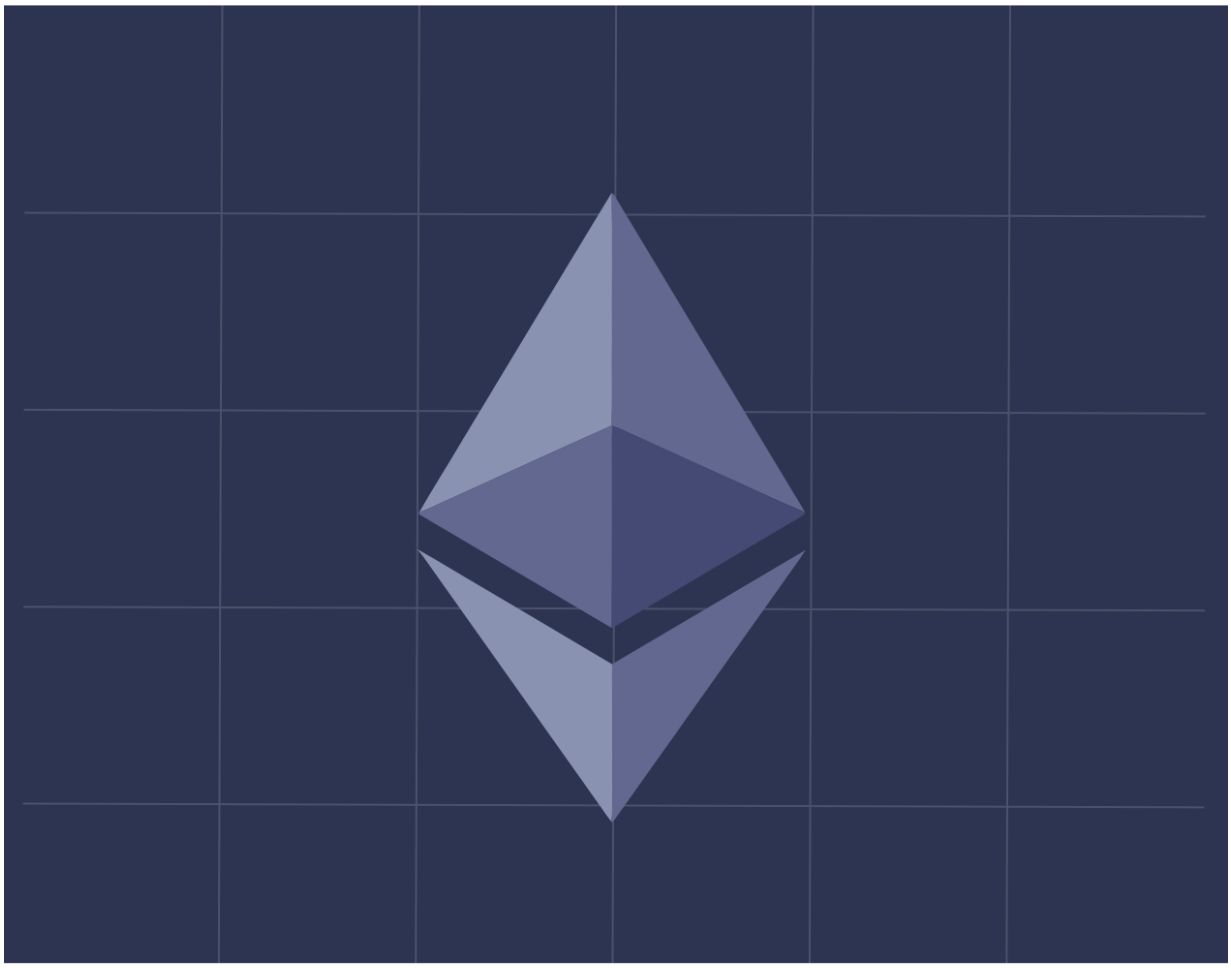

AN INTRODUCTION TO ETHEREUM AND SMART CONTRACTS

By Sebastián Peyrott

Auth0

An Introduction to Ethereum and Smart Contracts

Sebastián E. Peyrott, Auth0 Inc.

Version 0.1.0, 2017

Contents

Special Thanks

In no special order: **Diego Poza**, for being patient while this handbook was being written; **Yohanna Etchemendy**, for turning my doodles into awesome graphics; **Prosper Otemuyiwa**, for reviewing and proofreading; **Matías Woloski**, for managing and reviewing this project; **GFT's Innovation Team (Ivo Zieliński, Konrad Kozioł, David Belinchon, and Nicolás González)**, for coming up with the idea for a practical authentication solution for high-value Ethereum accounts and then going ahead and implementing it as shown in this handbook; **Manuel Aráoz**, for reviewing early drafts of this handbook and providing insight.

Foreword

This handbook is a collection of three short articles written for the Auth0 Blog[1]. The first of these articles[2] focuses on the concept of *blockchain*, a decentralized, verifiable, and ordered database of operations. To get a better sense of how blockchains operate, the first article explores Bitcoin, the first popular and successful implementation of a blockchain.

The second article[3] explores Ethereum, a different blockchain implementation focused on decentralized applications. At the end of the article an example of a decentralized application is presented. This example attempts to explore how a decentralized authentication solution for Ethereum users could be implemented. It also details many of its shortcomings and mentions that a better alternative will be presented in the next article.

Finally, the third article[4] presents the work of GFT's Innovation Team on creating a practical authentication solution for Ethereum users with high-value accounts. The article shows how blockchain technologies can be integrated into classical projects, and explores some of the intricacies of blockchains applied to decentralized applications. The solution presented in this article solves many of the problems presented in the second article.

This handbook is targeted at people wanting to learn more about blockchain technologies, making a strong emphasis on practical applications of Ethereum. It is intended for technical and non-technical minds alike. Some of the articles included in this handbook originally used videos to show certain things. These videos are available online and it is recommended that you watch them as you find them while reading.

[1] https://auth0.com/blog/
[2] https://auth0.com/blog/an-introduction-to-ethereum-and-smart-contracts/
[3] https://auth0.com/blog/an-introduction-to-ethereum-and-smart-contracts-part-2/
[4] https://auth0.com/blog/an-introduction-to-ethereum-and-smart-contracts-part-3/

Chapter 1

Bitcoin and The Blockchain

1.1 Bitcoin and the Double-Spending Problem

In 2009, someone, under the alias of Satoshi Nakamoto, released this iconic Bitcoin whitepaper[1]. Bitcoin was poised to solve a very specific problem: how can the double-spending problem[2] be solved without a central authority acting as arbiter to each transaction?

To be fair, this problem had been in the minds of researchers[3] for some time[4] before Bitcoin was released. But where previous solutions were of research quality, Bitcoin succeeded in bringing a working, production ready design to the masses.

> The earliest references to some of the concepts directly applied to Bitcoin are from the 1990s. In 2005, Nick Szabo, a computer scientist, introduced the concept of Bitgold[5], a precursor to Bitcoin, sharing many of its concepts. The similarities between Bitgold and Bitcoin are sufficient that some people have speculated he might be Satoshi Nakamoto[6].

The double-spending problem is a specific case of transaction processing[7]. Transactions, by definition, must either happen or not. Additionally, some (but not all) transactions must provide the guarantee of happening before or after other transactions (in other words, they must be atomic). Atomicity gives rise to the notion of ordering: transactions either happen or not before or after other transactions. A lack of atomicity is precisely the problem of the double-spending problem: "spending", or sending money from spender A to receiver B, must happen at a specific point in time, and before and after any other transactions. If this were not the case, it would be possible to spend money more than once in separate but simultaneous transactions.

[1] https://bitcoin.org/bitcoin.pdf
[2] https://en.wikipedia.org/wiki/Double-spending
[3] http://eprint.iacr.org/2015/464.pdf
[4] http://ieeexplore.ieee.org/document/4268195/?reload=true
[5] http://unenumerated.blogspot.com/2005/12/bit-gold.html
[6] https://dave.liberty.me/who-is-satoshi-nakamoto/
[7] https://en.wikipedia.org/wiki/Transaction_processing

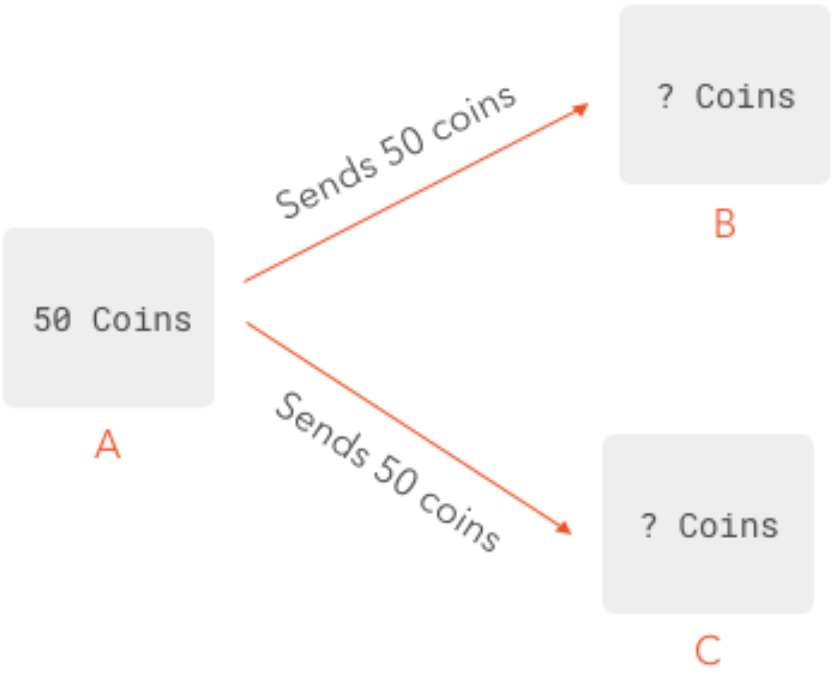

Figure 1.1: Double-spending

When it comes to everyday monetary operations, transactions are usually arbitrated by banks. When a user logs in to his or her home banking system and performs a wire transfer, it is the bank that makes sure any past and future operations are consistent. Although the process might seem simple to outsiders, it is actually quite an involved process with clearing procedures[8] and settlement requirements[9]. In fact, some of these procedures consider the chance of a double-spending situation and what to do in those cases. It should not come as a surprise that these are quite involved processes, resulting in considerable but seemingly impossible to surmount delays, were the target of computer science researchers.

[8] https://en.wikipedia.org/wiki/Clearing_%28finance%29
[9] https://en.wikipedia.org/wiki/Settlement_%28finance%29

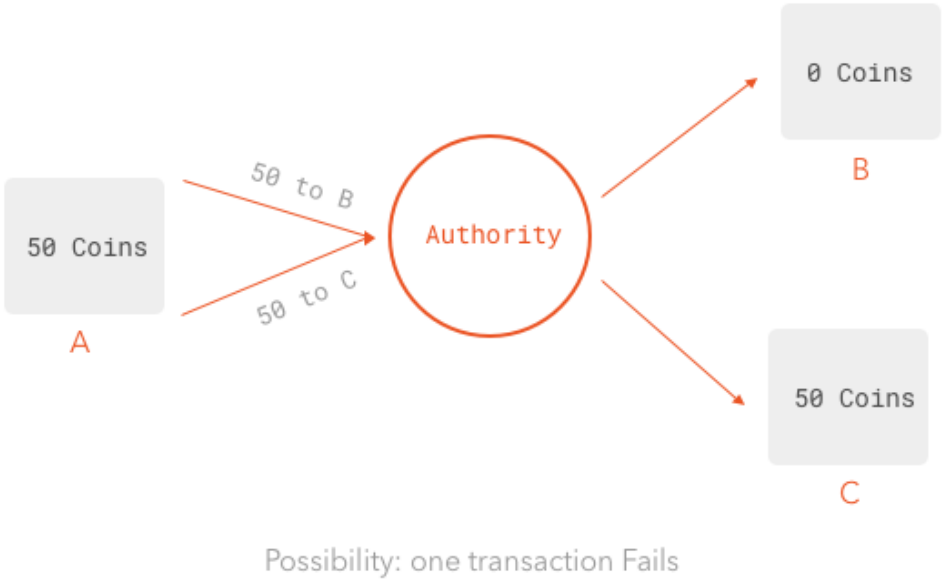

Figure 1.2: Double-spending using a central authority

1.2 The Blockchain

So, the main problem any transactional system applied to finance must address is "how to order transactions when there is no central authority". Furthermore, there can be no doubts as to whether the sequence of past transactions is valid. For a monetary system to succeed, there can be no way any parties can modify previous transactions. In other words, a "vetting process" for past transactions must also be in place. This is precisely what the blockchain system in Bitcoin was designed to address.

> If you are interested in reading about systems that must reach consensus and the problems they face, the paper for The Byzantine Generals Problem[10] is a good start.

Although at this point the concept of what a blockchain is is still murky, before getting into details about it, let's go over the problems the blockchain attempts to address.

[10] http://lamport.azurewebsites.net/pubs/byz.pdf

1.2.1 Validating Transactions

Public-key cryptography[11] is a great tool to deal with one of the problems: validating transactions. Public-key cryptography relies on the asymmetrical mathematical complexity of a very specific set of problems. The asymmetry in public-key cryptography is embodied in the existence of two keys: a public and a private key. These keys are used in tandem for specific purposes. In particular:

- Data encrypted with the public-key can only be decrypted by using the private-key.
- Data signed with the private-key can be verified using the public-key.

The private-key cannot be derived from the public-key, but the public-key can be derived from the private-key. The public-key is meant to be safely shared and can usually be freely exposed to anyone.

Of interest for creating a verifiable set of transactions is the operation of signing data. Let's see how a very simple transaction can be verified through the use of public-key cryptography.

Let's say there is an account holder A who owns 50 coins. These coins were sent to him as part of a previous transaction. Account holder A now wants to send these coins to account holder B. B, and anybody else who wants to scrutinize this transaction, must be able to verify that it was actually A who sent the coins to B. Furthermore, they must be able to see B redeemed them, and no one else. Obviously, they should also be able to find the exact point in time, relative to other transactions, in which this transaction took place. However, at this point we cannot do this. We can, fortunately, do everything else.

For our simple example, let's say the data in the transaction is just an identifier for the previous transaction (the one that gave A 50 coins in first place), the public-key of the current owner and the signature from the previous owner (confirming he or she sent those coins to A in first place):

```
{
    "previous-transaction-id": "FEDCBA987654321...",
    "owner-pubkey": "123456789ABCDEF...",
    "prev-owner-signature": "AABBCCDDEEFF112233..."
}
```

The number of coins of the current transaction is superfluous: it is simply the same amount as the previous transaction linked in it.

Proof that A is the owner of these coins is already there: his or her public-key is embedded in the transaction. Now whatever action is taken by A must be verified in some way. One way to do this would be to add information to the transaction and then produce a new signature. Since A wants to send money to B, the added information could simply be B's public-key. After creating this new transaction it could be signed using A's private-key. This proves A, and only A, was involved in the creating of this transaction. In other words, in JavaScript based pseudo-code:

```
function aToB(privateKeyA, previousTransaction, publicKeyB) {
    const transaction = {
        "previous-transaction-id": hash(previousTransaction),
        "owner-pubkey": publicKeyB
```

[11] https://en.wikipedia.org/wiki/Public-key_cryptography

```
    };

    transaction["prev-owner-signature"] = sign(privateKeyA, transaction);

    return transaction;
}
```

An interesting thing to note is that we have defined transactions IDs as simply the hash of their binary representation. In other words, a transaction ID is simply its hash (using an, at this point, unspecified hashing algorithm). This is convenient for several reasons we will explain later on. For now, it is just one possible way of doing things.

Let's take the code apart and write it down step-by-step:

1. A new transaction is constructed pointing to the previous transaction (the one that holds A's 50 coins) and including B's public signature (new transaction = old transaction ID plus receiver's public key).
2. A signature is produced using the new transaction and the previous transaction owner's private key (A's private key).

That's it. The signature in the new transaction creates a verifiable link between the new transaction and the old one. The new transaction points to the old one explicitly and the new transaction's signature can only be generated by the holder of the private-key of the old transaction (the old transaction explicitly tells us who this is through the `owner-pubkey` field). So the old transaction holds the public-key of the one who can spend it, and the new transaction holds the public-key of the one who received it, along with the signature created with the spender's private-key.

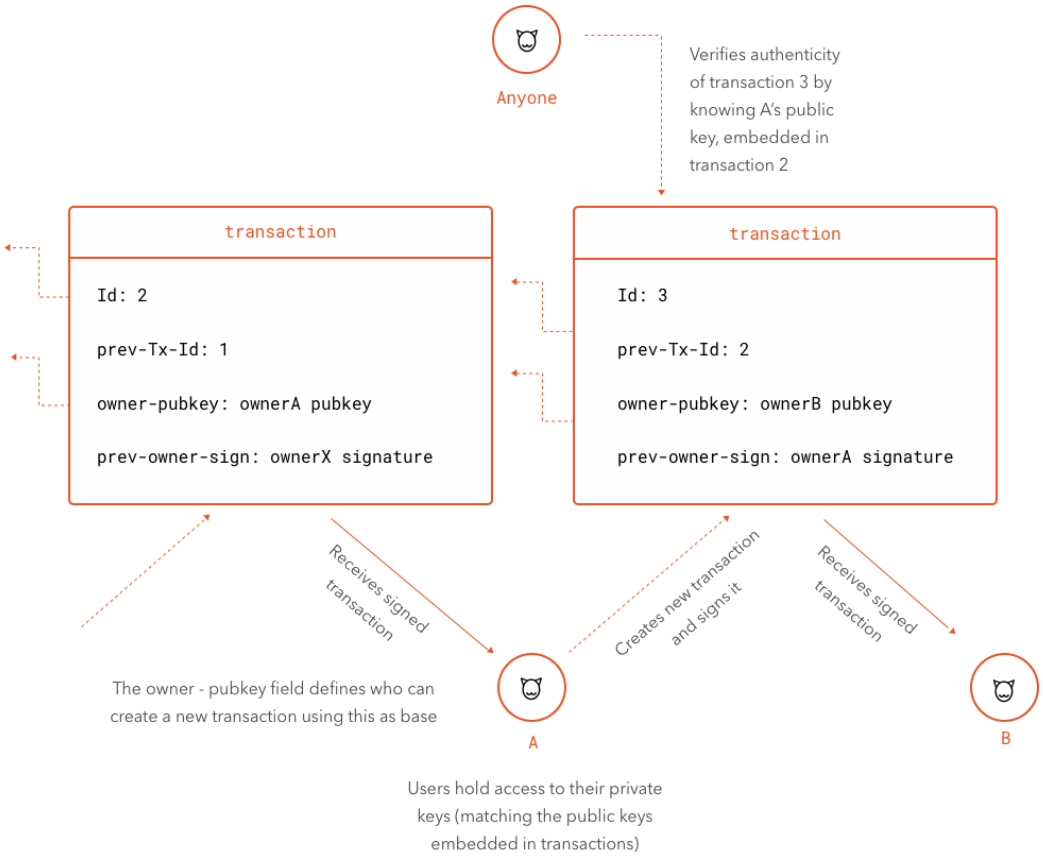

Figure 1.3: Verified transactions

If this seems hard to grasp at this point, think of it this way: it is all derived from this simple expression: *data signed with the private-key can be verified using the public-key.* There is nothing more to it. The spender simply signs data that says "I am the owner of transaction ID XXX, I hereby send every coin in it to B". B, and anybody else, can check that it was A, and only A, who wrote that. To do so, they need only access to A's public-key, which is available in the transaction itself. It is mathematically guaranteed that no key other than A's private-key can be used in tandem with A's public-key. So by simply having access to A's public-key anyone can see it was A who sent money to B. This makes B the rightful owner of that money. Of course, this is a simplification. There are two things we have not considered: who said those 50 coins where of A's property (or, in other words, did A just take ownership of some random transaction, is he or she the rightful

owner?) and when exactly did A send the coins to B (was it before or after other transactions?).

> If you are interested in learning more about the math behind public-key cryptography, a simple introduction with code samples is available in chapter 7 of The JWT Handbook[12].

Before getting into the matter of ordering, let's first tackle the problem of *coin genesis*. We assumed A was the rightful owner of the 50 coins in our example because the transaction that gave A his or her coins was simply modeled like any other transaction: it had A's public-key in the owner field, and it did point to a previous transaction. So, who gave those coins to A? What's more, who gave the coins to that other person? We need only follow the transaction links. Each transaction points to the previous one in the *chain*, so where did those 50 coins come from? At some point that chain must end.

To understand how this works, it is best to consider an actual case, so let's see how Bitcoin handles it. Coins in Bitcoin were and are created in two different ways. First there is the unique *genesis block*. The genesis block is a special, hardcoded transaction that points to no other previous transaction. It is the first transaction in the system, has a specific amount of Bitcoins, and points to a public-key that belongs to Bitcoin creator Satoshi Nakamoto. Some of the coins in this transaction were sent to some addresses, but they never were really used that much. Most of the coins in Bitcoin come from another place: they are an *incentive*. As we will see in the next section about ordering transactions, the scheme employed to do this requires nodes in the network to contribute work in the form of computations. To create an incentive for more nodes to contribute computations, a certain amount of coins are awarded to contributing nodes when they successfully complete a task. This incentive essentially results in special transactions that give birth to new coins. These transactions are also ends to links of transactions, as well as the genesis block. Each coin in Bitcoin can be traced to either one of these incentives or the genesis block. Many cryptocurrency systems adopt this model of coin genesis, each with its own nuances and requirements for coin creation. In Bitcoin, per design, as more coins get created, less coins are awarded as incentive. Eventually, coin creation will cease.

1.2.2 Ordering Transactions

The biggest contribution Bitcoin brought to existing cryptocurrency schemes was a decentralized way to make transactions atomic. Before Bitcoin, researchers proposed different schemes to achieve this. One of those schemes was a simple voting system. To better understand the magic of Bitcoin's approach, it is better to explore these attempts.

In a voting system, each transaction gets broadcast by the node performing it. So, to continue with the example of A sending 50 coins to B, A prepares a new transaction pointing to the one that gave him or her those 50 coins, then puts B's public-key in it and uses his or her own private-key (A's) to sign it. This transaction is then sent to each node known by A in the network. Let's say that in addition to A and B, there are three other nodes: C, D, E.

[12] https://auth0.com/e-books/jwt-handbook

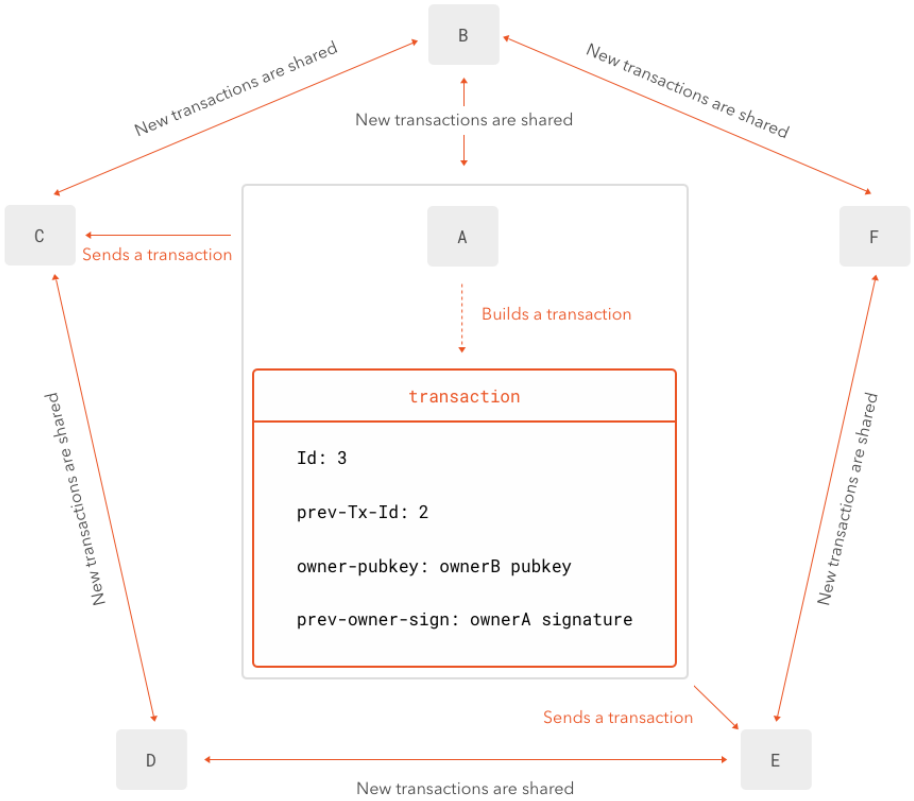

Figure 1.4: A broadcasts the transaction

Now let's imagine A is in fact a malicious node. Although it appears A wants to send B 50 coins, at the same time A broadcasts this transaction, it also broadcasts a different one: A sends those same 50 coins to C.

```
const aToB = {
    "previous-transaction-id": "FEDCBA987654321...",
    "owner-pubkey": "123456789ABCDEF...", // B
    "prev-owner-signature": "..."
};

const aToC = {
    "previous-transaction-id": "FEDCBA987654321...",
    "owner-pubkey": "00112233445566...", // C
    "prev-owner-signature": "..."
```

```
};
```

Note how `previous-transaction-id` points to the same transaction. `A` sends simultaneously this transaction to different nodes in the network. Who gets the 50 coins? Worse, if those 50 coins were sent in exchange for something, A might get goods from B and C although one of them won't get the coins.

Since this is a distributed network, each node should have some weight in the decision. Let's consider the voting system mentioned before. Each node should now cast a vote on whether to pick which transaction goes first.

Node	Vote
A	A to B
B	A to B
C	A to C
D	A to C
E	A to B

Each node casts a vote and `A to B` gets picked as the transaction that should go first. Obviously, this invalidates the `A to C` transaction that points to the same coins as `A to B`. It would appear this solution works, but only superficially so. Let's see why.

First, let's consider the case A has colluded with some other node. Did E cast a random vote or was it in some way motivated by A to pick one transaction over the other? There is no real way to determine this.

Secondly, our model does not consider the speed of propagation of transactions. In a sufficiently large network of nodes, some nodes may see some transactions before others. This causes votes to be unbalanced. It is not possible to determine whether a future transaction might invalidate the ones that have arrived. Even more, it is not possible to determine whether the transaction that just arrived was made before or after some other transaction waiting for a vote. Unless transactions are seen by all nodes, votes can be unfair. Worse, some node could actively delay the propagation of a transaction.

Lastly, a malicious node could inject invalid transactions to cause a targeted denial of service. This could be used to favor certain transactions over others.

Votes do not fix these problems because they are inherent to the design of the system. Whatever is used to favor one transaction over the other cannot be left to choice. As long as a single node, or group of nodes, can, in some way, favor some transactions over others, the system cannot work. It is precisely this element that made the design of cryptocurrencies such a hard endeavor. A strike of genius was needed to overcome such a profound design issue.

> The problem of malicious nodes casting a vote in distributed systems is best known as The Byzantine Generals Problem[13]. Although there is mathematical proof that this problem can be overcome as long as there is a certain ratio of non-malicious nodes, this

[13] http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf

does not solve the problem for cryptocurrencies: nodes are cheap to add. Therefore, a different solution is necessary.

1.2.2.1 Physics to the Rescue

Whatever system is used to ensure some transactions are preferred over others, no node should be able to choose which of these are with 100% certainty. And there is only one way one can be sure this is the case: if it is a *physical impossibility* for the node to be able to do this. Nodes are cheap to add, so no matter how many nodes a malicious user controls, it should still be hard for him or her to use this to his or her advantage.

The answer is CPU power. What if ordering transactions required a certain amount of work, verifiable work, in such a way that it would be hard to perform initially, but cheap to verify. In a sense, cryptography works under the same principles: certain related operations are computationally infeasible to perform while others are cheap. Encrypting data is cheap next to brute-forcing the encryption key. Deriving the public-key from the private-key is cheap, while it is infeasible to do it the other way around. *Hashing data is cheap, while finding a hash with a specific set of requirements (by modifying the input data) is not.* And that is the main operation Bitcoin and other cryptocurrencies rely on to make sure no node can get ahead of others, on average. Let's see how this works.

First, let's define what a block is. A block is simply a group of transactions. Inside the block, these transactions are set in a specific order and fulfill the basic requirements of any transaction. In particular, an invalid transaction (such as one taking funds from an account with no funds) cannot be part of a block. In addition to the transactions, a block carries something called *proof-of-work*. The proof-of-work is data the allows any node to verify that the one who created this block performed a considerable amount of computational work. In other words, no node can create a valid block without performing an indefinite but considerable amount of work. We will see how this works later, but for now know that creating any block requires a certain amount of computing power and that any other node can check that that power has been spent by whomever created the block.

Now let's go back to our previous example of a malicious node, A, double-spending 50 coins by trying to create to two separate transactions at the same time, one sending money to B and the other to C. After A broadcasts both transactions to the network, every node working on creating blocks (which may include A) pick a number of transactions and order them in whichever way they prefer. These nodes will note that two incompatible transactions are part of the same block and will discard one. They are free to pick which one to discard. After placing these transactions in the order they chose, each node starts solving the puzzle of finding a hash for the block that fits the conditions set by the protocol. One simple condition could be "find a hash for this block with three leading zeroes". To iterate over possible solutions for this problem, the block contains a special variable field known as the "nonce". Each node must iterate as many times as necessary until they find the nonce that creates a block with a hash that fits the conditions set by the protocol (three leading zeroes). Since each change in the nonce basically results in a random output for a cryptographically secure hash function, finding the nonce is a game of chance and can only be sped up by increasing computation power. Even then, a less powerful node might find the right nonce before a more powerful node, due to the randomness of the problem.

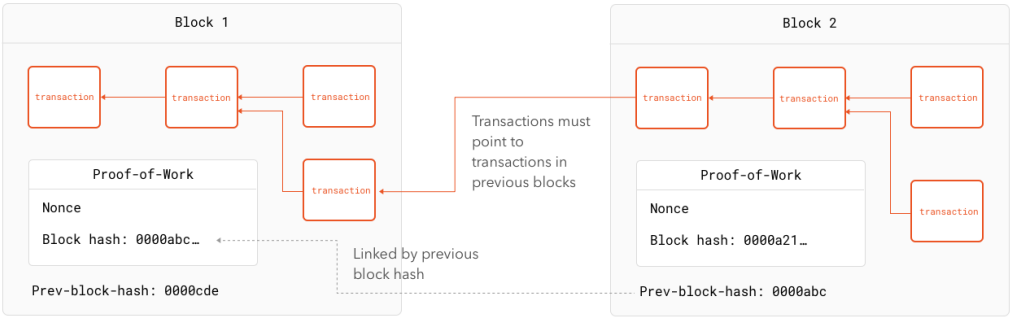

Figure 1.5: A sample block

This creates an interesting scenario because even if A is a malicious node and controls another node (for instance, E) any other node on the network still has a chance of finding a different valid block. In other words, this scheme makes it hard for malicious nodes to take control of the network.

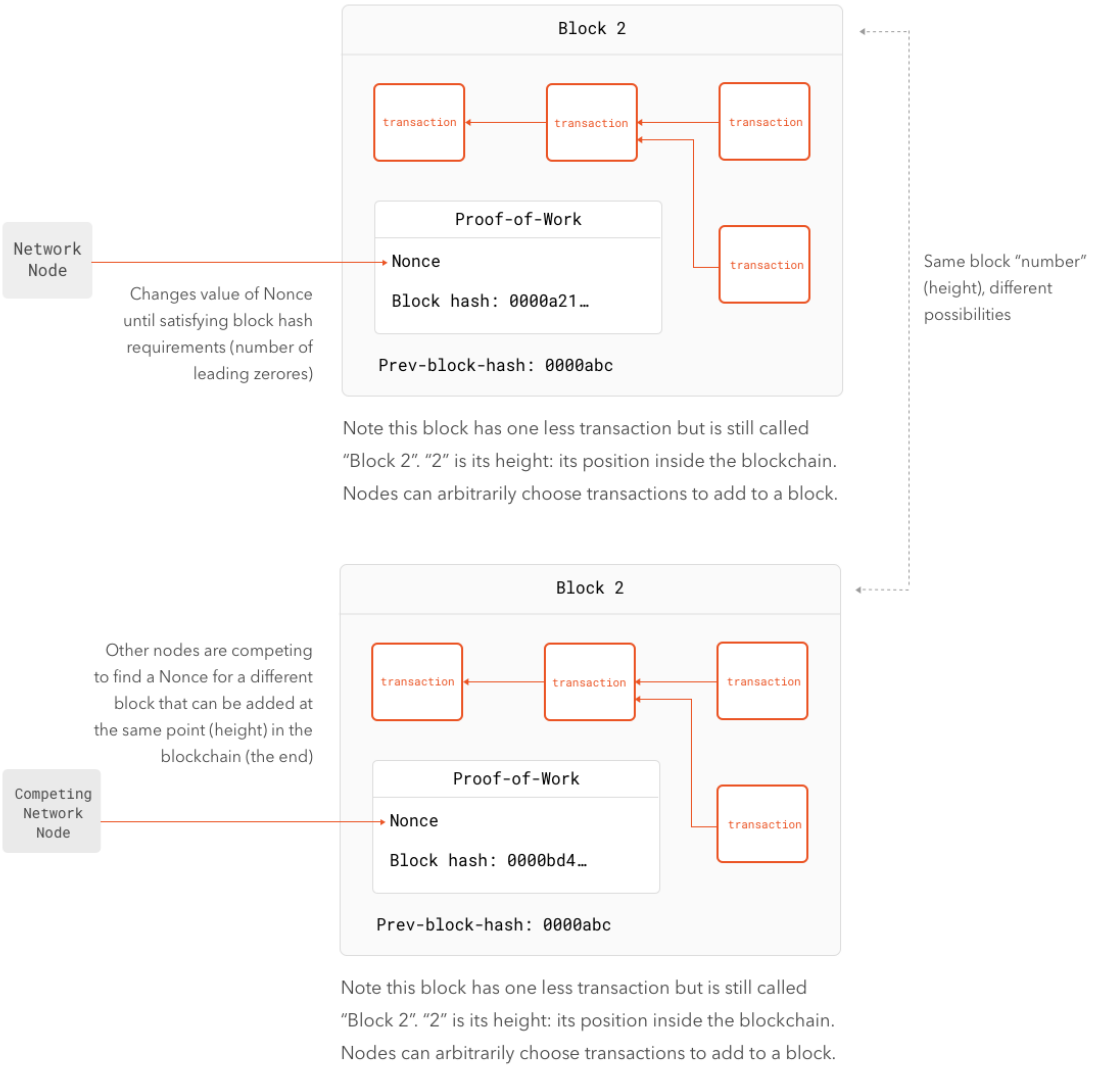

Figure 1.6: Proof-of-work

Still, the case of a big number of malicious nodes colluding and sharing CPU power must be considered. In fact, an entity controlling a majority of the nodes (in terms of CPU power, not number) could exercise a double-spending attack by creating blocks faster than other nodes. Big enough networks rely on the difficulty of amassing CPU power. While in a voting system an attacker need only add nodes to the network (which is easy, as free access to the network is a design target), in a CPU power based scheme an attacker faces a physical limitation: getting access to more and

more powerful hardware.

1.2.3 Definition

At last we can attempt a full definition of what a blockchain is and how it works. A blockchain is a verifiable transaction database carrying an ordered list of all transactions that ever occurred. Transactions are stored in blocks. Block creation is a purposely computationally intensive task. The difficulty of creation of a valid block forces anyone to spend a certain amount of work. This ensures malicious users in a big enough network cannot easily outpass honest users. Each block in the network points to the previous block, effectively creating a chain. The longer a block has been in the blockchain (the farther it is from the last block), the lesser the probability it can ever be removed from it. In other words, the older the block, the more secure it is.

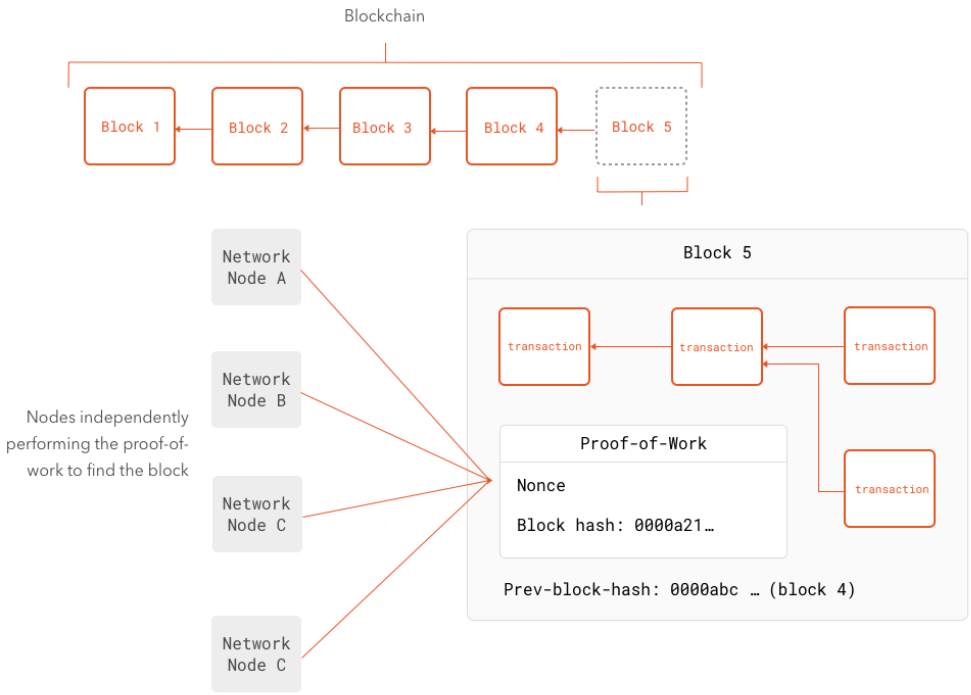

Figure 1.7: The blockchain

One important detail we left in previous paragraphs is what happens when two different nodes find different but still valid blocks at the same time. In a sense, this looks like the same problem

transactions had: which one to pick. In contrast with transactions, the proof-of-work system required for each block lets us find a convenient solution: since each block requires a certain amount of work, it is only natural that the only valid blockchain is the one with most blocks in it. Think about it: if the proof-of-work system works because each block demands a certain amount of work (and time), the longest set of valid blocks is the *hardest* to break. If a malicious node or group of nodes were to attempt to create a different set of valid blocks, by always picking the longest blockchain, they would always have to redo a bigger number of blocks (because each node points to the previous one, changing one block forces a change in all blocks after it). This is also the reason malicious groups of nodes need to control over 50% of the computational power of the network to actually carry any attack. Less than that, and the rest of the network will create a longer blockchain faster.

Valid blocks that are valid but find their way into shorter *forks* of the blockchain are discarded if a longer version of the blockchain is computed by other nodes. The transactions in the discarded blocks are sent again to the pool of transactions awaiting inclusion into future blocks. This causes new transactions to remain in an *uncofirmed* state until they find their way into the longest possible blockchain. Nodes periodically receive newer versions of the blockchain from other nodes.

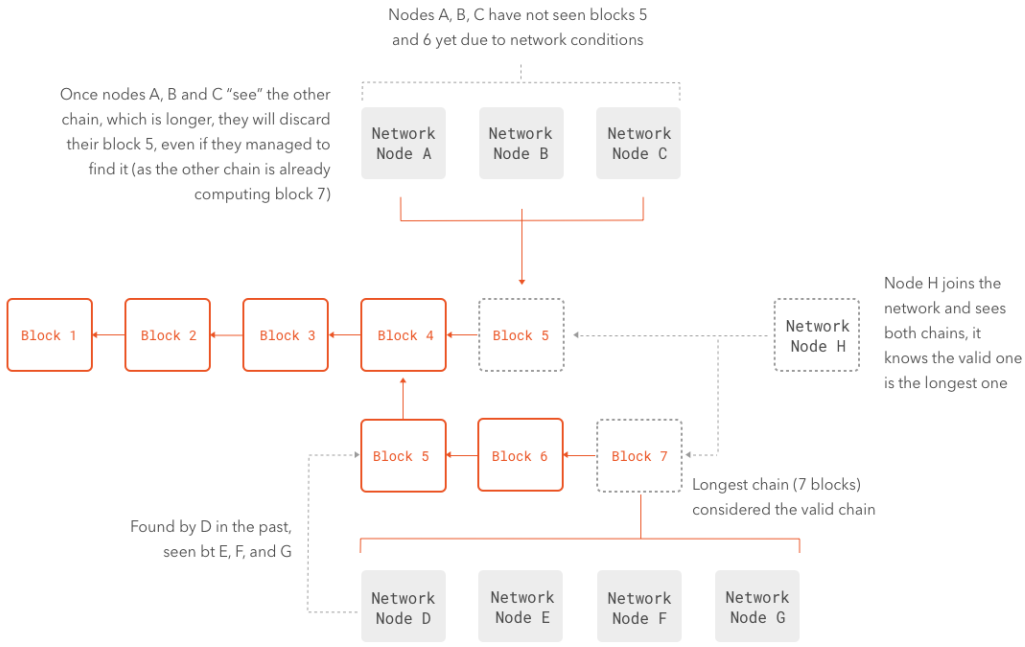

Figure 1.8: Blockchain forks

It is entirely possible for the network to be forked if a sufficiently large number of nodes gets disconnected at the same time from another part of the network. If this happens, each fork will continue creating blocks in isolation from the other. If the networks merge again in the future, the nodes will compare the different versions of the blockchains and pick the longer one. The fork with the greater computational power will always win. If the fork were to be sustained for a long enough period of time, a big number of transactions would be undone when the merge took place. It is for this reason that forks are problematic.

Forks can also be caused by a change in the protocol or the software running the nodes. These changes can result in nodes invalidating blocks that are considered valid by other nodes. The effect is identical to a network-related fork.

1.3 Example: a Perpetual Message System Using Webtasks and Bitcoin

Although we have not delved into the specifics of how Bitcoin or Ethereum handle transactions, there is a certain *programmability* built into them. Bitcoin allows for certain conditions to be specified in each transaction. If these conditions are met, the transaction can be spent. Ethereum, on the other hand, goes much further: a Turing-complete programming language is built into the system. We will focus on Ethereum in the next chapter, but for now we will take a look at creative ways in which the concepts of the blockchain can be exploited for more than just sending money. For this, we will develop a simple perpetual message system on top of Bitcoin. How will it work?

We have seen the blockchain stores transactions that can be verified. Each transaction is signed by the one who can perform it and then broadcast to the network. It is then stored inside a block after performing a proof-of-work. This means that any information embedded in the transaction is stored forever inside the blockchain. The timestamp of the block serves as proof of the message's date, and the proof-of-work process serves as proof of its immutable nature.

Bitcoin uses a scripting system that describes steps a user must perform to spend money. The most common script is simply "prove you are the owner of a certain private-key by signing this message with it". This is known as the "pay to pubkey hash" script. In decompiled form it looks like:

```
<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Where `<sig>` and `<pubKey>` are provided by the spender and the rest is specified by the original sender of the money. This is simply a sequence of mixed data and operations. The interpreter for this script is a stack-based virtual machine. The details of execution are out of scope for this handbook, but you can find a nice summary at the Bitcoin Wiki[14]. The important take from this is that transactions can have data embedded in them in the scripts.

In fact, there exists a valid opcode for embedding data inside a transaction: the `OP_RETURN` opcode. Whatever data follows the `OP_RETURN` opcode is stored in the transaction. Of course, there is a limit for the amount of data allowed: 40-bytes. This is very little, but still certain interesting applications can be performed with such a tiny amount of storage. One of them is our perpetual message system. Another interesting use case is the "proof of existence" concept. By storing a hash of an asset in the blockchain, it serves as proof of its existence at the point it was added to a block. In fact, there already exists such a project[15]. There is nothing preventing you from using our perpetual message system for a similar use. Yet other uses allow the system to prepare transactions that can only be spent after conditions are met, or when the spender provides proof of having a certain digital asset, of when a certain minimum number of users agree to spend it. Programmability opens up many possibilities and makes for yet another great benefit of cryptocurrencies in contrast with traditional monetary systems.

[14] https://en.bitcoin.it/wiki/Script#Standard_Transaction_to_Bitcoin_address_.28pay-to-pubkey-hash.29
[15] https://proofofexistence.com

1.3.1 The Implementation

Our system will work as an HTTP service. Data will we passed in JSON format as the body of `POST` requests. The service will have three endpoints plus one for debugging.

1.3.1.1 The `/new` endpoint

It creates a new user using the username and password passed in. Sample body:

```
{
    "id": "username:password", // password is not hashed for simplicity,
                               // TLS is required!
    "testnet": true            // True to use Bitcoin's test network
}
```

The response is of the form:

```
{
    "address": "..."            // A Bitcoin address for the user just created
}
```

1.3.1.2 The `/address` endpoint

Returns the address for an existing user. Sample body:

```
{
    "id": "username:password", // password is not hashed for simplicity,
                               // TLS is required!
}
```

The response is identical to the `/new` endpoint.

1.3.1.3 The `/message` endpoint

Broadcasts a transaction to the Bitcoin network with the message stored in it. A fee is usually required for the network to accept the transaction (though some nodes may accept transactions with no fees). Messages can be at most 33 bytes long. Sample body:

```
{
    "id": "username:password",
    "fee": 667,
    "message": "test"
}
```

The response is either a transaction id or an error message. Sample of a successful response:

```
{
    "status": "Message sent!",
    "transactionId": "3818b4f03fbbf091d5b52edd0a58ee1f1834967693f5029e5112d36f5fdbf2f3"
}
```

Using the transaction id one can see the message stored in it. One can use any publicly available blockchain explorer to do this.

1.3.1.4 The `/debugNew` endpoint

Similar to the `/new` endpoint but allows one to create an user with an existing Bitcoin private key (and address). Sample body:

```
{
    "id": "username:password", // password is not hashed for simplicity,
                               // TLS is required!
    "testnet": true,           // True to use Bitcoin's test network
    "privateKeyWIF": "..."     // A private key in WIF format.
                               // Note testnet keys are different from livenet keys,
                               // so the private key must agree with the
                               // value of the "testnet" key in this object
}
```

The response is identical to the `/new` endpoint.

1.3.2 The Code

The most interesting endpoint is the one that builds and broadcasts the transaction (`/message`). We use the `bitcore-lib` and `bitcore-explorers` libraries to do this:

```
getUnspentUtxos(from).then(utxos => {
    let inputTotal = 0;
    utxos.some(utxo => {
        inputTotal += parseInt(utxo.satoshis);
        return inputTotal >= req.body.fee;
    });
    if(inputTotal < req.body.fee) {
        res.status(402).send('Not enough balance in account for fee');
        return;
    }

    const dummyPrivateKey = new bitcore.PrivateKey();
    const dummyAddress = dummyPrivateKey.toAddress();

    const transaction =
        bitcore.Transaction()
                .from(utxos)
```

```
                .to(dummyAddress, 0)
                .fee(req.body.fee)
                .change(from)
                .addData(`${messagePrefix}${req.body.message}`)
                .sign(req.account.privateKeyWIF);

    broadcast(transaction.uncheckedSerialize()).then(body => {
        if(req.webtaskContext.secrets.debug) {
            res.json({
                status: 'Message sent!',
                transactionId: body,
                transaction: transaction.toString(),
                dummyPrivateKeyWIF: dummyPrivateKey.toWIF()
            });
        } else {
            res.json({
                status: 'Message sent!',
                transactionId: body
            });
        }
    }, error => {
        res.status(500).send(error.toString());
    });
}, error => {
    res.status(500).send(error.toString());
});
```

The code is fairly simple:

1. Gets the unspent transactions for an address (i.e. the coins available, the balance).
2. Build a new transaction using the unspent transactions as input.
3. Point the transaction to a new, empty address. Assign 0 coins to that address (do not send money unnecessarily).
4. Set the fee.
5. Set the address where the unspent money will get sent back (the change address).
6. Add our message.
7. Broadcast the transaction.

Bitcoin requires transactions to be constructed using the money from previous transactions. That is, when coins are sent, it is not the origin address that is specified, rather it is the transactions pointing to that address that are included in a new transaction that points to a different destination address. From these transactions is subtracted the money that is then sent to the destination. In our case, we use these transactions to pay for the fee. Everything else gets sent back to our address.

1.3.3 Deploying the Example

Thanks to the power of Webtasks[16], deploying and using this code is a piece of cake. First clone the repository:

```
git clone git@github.com:auth0-blog/ethereum-series-bitcoin-perpetual-message-example.git
```

Now make sure you have the Webtask command-line tools installed:

```
npm install -g wt-cli
```

If you haven't done so, initialize your Webtask credentials (this is a one time process):

```
wt init
```

Now deploy the project:

```
cd ethereum-series-bitcoin-perpetual-message-example
wt create --name bitcoin-perpetual-message \
--meta 'wt-node-dependencies={"bcryptjs":"2.4.3","bitcore-lib":"0.13.19",\
"bitcore-explorers-bitcore-lib-0.13.19":"1.0.1-3"}' app.js
```

Your project is now ready to test! Use CURL to try it out:

```
curl -X POST \
https://wt-sebastian_peyrott-auth0_com-0.run.webtask.io/bitcoin-perpetual-message/new \
-d '{ "id":"test:test", "testnet":true }' -H "Content-Type: application/json"

{"address":"mopYghMw5i7rYiq5pfdrqFt4GvBus8G3no"} # This is your Bitcoin address
```

You now have to add some funds to your new Bitcoin address. If you are on Bitcoin's testnet, you can simply use a faucet[17].

> Faucets are Bitcoin websites that give free coins to addresses. These are easy to get for the testnet. For the "livenet" you need to buy Bitcoins using a Bitcoin exchange[18].

Now send a message!

```
curl -X POST \
https://wt-sebastian_peyrott-auth0_com-0.run.webtask.io/bitcoin-perpetual-message/message \
-d '{ "id":"test:test", "fee":667, "message":"test" }' -H "Content-Type: application/json"

{"status":"Message sent!",\
"transactionId":"3818b4f03fbbf091d5b52edd0a58ee1f1834967693f5029e5112d36f5fdbf2f3"}
```

Now you can look at the transaction[19] using a blockchain explorer and the transaction id. If you go down to the bottom of the page in the link before you will see our message with a prefix `WTMSG:test`. This will get stored in the blockchain forever.

[16] https://webtask.io
[17] https://testnet.manu.backend.hamburg/faucet
[18] https://en.wikipedia.org/wiki/Digital_currency_exchange
[19] https://www.blocktrail.com/tBTC/tx/3818b4f03fbbf091d5b52edd0a58ee1f1834967693f5029e5112d36f5fdbf2f3

Try it yourself! The webtask at `https://wt-sebastian_peyrott-auth0_com-0.run.webtask.io/bitcoin-perpetual-` is live. You will need to create your own account and fund it, though.

You can also get the full code[20] for this example and run it!

1.4 Summary

Blockchains enable distributed, verified transactions. At the same time they provide a creative solution to the double-spending problem. This has enabled the rise of cryptocurrencies, of which Bitcoin is the most popular example. Millions of dollars in Bitcoins are traded each day, and the trend is not giving any signs of slowing down. Bitcoin provides a limited set of operations to customize transactions. Still, many creative applications have appeared through the combination of blockchains and computations. Ethereum is the greatest example of these: marrying decentralized transactions with a Turing-complete execution environment. In the next chapter we will take a closer look at how Ethereum differs from Bitcoin and how the concept of decentralized applications was brought to life by it.

[20] https://github.com/auth0-blog/ethereum-series-bitcoin-perpetual-message-example

Chapter 2

Ethereum: a Programmable Blockchain

2.1 Introduction

In the previous chapter, we took a closer look at what blockchains are and how they help in making distributed, verifiable transactions a possibility. Our main example was Bitcoin: the world's most popular cryptocurrency. Millions of dollars, in the form of bitcoins, are traded each day, making Bitcoin one of the most prominent examples of the viability of the blockchain concept.

Have you ever found yourself asking this question: "what would happen if the provider of this service or application disappeared?" If you have, then learning about Ethereum can make a big difference for you. Ethereum is a platform to run decentralized applications: applications that do not rely on any central server. In this chapter we will explore how Ethereum works and build a simple PoC application related to authentication.

2.1.1 Blockchain Recap

A blockchain is a distributed, verifiable datastore. It works by marrying public-key cryptography with the nobel concept of the *proof-of-work*.

Each transaction in the blockchain is signed by the rightful owner of the resource being traded in the transaction. When new coins (resources) are created they are assigned to an owner. This owner, in turn, can prepare new transactions that send those coins to others by simply embedding the new owner's public key in the transaction and then signing the transaction with his or her private-key. In this way, a verifiable link of transactions is created; each new transaction, with a new owner, pointing to the previous transaction, with the previous owner.

To order these transactions and prevent the double-spending problem[1], blockchains use the *proof-*

[1] https://en.wikipedia.org/wiki/Double-spending

of-work. The proof-of-work is a procedure that establishes a cost for grouping transactions in a certain order and adding them to the blockchain. These groups of transactions are called *blocks*. Each block points to a previous block in the chain, thus the name *blockchain*. By making blocks costly to make and making sure each new block points to the previous block, any potential attacker wanting to modify the history of transactions as represented by the blockchain must pay the cost of each block modified. Since blocks point to previous blocks, modifying an old block requires paying the cost for *all* blocks after it, making changes to old blocks very costly. A blockchain compounds the difficulty of modifying the blockchain by making the cost of creating blocks be of computational nature. In other words, to create new blocks, a certain amount of CPU power must be spent. Since CPU power is dependent on the advancement of technology, it is very hard for any single malicious entity to amass enough CPU power to outspend the rest of the network. A practical attack against a blockchain-based network usually requires a single entity controlling more than 50% of the combined CPU power of the network. The bigger the network, the harder it is to perform.

But, as we saw in chapter 1, blockchains are more than just that. Transactions, by their very nature, can do more than just send resources from owner A to owner B. In fact, the very act of doing so can be described as a very simple program: the sender produces a computation (transaction) that can only be performed if the receiver produces, at some point in the future, the right inputs. In the case of a standard monetary transaction, the right input would be the proof of ownership from the receiver. In other words, the receiver can only spend the coins he received if he proves he is the rightful owner of those coins. It may seem a bit contrived but it really isn't. When you perform a wire transfer, you prove you are the owner of an account through some sort of authentication procedure. For a home-banking system that could simply be a username and a password. At a bank, it would be your ID or debit-card. These procedures are usually hardwired into the system, but with blockchains it needn't be so.

In chapter 1 we also took a cursory look at this. We first showed how Bitcoin transactions are in fact small programs that are intepreted by each node using a simple stack-based virtual-machine.

```
<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

This virtual-machine, in the case of Bitcoin, is limited by design. It is not Turing-complete and can only perform a limited number of operations. Still, its flexibility opened up the possibility for many interesting uses. The small script above, a.k.a. smart contract, is the standard "pay to pubkey hash" Bitcoin script[2]. It describes a small program that allows a sender to send coins to a receiver by verifying his identity with a public-key: the standard A to B monetary transaction, with ID cards substituted with public and private-keys. However, there's nothing preventing other uses, as long as you stick to the available operations supported by the virtual-machine. We took a look at a possible use in the previous chapter, where we created a perpetual-message system: immutable messages timestamped and forever embedded in the blockchain. The older they get, the harder it is for them to ever be changed. Nifty.

Now, we'll take a look at how Ethereum amplifies these concepts.

[2] https://en.bitcoin.it/wiki/Transaction#Pay-to-PubkeyHash

2.2 Ethereum: a Programmable Blockchain

Although the concept of the blockchain was born out of the research into cryptocurrencies, they are much more powerful than just that. A blockchain essentially encodes one thing: state transitions. Whenever someone sends a coin in Bitcoin to someone else, the global state of the blockchain is changed. Moments before account A held 50 coins, now account A is empty and account B holds 50 coins. Furthermore, the blockchain provides a cryptographically secure way of performing these state transitions. In other words, not only the state of the blockchain can be verified by any outside party, but any state transitions initiated by blockchain users can only be performed in a secure, verifiable manner.

> An interesting way to think of a blockchain is as a never-halting computation: new instructions and data are fetched from a pool, the pool of unconfirmed transactions. Each result is recorded in the blockchain, which forms the state of the computation. Any single snapshot of the blockchain is the state of the computation at that point.

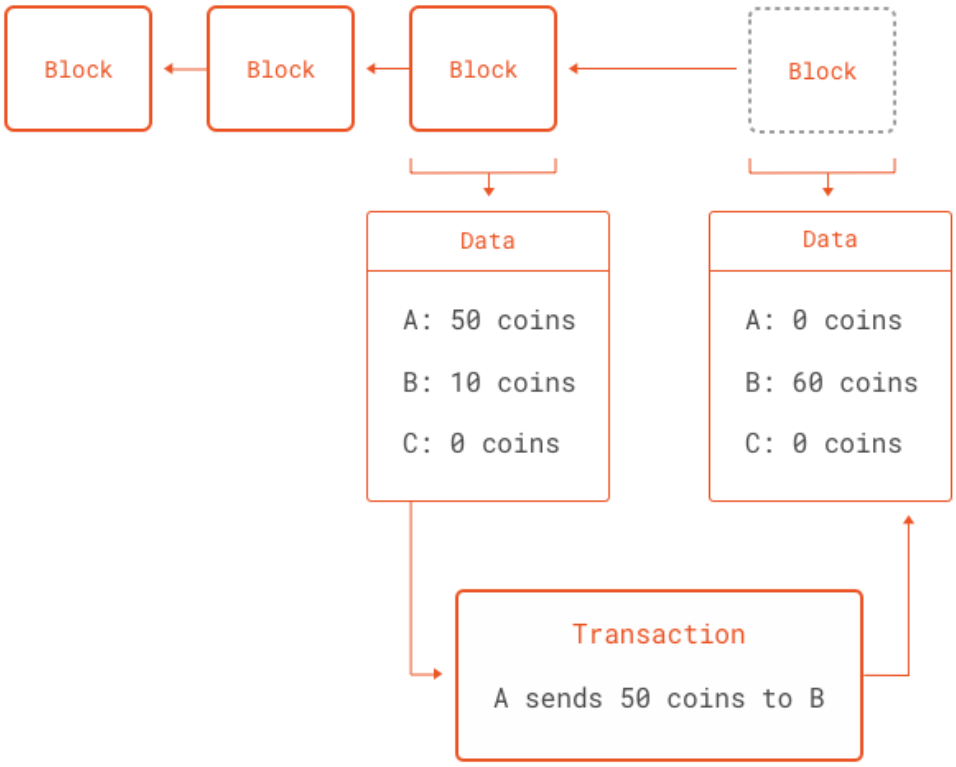

Figure 2.1: Transactions as computations

All software systems deal in some way or another with state transitions. So what if we could generalize the state transitions inside a blockchain into any software we could think of. Are there any inherent limitations in the blockchain concept that would prevent state transitions from being something different than sending coins? The answer is no. Blockchains deal with reaching consensus for decentralized computations, it does not matter what those computations are. And this is exactly what the Ethereum network brings to the table: a blockchain that can perform any computation as part of a transaction.

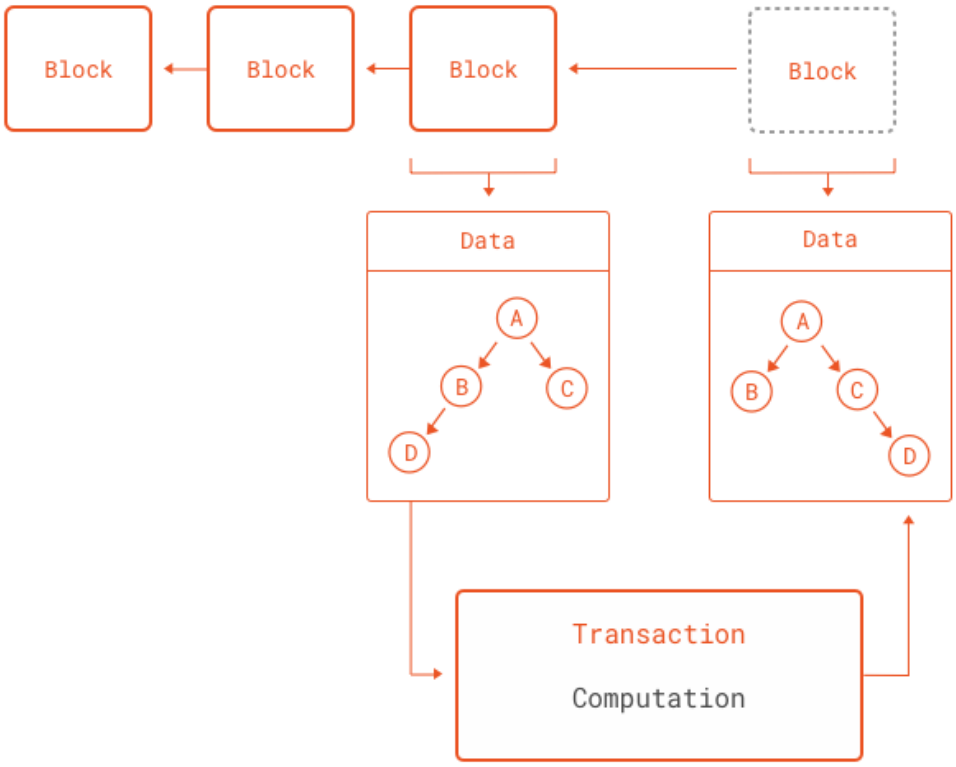

Figure 2.2: Transactions as general computations

It is easy to get lost in the world of cryptocurrencies and simple exchanges of value between two users, but there are many other applications where distributed, secure computations make sense. It is this system that allows for things like:

- Secure deposits that get returned to the payer if conditions are met (or not)
- Money that cannot be spent unless a certain number of users agree to spending it[3]
- Money that can only be spent after producing external data that satisfies rules set in the script

Given a Turing-complete system for computations associated to a blockchain, many more applications are possible. This is Ethereum.

[3] https://en.bitcoin.it/wiki/Multisignature

Take a look[4] at the things the community is working on to get a sense of the many useful ideas that can be run as decentralized applications.

2.2.1 Ether

Although Ethereum brings general computations to the blockchain, it still makes use of a "coin". Its coin is called "ether", and, as any coin, it is a number that can be stored into account addresses and can be spent or received as part of transactions or block generation. To run certain transactions, users must spend Ether. But why is this the case?

A Turing-complete language[5] is a language that, by definition, can perform any computation. In other words, if there is an algorithm for something, it can express it. Ethereum scripts, called *smart contracts*, can thus run any computation. Computations are run as part of a transaction. This means each node in the network must run computations. Any machine capable of running a Turing-complete language (i.e. a Turing machine) has one problem: the halting problem[6]. The halting problem essentially states that no Turing machine can determine beforehand whether a program run in it will either terminate (halt) or run forever. In other words, the only way of finding out if a piece of code loops forever or not is by running that code. This poses a big problem for Ethereum: no single node can get caught up in an infinite loop running a program. Doing so would essentially stop the evolution of the blockchain and halt all transactions. But there is a way around that.

Since computation is costly, and it is in fact rewarded by giving nodes that produce blocks ether (like Bitcoin), what better way to limit computations than by requiring ether for running them. Thus Ethereum solves the problem of denial of service attacks through malicious (or bugged) scripts that run forever. Every time a script is run, the user requesting the script to run must set a limit of ether to spend in it. Ether is consumed by the script as it runs. This is ensured by the virtual machine that runs the scripts. If the script cannot complete before running out of ether, it is halted at that point. In Ethereum the ether assigned to an script as a limit is known as *gas* (as in gasoline).

As ether represents value, it can be converted to other coins. Exchanges exist to trade ether for other coins. This gives ether a real money valuation[7], much like coins from Bitcoin.

2.2.2 Smart Contracts

Smart contracts are the key element of Ethereum. In them any algorithm can be encoded. Smart contracts can carry arbitrary state and can perform any arbitrary computations. They are even able to call other smart contracts. This gives the scripting facilities of Ethereum tremendous flexibility.

Smart contracts are run by each node as part of the block creation process. Just like Bitcoin, block creation is the moment where transactions actually take place, in the sense that once a transaction takes place inside a block, global blockchain state is changed. Ordering affects state changes, and

[4] http://dapps.ethercasts.com
[5] https://en.wikipedia.org/wiki/Turing_completeness
[6] https://en.wikipedia.org/wiki/Halting_problem
[7] https://coinmarketcap.com/currencies/ethereum/

just like in Bitcoin, each node is free to choose the order of transactions inside a block. After doing so (and executing the transactions), a certain amount of work must be performed to create a valid block. In contrast to Bitcoin, Ethereum follows a different pattern for selecting which blocks get added to the valid blockchain. While in Bitcoin the longest chain of valid blocks is always the rightful blockchain, Ethereum follows a protocol called GHOST[8] (in fact a variation thereof). The GHOST protocol allows for stale blocks, blocks that were computed by other nodes but that would otherwise be discarded since others have computed newer blocks, to be integrated into the blockchain, reducing wasted computing power and increasing incentives for slower nodes. It also allows for faster confirmation of transactions: whereas in Bitcoin blocks are usually created every 10 minutes, in Ethereum blocks are created within seconds. Much discussion[9] has gone into whether this protocol is an improvement over the much simpler "fastest longest chain" protocol in Bitcoin, however this discussion is out of scope for this handbook. For now this protocol appears to run with success in Ethereum.

An important aspect of how smart contracts work in Ethereum is that they have their own address in the blockchain. In other words, contract code is not carried inside each transaction that makes use of it. This would quickly become unwieldy. Instead, a node can create a special transaction that assigns an address to a contract. This transaction can also run code at the moment of creation. After this initial transaction, the contract becomes forever a part of the blockchain and its address never changes. Whenever a node wants to call any of the methods defined by the contract, it can send a message to the address for the contract, specifying data as input and the method that must be called. The contract will run as part of the creation of newer blocks up to the *gas limit* or completion. Contract methods can return a value or store data. This data is part of the state of the blockchain.

2.2.2.1 State

An interesting aspect of contracts being able to store data is how that can be handled in an efficient way. If state is mutated by contracts, and the nature of the blockchain ensures that state is always consistent across all nodes, then all nodes must have access to the whole state stored in the blockchain. Since the size of this storage in unlimited in principle, this raises questions with regards to how to handle this effectively as the network scales. In particular, how can smaller and less powerful nodes make use of the Ethereum network if they can't store the whole state? How can they perform computations? To solve this, Ethereum makes use of something called Merkle Patricia Trees[10].

A Merkle Patricia Tree is a special kind of data structure that can store cryptographically authenticated data in the form of keys and values. A Merkle Patricia Tree with a certain group of keys and values can only be constructed in a single way. In other words, given the same set of keys and values, two Merkle Patricia Trees constructed independently will result in the same structure bit-by-bit. A special property of Merkle Patricia Trees is that the hash of the root node (the first node in the tree) depends on the hashes of all sub-nodes. This means that any change to the tree results in a completely different root hash value. Changes to a leaf node cause all hashes leading

[8] https://www.cryptocompare.com/coins/guides/what-is-the-ghost-protocol-for-ethereum/
[9] https://news.ycombinator.com/item?id=7553418
[10] https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie/

to the root hash through that and sister branches to be recomputed. What we have described is in fact the "Merkle" part of the tree, the "Patricia" part comes from the way keys are located in the tree. Patricia trees are tries[11] where any node that is an only child is merged with its parent. They are also known as "radix trees" or "compact prefix trees". A trie is a tree structure that uses prefixes of the keys to decide where to put each node.

The Merkle Patricia Trees implemented in Ethereum have other optimizations that overcome inefficiencies inherent to the simple description presented here.

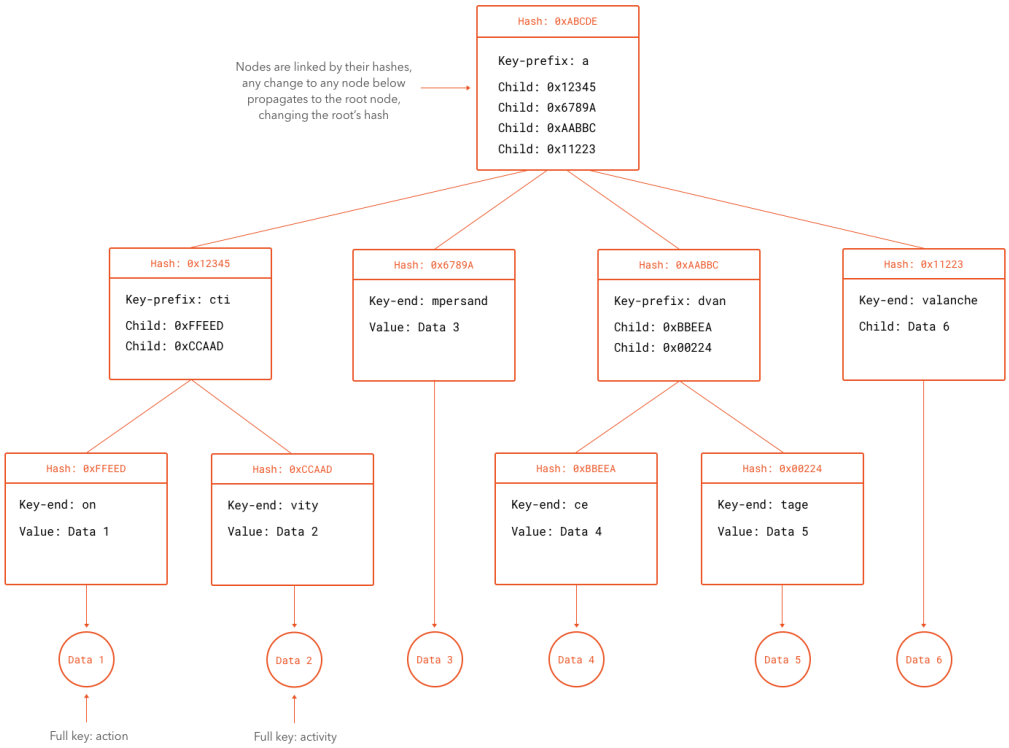

Figure 2.3: Simplified Merkle Patricia Tree

For our purposes, the Merkle aspect of the trees is what matters in Ethereum. Rather than keeping the whole tree inside a block, the hash of its root node is embedded in the block. If some malicious node were to tamper with the state of the blockchain, it would become evident as soon as other nodes computed the hash of the root node using the tampered data. The resulting hash would simply not match with the one recorded in the block. At this point we should find ourselves asking a big question: why not simply take the hash of the data? Merkle Patricia Trees are used in

[11] https://en.wikipedia.org/wiki/Trie

Ethereum for a different, but very important reason: most of the time, nodes do not need a full copy of the whole state of the system. Rather, they want to have a partial view of the state, complete enough to perform any necessary computations for newer blocks or to read the state from some specific address. Since no computations usually require access to the whole state stored in the blockchain, downloading all state would be superfluous. In fact, if nodes had to do this, scalability would be a serious concern as the network expanded. To verify a partial piece of the state at a given point, a node need only download the data necessary for a branch of the tree and the hashes of its siblings. Any change in the data stored at a leaf would require a malicious node to be able to carry a preimage attack[12] against the hashing algorithm of the tree (to find the values for the siblings that combined with the modified data produce the same root hash as the one stored in the block).

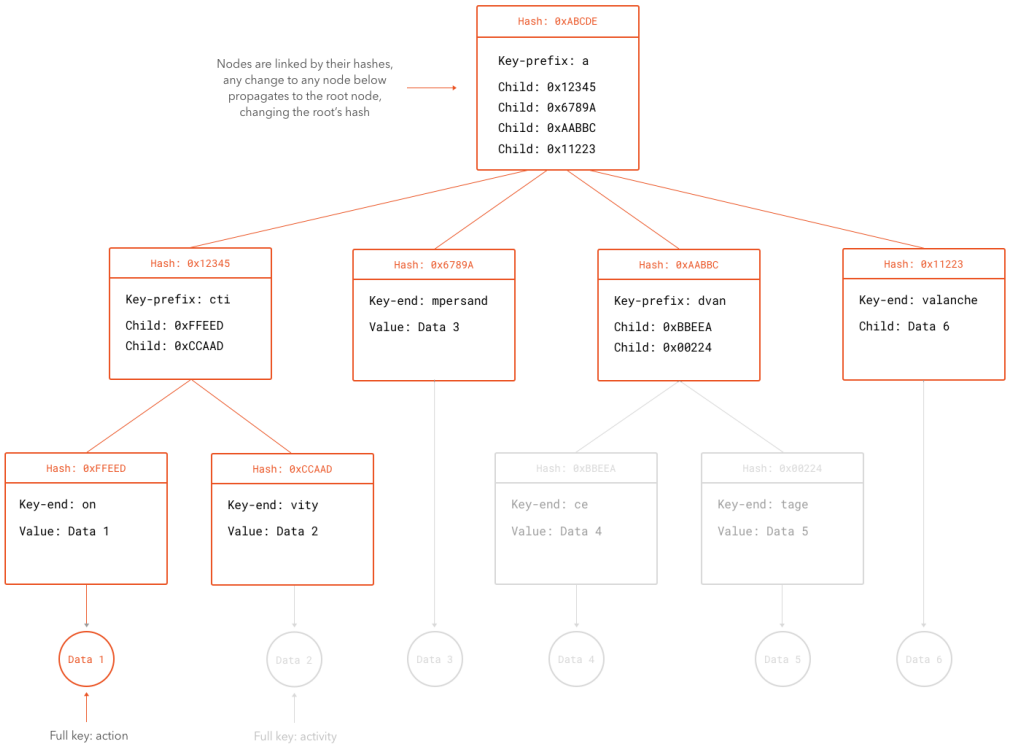

Figure 2.4: A Partial Simplified Merkle Tree

All of this allows efficient operations on the state of the blockchain, while at the same time keeping its actual (potentially huge) data separate from the block, still the center piece of the security

[12] https://en.wikipedia.org/wiki/Preimage_attack

scheme of the blockchain.

2.2.2.2 History

Much like Bitcoin, the blockchain can be used to find the state of the system at any point in time. This can be done by replaying each transaction from the very first block up to the point in question. However, in contrast to Bitcoin, most nodes do not keep a full copy of the data for every point in time. Ethereum allows for old data to be *pruned* from the blockchain. The blockchain remains consistent as long as the blocks are valid, and data is stored outside of the blocks, so technically it is not required to verify the proof-of-work chain. In contrast to Bitcoin, where to find the balance of an account a node must replay all transactions leading up to that point, Ethereum stores state by keeping the root hash of the Merkle Patricia Tree in each block. As long as the data for the last block (or any past blocks) is available, future operations can be performed in the Ethereum network. In other words, it is not necessary for the network to replay old transactions, since their result is already available. This would be akin to storing the balance of each account in each block in the Bitcoin network.

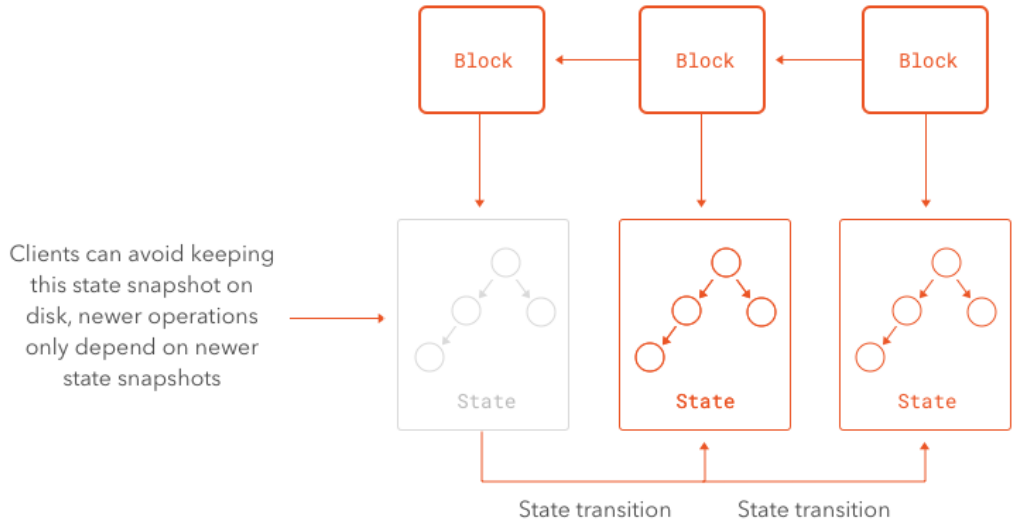

Figure 2.5: Partial historical state in the blockchain

There are, however, nodes that store the whole copy of the historical state of the blockchain. This serves for historical and development purposes.

2.2.2.3 Solidity and a Sample Smart Contract

Smart contracts run on the Ethereum Virtual Machine, which in turn runs on each node. Though powerful, the Ethereum Virtual Machine works at a level too low to be convenient to directly program (like most VMs). For this reason, several languages for writing contracts have been developed. Of these, the most popular one is Solidity[13].

Solidity is a JavaScript-like language developed specifically for writing Ethereum Smart Contracts. The Solidity compiler turns this code into Ethereum Virtual Machine bytecode, which can then be sent to the Ethereum network as a transaction to be given its own address.

To better understand Solidity, let's take a look at one example:

```
pragma solidity ^0.4.2;

contract OwnerClaims {

    string constant public defaultKey = "default";

    mapping(address => mapping(string => string)) private owners;

    function setClaim(string key, string value) {
        owners[msg.sender][key] = value;
    }

    function getClaim(address owner, string key) constant returns (string) {
        return owners[owner][key];
    }

    function setDefaultClaim(string value) {
        setClaim(defaultKey, value);
    }

    function getDefaultClaim(address owner) constant returns (string) {
        return getClaim(owner, defaultKey);
    }

}
```

This is a simple owner claims contract. An owner claims contract is a contract that lets any address owner to record arbitrary key-value data. The nature of the blockchain certifies that the owner of certain address is the only one who can set claims in connection to that address. In other words, the owner claims contract allows anyone who wants to perform transactions with one of your addresses

[13] https://solidity.readthedocs.io/en/develop/

to know your claims. For instance, you can set a claim called "email", so that anyone that wants to perform a transaction with you can get your email address. This is useful, since an Ethereum address is not bound to an identity (or email address), only to its private-key.

The contract is as simple as possible. First there is the `contract` keyword that signals the beginning of a contract. Then comes `OwnerClaims`, the contract name. Inside the contract there are two types of elements: variables and functions.

Among variables there are two types as well: constants and writable variables. Constants are just that: they can never be changed. Writable variables, however, save state in the blockchain. It is these variables that encode the state saved in the blockchain, nothing more.

Functions are pieces of code that can either read or modify state. Read-only functions are also marked as `constant` in the code and do not require `gas` to run. On the other hand, functions that mutate state require `gas`, since state transitions must be encoded in new blocks of the blockchain (and these cost work to produce).

Values returned from functions are returned to the caller.

The `owners` variable in our contract is a map[14], also known as associative array or dictionary. It matches a key to a value. In our case, the key is an `address`. Addresses in Ethereum are the identifiers of either normal accounts (usually managed by users) or other contracts. When an owner of an address decides to set a claim, it is this mapping from address to a claim that we are interested in. In fact, we are not simply mapping an address to a claim, but to a group of key-values that constitute a group of claims (in the form of another map). This is convenient because an address owner might want to make several details about himself known to others. In other words, address owners might want to make their email address and their cellphone number available. To do so, they might create two claims: one under the "email" key, and the other under the "phone" key.

The contract leaves to each owner to decide what entries to create, so the names of the keys are not known in advance. For this reason, a special "default" key is available, so any reader might know at least one claim if he doesn't know what keys are available. In truth, this key is also in place for a different reason: Solidity does not make it practical to return bulk data from functions. In other words, it is not easy to return all claims connected to an address in a single function call. In fact, the `mapping` type does not even have an iteration operation (although one can be coded if needed), so it is not possible to know what keys are inside a mapping. It is left as an exercise for the reader to find ways to improve this if needed.

2.2.3 Current and Potential Uses

What we just saw with our simple example gave us a taste of what is possible with Ethereum. Do note it has nothing to do with exchanging money! Although ether is necessary to perform mutations on the network, our contract is strictly concerned with securely establishing a series of claims connected to an Ethereum address. Nothing more. Not only the result is mathematically verifiable (no other person other than the owner of the address can set claims), but is also very hard to erase: it is recorded in a globally distributed database with no central node!

[14] https://en.wikipedia.org/wiki/Associative_array

Having access to a distributed, Turing-complete computing engine with verifiable semantics opens a world of possibilities. Let's take a look at interesting ideas already implemented or under implementation in Ethereum.

2.2.3.1 The Decentralized Autonomous Organization (DAO)

The DAO is, literally, an organization. It has members, it has a central authority (the owner), members can cast votes and the organization itself can perform any operations any other account could do. Members can create proposals, in the form of transactions, and voting members from the organization can cast votes to either approve the proposal or dismiss it. Proposals have a limit of time after which votes are counted and a decision is taken. The decision to perform or dismiss the proposal is carried by the contract of the DAO. In other words, no central authority can decide the fate of a proposal, and this is certified by the contract and the nature of the blockchain. The owner can be changed by a proposal. The only privilege the owner has is the ability to add or remove voting members.

In fact, the DAO we have just described is only one of the possible implementations. There are many improvements or modifications that can be performed to create whatever type of hierarchy. A Congress, a shareholder association, a democracy, these are all possibilities.

To learn more about DAOs, the main Ethereum website has a whole area[15] dedicated to them.

2.2.3.2 A Central Bank or Your Own Coin

Although ether has real value and can be traded for other coins, other coin systems can be implemented on top of Ethereum. For instance, you could design your own coin with a central authority that can create money, authorize transactions or arbitrate disputes. Take a look at a possible implementation by following this tutorial[16].

2.2.3.3 A Crowdfunding System

Crowdfunding lets donors send money for a project that has not been completed or even started. In this way, funding for projects of different sizes is possible. The amount of money donated for the project is what usually decides the fate of the project. The usual problem with crowdfunding is the need for a central figure to hold founders responsible in case a project is not satisfactorily completed after funding, or to make sure all the money donated actually arrives at the hands of the founders. In other words, crowdfunding requires a considerable amount of trust to be placed in both the founder of a project and the central authority. But with Ethereum this needn't be so.

With Ethereum, it is possible to design a contract that takes a certain amount of money from donors and stores it in an account. The funds in this account can be kept away from the hands of the founders until they provide proof of their progress. When a certain milestone is achieved, the funds can be released. On the other hand, if the founders fail to provide proof of their progress

[15] https://www.ethereum.org/dao
[16] https://www.ethereum.org/token

in a reasonable timeframe, donated funds can be automatically returned to the donors. All of this logic of handling funds can be performed without trust in a central authority. Donors can be sure their money won't be spent until proof-of-work is provided, and they can be sure they will always get their money back otherwise. They can also be 100% certain each donor's money will go into the right hands.

An example implementation of a crowdsale[17] is available in the Ethereum page.

2.2.3.4 Prove That You Said Something in the Past

An interesting aspect of the blockchain is that its mere existence is proof that every transaction in it happened at some point in time. Although a certain variance in the timestamp of a transaction is expected (as it will get set by the node that creates the block that contains it), anything recorded in the blockchain happened at some point in the past. In fact, it is possible to assert it happened before or after other events also recorded or linked in some way to the blockchain. Since the blockchain allows for arbitrary state to be stored in it, it is possible to link an arbitrary message to an address. Anyone can confirm by looking at the blockchain that that message was produced at some point in the past by the owner of an address. All the owner needs to do is prove he is the owner of the address that produced the same message in the past. This can simply be done by performing a transaction using the same address as before.

Suppose you wrote a book. Before sending copies to your friends and editors, you decide to prove it was you who wrote it by storing its proof of existence in the blockchain. If your book gets plagiarized before getting published (by one of the editors, for instance), you can prove it was you who wrote it by showing you linked its hash to an Ethereum address. When anyone wants to confirm you own the address, you can show it to them through any transaction of their choice. The blockchain ensures any person in doubt can see the association between the hash of the book and your address, proving you had access to the full copy of the book at some point in the past.

2.2.3.5 Proof of Existence for Digital Assets

The concept of the previous example can be extended to a proof of the existence of anything that can be hashed. In other words, anything with a single digital representation can be hashed and stored in the blockchain, just like the arbitrary message from above. Later, any user can query whether the element was hashed and added to the blockchain.

Here[18] is one working example of this concept.

There are many more examples of things that can be implemented with Ethereum, check them out[19]!

[17] https://www.ethereum.org/crowdsale
[18] https://chainy.info
[19] http://dapps.ethercasts.com

2.3 Example: A Simple Login System using Ethereum

One of the cool things about Ethereum is that addresses are, by definition, systems to prove ownership. Whomever can perform operations with an Ethereum address is the rightful owner of that address. This is, of course, the consequence of the underlying public-key infrastructure used to verify transactions. We can exploit this to create a login system based on Ethereum addresses. Let's see how.

Any login system is mainly concerned with creating a unique identity that can be managed by whomever can pass a certain "login challenge". The login challenge is the method to prove that the same entity that created the account in the first place is the same entity doing operations now. Most systems rely on the classic username + password login challenge: a new user registers by choosing a unique username and a password, then, anytime the system requires proof that the user is in fact who he says he is, it can request the password for that username. This system works. But with Ethereum we already have a system for proving identities: public and private keys!

We'll design a simple contract that can be used by any user to validate his ownership of an address. The login process will be as follows:

1. A user accesses a website that requires him or her to login. When the user is not logged in, the website requests the user to enter his or her Ethereum address.
2. The backend for the website receives the address for the user and creates a challenge string and a JWT. Both of these are sent back to the user.
3. The user sends the challenge string to the `Login` contract and stores the JWT for later use locally.
4. The backend listens for login attempts using the challenge string at the Ethereum network. When an attempt with the challenge string for the right user is seen, it can assume the user has proved his or her identity. The only person that can send a message with an Ethereum address is the holder of the private key, and the only user that knows the challenge string is the user that received the challenge through the login website.
5. The user gets notified or polls the website backend for confirmation of his or her successful login. The user then proceeds to use the JWT issued in step 2 for accessing the website. Alternatively, a new JWT can be issued after a successful login.

To that end, this is the Ethereum contract we will use:

```
pragma solidity ^0.4.2;

contract Login {

    event LoginAttempt(address sender, string challenge);

    function login(string challenge) {
        LoginAttempt(msg.sender, challenge);
    }

}
```

The contract is extremely simple. `Events` are special elements in Solidity that are mapped to a

system in Ethereum that allows special data to be logged. Events are generally watched by clients monitoring the evolution of the blockchain. This allows actions to be taken by clients when events are created. In our case, whenever a user attempts to login, an event created with the challenge is broadcast. We only care about receiving a call from the rightful owner of the Ethereum address that was passed to the third party website. And, thanks to the way Ethereum works, we can be sure the sender was the one who performed the call.

In addition to the sender's address, the challenge is also broadcast. This means anyone watching the blockchain now knows the challenge. However, this cannot be used on its own to impersonate a user: a user can only interact with the backend through the session JWT. This means an attacker must know three pieces of information to impersonate a user: the Ethereum address, the challenge AND the JWT issued with the challenge. Since JWTs are signed, an attacker cannot create a valid JWT to impersonate an user, even with access to the challenge.

What follows is our backend code. First, let's see how to watch for Ethereum events:

```
const LoginContract = require('./login_contract.js');

const loginContract = LoginContract.at(process.env.LOGIN_CONTRACT_ADDRESS ||
                      '0xf7b06365e9012592c8c136b71c7a2475c7a94d71');

// LoginAttempt is the name of the event that signals logins in the
// Login contract. This is specified in the login.sol file.
const loginAttempt = loginContract.LoginAttempt();

const challenges = {};
const successfulLogins = {};

loginAttempt.watch((error, event) => {
    if(error) {
        console.log(error);
        return;
    }

    console.log(event);

    const sender = event.args.sender.toLowerCase();

    // If the challenge sent through Ethereum matches the one we generated,
    // mark the login attempt as valid, otherwise ignore it.
    if(challenges[sender] === event.args.challenge) {
        successfulLogins[sender] = true;
    }
});
```

The `login_contract.js` file contains what is needed to inter-operate with our contract. Let's take a look:

```
// web3 is an Ethereum client library
```

```
const Web3 = require('web3');
const web3 = new Web3();

web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545'));

// This file is generated by the Solidity compiler to easily interact with
// the contract using the web3 library.
const loginAbi = require('../solidity/build/contracts/Login.json').abi;
const LoginContract = web3.eth.contract(loginAbi);

module.exports = LoginContract;
```

Web3[20] is the official client library to interact with Ethereum nodes. An Ethereum node is what actually connects to the rest of the Ethereum network. It performs "mining" (block generation), transaction operations (create and send) and block verification.

The `Login.json` file is generated by the Solidity contract compiler, part of the standard Ethereum development tools. The Solidity compiler takes Solidity source code and turns it into Ethereum Virtual Machine bytecode and an interface description file that can be used by Web3 to interact with the contract once it is uploaded to the network.

And here are our HTTP endpoints:

```
app.post('/login', (req, res) => {
    // All Ethereum addresses are 42 characters long
    if(!req.body.address || req.body.address.length !== 42) {
        res.sendStatus(400);
        return;
    }

    req.body.address = req.body.address.toLowerCase();

    const challenge = cuid();
    challenges[req.body.address] = challenge;

    const token = jwt.sign({
        address: req.body.address,
        access: 'finishLogin'
    }, secret);

    res.json({
        challenge: challenge,
        jwt: token
    });
});

app.post('/finishLogin', validateJwt, (req, res) => {
```

[20] https://github.com/ethereum/web3.js/

```
    if(!req.jwt || !req.jwt.address || req.jwt.access !== 'finishLogin') {
        res.sendStatus(400);
        return;
    }

    if(successfulLogins[req.jwt.address]) {
        delete successfulLogins[req.jwt.address];
        delete challenges[req.jwt.address];

        const token = jwt.sign({
            address: req.jwt.address,
            access: 'full'
        }, secret);

        res.json({
            jwt: token,
            address: req.jwt.address
        });
    } else {
        // HTTP Accepted (not completed)
        res.sendStatus(202);
    }
});

app.post('/apiTest', validateJwt, (req, res) => {
    if(req.jwt.access !== 'full') {
        res.sendStatus(401); //Unauthorized
        return;
    }

    res.json({
        message: 'It works!'
    });
});
```

The `/login` endpoint receives a login request carrying an Ethereum address for the user that wants to login. The user must be the owner of such Ethereum address. It generates a JWT and a challenge. The JWT can only be used to access the `/finishLogin` endpoint.

Before the user can call the `/finishLogin` endpoint he or she must prove his or her identity by making a call to the `login` method of the `Login` contract. The `login` method receives a single parameter: the challenge returned by the `/login` endpoint. He must perform this call using the same account address that was passed to the `/login` endpoint. He or she can use any Ethereum wallet or client to do this.

After making the call to the `login` method of the `Login` contract, the user can complete the login by using the `/finishLogin` endpoint. He or she must pass the JWT returned by the `/login` endpoint to it. If the login is successful, a new JWT with full access is returned. Otherwise, if the login is

still pending, an accepted HTTP status (202) is returned signalling proper verification of the login request is still pending. If the JWT passed to `/finishLogin` is invalid, an unauthorized HTTP status code is returned (401).

After the `/finishLogin` endpoint is called and the login process is completed, the returned JWT can be used to access other parts of the API. In this case, the `/apiTest` endpoint is available. It simply returns "It works!" wrapped in a JSON object if the user is logged-in.

Grab the full example.[21]

2.3.1 Running the Example

Building and deploying the example is not as straightforward as it may seem due to the nature of Ethereum and current development tools. Here are the steps we used to test the example above.

2.3.1.1 1. Get an Ethereum node client

There are several Ethereum node clients. A popular one is go-ethereum[22], a client written in Go. Download and install it.

Ethereum, as other cryptocurrencies do, has different versions of the blockchain with different parameters. There are essentially two blockchains: the main official blockchain and a test blockchain. The main blockchain never undoes operations once they are confirmed. Since some operations require money, the main blockchain is not ideal for testing. The test blockchain, on the other hand, is much less strict about forks and changes. It is also simpler to mine "Ether", Ethereum's currency.

We could use the test network for our example here. However, running a client node for any of the public networks is problematic for one reason: to be able to start doing transactions, the client must first *verify* all previous transactions in the blockchain. That means that bootstrapping a new client node takes quite a bit of time. Fortunately there is an alternative: we can create a new, pristine private Ethereum blockchain to run our tests. To do so, run `go-ethereum` using the following command line:

```
./geth --rpc --nat none --dev
```

2.3.1.2 2. Create a new Ethereum account to mine some Ether

The `geth` command can also be used to interact with a running client. Launch an interactive console connected to the running client:

```
/geth attach ipc:/var/folders/ts/7xznj_p13xb7_5th3w6yjmjm0000gn/T/ethereum_dev_mode/geth.ipc
```

The IPC file mentioned in the command can be found in the output from running the node in our first step. Look for the line that reads:

```
IPC endpoint opened: /var/folders/ts/7xznj_p13xb7_5th3w6yjmjm0000gn/T/ethereum_dev_mode/geth.ipc
```

[21] https://github.com/auth0-blog/ethereum-login-sample
[22] https://github.com/ethereum/go-ethereum

Now in the Geth console type:

```
personal.newAccount()
```

After hitting `ENTER` a prompt will appear requesting a passphrase. This is the passphrase that will be used to perform any operations using this account. You can think of this as the passphrase required to decrypt the private-key used to sign Ethereum transactions. Do not leave the prompt empty, choose a simple passphrase for testing instead. A new Ethereum address will be returned by the function. If at any point you forget this address, you can list accounts by inspecting `personal.listAccounts` (it's a variable, not a function, so don't add `()` at the end).

The `geth` console is a JavaScript interpreter.

2.3.1.3 3. Start mining some Ether

Now it's time to add some Ether to our new account. Ether is required to perform operations in the Ethereum blockchain, so it is necessary to perform this step. Ether can be gathered in two ways: by receiving it from another account or by mining it. Since this is a private network, we will need to mine it. Don't worry, the private network is by default configured to be able to mine Ether easily. Let's do it:

```
miner.setEtherbase(personal.listAccounts[0]) // Hit ENTER
miner.start() // Hit ENTER
```

Now wait a few seconds (or minutes depending on your hardware) and then confirm you have some Ether in your account:

```
eth.getBalance(personal.listAccounts[0]) // Hit ENTER
```

2.3.1.4 4. Compile and deploy our Login contract

To simplify the process of compiling and deploying contracts, we will use `truffle`. Truffle is a development framework for Ethereum, simplifying many common tasks. Install it:

```
npm install -g truffle
```

Before using truffle to deploy contracts, it is necessary to "unlock" our account in our Ethereum node client. Unlocking is the process of decrypting the private-key and holding it in memory using the passphrase used to create it. This allows any client libraries (such as Truffle) connecting to the node to make operations on behalf of the unlocked account. Go to the `geth` console and type:

```
personal.unlockAccount(personal.listAccounts[0]) // Hit ENTER
```

Now switch to the `solidity` directory of our sample application. Edit the `truffle.js` file and set your newly created address as the `from` key. Then run:

```
truffle migrate
```

The `migrate` command compiles and deploys the contracts to the Ethereum network on behalf of the account set in `truffle.js`. As a result you will get the address of the newly deployed contract. Take note of it.

2.3.1.5 5. Install an Ethereum wallet

Ethereum wallets are convenient interfaces for users to interact with the Ethereum network. Sending and receiving Ether, deploying contracts or making calls to them are all operations usually supported by wallets. Mist is the official Ethereum wallet. Download it and install it.

Once installed, we will need to tell Mist to connect to our private network rather than the public main or test networks. To do this, run Mist from the command line like so:

```
./Ethereum\ Wallet --rpc \
/var/folders/ts/7xznj_p13xb7_5th3w6yjmjm0000gn/T/ethereum_dev_mode/geth.ipc
```

The IPC file is the same file used by the `geth` console and can be gathered from the `geth` output logs.

2.3.1.6 6. Tell the Ethereum wallet of the contract

Many contracts live in the Ethereum network. Wallets need to know a contract's address and interface before being able to interact with them. Let's tell Mist about our Login contract. Go to `Contracts -> Watch Contract` (top right, then bottom left).

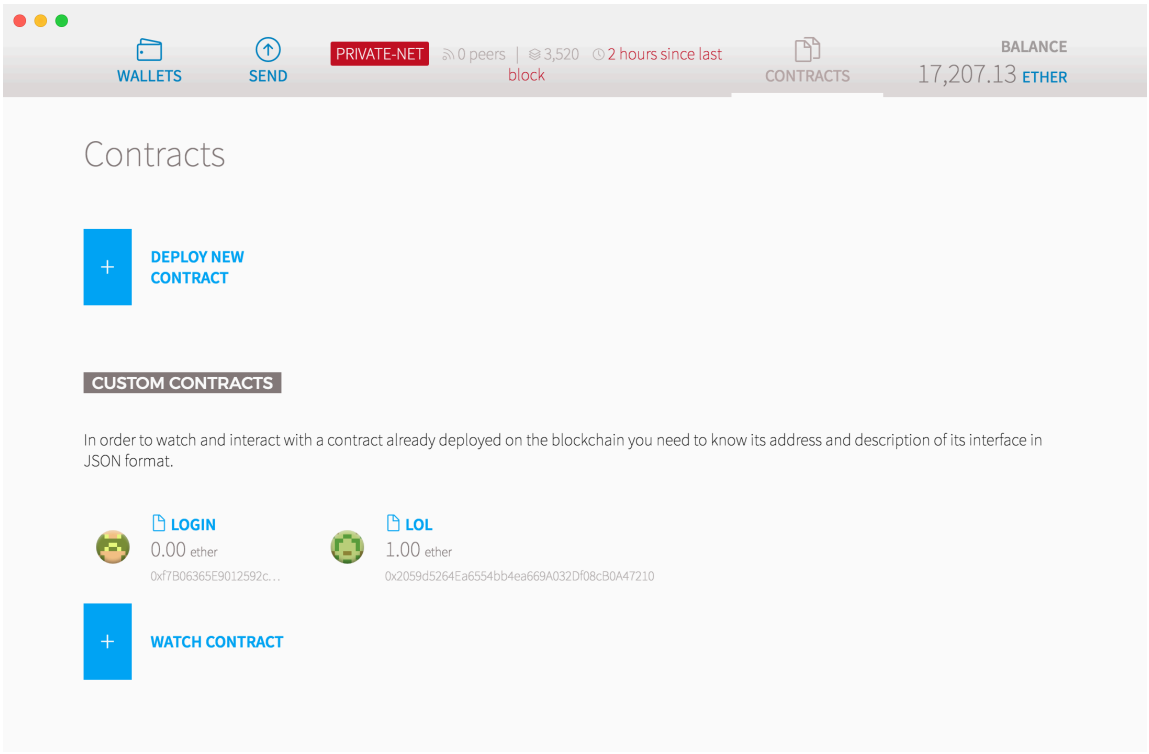

Figure 2.6: Watch contract

Complete the fields as follows:

- Name: Login
- Contract Address:
- JSON Interface: the ABI from `Login.json`. For convenience it is pasted below. Copy and paste it in Mist.

```
[{
    "constant": false,
    "inputs": [{
        "name": "challenge",
        "type": "string"
    }],
    "name": "login",
    "outputs": [],
    "payable": false,
    "type": "function"
}, {
    "anonymous": false,
```

```
    "inputs": [{
        "indexed": false,
        "name": "sender",
        "type": "address"
    }, {
        "indexed": false,
        "name": "challenge",
        "type": "string"
    }],
    "name": "LoginAttempt",
    "type": "event"
}]
```

The contract ABI (application binary interface) is a file that gets generated when compiling a contract. This file describes in which way contracts can interact with one another.

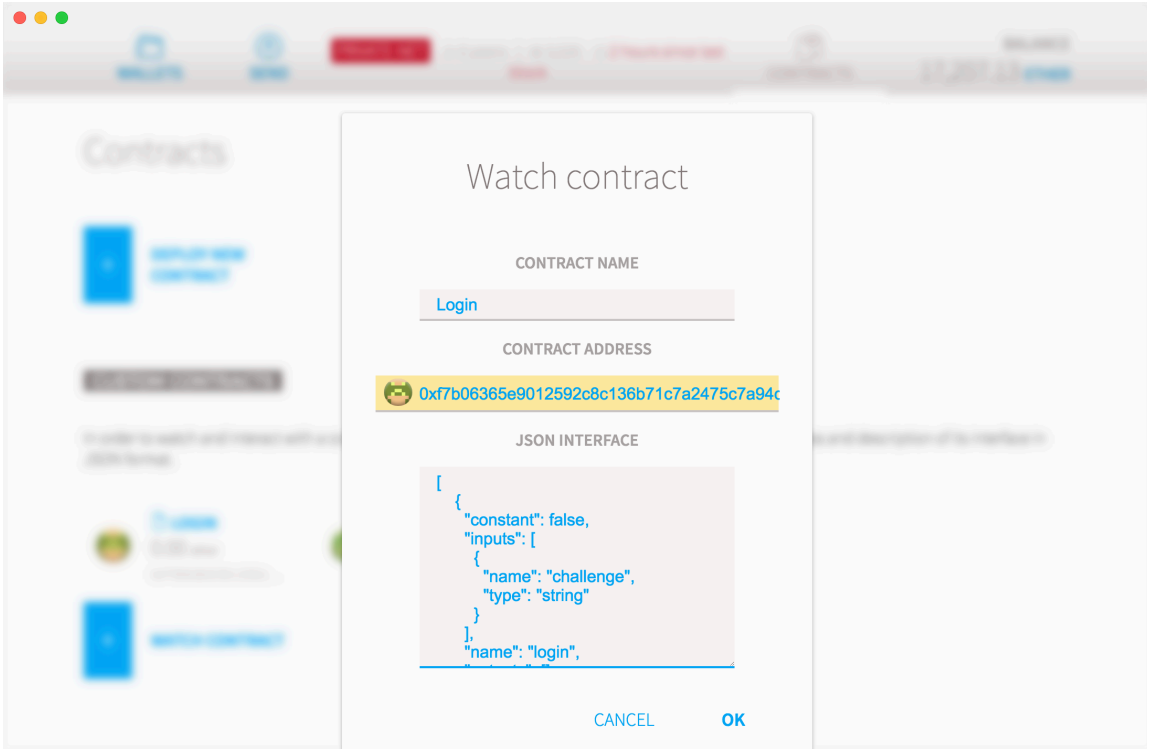

Figure 2.7: Watch contract

As a test, now try to send some Ether to the contract: `Contracts -> Login -> Transfer Ether & Tokens`. Send `1 Ether` or any other amount less than your balance. You will need to provide

the passphrase for your account.

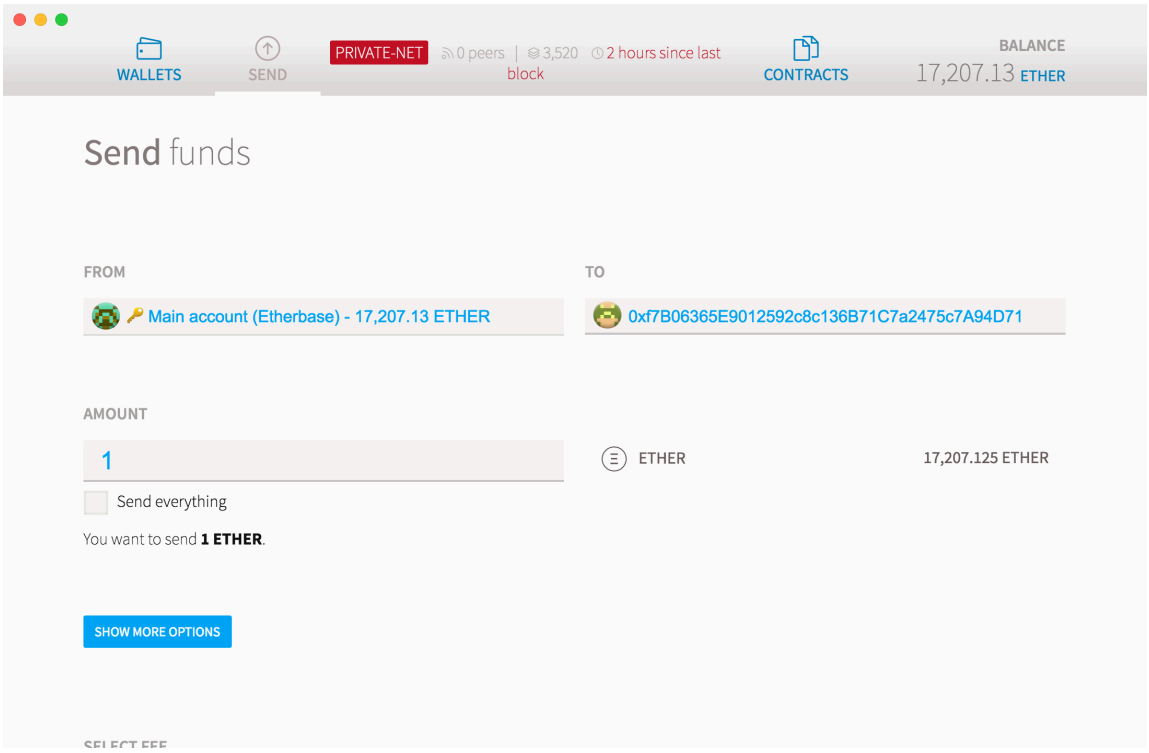

Figure 2.8: Send Ether

2.3.1.7 7. Deploy the backend

Go to the `backend` folder and run:

```
npm install
node app.js
```

2.3.1.8 8. Serve the frontend

Go to the `frontend` folder and run:

```
npm install -g static-serve
static-serve
```

You may use any other simple static HTTP server such as Python's `SimpleHTTPServer`. If you do so, make sure to serve the app in port 9080. This is important due to CORS.

2.3.1.9 9. Test everything together!

Open your browser at http://localhost:9080. Now attempt to login by putting your Ethereum address in the input field. A challenge text will be generated. Go to the Mist (Ethereum Wallet) and go to the Login contract. To the right you will see "WRITE TO CONTRACT". Select the `login` function and paste the challenge in the text fill that appears there. Then click on `Execute`. Input your passphrase and send the transaction.

Now switch back to the login page. After a few seconds the login will be completed and a welcome message will appear. Voilà!

> Watch video: "Login"[23]

This example shows how a typical Ethereum user can use his existing Ethereum account to log in to any third party website supporting Ethereum. And all of this is done without a central server. Although authentication is not performed by the owner of the website, there is no central authority validating the user: it is the Ethereum network that does so.

Grab the full example.[24]

2.4 Summary

We have taken a deeper look at Ethereum: a decentralized, blockchain-based framework for developing applications. Applications run on each node, and each state transition produced by them is validated and recorded by the blockchain. The power of the approach extends the concepts of Bitcoin to more than just monetary transactions or simple non-Turing complete contracts. The power of distributed apps is just beginning to be tapped. In the next chapter we will take a look at an actual application developed on the Ethereum network: a two-factor authentication system for Ethereum users using a mobile validator application. Stay tuned!

[23] https://cdn.auth0.com/blog/ethereum2/login.mp4
[24] https://github.com/auth0-blog/ethereum-login-sample

Chapter 3

A Practical Authentication Solution for Ethereum Users

3.1 Introduction

In the previous chapter we took a closer look at Ethereum, a decentralized, Turing-complete platform for developing applications using a blockchain. In Ethereum, applications run on each node in the network. The results of those computations are then encoded in blocks, which, through the proof-of-work system, are validated by each node in the network. Furthermore, these operations (transactions) are carried on out on behalf of users. Users must sign each transaction with their private-key, thus making it possible to track whether a certain user can perform certain operations or not. In particular, transactions have a cost, and users must be able to pay that cost by spending Ethereum's cryptocoin: **Ether**.

In the previous chapter we also had a look at practical applications of Ethereum. The Decentralized Autonomous Organization (DAO)[1], a central bank[2], a crowdfunding system[3], a proof of existence system[4], and even our own simple authentication system[5]. All of these examples run without a central authority holding any control over them. All operations are carried out by each node on the network, and these are only effective after all nodes agree on the results. This makes Ethereum particularly powerful for applications were no single entity must be able to validate or approve operations.

Our simple login system on Ethereum did work as expected, but it was less than ideal. Let's take a look at how it worked:

Watch video: "Login"[6]

[1] https://www.ethereum.org/dao
[2] https://www.ethereum.org/token
[3] https://www.ethereum.org/crowdsale
[4] https://chainy.info
[5] https://auth0.com/blog/an-introduction-to-ethereum-and-smart-contracts-part-2/
[6] https://cdn.auth0.com/blog/ethereum2/login.mp4

The objective of the system is to make it possible for any third-party to allow users to log into their website using an Ethereum address as an identifier. No username or password is required. We assume a user attempting to login with an Ethereum address is a user who currently holds an Ethereum address with some Ether (that is, a user that holds an Ethereum account for other uses). Based on these assumptions, this is how our sample system worked:

1. A user browses to a third-party website that requires login. An input text area for the user's Ethereum address is displayed.
2. The user inputs his or her Ethereum address and clicks "login".
3. The third-party backend produces a challenge string and signs a JWT with the challenge embedded in it.
4. The user sends the challenge string to the `login` method of the `Login` contract already available on Ethereum.
5. The backend watches the Ethereum network for the challenge string. It must be sent by the owner of the Ethereum address that was input in step 2.
6. If the challenge is seen by the backend within a reasonable timeframe, the user is then marked as logged in using the Ethereum address from step 2 as the identifier. A new JWT with full access to the third-party website is issued.

There are a series of problems with this approach. Namely:

- The user must manually make a call to the `login` method of the `Login` contract using an Ethereum wallet of his or her choice.
- The user must know the address and the interface of the `Login` contract beforehand.
- The user must spend some Ether to login because the contract relies on `events` that are logged to the blockchain (that is, they perform writes). This makes the contract require `gas` to run.
- The backend must wait for a new block to be mined and propagated through the network before the login is completed (minimum latency in the order of 12 seconds or more).

As you can imagine, these limitations make our simple authentication example impractical. So what can we do about them?

3.2 Towards a Practical Authentication Solution for Ethereum Users

Authentication is what we do at Auth0, so we teamed up with the guys from GFT's Innovation Team (Ivo Zieliński, Konrad Kozioł, David Belinchon and Nicolás González)[7] to think of a better way of using Ethereum for this purpose. We came up with a proof of concept which we will share with you in this chapter. First, let's describe the design goals for our system:

- It should allow users with an Ethereum address to use that address to login to a third party website (that supports this login method).
- It should be easy to use and reasonably easy to setup.
- It should not compromise the security of the user's Ethereum account.

[7] http://www.gft.com/

- It should allow users to recover their credentials in case of loss or theft.
- It should not require knowledge of contracts or manually calling contract methods.
- It should have reasonable latency for a login system (no more than a couple of seconds).
- It should not cost users gas (or money) to login.
- It should be reasonably easy for developers to implement in their apps.

One of the biggest problems with our initial approach was that writes were necessary. This forced users who wanted to log in to spend Ether to do so. Worse, they had to wait for confirmation of the transaction before the login was successful. On the other hand, this made the login process truly decentralized.

Writes were a requirement for our previous system due to the way Ethereum events work. Events are special operations in the Ethereum network that can be watched by nodes. Internally, events are Ethereum Virtual Machine ops that create data that is added to the transaction when it is mined. Events do not work on read-only (constant) Solidity functions, since they are added to a transaction when it is mined, this forces users of our first system to pay to generate a `LoginAttempt` event.

This limitation forced us to make a compromise: rather than remain entirely decentralized, we added a server to handle authentication requests. In turn, this server relies on data stored in the Ethereum blockchain. However, our system does retain the ability to allow for serverless logins. We will see how that works later on.

Another big limitation of our first system was that the user needed access to his Ethereum wallet to login. This is impractical for several reasons:

- Users usually keep a single wallet. In other words, they do not carry around their private keys to easily use them on different devices.
- If a user loses his or her Ethereum private key, he may never be able to authenticate again with that address to a third party service, not even to switch his main address or recover his information. This poses a problem for long term use of the system.
- Requiring a user to use his or her private key for each login can be a security issue for accounts holding big amounts of value. For those accounts, private keys may be stored safely and used only when necessary. Requiring their use for each login is less than ideal.

So some way of using an Ethereum address to login without requiring the private key for that address must be implemented for our new system.

3.3 A Login System for Ethereum Users

So, here is what we implemented. Our system relies on three key elements: an authentication server, a mobile application, and the Ethereum network. Here's how they play together.

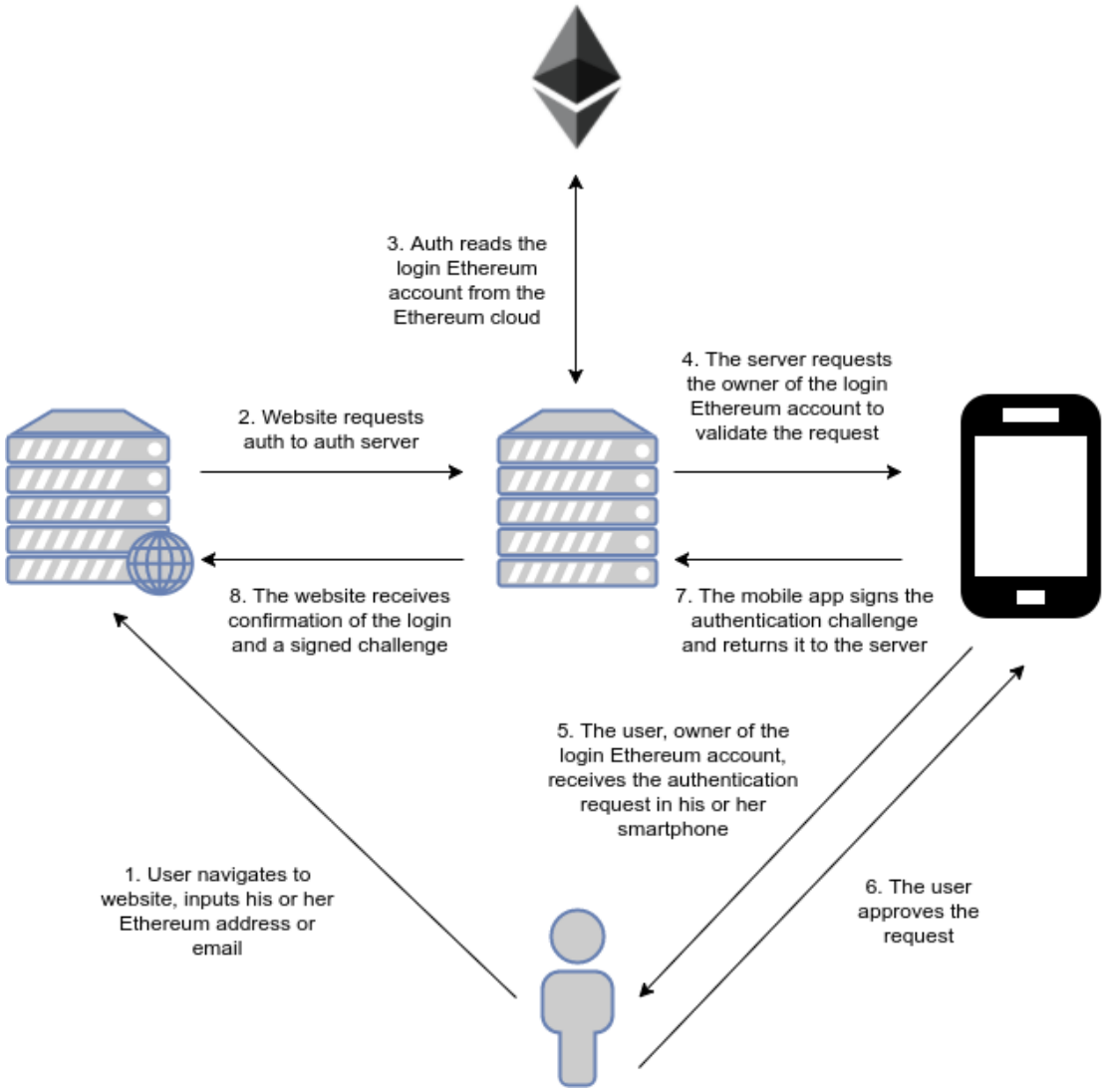

Figure 3.1: Architecture

To keep the user's Ethereum address separate from the authentication process, a different, authentication only, Ethereum address is generated by the system. This address is associated to the user's Ethereum address using an Ethereum contract. In other words, a mapping between the user's Ethereum address and the system's login-only address is established. This mapping is stored in Ethereum's blockchain with the help of a contract.

```
pragma solidity ^0.4.2;
```

```
contract Mapper {

    event AddressMapped(address primary, address secondary);
    event Error(uint code, address sender);

    mapping (address => address) public primaryToSecondary;
    mapping (address => bool) public secondaryInUse;

    modifier secondaryAddressMustBeUnique(address secondary) {
        if(secondaryInUse[secondary]) {
            Error(1, msg.sender);
            throw;
        }
        _;
    }

    function mapAddress(address secondary)
        secondaryAddressMustBeUnique(secondary) {
        // If there is no mapping, this does nothing
        secondaryInUse[primaryToSecondary[msg.sender]] = false;

        primaryToSecondary[msg.sender] = secondary;
        secondaryInUse[secondary] = true;

        AddressMapped(msg.sender, secondary);
    }
}
```

Although this contract is a bit more complex than we have seen so far, it remains fairly accessible. Let's break it down:

- There are two events: `AddressMapped` and `Error`. The `AddressMapped` event is generated any time a user's primary Ethereum address is mapped to a secondary, login-only, address. The `Error` event is only generated in case of errors, such as when a mapping using an existing secondary address already exists.
- Then two variables are declared: `primaryToSecondary` and `secondaryInUse`. `primaryToSecondary` is a map of addresses: given the primary address, it can tell the secondary address mapped to it. `secondaryInUse` is a map of addresses to booleans, used to check whether a secondary address is already in use.
- Next comes `secondaryAddressMustBeUnique`. This special function is a *modifier*. Modifiers in Solidity are special functions that can be attached to contract methods. These run before the method code and can be used to *modify* their behavior. In this case, `secondaryAddressMustBeUnique` uses the `secondaryInUse` variable to confirm whether the secondary address passed as parameter is in use. If it is, this is flagged as an error and the `Error` event is emitted. If it is not in use, then execution continues. The `_` placeholder is where the code from the modified function is logically inserted.
- And lastly there is the `mapAddress` method. This method takes a secondary address and maps

it to the address of the *sender* or *caller* of this method. The semantics of Ethereum make sure that the sender is who he says he is. In other words, only the owner of the private key for an address can make calls as the *sender* to a Solidity method. This makes sure, without any special check, that only the rightful owner of the primary address can establish a mapping between it and a secondary address used for logins. This is the crux of our system.

In summary, our contract does four things:

- It establishes a mapping between two Ethereum addresses: one high value address (the primary address) and a low value, login-only, secondary address.
- It certifies only the owner of the primary address can establish this mapping.
- It records this information publicly in the blockchain.
- It emits events to monitor and react to changes in the data stored in it.

This is all we need to make our system work. Let's go over the full registration and authentication flow to see how it all works together. We assume the user is the rightful owner of an Ethereum account with a certain amount of Ether.

3.3.1 Registration

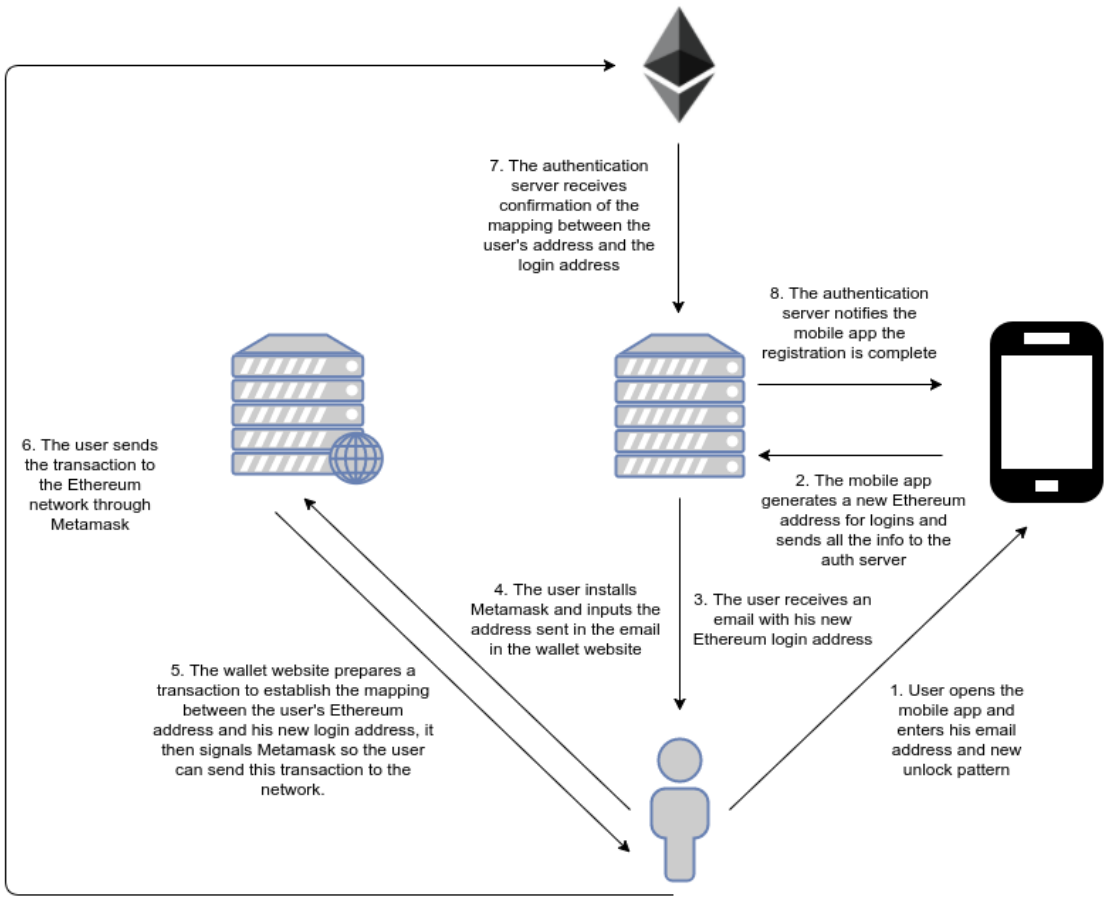

Figure 3.2: Registration

This is a one time only step to be performed the first time the user tries to use the system. Once registered, the user can use his or her Ethereum address with **any** third-party website. In other words, this is a system-wide, one time only step.

To simplify the authentication experience, our implementation uses a mobile application to authorize or deny authentication requests. A user who wants to enable his Ethereum account for use as an authentication factor first registers through the mobile application.

Registration is performed by following these steps:

1. The user opens the **mobile application**.
2. The user enters his or her email address and an *unlock pattern*.

3. A new Ethereum address is generated behind the scenes by the **mobile application**. This is the **secondary address**. This address is sent to the user to his or her email for convenience.
4. The **user** establishes a link between his or her primary Ethereum address and this secondary address. To do so the user can manually call the `mapAddress` method of the `Mapper` contract or use a special **wallet app** developed for this purpose. This step requires the user to spend a minimum amount of gas from his primary account.
5. Once the link between addresses is established, the mobile application will show a confirmation dialog. If the user confirms, the mapping is established and the process is complete.

One of the added benefits of this approach is that it makes throwaway accounts harder to use. Point 4 forces the user to spend Ether to establish the mapping between his personal Ethereum address and the login-only address. This way, third-parties can be sure that the Ethereum account used by the user is not a throwaway account (i.e. a spam account).

3.3.2 Authentication

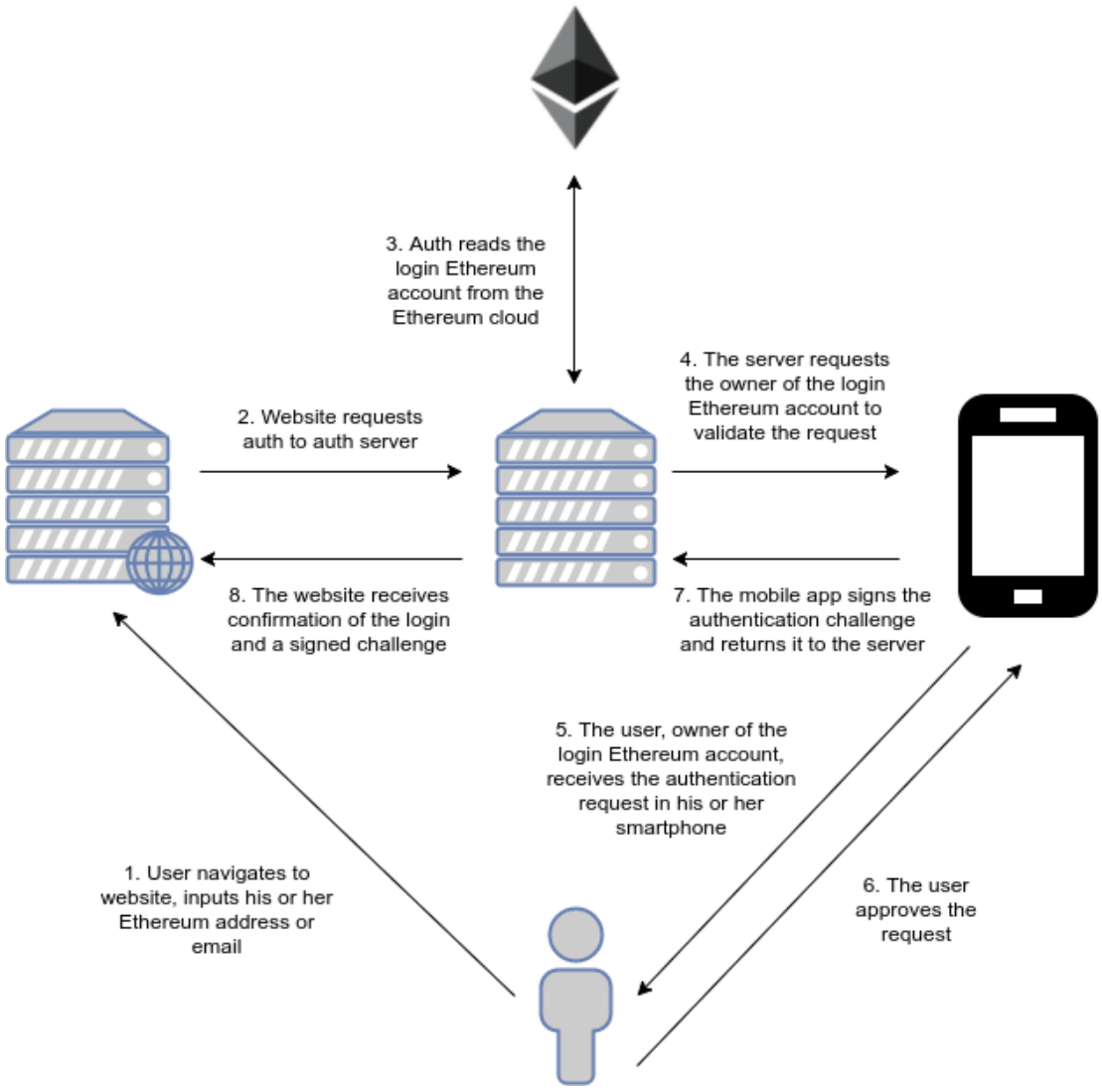

Figure 3.3: Authentication

Whenever a user who has already registered wants to use his or her Ethereum account to login to a **third party website**, he or she follows this process:

1. The user inputs his or her primary Ethereum address or his or her email in an input field and clicks “Login”.

2. The **third-party website** contacts the **authentication server** requesting authentication for that address. To do so the third-party website generates a challenge string with a specific format and passes it to the authentication server.
3. The **authentication server** checks the Ethereum network for the current secondary address of the user. It then checks the internal database for the necessary data to contact the mobile device associated to that address.
4. The user receives a **mobile push notification** to confirm or deny the login request.
5. If the user accepts, the private key of the secondary address is used to **sign the challenge**. The signed challenge is then sent back to the **authentication server**.
6. The **authentication server** verifies the signature and if it is valid and the challenge matches, it considers the login successful. It then sends back the signed challenge to the third-party website for optional independent confirmation.

That is all there is to it, really! This scheme separates the signature process from a sensitive primary address, preventing the exposure of a potentially important private key while still giving the third party site confirmation that the user is the rightful owner of that address. Furthermore, although it relies on the authentication server for convenience, it can still work without it and does not require trust to be placed in it (the third party website can check the signature itself). Thus it remains decentralized in worst-case scenarios (authentication server down) and convenient for the common case.

As an added benefit, this system can easily be adapted to work like "Login with Facebook" or "Login with Google" do. In fact, a future version could be included in Auth0!

3.3.3 Cons

As we have seen so far, our system appears to be more convenient than our initial, simple approach from part two of this series. However, it does come with a few limitations of its own. Let's take a brief look at them.

Our initial approach sported a key element from blockchain based systems: it was entirely decentralized. Our newer approach relies on an authentication server for convenience. Although it is possible for the system to work without the authentication server, it is not convenient to use it this way. This is by design and must be considered if convenient decentralized operation is mandatory in all cases. In every case, however, no trust is placed in the server.

3.4 Try it out!

Since this is just a proof-of-concept and getting your feet wet with Ethereum can be a bit hard at first, here is a step by step guide for new users to test the system. Please note that this is just a test system so it uses Ethereum's testnet. In other words, no hard guarantees are provided with regard to the integrity of the data stored in the Ethereum testnet, do not put important stuff in the accounts created in this guide, they won't be protected by the same guarantees as the Ethereum mainnet.

3.4.1 Get an Ethereum Wallet

To perform operations in Ethereum you need a wallet. A wallet is an application that allows you to interact with the rest of the network. Wallets store the private-keys for your Ethereum addresses. For simplicity we will be using Metamask[8]. Metamask is a browser-based wallet that runs locally as a Chrome extension. Keys are stored locally and transactions are signed with them. These are then sent to the rest of the network through a Metamask operated public node.

3.4.1.1 1. Get Metamask

Go to the Chrome Webstore[9] and install Metamask.

3.4.1.2 2. Create a New Account

Click on the Metamask icon on the top right corner of your Chrome windows and follow the wizard to create an account. Make sure it is created in the **Rinkeby** testnet. To check this, after creating the account, click on the icon next to the Metamask fox, on the top left corner of the Metamask window. If you are using another network, just switch to `Rinkeby` and then follow the wizard again.

Watch video: "Creating a Rinkeby Account using Metamask"[10]

3.4.1.3 3. Get Some Ether

To register you will need a minimum amount of Ether. Fortunately, this is easy to get in the testnet (in the mainnet you must either buy it or be lucky enough to be able to mine it). For the testnet it is possible to use "faucets". Faucets are places to get free Ether. The most popular Rinkeby faucet[11] requires users to create a GitHub gist[12]. This is a simple way to limit misuse of the faucet. Creating gists is easy, you only require a GitHub account. Create a public GitHub gist and paste your Metamask Rinkeby address in it. Then go back to the faucet and place the link to the gist in the required field, then click on "Give Me Ether" (the faucet is located in the `crypto faucet` section on the left bar).

After a bit, you should see your newly acquired Ether in Metamask.

To get your Rinkeby Ethereum address, go to Metamask and then click on the "copy" icon next to your account name. This will be your *primary* Ethereum address. In an actual production system, this would be the address of an account with lots of Ether in it. One that you would not want to expose every time you want to log in to some third party site using your Ethereum address.

Watch video: "Using a Rinkeby Faucet to Obtain Ether"[13]

[8] https://metamask.io/
[9] https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn
[10] https://cdn.auth0.com/blog/ethereum3/Metamask-Account-Rinkeby.mp4
[11] https://www.rinkeby.io/
[12] https://gist.github.com/
[13] https://cdn.auth0.com/blog/ethereum3/Metamask-Faucet-Rinkeby.mp4

3.4.2 Get the Mobile Authenticator App

Now it's time to set up your secondary address and login helper app. This application will be the authentication factor used to confirm your login request. Any time you want to login to some site, you will receive a notification through this app. This notification will allow you to either confirm or deny the authentication request.

3.4.2.1 1. Get the App

Go to the Android Play Store and download our Auth0 PoC app[14].

3.4.2.2 2. Register

Open the app and input your email address. Now choose an unlock pattern. You will be asked to input this same pattern any time you want to log in to a site. Then click `Register`. You will be asked to confirm the registration through the mobile app. Click `Sign` to confirm it.

The mobile app is now set, let's enable your Ethereum account for logins.

> Watch video: "Register Using the Mobile App"[15]

3.4.3 Enable Your Ethereum Address for Logins

This step, like the previous ones, is only performed once. This sets up the mapping between your primary address and the login address. In other words, it connects your Metamask account to the mobile app in your smartphone.

3.4.3.1 1. Get Your Mobile App (Secondary) Address

If you now look at your emails (please check spam, promotions, etc.) you will find your Ethereum secondary address. This is the address of the account managed through your smartphone. Just copy it to the clipboard.

[14] https://play.google.com/store/apps/details?id=block.chain.auth.zero
[15] https://cdn.auth0.com/blog/ethereum3/Mobile-Register.mp4

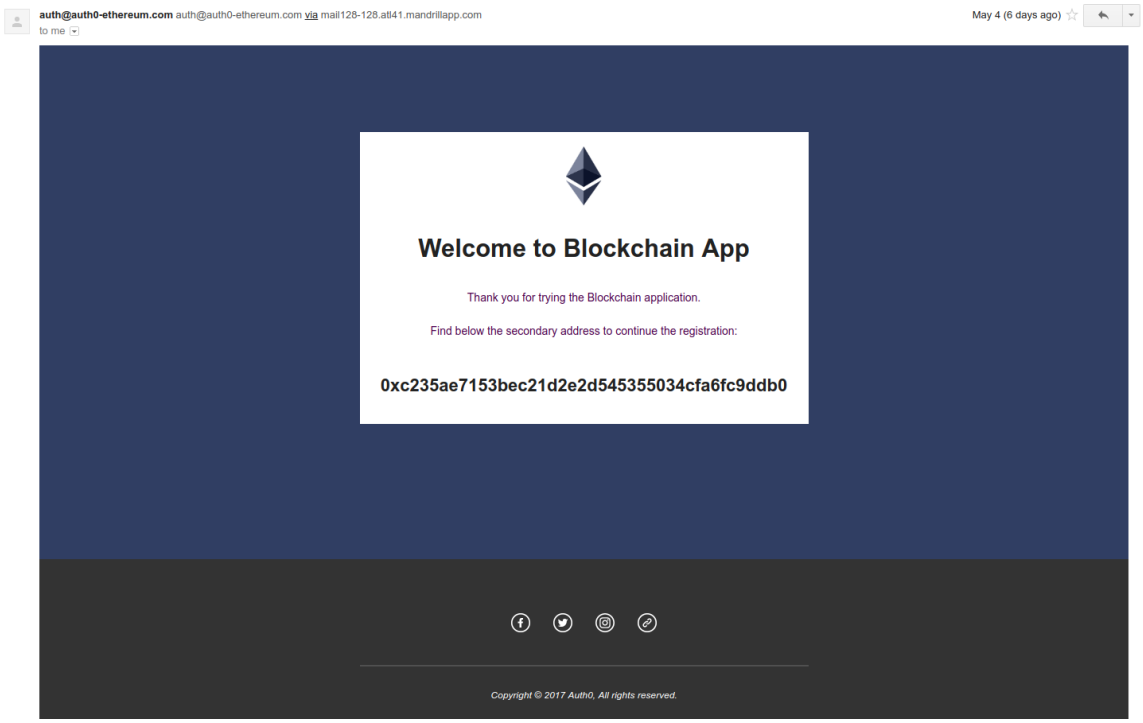

Figure 3.4: Registration email

3.4.3.2 2. Call the Contract!

If you are an Ethereum user and you have your own wallet, you can perform this step manually. For simplicity, however, we have set up a site that will do the hard work for you. Using the same Chrome instance where you installed Metamask, navigate to our PoC wallet[16]. This site is a simple local-only wallet-like app that creates the Ethereum transaction necessary to call the contract. This site communicated with Metamask so that you don't have to input your account details manually.

Once you are in the site, paste the Ethereum address you copied from the email in the previous step and click `Register`. A Metamask window will pop-up. This is a confirmation that you are about to make a transaction from your primary account that will spend Ether. Click `Sign`. After a while your primary and secondary accounts will be connected! The time for this to happen depends on the Ethereum network. In general it is just a few seconds.

In case you are already experienced with Ethereum you may want to perform this step manually. Call the `mapAddress` method of the `Mapper` contract located at `0x5e24bf433aee99227737663c0a387f02a9ed4b8a`. You can get the JSON API here[17]. The only parameter is the address you got in your email.

[16] http://auth0-ethereum.com:3002/wallet/
[17] https://github.com/auth0/ethereum-auth-client/blob/master/config/abi.json

After this is done everything is set!

Watch video: "Register Using the Wallet Webapp and Metamask"[18]

3.4.4 Login to Our Test Site

You may now login to any third party site that supports this authentication method using either your email address or your primary Ethereum address as a credential. Go to our sample website[19], put your email address and click `Login`. Watch your smartphone for notifications to approve your login.

You will notice there is a checkbox labeled `Trustless Authentication`. As explained before, third parties may opt for different levels of security. They can opt to trust the authentication server when it says the login is valid (trustful authentication) or they may opt to not trust the authentication server and validate the signature internally. In this case, the third party website must validate the signature of the secondary address itself, first be querying the secondary address using the `Mapper` contract (which is publicly available) and then by verifying the signature of the returned data using the secondary address to find the public key of the secondary address. This provides the highest level of security and uses the authentication server as simply a messenger.

Watch video: "Login"[20]

3.4.5 Explore the Code

If you are interested in taking a closer look at how our PoC works, here are all the repositories:

- The authentication server[21]
- The mobile app[22]
- The sample third party web app[23]
- The registration wallet using Metamask[24]
- Docker scripts for easy testing[25]

There are also a couple of helper libraries that were developed for this PoC, these are used by the repositories above:

- Ethereum crypto helper lib[26]
- JavaScript library for doing auth as used by this PoC[27]
- A simple database abstraction helper[28]

[18] https://cdn.auth0.com/blog/ethereum3/Wallet-Register-Rinkeby.mp4
[19] https://auth0-ethereum.com/authzero
[20] https://cdn.auth0.com/blog/ethereum3/Login.mp4
[21] https://github.com/auth0/ethereum-authentication-server
[22] https://github.com/auth0/ethereum-authenticator-app-public
[23] https://github.com/auth0/ethereum-sample-web
[24] https://github.com/auth0/ethereum-browser-wallet
[25] https://github.com/auth0/ethereum-docker-deployment
[26] https://github.com/auth0/ethereum-crypto
[27] https://github.com/auth0/ethereum-auth-client
[28] https://github.com/auth0/ethereum-user-db-service

- Preconfigured Ethereum client node for this PoC[29]

3.5 Summary

We have taken our simple authentication for Ethereum accounts concept from our previous chapter and expanded it to make it more convenient. Let's review our design goals from the beginning of this chapter:

- **It should allow users with an Ethereum address to use that address to login to a third party website (that supports this login method)**. After registration, users can login to any site implementing this protocol using their Ethereum address or email address.
- **It should be easy to use and reasonably easy to setup**. It is simpler than our previous example and simple enough for typical Ethereum users: one mobile app to install, one transaction to execute once.
- **It should not compromise the security of the user's Ethereum account**. Logins are now handled using a separate Ethereum account so the user does not need to expose his valuable Ethereum account.
- **It should allow users to recover their credentials in case of loss or theft**. In case of theft of the mobile device, the user can create a mapping to a new account for logins using his primary Ethereum address.
- **It should not require knowledge of contracts or manually calling contract methods**. The mobile wallet app and Metamask combined isolate users from interacting with contracts directly.
- **It should have reasonable latency for a login system (no more than a couple of seconds)**. Logins are only affected by network latency between the authentication server and the mobile device. In other words, they are as fast as any login system.
- **It should not cost users gas (or money) to login**. Users only spend Ether once when first setting up their account. After that, logins to any third party websites do not use gas or Ether.
- **It should be reasonably easy for developers to implement in their apps**. Developers can implement this by calling two endpoints of a RESTful API. Really simple.

Not bad for our initial research into integrating Ethereum with classic technologies. This shows Ethereum can be integrated into traditional applications today. The platform works, and the concept of decentralized applications is picking up steam.

Another interesting approach to Ethereum authentication is currently under development by uPort[30]. The landscape of blockchain based applications is still being explored.

[29] https://github.com/auth0/go-ethereum
[30] https://www.uport.me/

Many thanks to GFT's Innovation Team (Ivo Zieliński, Konrad Kozioł, David Belinchon and Nicolás González) for doing an amazing job developing this proof-of-concept, and to Manu Aráoz[31] for reviewing and providing insight for this chapter.

[31] https://twitter.com/maraoz