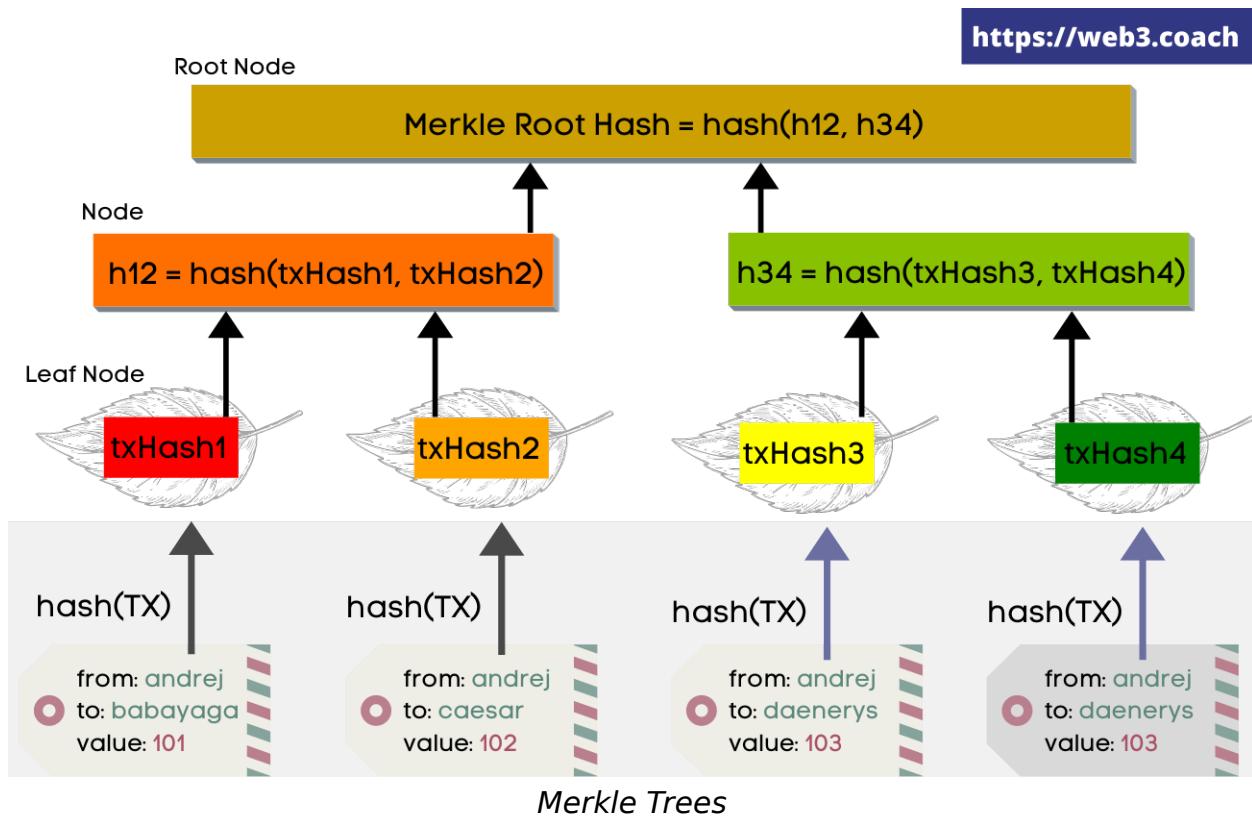
¹¹⁸ <https://web3.coach/golang-blockchain-filesystem-and-merkle-trees>

Merkle Tree is a data structure combining the efficiency of a tree, with the advantages of crypto hashing. Invented by a computer scientist Ralph Merkl in 1979.

The main value proposition is to separate data fingerprints (hashes) and data itself while being efficient in performing **Merkle Proofs**.

Leaf nodes are grouped in a couple and recursively hashed until there is only one final Root Hash.

Fun fact, if the number of leaves is odd, you must duplicate the odd leaf node.



Continue reading this chapter in the form of a blog post:

<https://web3.coach/golang-blockchain-filesystem-and-merkle-trees>¹¹⁹

¹¹⁹ <https://web3.coach/golang-blockchain-filesystem-and-merkle-trees>

17 | Bonus: Blockchain Storage (complexity level 99)



<https://arxiv.org/pdf/2108.05513.pdf>¹²⁰.

The Blockchain Bar uses a simplified append-only database containing JSON encoded blocks for educational purposes.

However, production-ready blockchains like Ethereum need a sophisticated, **storage systems supporting high write-throughput** and parallel, **efficient read access to calculate millions of possible state transitions for the latest block**. Archive nodes must even optimize for serving requests from analytical DApps and compute the state at ANY past block. How?

¹²⁰ <https://arxiv.org/pdf/2108.05513.pdf>

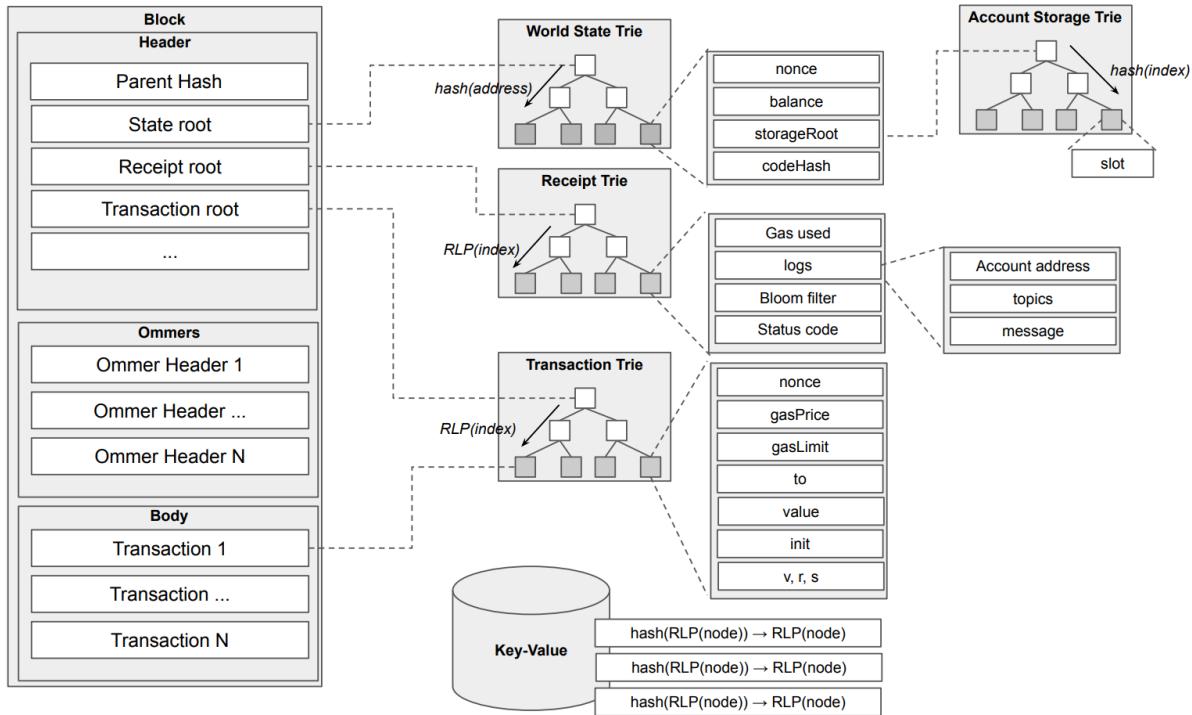


Fig. 6. Ethereum Data Structures

Implementing this storage layout is out of scope of this eBook, but I find it essential to share this knowledge with you already by at least directing you to the work of KAMIL JEZEK from University of Sydney who did an outstanding job explaining the dynamics in their paper: Ethereum Data Structures.

[https://arxiv.org/pdf/2108.05513.pdf¹²¹](https://arxiv.org/pdf/2108.05513.pdf)

PS: If time permits, I will try to integrate the Ethereum's Merkle Patricia Trie into The Blockchain Bar during my next holidays.

¹²¹ <https://arxiv.org/pdf/2108.05513.pdf>

Official TBB Training Ledger

>_ git checkout c16_blockchain_explorer

Congratulations!

You have made it till the end!

The blockchain ecosystem is an endless rabbit hole of new programming concepts, and this book could easily be 1000 pages long. We still have to cover some important concepts such as Merkle Trees for storing and verifying the blockchain database in the file system efficiently, but that's advanced thoughts for the next chapters that will be released in the upcoming months.

Now, it's time for you to abandon your local blockchain network, and **join Andrej, BabaYaga, Caesar, Daenerys and other TBB students by connecting to the official TBB training ledger.**

We will use this test network to continue improving our blockchain, securing it, and learning by fixing the existing bugs. A lot of new practical exercises are on the way!

I have created a new Genesis file and a new wallet account holding 1M TBB tokens in branch `c16_blockchain_explorer`. Time to compile

the latest branch, create a new node, and synchronize with the rest of the training network.

```
var genesisJson = `{
  "genesis_time": "2020-06-01T00:00:00.000000000Z",
  "chain_id": "the-blockchain-bar-ledger",
  "symbol": "TBB",
  "balances": {
    "0x09eE50f2F37FcBA1845dE6FE5C762E83E65E755c": 1000000
  }
}`
```

Create a New Account

```
>_ tbb wallet new-account --datadir=$HOME/.tbb
```

Please enter a password to encrypt the new wallet:

Repeat password:

```
New account created: 0xYOUR_ADDRESS
Saved in: $HOME/.tbb/keystore
```

Connect Your Node to the TBB Training Ledger

```
>_ tbb run --datadir=$HOME/.tbb --ip=127.0.0.1 --port=8080
--miner=0xYOUR_ADDRESS --disable-ssl
```

Your node will connect to my bootstrap node and synchronize all the blocks, pending transactions and known peers.

```
Launching TBB node and its HTTP API...
Listening on: 127.0.0.1:8080
Blockchain state:
  - height: 0
  - hash: 000000...000000

Searching for new Peers, Blocks: 'node.tbb.web3.coach'
Found **37 new blocks** from Peer node.tbb.web3.coach:443
Importing blocks from Peer node.tbb.web3.coach...
```

Query The Blockchain Bar Customer Balances

>_ curl -X GET http://localhost:8080/balances/list | jq

```
{
  "block_hash": "000000a0718c1f556177e470dc42a486f04226fa1a39c12344658\5457597b5c1",
  "balances": {
    "0x09ee50f2f37fcba1845de6fe5c762e83e65e755c": 987945,
    "0x0fc524c45b51e3215701ef7c12e58c781b0642ac": 1000,
    "0x19a08bd5df98c09f07913b2f0d71c0db9b0e561b": 1000,
    "0x22ba1f80452e6220c7cc6ea2d1e3eeddac5f694a": 5,
    "0x3a8072e42b2402aaa9cf58c2e0cdde617cc5e76d": 150,
    "0x9bbd16fd48ec35d5e3bde307f451ead7e7593461": 1000,
    "0x9f23369f02924f94765711bb09ebe1937123e37d": 1000,
    "0xdd1fb71d7f78a810ae10829220fbb7bd2f1818b": 150,
    "0xe16b3d233331688b44fe4329644dd4956df40a15": 1000,
```

```
"0xf04679700642fd1455782e1acc37c314aab1a847": 1550,  
"0xf336ed6a7c7529df70552377d641088be4dc4519": 1000  
}  
}
```

Request 1000 TBB Testing Tokens

Write a tweet and let me know how did you like this book! Tag me in it [@Web3Coach¹²²](#) and include your account address 0xYOUR_ADDRESS.

See you on Twitter - [@Web3Coach](#).¹²³

¹²² <https://twitter.com/Web3Coach>

¹²³ <https://twitter.com/Web3Coach>

El Fin – Congrats!

You finished the eBook + all the bonus chapters!

Did you enjoy exploring the magic behind blockchain system?

I worked 1000+ hours on “How to Build a Blockchain from Scratch in Go”. 2 years of my life so your transition to Web3 is easier. Hopefully, you now find the blockchain components as fascinating as I do, and you learned a ton of new concepts and paradigms that will expand your programming career over the next years.

Can I ask you for a favor?

Recommend this eBook to 3 of your dev friends by sending them this link right now: <https://web3coach.gumroad.com/l/build-a-blockchain-from-scratch-in-go>¹²⁴, I will be forever grateful. THANKS.

Technically, this doesn't have to be the end. If you have any more blockchain questions, or simply want to get in touch to chat about Web3, my Twitter DMs are open: <https://twitter.com/Web3Coach>¹²⁵,

Lukas

¹²⁴ <https://web3coach.gumroad.com/l/build-a-blockchain-from-scratch-in-go>

¹²⁵ <https://twitter.com/Web3Coach>