Offset

Freedomlayer

May 12, 2021

INTRODUCTION

Offset is a credit card powered by trust between people.

CHAPTER

ONE

WHAT IS OFFSET?

Warning: Offset is experimental technology, and not yet ready to use in production!

Offset is a credit card powered by trust between people. Payments in Offset are secure, fast and *privacy respecting*. In addition, Offset lets you pay with any currency you like.

You can pay with Offset at the local grocery store by scanning a QR code, or buy something online from the other side of the world, even from someone you have never met. As a merchant, Offset gives you the freedom to sell without having to pay the extra fees to a payment processor.

Most importantly, Offset gives **you** the control. Offset is free and open source. There is no single "Offset server", as Offset is fully decentralized. Your Offset credits fit nicely into your mobile phone, and you are the only one having control over those credits.

To use Offset you don't need a previous credit history, a bank account, id card, phone number or even an email address. You can open as many Offset cards as you like, without having to pay any fees. Your Offset cards can not be frozen, and your transactions can not be tracked.

Offset moves the power of issuing money *from the state to the market participants*. Therefore, Offset credits are *protected from inflation*. While the currencies of the world are constantly depreciating in value, Offset credits hold to their original value.

CHAPTER

TWO

OFFSET APP

View the full video tutorial at peertube.

2.1 Setting up Offset

2.2 Payment using Offset

CHAPTER

THREE

ECONOMIC IDEA

In traditional monetary systems, money is issued by the state. Participants of the market can then use the money issued by the state to buy and sell goods and services in the market.

Offset moves the power of issuing money from the state to the market participants.

3.1 Trading without money

To understand how Offset works, we begin with an example. Consider two local merchants, Charli and Bob, who work on the same street. Charli makes chocolate bars and sells them. Bob bakes bread. Living in the same community, Charli often buys bread from Bob's store, and Bob often buys chocolate from Charli's store.

To clarify our understanding about the trading process, let's *imagine that Bob and Charli have no money at all*. Now, given that Charli has something that Bob wants and vice versa, how can they perform the exchange without money?

3.1.1 Traditional Loans

In the traditional money system, all money is created through borrowing from banks. Hence, a possible solution for Bob's and Charli's trade problem is to take a loan. Charli could go to a bank, ask for a loan[1], and then use that money to buy some bread from Bob. At his moment, new money has entered the system. Bob can later use that money to buy chocolate bars from Charli. If Charli could sell enough chocolate bars, she will be able eventually to repay the loan to the bank[2].

The loan solution has some major disadvantages for Charli and Bob: When Charli took the loan, she promised to the bank to return more money at a later time, usually in the form of interest rate. As a result, Charli will have to increase her chocolate bar prices to be able to return the loan she took from the bank. Bob will also have to increase his bread prices if he wants to keep buying the same amount of chocolate bars from Charli.

In other words, the exchange between Bob and Charli becomes less efficient, as some of the value is leaking from the mutual relationship with an external entity: the bank.

[1] Considering a closed system, including only Charli and Bob. If Charli was able to repay the loan (with interest) then it definitely means Bob has less than 0 "money", which means he is bankrupt! How can this be? In our modern economy, more and more money is created all the time. This is a strategy called inflation.

[2] Or borrowing credit from a credit card company.

3.1.2 Barter

What if Bob and Charli lived in a place without money? Could they still manage to exchange without an external entity that creates money?

Charli could ask Bob to directly exchange one bar of chocolate from her store for two loaves of bread from Bob's bakery[3]. If Bob agrees, the a deal can be made. This sort of direct exchange is called barter.

Barter allows to trade in a moneyless world, but by itself it is not always a convenient solution.

First, the value of goods do not always behave nicely as in the example above. What if one bar of Charli's chocolate has the same value of one and a half Bob's loaves of bread? To resolve this issue, Charli might trade two chocolate bars for three loaves of bread, but this is inefficient.

Second, barter trade requires that each side wants something that the other side wants, at about the same time. What if Charli wants to buy bread today, but Bob doesn't want to buy chocolate bars today? Exchange will not be possible.

3.1.3 Mutual credit

To release themselves from the strictness of the barter system, Bob and Charli can come up with the following system: they maintain a single paper that contains the current mutual balance between Bob and Charli.

At first, the balance will be zero:

Now when Bob wants to buy a chocolate bar from Charli's store, Charli hands Bob a chocolate bar, and they both agree to move the balance to 2USD[4]. **At that moment, new money was created.** What this balance means is that Bob now owes Charli 2USD, and Charli can use those 2USD anytime she wants to buy bread at Bob's bakery. We call this idea Mutual credit.

Mutual credit allows Bob and Charli to split the transaction into two separate parts that could happen in a separate place and time, compared to barter which requires everything to happen at once.

There are a few points to consider here. First, the creation of money: Initially Bob and Charli both had no money, but Bob was able to create new money during the purchase of the chocolate bar from Charli. Bob didn't need any help from external entities to create the money.

If in the future Charli ever arrives at Bob store and buys two loaves of bread for exactly 2USD, the balance between Bob and Charli will again be restored to 0. **At that moment, money was destroyed.**

[3] Charli happens to own one of the only chocolate stores in town, and therefore she can price her chocolate bars higher than what Bob can price his loaves of bread.

[4] In fact, Bob and Charli could decide upon any currency that fits them, or even invent a new currency. USD was chosen here because of the assumption most readers are familiar with it.

3.2 Credit limits

One thing we haven't talked about yet, is what could go wrong.

Consider the mutual credit line between Bob (the baker) and Charli (the chocolate artist). How can Bob be sure that Charli doesn't one day buy lots of bread from his bakery, runs away with the bread and never comes back? Bob will be left with a piece of paper saying that Charli owes him money, but this piece of paper does not grant Bob any buying power.

To handle this issue, we introduce credit limits. For example, Bob will declare ahead of time that Charli can owe him a maximum of 200USD, and Charli will declare that Bob can owe her a maximum of 150USD[5]. Therefore, if Charli will keep on buying bread without selling chocolate, eventually Bob will not be willing to provide any more bread.

Fig. 1: Above: Charli set up a 150USD credit limit. This means 150USD is the maximum amount of value Charli can accumulate through this relationship. Bob set up a credit limit of 200USD. Note that only Charli has control over the right credit limit, and only Bob has control over the left credit limit.

Charli might still decide to take as much bread as it can and disappear one day, but with credit limits, Bob knows that the maximum amount he may lose from the relationship with Charli is 200USD.

By disappearing and not respecting her credit line with Bob, Charli has lost the mutual relationship with Bob. The overall cost of the lost relationship probably costs more than the money gained by Charli.

3.3 Offset: Mutual credit at scale

Earlier we described how two merchants can manage a mutual credit line. Using a piece of paper to write the mutual balance can be a reasonable solution for a market of two people, but it could get difficult quickly in a market with multiple participants. **Offset was designed to allow using mutual credit at scale.**

Continuing our story, consider an extra market participant, called Daniel. Daniel is a mechanic, repairing cars for a living. Daniel lives in the same community as Bob (the baker), and Charli (the chocolate artist). Daniel wants to be able to buy from Bob and Charli, and also provide his car repair services.

Bob and Charli manage a mutual credit line between each other. Let us assume that Daniel and Charli are good friends, and they also set up a mutual credit line. This is the resulting graph of relationships:

In the figure above, we denote B: Bob, C: Charli and D: Daniel. We assume that initially the balance between Bob and Charli is 0, and that the balance between Charli and Daniel is also 0. We also assume that both Bob and Charli, and Charli and Daniel have set up credit limits.

In Offset we denote the relationship between Bob and Charli, or between Charli and Daniel, as **friendship**.

We have already seen how Bob and Charli can trade, and in the same way Charli and Daniel can trade. What is new about this configuration is the discovery that Bob and Daniel can also trade, although they do not have a direct mutual credit line between each other.

Assume that Bob arrives at Daniel's garage to repair his car, and the repair cost 100USD. Bob can push the credits all the way to Daniel through Charli: 1. Bob owes 100USD to Charli 2. Charli owes 100USD to Daniel

resulting state will look like this:

[5] The credit limits don't have to be equal! In some cases it might be possible that one party trusts the other party more than the other way around. It might also be true that certain businesses might have different turnover, and therefore might need different amount of credit to operate.

Note that the total balance of Charli (-100USD + 100USD = 0USD) hasn't changed as a result of the transaction between Bob and Daniel. Bob's total balance decreased by 100USD, and Daniel's total balance increased by 100USD.

As a market gets larger, routes of mutual credit lines between people might get longer and more dynamic, hence more difficult to discover. In addition, it might become more difficult to ensure a transaction performed along a long route is not stalled, or fails due to lack of synchronization. Offset is a technology that solves those issues, allowing automatic discovery of routes and synchronization guarantees for payments.

3.4 Fees

Earlier we described how Bob can buy from Daniel, through a route along Charli.

With Offset, Charli's phone (or computer) will mediate the transaction automatically, without any human intervention. In some cases Charli might decide to collect fees for mediating the transaction. This could be to mitigate risk, or for example, due to the expenses of running an Offset card in the cloud. The default value for fees in Offset is `0`.

Offset allows setting up fee in the form of `a% + b`, where `a` is the amount of percents taken from the transaction, and `b` is a constant amount. For example, `0.5% + 0.01` that for a `100USD` transaction sent from Bob to Daniel, Bob will have to pay Charli an extra of `0.5 + 0.01 = 0.51USD`.

Offset's algorithm for discovering routes for payment generally prefers routes with lower fees over routes with higher fees. This allows open competition for fees.

CHAPTER

FOUR

PROTECTION FROM INFLATION

Offset moves the power of issuing money from the state to the market participants. Therefore, Offset credits are protected from inflation.

4.1 What is inflation?

Imagine living in 1970, having 1000 US dollars in your bank account. Now imagine you fell asleep for 50 years, waking up in the year 2020.

After the first moments of cultural shock, witnessing all the new roads, cars, skyscrapers and mobile phones, at some point you will (unhappily) notice that your 1000 US dollars you have in the bank have became much less valuable than they used to be. An item that costs 10 US dollars in 2020 probably costed only 1.5 US dollars back in 1970!

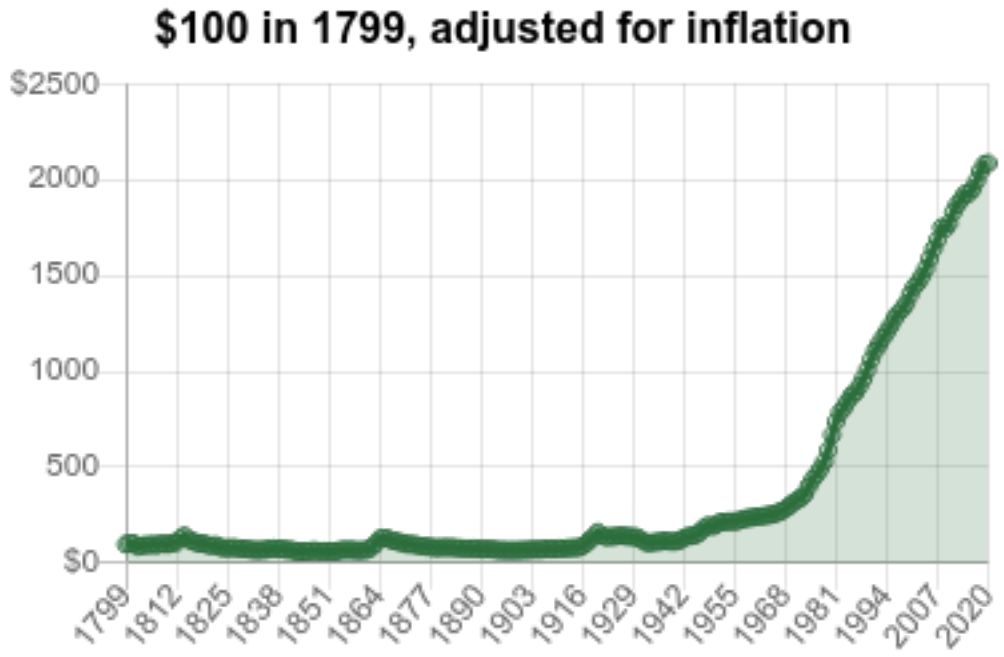

Fig. 1: Changes in the price of an item that costs $100 since the year 1799. The price arrives at $2084.46 in the year 2020. Figure taken from www.officialdata.org

How can the value of the currency be measured? One way to do it is to track, over a long period of time, how much it costs to buy an average basket of goods, and see how it changes over time. We deduce that money loses its value over time, if the price of goods generally increases over time.

Money depreciation is not special to US dollars. People in most countries of the world experience a similar phenomenon. Money depreciates in value as time passes. This phenomenon is called inflation.

4.2 Control over money supply

Governments in modern countries usually have control over the money supply[1]. This means that governments can create new money and insert it into the market. Inflation is usually caused by the state, inserting money into the market faster than the market produces new goods. In small doses, inflation is considered to be a healthy enabler of long term market growth.

By handing the government the power of creating money, we trust our governments to use this power wisely, helping create healthy economic markets. Unfortunately, along the history, governments again and again abused their power of controlling the money supply for short term, and sometimes corrupt motives.

Assume for a moment that you were the king of a country, wanting to wage war over some other country. Wars are very expensive, and you find that you do not have enough money to fund the war. What could you do to get more money?

One way would be to ask for higher taxes from your citizens, but then the citizens of the country could protest against the high taxes. In fact, finding out about the high costs of the war, some citizens might prefer to not start a war at all.

A much easier way to collect the money is to use your control over the money supply, and just print it. You could print many new bills, and use those bills to pay for the war[2]. This is a stealthy way to tax all the citizens of the country.

At first the citizens will not notice that they were taxed. As the new money bills enter the market (faster than the market grows), the prices of all the goods in the market will start rising, causing inflation. By the time inflation hits enough time have passed. The citizens of the country might not be able to connect between the money printing and the inflation.

Except for wars, printing new money could also be used as a band aid for overspending of the government, or for dealing with unexpected disasters.

4.3 Dangers of inflation

High inflation rates distort the value of money and damage the ability to trade.

Let's look at an example. Consider two merchants, Bob and Charli, living in the same community. Bob owns a bakery, and Charli is a chocolate artist, selling her famous chocolate bars.

Suppose that one loaf of bread at Bob's bakery costs 1USD, and one chocolate bar at Charli's shop costs 2USD.

As part of party preparations, Bob buys 50 units of Charli's chocolate bars, 2USD each, for a total of 100USD. Charli now has extra 100USD, which she could spend to buy anything. For example: Charli could use those 100USD to buy 100 loafs of bread from Bob's bakery. Charli keeps those extra 100USD to be used later.

One week later, for reasons absolutely unrelated to bread and chocolate bars, the government of Bob and Charli's country experienced economical difficulty. This could be due to over spending of the government during the last year, changes in the prices of oil around the world, a fast spreading epidemic or a war with a neighboring country.

In attempt to rescue itself from the economic difficulties, the government decides to create new money, and use it to pay its debts. As a result, new money enters the market. Suppose for the sake of simplicity that the government has created money worth 100% of all the previous money circulating in the market. In other words, the amount of money circulating in the market has doubled.

[1] The control over the money supply sometimes take other forms than the direct ability to print new money.
[2] When asked, you could delighfully reply that "by printing new money you make the country richer".

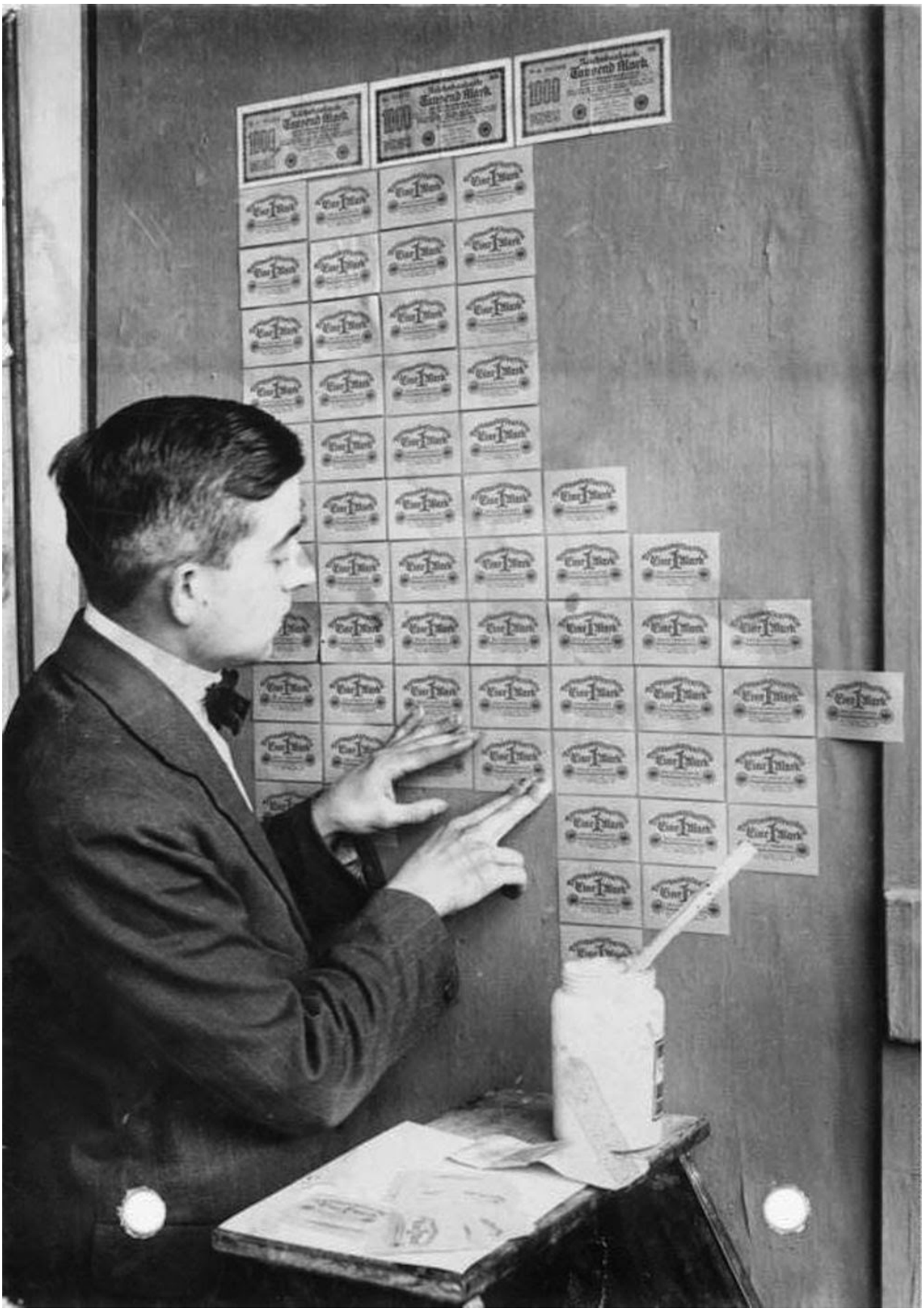

Fig. 2: A person using bills as wallpaper during German hyperinflation in 1923, after the first world war. Taken from: rarehistoricalphotos.com

The total amount of goods in the market hasn't changed, but the amount of money cirulating in the market has doubled. This means that every new USD has half the value of an old USD.

We now go back to Bob and Charli. As the money changed its value, but the value of one loaf of bread, or of one chocolate bar hasn't changed, Bob and Charli decide to double the price of everything in the store. So now Bob sells one loaf of bread for 2USD, and Charli sells one chocolate bar for 4USD.

Remember that Charli saved an extra 100USD? Before the government created new money, Charli could use those 100USD to buy 100 loafs of bread. But now, as the price of bread at Bob's bakery doubled from 1USD to 2USD, Charli can use the same 100USD to buy only 50 loafs of bread!

In the short term, the government bailed itself from debt, but Charli's business was severely impacted.

By creating new money, the government has distorted the value of money. If such events happen very often in Bob's and Charli's country, people might conclude that holding to money is very risky. Lending money also becomes risky, as the value of the currency might change from the time of issuing the loan to the time of when the loan is repaid.

4.4 Protection from inflation

Money created using Offset is protected from inflation.

Distortion of value of money happens when the amount of money circulating in the market changes disproportionally to the amount of goods produced in the market. If the amount of money in the market doubles, and the amount of goods in the market also doubles at the same time, the value of money should not change. Problems arise when new money enters the market, but the amount of goods created in the market do not change.

In traditional money systems, only the state can create money. However, the state does not offer goods or services in return for the money created. In other words, the state is not an active participant in the market. As a result, new money can enter the market, without extra goods to back it up.

Offset moves the power of issuing money from the state to the market participants. In Offset, new money is created when Charli (the chocolate artisan) buys the cocoa powder and milk required to create chocolate bars, and money is destroyed when Charli finally sells her chocolate bars.

Consider the example discussed in the previous section: Money printed by the state distorts Charli's savings. Let's review this example, and see what happens if Bob and Charli used Offset to maintain their economic relationship.

1. Bob and Charli create Offset friendship, having an initial balance of 0. Bob and Charli use a currency they call OUSD (Offset USD). This is a currency that has the value of one USD at the time the relationship between Bob and Charli was established. Bob sells one loaf of bread for 1 OUSD or 1 USD, and Charli sells one chocolate bar for 2 OUSD or 2 USD.
2. Bob buys 50 of Charli's chocolate bars, for a total of 100 OUSD. The balance between Bob and Charli is now -100 OUSD. (Bob owes Charli 100 OUSD).
3. Bob's and Charli's government prints 100% more USD, which makes one USD twice less valuable. This means now 1 OUSD = 2 USD. Bob now sells 1 loaf of bread for 1 OUSD or 2 USD, and Charli sells one chocolate bar for 2 OUSD or 4 USD.
4. Although the value of the USD has decreased, Charli can still use her mutual Offset balance with Bob of +100 OUSD to buy 100 loafs of bread.

If the whole community where Bob and Charli live used Offset, they will all be protected from the effect of inflation created by money printed by the state.

4.5 Further reading

- The New Approach to Freedom (E.C. Riegel)
- Flight from Inflation (E.C. Riegel)

CHAPTER

FIVE

OFFSET VS BLOCKCHAIN

The recent rise in popularity of digital currencies might make it difficult to understand what makes Offset different in its approach to money and payments. We present here the differences between Offset and a blockchain based digital currency.

5.1 Summary

Table 1: Offset - Blockchain[1] comparison

	Blockchain	Offset
Global consensus	Yes	No
Security foundation	Proof of work (Mining)	Trust between people
Origin of money	Created through mining	Created and destroyed by users. Automatically adjusts to market size.
Incentives	Rewards first adopters	Early and late adopters have the same money creation power
Efficiency	Transactions are very expensive	Transactions are cheap
Transaction speed	A few minutes, up to a few hours	Less than a few seconds
Transaction certainty	Certainty increases as time progresses, but never reaches 100%	100% certainty when completed
Storage	Blockchain size increases at a rate of a few GBs every month	Small constant size (A few KBs)
Fees	High fees due to mining difficulty	Low fees
Receive payments when offline	Yes	No

5.2 Global consensus

Global consensus is a core idea powering blockchain based digital currencies: All the participants in a blockchain network try to maintain together a single view of the current state of balances for all users. Blockchain based currencies usually use proof of work as a technology to achieve consensus: the ledger that took the most effort to create is chosen as the global truth.

Contrast with blockchain currencies, Offset does not attempt to achieve global consensus. Instead, every Offset user maintains synchronized balances with a few selected Offset friends. In other words, each Offset user has a local view

[1] There are many blockchain based digital currencies, therefore the comparison might fail to generalize over all of them. When in doubt, the comparison refers to the characteristics of Bitcoin.

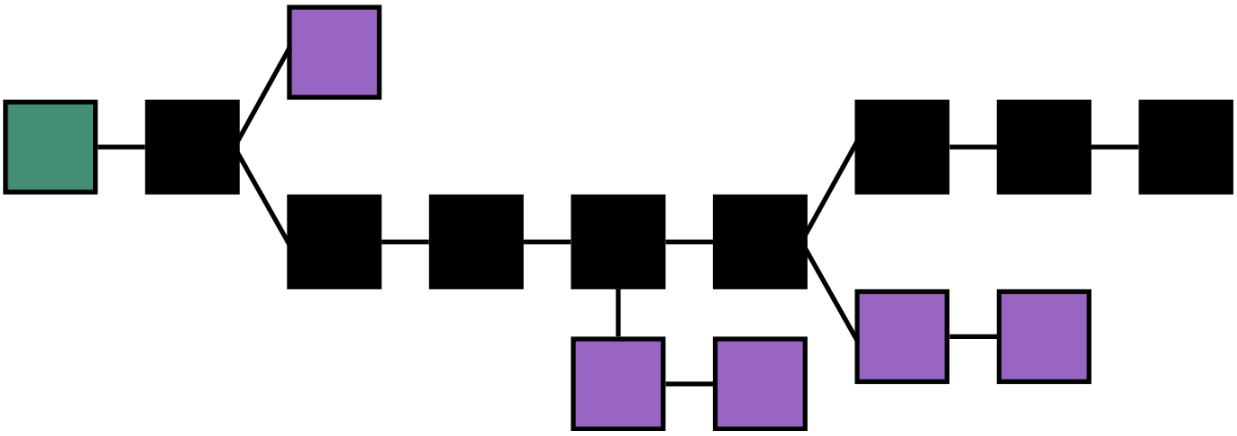

Fig. 1: Longest chain rule: The black chain is chosen to be the new agreed upon state, because it took the most effort to create. (source)

of his own balances, and not a global view of the balances of all Offset users. **It turns out that secure payments are possible even without a global consensus system!**

Fig. 2: The figure shows how Offset keeps balances in a decentralized manner. Blue dots are Offset nodes, brackets represent credit limits, and green arrows represent the current balance between a pair of nodes. Every Offset node only has to maintain balance information with nodes he has direct relation to.

Offset's approach makes it much more efficient than its blockchain counterparts. The energy footprint for every transaction is small, transactions are faster, and the storage required for every user is of a very small constant size.

5.3 Security foundation

Decentralized network can be subverted when populated by large amounts of identities all belonging to a single malicious adversary. This kind of attack is called a Sybil attack. We compare here the mitigations used in blockchain systems and in Offset against sybil attacks.

Blockchain systems use proof of work as a safeguard against Sybil attacks. This idea can be simply described as: "one processor, one vote". **blockchain networks rely on the fact that computation power is rare.**

Therefore an adversary has to gain meaningful computation power before he can obtain influence over a blockchain network. In blockchain based network, having large computation power can provide an adversary with the ability to double spend money.

Offset does not make use of Proof of work. Instead, Offset uses trust between people as a safeguard against Sybil attacks. In order to use Offset, a user has to set up mutual credit lines with a few Offset friends. Friends should be chosen carefully! Friends will usually be people the user has real world familiarity with, or possibly a trusted local hub.

For every Offset friend, the user sets up a credit limit. The credit limit is the maximum amount of money the friend might owe the user. It is also the maximum amount that the user will lose in case the relationship with this friend is lost. Hence, **Offset relies on the fact that real life relationships are rare**. An Offset user can spend money from his mutual credit relationships and disappear, but it will cost him relationships that might be more valuable than the money he spent.

5.4 Origin of money

5.4.1 Money creation in blockchains

Blockchain systems have a mining mechanism for the creation of new money. Mining is a computationaly expensive process that fills multiple roles:

- Inserting new money (Miners are rewarded with the newly created money)
- Maintaining the blockchain consensus.

Blockchains are usually designed such that mining is initially more rewarding to miners, and as time goes by it becomes less and less profitable. For example, in Bitcoin, mining is designed to become 50% less profitable every 210000 blocks, and the total amount of Bitcoins ever created is limited to about 21 million.

This property of blockchains makes it more appealing for people to join early, with the hope of becoming rich as more users join the network.

5.4.2 Money creation in Offset

Money in Offset is created and destroyed by users. Offset is designed so that the money supply changes to match the market. As the market expands, the money supply increases. When the market shrinks, money is destroyed. Therefore, **You will not become rich by joining Offset early**.

The total sum of balances in Offset is always zero. Consider two Offset friends: Bob and Charli. If Bob's balance with respect to Charli is `x`, then Charli's balance with respect to Bob is `-x`. The sum of those two balances is always `0`.

We count the amount of money in an Offset network by summing all the positive balances. For example purposes, consider again the two Offset friends: Bob and Charli. Suppose that initially the balance between Bob and Charli is `0`.

Next, assume that Bob buys a chocolate bar from Charli for the price of $2. Now the balance between Bob and Charli is -$2 from Bob's point of view, and +$2 from Charli's point of view. In the moment of purchase, new money was created by Bob. In this case we can say that the total amount of money in the market is $2.

The money created by Bob's purchase will be destroyed when a complete buying cycle is complete: For example, Charli will use the newly created money to buy something from Dan, which will use the money to buy something from Eve, which will eventually buy services from Bob. When Eve buys from Bob, the money is destroyed.

Fig. 3: The figure shows the full cycle of money creation and destruction in Offset. Bob created new money when he wanted to buy something but didn't have any money. The money created by Bob was destroyed when Eve finally used that same money to buy something from Bob.

5.5 Incentives

Most blockchain based digital currencies reward first adopters: New money is easier to create in the beginning. Therefore people want to join early, in the hope of becoming rich when late users join the network.

In Bitcoin, for example, mining is designed to become 50% less profitable every 210000 blocks, and the total amount of Bitcoins ever created is limited to about 21 million.

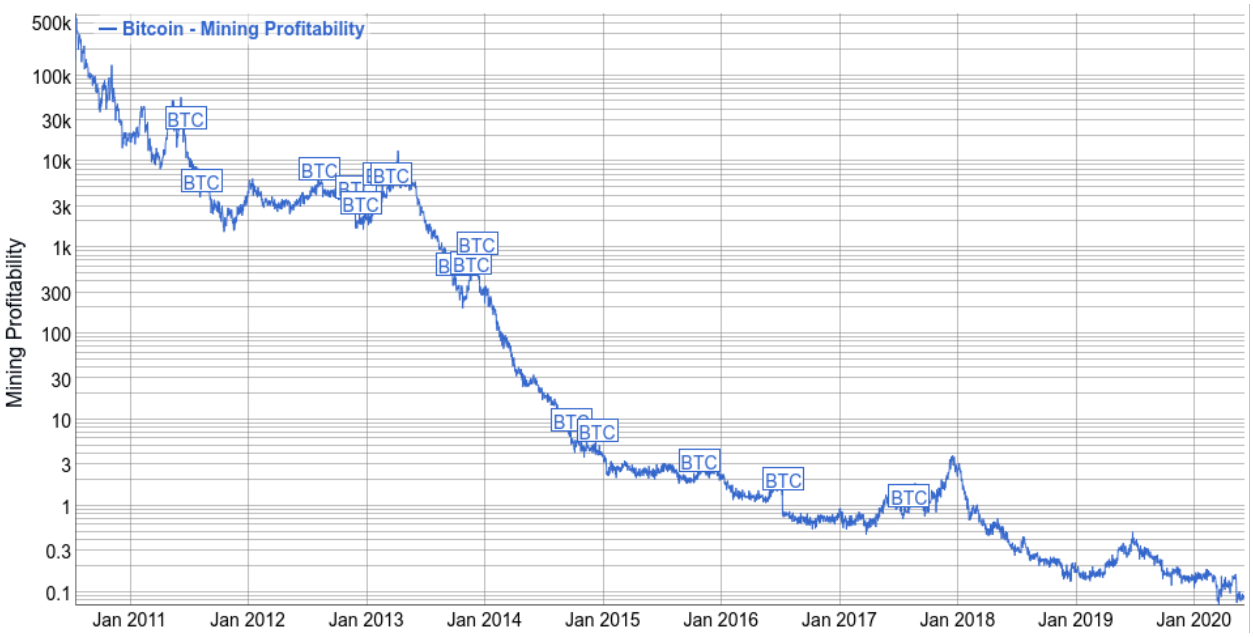

Fig. 4: Bitcoin mining profitability historical chart, USD/day for 1 THash/s, shown on a logarithmic scale. The first adopters in 2011 could 500k US dollars per day with computation power of 1THash/s. Miners in 2020 can make less than half a dollar with the same computation power. Chart Taken from from bitinfocharts.

Contrast with blockchain based currencies, **you will not become rich by joining Offset early**. Early and late Offset users have the same money creation power.

The money supply in Offset matches the size of the market, and so Offset currencies stick to their original value. Unlike blockchain based currencies, there is no point in speculating or gambling on the future value of Offset currencies.

Offset offers new users interest free credit, based on trust. A new user can start using Offset by establishing Offset friendship with another Offset user. New users do not need to spend any (traditional) money to start playing the Offset "game".

5.6 Efficiency

Performing transactions in a blockchain based currency is very expensive. Consider a new transaction being issued to a blockchain network. In the typical blockchain currency, the transaction is sent to all the network participants. Each participant has to verify the transaction.

The network participants then have to perform expensive proof of work to maintain consensus over the current state of the blockchain. Taking Bitcoin for example:

> "The digital currency consumes 511 kilowatt hours of electricity for one coin to change hands, according to research by digiconomist. That is equivalent to 330,000 Visa transactions, making it the most energy-intensive form of electronic trading known today" (source)

Finally, every participant in a blockchain network has to remember the full blockchain (Or large part of it) in order to verify future transactions. This means that every new transaction will have to be stored on the machines of all the network participants forever. In Bitcoin for example, the size of the blockchain grows by a few gigabytes every month. Those same gigabytes are stored on all the machines running a Bitcoin client.

Offset transactions are efficient, as Offset does not rely on proof of work or global consensus. Every Offset transaction involves communication between a few select machines, without any significant amount of computation. The amount of data Offset nodes has to maintain is small and constant sized.

Fig. 5: During an Offset payment, a few balances between a few select nodes are affected. The rest of the network is unaware of the transaction.

5.7 Transaction speed

In a blockchain based digital currency, every batch of transactions has to propagate through all the participants of the blockchain network. As a means of avoiding money double spending, participants in the blockchain network have to perform expensive proof of work and achieve global consensus over the new state of the blockchain.

A blockchain transaction is considered complete only when there is enough certainty that it will stay inside the blockchain, and this might take a long time to happen.

For example, in Bitcoin new blocks are added to the blockchain at a rate of about 1 block every 10 minutes. For small transactions most users will want to wait at least one block, and for larger transactions where stronger certainty is required, users will sometimes prefer to even wait 6 blocks (about one hour).

Offset transactions are very efficient with respect to their blockchain based counterparts. This is possible because Offset does not rely on a global consensus to operate.

It usually takes no more than a few seconds for an Offset transaction to complete. an Offset transaction will usually pass through only a few computers in the network that are relevant to the transaction. Offset doesn't have to maintain any shared ledger, and therefore no consensus or proof of work are required.

5.8 Transaction certainty

Payments with blockchain based currencies have some amount of uncertainty. When you send money using blockchain currencies, you have to wait for a while. The more you wait, the more certain you are that the transaction completed successfuly, though you will never become 100% sure.

This phenomenon is inherent to the blockchain design. When a new transaction is added as part of a new block on the blockchain, it is still possible that a "longer chain" not containing the new transaction will appear. A transaction is considered to be more and more certain as new blocks are added on top of it.

Most blockchain based currencies allows the sender of money to add transaction fees. The fees are paid to the miners that run the expensive consensus computation (proof of work), hence miners prioritize transactions with higher fees. Paying higher fees for a transaction makes it get into the blockchain faster, hence increasing the certainty that it will complete successfuly in a timely manner.

Unlike blockchain based transactions, Offset transactions do not have an element of uncertainty. Offset transactions are 100% certain when complete. We call this characteristic **transaction atomicity**.

An Offset transaction changes a list of balances along a path, atomically. When the buyer hands an Offset Commitment to the seller, the transaction is complete with 100% certainty.

Fig. 6: A diagram with thumb rules of how many bitcoin confirmations (blocks) are enough to be sure a transaction is complete. Taken from buybitcoinworldwide.

5.9 Storage

To operate a blockchain, every network node has to store the full blockchain. For example, the size of the bitcoin blockchain in May 2020 is more than 270GB, and it keeps growing in the rate of about 5GB every month.

Offset is storage efficient. In comparison, every Offset user has to save only a few kilobytes of information about his balances and current state, and that amount stays constant.

5.10 Fees

Sending money using a blockchain based currency usually requires extra transaction fees. This is extra money paid to make sure the transaction succeeds. Why are those fees required?

Blockchain based currencies are usually operated by miners: Those are machines that run the computationaly expensive global consensus algorithm, Also known as proof of work.

Running a miner is expensive, as it requires electricity, proper cooling and other maintanence. To cover those expenses, miners are incentivized by receiving transaction fees. In Bitcoin, for example, the average transaction fee (5/2020) is a few US dollars.

Miners will usually prioritize transactions with higher fees over transactions with lower fees. Hence users that want to make sure their transaction is processed quickly have to provide large enough fees.

If transaction fees are too low, it will become not profitable to run a miner. Therefore blockchain based networks have a theoretical lower bound over the transaction fees[2]. This lower bound is related to the amount of miners in the network, the amount of transactions (per unit of time) and to the costs of computation.

Offset does not require a global consensus, and has no miners. It is extremely cheap to run an Offset node, and so Offset fees are mostly unrelated to computation costs. Offset fees are determined by Offset users. Every user can decide the fees required for a certain Offset friend to transfer a transaction through him.

[2] Unless miners are willing to lose money

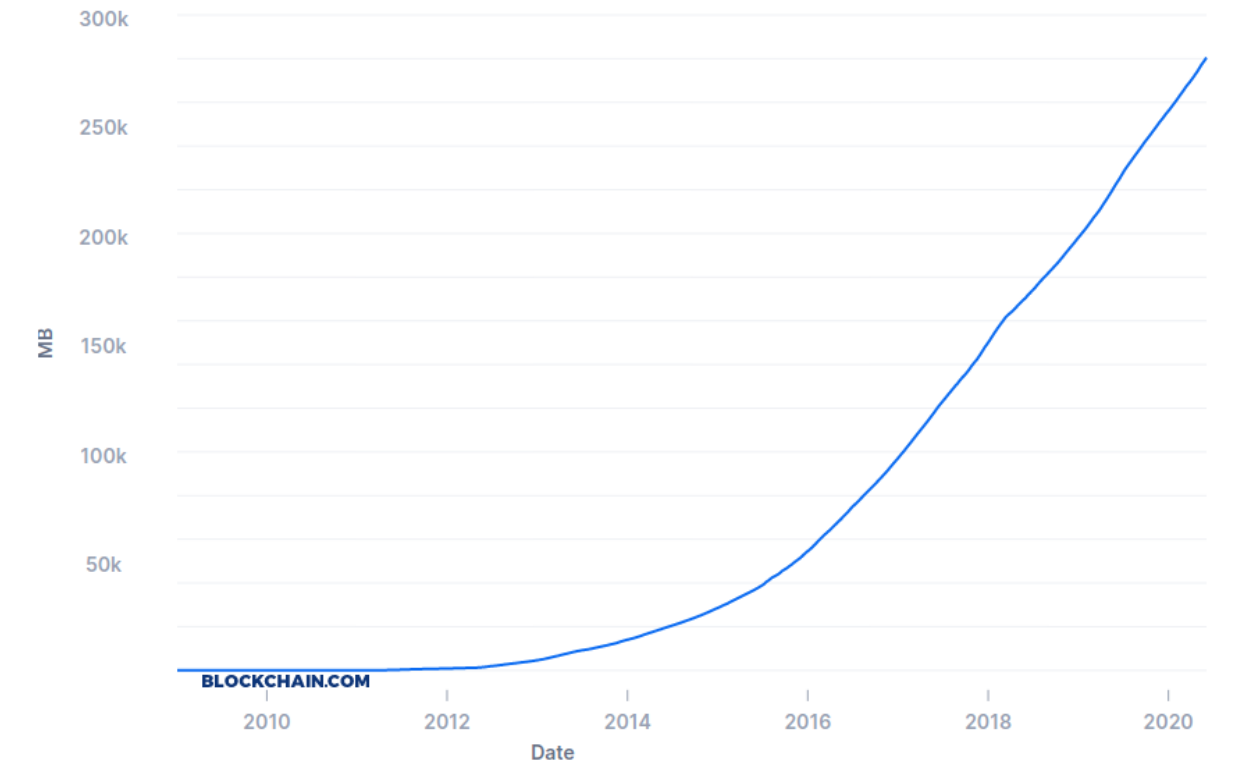

Fig. 7: A chart showing historical data for Bitcoin's blockchain size. Taken from blockchain.com.

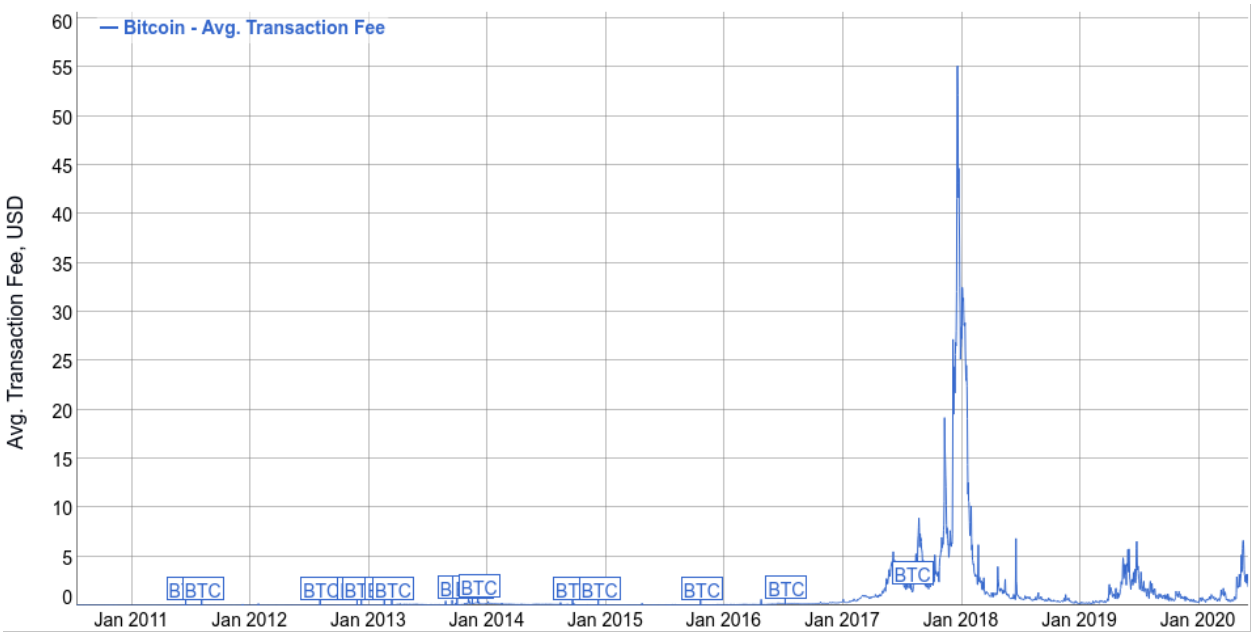

Fig. 8: Historical Bitcoin average transaction fee. Note the peak at the end of 2017, happening due to large amount of transactions during the "Crypto boom". At that time the avarage transaction fee reached $55.16 per transaction. Chart Taken from from bitinfocharts.

It is too early to know, though we believe that Offset fees will be mostly related to risk management. For example, a large Offset hub might take larger fees, because of the greater risk taken when dealing with many people, while two family relatives might set up 0 fees, because the risk is much lower. Generally speaking, the larger the trust between people, the lower the fees.

5.11 Receive payments when offline

The blockchain approach allows users to collect payments even when they are offline. For example, it is possible to send money to a Bitcoin address even if the recipient is currently not connected to the Internet.

One downside of Offset design is that Offset users have to be online in order to collect payments. This happens because Offset payments require the recipient to sign using his private key. The recipient is the only one knowing his private key, and therefore he has to be online in order to collect the incoming payment.

CHAPTER

SIX

PRIVACY

> "Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say." – Edward Snowden

Privacy is a core value in the design of Offset. This document discusses what information Offset allows you to keep private, and what bits of information might leak when you use Offset.

6.1 Summary

- The only unique id representing your Offset identity is a public key.
- Offset does not require any of the following information:
  - Real name
  - Bank account information
  - Credit card number
  - Phone number (Sim card is also not required)
  - Email address
- Transactions
  - In most cases when you buy something with Offset, your identity will be kept secret.
  - When you sell something using Offset, the buyer learns your public key.
- Index servers
  - Know your public key
  - Know the public keys of your Offset friends
  - Know approximate balances between you and your Offset friends.
  - Know when you are online
- Relays
  - Know your public key
  - Know when you are online
  - Might know the public keys of some of your Offset friends
- Offset friends

  - Know your public key
  - Know when you are online
  - Usually do not know your IP address
- Strangers
  - Can't do much if don't know your public key
  - If a stranger knows your public key, he might be able to deduce:
    * The public keys of your Offset friends
    * Approximate balances between you and your Offset friends.
    * When you are online
- The Offset app
  - Contains no tracking code
  - Stores your identity locally.
  - Does not send your information to any third party, except for servers and Offset friends you choose explicitly.

6.2 Identity

When you create a new Offset credit card, a new key pair (private and public key) is created for you automatically. Offset does not require that you provide your real name, id number, bank account, credit card number, phone number or email address. Therefore, non of those pieces of information are linked to your Offset identity.

The newly created public key is the only unique id representing your identity in almost all of Offset operations.

6.3 Transactions

When you initiate **a sale** by creating an invoice and sending it, you expose your Offset public key. This happens because your public key is included inside the invoice file. Unless your public key is provided, the buyer (And the rest of nodes along the transaction chain) will not be able to verify your signature.

When you buy from someone using Offset, you send the funds along a chain of Offset nodes. The seller can only see the public key of the last node on the chain, and therefore if the chain is of size at least 3, the seller will not be able to learn the buyer's public key. Hence, the buyer can have certain privacy when buying goods.

6.4 Index servers

Index servers has the job of knowing the full state of an Offset network, allowing nodes to find routes of certain credit capacity. As such, index servers need to know about every Offset friendship, and an approximate balance for every currency in those friendships.

As a result, given a node's public key, an index server can know whether a node is online, list the node's friends public keys, and provide approximate balances with those friends.

6.5 Relays

Relays facilitate communication between nodes. A node typically listens on multiple relays to accept TCP connections from friends. As communication between nodes is usually not direct, in most cases Offset friends can not learn each other's IP address.

As a node is constantly connected to a few relay servers, a relay server can know whether or not a node is online. (Though not every relay will have this information about every node).

6.6 Friends

Your Offset friends know your Offset public key (It is included inside your friend ticket).

Knowing the liveness status of an Offset friend is crucial for the operation of Offset, because Offset nodes should not forward transactions to friends that seem to be offline. Your Offset friends get information from their relays about your liveness status. Therefore, Offset friends can always know if you are online.

6.7 Strangers

A stranger that does not know your Offset public key will not be able to obtain any information about your presence in Offset.

A stranger that knows your Offset public key can query index servers, to be able to deduce some information like your friends' public keys and possibly deduce approximate balances between you and your friends.

6.8 Offset app

The Offset app does not include any tracking functionality, and does not send your information to any third party without your explicit consent. When you create an Offset local card, your identity is kept inside your mobile device. There is no "Offset app backend", as Offset is fully decentralized.

When you connect with an index server, relay or an (Offset) friend, you share some information with them, as described earlier in this document.

CHAPTER

SEVEN

ABOUT

The idea for Offset began somewhere in 2010 during a call to my bank, where I was asking to cancel my credit card. After some inquiries, the banker on the other side of the line finally concluded he is not willing to cancel my credit card.

Baffled by the banker's response, I asked how is it possible that I can not cancel my own credit card. The banker laughed and answered that the credit card is not truly mine, it belongs to the bank. He then pursued asking if I happen to be interested in some kind of loan, because according to my history of transactions, I might be eligible for one.

I hung up the phone feeling confused. That conversation set me up for a journey learning about the nature of money, and how to decentralize the power over it back to the people.

10 years later, with the help of many friends along the way, the basic technology is complete. For the first time I can hold in my hand a credit card that is truly mine.

– real

CHAPTER

EIGHT

CONTACT

For any ideas, suggestions or questions, please contact us!

- Community forum
- Email: real@freedomlayer.org
- **Github:**
  - Offset core
  - Offset mobile app
  - Offset documentation
  - Offset website
- Subscribe to our mailing lists to get updates:

CHAPTER

NINE

NETWORK STRUCTURE

Offset's network is fully decentralized. This means that there are no single Offset server. Instead, there are multiple instances, run by multiple people and organizations, cooperating together to make Offset work.

Decentralization is a core property of Offset. When control over money is decentralized, it becomes difficult to abuse.

For example, when you use a local Offset card inside your mobile phone, you have full control over your card, as it is fully hosted inside your phone. No single party can decide to confiscate your Offset credits or freeze your Offset card.

9.1 Main network components

The network consists of a few entities:

- Node (Equivalent to one Offset card)
- Relay server (`strelay`)
- Index server (`stindex`)
- Application (For example: `stctrl`)

9.2 Example network topology

Fig. 1: All lines in the diagram above represent encrypted TCP connections.

9.3 Node

The node is the core component of the Offset network. It is roughly equivalent to one Offset card. The node is responsible for all payment related logic.

To operate, a node needs the following information:

- An identity file: A private key used to authenticate the identity of the node.
- A database file: Used to save the current relationships (balances and open payment requests) with other nodes.
- A list of trusted applications and their permissions.

Nodes of Offset friends communicate with each other. Two nodes can communicate through a relay[1], using end-to-end encryption[2].

A node connects to relay servers to be able to accept communication from other nodes. In addition, a node connects to index servers for two purposes:

- Reporting current credit capacities: How much credit can be pushed through a certain Offset friend).
- Requesting for routes. Routes of friends are required for creating Offset transactions.

The node listens on a TCP port for Offset applications to connect. Applications must register ahead of time to be able to connect to the node. The communication interface between a node and an application allows an application to view the current node status, and perform various operations.

A node needs to be configured to have at least one relay server and one index server to be able to operate.

9.4 Relay server

In a perfect world nodes should have been able to directly communicate with each other. However, currently many of the Internet users are not able to receive connections directly.

As a workaround, the offset network uses relay servers. A relay server is a server that is used as a meeting point for communication between two nodes. A node connects to a relay server in one of two modes:

- Wait for incoming connections from remote nodes.
- Request to connect to a remote node.

When a node is configured to have a remote node as a friend, it must know one or more relays on which the remote node is listening for connections.

The relays model is decentralized. Anyone[3] can run his own relay. However, we realize that some users might not be willing (or able) to run their own relay servers. Instead, it is possible to use the services of a public relay.

9.5 Index server

Payments in offset rely on pushing credits along a routes of nodes. We did not manage to find a fully decentralized solution for finding capacity routes between nodes. As a workaround, we use index servers.

A node is configured to know one or more index servers, and can ask information about routes from the index servers. A node also sends to the index servers periodic updates about his relationship with his friends.

The index servers form a **Federation**. Usually every node communicates with about one index server. The index servers then share the nodes information with each other. This means that the information sent from a node to one index server should eventually reach all other index servers that are reachable from that index server.

Index servers federate with other index servers only if configured to do so. For two index servers A and B, A and B will share information only if the two conditions hold:

- A trusts B

[1] We use relays because in the modern Internet it is in many cases difficult or impossible to set up a direct connection between two user owned devices. Most user devices connected to the Internet, in particular mobile phones, are behind NATs. As a result, those devices do not own a public IP address, which makes it difficult to have direct communication. Maybe this could change in the future, and in that case, relays will not be required anymore.

[2] This means that the relay forwards the data between the two Nodes, but it can not read the data, because it is encrypted at the first Node and only decrypted at the second Node.

[3] *Almost* anyone can run a relay server. The only requirement is to have a public address on the Internet, for example: A public IP address, or a domain name (Though a certificate is not required).

- B trusts A

Every index server has a full picture of the whole nodes' funds network. This allows index servers to find routes of wanted capacity efficiently, using classical graph theoretic algorithms, like BFS.

Anyone can run his own index server, but to have any value, this index server must be part of the index servers federation. On his own, an index server will only have partial information about the nodes' network, and therefore will not be able to find routes to any place in the network.

9.6 Application

An Offset application connects to an Offset node, and allows viewing information or controlling the operation of an Offset node. An application connected to a node has the following capabilities:

- Obtain information about the node.
- Configure the node
- Request routes
- Send funds

To operate, an Application needs a private key, a target node's public key and and address (For example: IP address). In addition, the target node must register the Application's public key ahead of time.

An application can be any program that communicates with a node. Examples to applications are:

- Offset mobile app.
- `stctrl` command line util.

The network protocol between Applications and Nodes is an open standard. You can write your own application and connect it to your node.

CHAPTER

TEN

MUTUAL CREDIT PROTOCOL

10.1 Prior reading

To get the most of this document, you are recommended to first read:

- *Economic idea*
- *Network structure*

10.2 Intro

At its core, **Offset allows to manage mutual credit balances at scale**. Managing credit balance between only two people is a simple task, and can be done using a pen and a paper. However, managing mutual credit for a whole economy becomes more difficult, and requires specialized technology.

Recall the idea of mutual credit. Consider three B, C, D. Suppose that B and C maintain a shared balance, and also that C and D maintain shared balance. We can draw the relationships between B, C, D using this simplified graph:

```
B --- C --- D
```

Suppose that initially, B and C set the balance between them to be 0, and that C and D also set the balance between them to be 0. Assume that B wants to pay D 100 credits. In order to do this, the following needs to happen:

1. B and C set the balance between them to -100 (B owes C 100 credits)
2. C and D set the balance between them to -100 (C owes D 100 credits)

In total, we have the following balances:

- B: -100
- C: -100 + 100 = 0
- D: +100

Therefore, we consider the payment from B to D to be successful. The process described here is the core of what Offst does.

If paying is that simple, why do we need a sophisticated protocol? There are a few problems the Offset protocol attempts to solve:

- Correct incentives: What stops C from taking the money B has sent and run away with the money?
- Atomic payments along routes: What happens if in the middle of the payment, C suddenly become nonresponsive? How can we be sure that transactions are synchronized correctly, and don't get stuck?

The protocol described here is an abstraction of the communication protocol used between nodes. For brevity's sake, we neglect a few matters:

- Route discovery (Done using index servers)
- How communication between nodes is relayed using relay servers.
- **External "Token Channel" protocol**
  - Allows nodes to keep communication even after disconnection.
  - Synchronizes communication between the two nodes.
  - Multiplexes multiple currency balances.

The mutual credit protocol described here explains the inner workings of a single currency channel between two nodes.

10.3 Protocol outline

Consider a network of 4 parties: B, C, D, E, where the pairs: (B,C), (C,D), (D,E) are Offset friends.

```
B --- C --- D --- E
```

Suppose that B wants to send credits to E along this route. We call B the **buyer**, and E the **seller**. The following is an outline diagram for the payment protocol:

```
Invoice         <=====[inv]========    (Out of band)

Request         ------[req]------->
Response        <-----[resp]-------

Commit          ======[commit]====>    (Out of band)
Goods           <=====[goods]======    (Out of band)

Collect         <-----[collect]----
(Receipt)
                B --- C --- D --- E
```

Where single arrows (`--->`) represent communication done between nodes, and fat arrows (`===>`) represent communication done out of band. Out of band communication could be using QR codes, sending an instant message on a mobile phone, or using email.

For example, in the diagram above, the invoice will be sent from E to B using some external communication. However, the Request message sent from B to E is sent along the chain of nodes `B -- C -- D -- E`.

In the above diagram, the following operations occur:

1. E hands an Invoice to B. (Out of band)
2. B sends a Request message along a route to E.
3. E sends back a Response message along the same route to B.
4. B prepares a Commit message and hands it to E (Out of band).
5. E gives the goods to B (Out of band).
6. E sends a Collect message along the route to B.
7. B receives the Collect message, prepares a Receipt and keeps it.

From the point of view of an Offset user, this is how the transaction looks like:

```
Invoice         <=====[inv]========     (Out of band)
Commit          ======[commit]====>     (Out of band)
Goods           <=====[goods]======     (Out of band)
                B -- ...    ... -- E
```

The user only see the following steps:

(1) E hands an Invoice to B. (Out of band) (4) B prepares a Commit message and hands it to E (Out of band). (5) E gives the goods to B.

The event of B handing the commitment to E is "atomic". In other words, the moment E receives the commitment, E knows for sure that he will receive the money, and the transaction is considered successful. Note however, that it might take some time until the seller E will be able to collect his credits.

10.4 Message definitions

The Offset protocol contains 4 in band messages: Request, Response, Cancel and Collect, and two out of band messages: Invoice and Commit. We also describe here the structure of the Receipt message, which the buyer can compose after a successful transaction.

10.4.1 Invoice

```
<=======[inv]======
B                 E
```

The structure of an invoice:

```
struct Invoice {
    invoiceId: InvoiceId,
    currency: Currency,
    destPublicKey: PublicKey,
    destPayment: u128,
}
```

Description of invoice fields:

- `invoice_id` is a unique id representing this invoice. It is randomly generated by the seller.
- `currency` is a short string representing the name of the currency being used for this invoice.
- `destPublicKey` is the seller's public key. The buyer will search this public key using an index server to obtain a route from the buyer all the way to the seller, along Offset friends.
- `destPayment` is the total amount of credits to be paid.

The Invoice is an out of band message. It could be sent for example using a QR code, instant messaging, email.

10.4.2 Request message

```
 -------[req]----->
B --- C --- D --- E
```

This is the structure of the Request message:

```
struct RequestSendFundsOp {
        requestId @0: Uid;
        # Id number of this request. Used to identify the whole transaction
        # over this route.
        srcHashedLock @1: HashedLock;
        # A hash lock created by the originator of this request
        route @2: FriendsRoute;
        destPayment @3: CustomUInt128;
        totalDestPayment @4: CustomUInt128;
        invoiceId @5: InvoiceId;
        # Id number of the invoice we are attempting to pay
        leftFees @6: CustomUInt128;
        # Amount of fees left to give to mediators
        # Every mediator takes the amount of fees he wants and subtracts this
        # value accordingly.
}
```

Description of the message fields:

- **`requestId` is a unique id given to this request by the buyer. The** recommended way to create a requestId is to randomly generate it.
- **`srcHashedLock` is a hash over a secret value that only the buyer knows. The** buyer will expose this value only when he is ready to commit to the funds transfer.
- **`route` is a route of nodes, beginning from the buyer node, all the way to** the seller node. The route contains the public keys of all the node along the route.
- **`destPayment` contains the amount of credits (in a certain currency) being** sent in this request.
- **`totalDestPayment` contains the total amount of credits the buyer plans to** send to pay an invoice. Hence, the following should always hold: `destPayment <= totalDestPayment`. During single route payments, `destPayment == totalDestPayment`, but during multi route payments, we will have `destPayment < totalDestPayment`.
- `invoiceId` is the id of the invoice this Request message attempts to pay. This value is copied from the Invoice.
- `leftFees` contain the amount of fees we are willing to pay to transfer this request. This value is being reduced with every hop of forwarding the request message.

When a node receives a Request message, it verifies that there is enough capacity to make the payment along the route (Including capacity for the transaction fees). For example, if B wants to send 10 credits to E, paying 1 1 credit of fees for every node along the route, then during the Request message passage from B to E:

- B checks that B -> C has at least the capacity of 12 credits.
- C checks that C -> D has at least the capacity of 11 credits.
- D checks that D -> E has at least the capacity of 10 credits.

The extra credits are due to transaction fees to C and D (1 credit each).

If at any hop along the route a node finds out that the fees provided are not sufficient, or that there is not enough capacity to pass the Request message, a Cancel message is sent backwards, all the way to the buyer node.

The required amount of credits is then frozen. With respect to the example above:

- 12 credits are frozen in B -> C
- 11 credits are frozen in C -> D
- 10 credits are frozen in D -> E

When credits are frozen, they can not be used for other transactions. The credits will be unfrozen in one of two cases:

1. The transaction was successful. The credits will be unfrozen and moved to the next node.
2. The transaction was cancelled. The credits will be unfrozen and returned to their original owner.

The Request message contains a hash lock: `srcHashedLock`. This value is generated by the buyer by generating a random `srcPlainLock` value and hashing it: `srcHashedLock := hash(srcPlainLock)`. This mechanism is used to ensure transaction atomicity: The seller can not create a valid Collect message without knowing the secret value `srcPlainLock`.

10.4.3 Response message

If all went well during the Request stage, E (The seller) sends back a Response message along the same route, all the way back to B.

```
 <------[resp]-----
B --- C --- D --- E
```

Structure of a Response message:

```
struct ResponseSendFundsOp {
        requestId @0: Uid;
        destHashedLock @1: HashedLock;
        isComplete @2: Bool;
        # Has the destination received all the funds he asked for at the invoice?
        # Mostly meaningful in the case of multi-path payments.
        # isComplete == True means that no more requests should be sent.
        # The isComplete field is crucial for the construction of a Commit message.
        randNonce @3: RandValue;
        signature @4: Signature;
        # Signature{key=destinationKey}(
        #   sha512/256("FUNDS_RESPONSE") ||
        #   sha512/256(requestId || randNonce) ||
        #   srcHashedLock ||
        #   destHashedLock ||
        #   isComplete ||
        #   destPayment ||
        #   totalDestPayment ||
        #   invoiceId ||
        #   currency [Implicitly known by the mutual credit]
        # )
}
```

Description of the message fields:

- `requestId` must match the requestId value provided in the Request message.
- `destHashedLock` This value is created by hashing a secret generated by the seller: `destHashedLock := hash(destPlainLock)`. This secret will only be revealed when the Collect message is sent. We have this mechanism to defend against fake Receipt generated by the buyer. (A valid receipt must contain the secret `destPlainLock`).

- `isComplete`: This is a boolean value. It contains "true" only if the destination has received all of the funds he asked for in the invoice. Otherwise, it contains "false". During single route payments this value is always set to "true". During multi route payments, the seller will issue multiple Response messages, and only the last Response message will have `isComplete=true`.
- `randNonce` is a value randomly generated by the seller. We add this value to make sure the signature over this Response message (Signed by the seller) can not be reused.
- `signature`: This field is a signature, signed using the seller's key. Note that the signed buffer contains various fields from the previous Request message, and also from the newly created Response message. The signed buffer also contains an extra implicit value: currency. This is the name of the currency being used for this transaction.

Only the seller can create a valid signature for a Response message. Therefore, a valid signature is a proof that a Request message has reached all the way to the seller.

When a node receives a Response message, it first makes sure that a corresponding matching Request message was sent earlier in the opposite direction. In case of a match, and if the signature is valid, the Response message is forwarded to the next node along the reversed route.

10.4.4 Cancel message

A Cancel message may be sent back by any node during the Request period. After Request message arrived at the seller node and before the Collect message was sent, only the seller node may send a Cancel message. In addition, any node may send a Cancel message to cancel ongoing transactions in case of unfriending a node (As long as the Collect message was not yet received and forwarded).

After the Collect message was received, the transaction can not be cancelled.

If any node could not forward the Request message, or the seller decided to cancel the transaction, a Cancel message will be sent back, beginning from the failing node.

```
 <----[cancel]-----
B --- C --- D --- E
```

```
struct CancelSendFundsOp {
        requestId @0: Uid;
}
```

The only field present in a Cancel message is the `requestId`, which matches the requestId value sent inside the corresponding Request message.

10.4.5 Commit message

After receiving a Response message, the buyer node creates a Commit message. The Commit message is created by the buyer, using the corresponding Request and Response message. In case of a multi route payment, a Response message with `isComplete=true` must be used for the creation of the Commit message.

The Commit message is given to the seller (out of band), and at that moment the payment is considered successful.

```
======[commit]====>     (Out of band)
B                  E
```

Upon receipt of a valid Commit message, the seller will give the goods to the buyer, and send back (along the same route) a Collect message to collect his credits.

```
struct Commit {
        responseHash @0: HashResult;
        # sha512/256(requestId || randNonce) ||
        srcPlainLock @1: PlainLock;
        destHashedLock @2: HashedLock;
        destPayment @3: CustomUInt128;
        totalDestPayment @4: CustomUInt128;
        invoiceId @5: InvoiceId;
        currency @6: Currency;
        signature @7: Signature;
        # Signature{key=destinationKey}(
        #   sha512/256("FUNDS_RESPONSE") ||
        #   sha512/256(requestId || randNonce) ||
        #   srcHashedLock ||
        #   destHashedLock ||
        #   isComplete ||       (Assumed to be True)
        #   destPayment ||
        #   totalDestPayment ||
        #   invoiceId ||
        #   currency
        # )
}
```

Description of Commit fields:

- `responseHash` equals `sha512/256(requestId || randNonce)`. A trick used to make the Commit shorter.
- `srcPlainLock` is the secret originally chosen by the buyer, revealed. Only when this value reaches the seller, the seller is able to collect the funds. Corresponds to the srcHashedLock field from the Request message.
- `destHashedLock` equals the destHashedLock field from the Response message. The seller's secret will be revealed when he sends the Collect message.
- `destPayment` is the same field as destPayment from the corresponding Request message. In case of a multi route payment, this will contain the destPayment of the Response with `isComplete=true`.
- `totalDestPayment` is the same field as totalDestPayment from the corresponding Request message.
- `invoiceId` equals the invoiceId specified in the initial invoice.
- `currency` is the name of the currency being used. We add this value to allow third parties verify the Commit too. (Compare to the Response message, where the value of currency was implicitly known by the two communicating nodes, so it wasn't included in the message).
- `signature` is same signature as in the Response message. For the Commit to be valid, the signed buffer must have `isComplete=true`.

Verification of a Commit message is done as follows:

- InvoiceId matches an originally issued invoice.
- Check that `destPayment <= totalDestPayment` holds.
- The revealed lock is valid: `hash(srcPlainLock) == srcHashedLock`
- Signature is valid, assuming that `isComplete=true`.

10.4.6 Collect message

After receiving a confirmation message from the buyer, the seller gives the goods to the buyer and sends back a Collect message to collect his credits.

```
 <----[collect]----
B --- C --- D --- E
```

A Collect message completes the transaction. For example, when the Collect message is sent from E to D, the credits that were frozen between D and E become unfrozen, and the payment is irreversible. The Collect messages continues all the way (along the original route) to the source of the payment.

```
struct CollectSendFundsOp {
        requestId @0: Uid;
        srcPlainLock @1: PlainLock;
        destPlainLock @2: PlainLock;
}
```

Collect message fields:

- `requestId` matches the requestId of the Request message.
- `srcPlainLock` corresponds to srcHashedLock from the Request message. The following should hold: `hash(srcPlainLock) = srcHashedLock`.
- `destPlainLock` corresponds to destHashedLock from the Response message. The following should hold: `hash(destPlainLock) = destHashedLock`.

Note that the Collect message can only be sent by the seller after it has received the Commit message, because the Commit message is the first message where the buyer reveals srcPlainLock.

In the case of multi route payment, a single Collect message is sent along the reversed route of the corresponding Request message.

10.4.7 Receipt

Upon receiving the Collect message, the buyer of the payment can compose a Receipt. The Receipt is a cryptographic artifact, proving that the transaction has occured.

```
# A receipt for payment to the Funder
struct Receipt {
        responseHash @0: HashResult;
        # = sha512/256(requestId || randNonce)
        invoiceId @1: InvoiceId;
        currency @2: Currency;
        srcPlainLock @3: PlainLock;
        destPlainLock @4: PlainLock;
        isComplete @5: Bool;
        destPayment @6: CustomUInt128;
        totalDestPayment @7: CustomUInt128;
        signature @8: Signature;
        # Signature{key=destinationKey}(
        #   sha512/256("FUNDS_RESPONSE") ||
        #   sha512/256(requestId || sha512/256(route) || randNonce) ||
        #   srcHashedLock ||
        #   dstHashedLock ||
        #   isComplete ||       (Assumed to be True)
```

(continues on next page)

(continued from previous page)

```
        #   destPayment ||
        #   totalDestPayment ||
        #   invoiceId ||
        #   currency
        # )
}
```

The fields of the Receipt are taken from the corresponding Request, Response and Collect message. The Receipt can be constructed only after the Collect message was received, because the Collect message is where the seller reveals destPlainLock for the first time.

Note that payment was considered successful the moment the buyer hands the seller a valid Commit message. At that moment the goods can already be transferred. However, it might take a while until the buyer can compose a Receipt message, because the Collect message sent from the seller along the reversed route might take a while to arrive.

10.5 Cancellation

Cancellation can happen at any time after the Request message was sent from the buyer and before the Collect message was sent by the seller.

During the Request stage a Cancel message could be sent from any node forwarding the Request message. However, after the Request message arrives at the seller node, only the seller node may issue a Cancel message. (This rule has one exception that happens during unfriending, see below).

10.5.1 Examples for cancellation

- An intermediate node cancels the transaction during the Request period. This can happen for example if there is not enough capacity for pushing credits forward, or the provided fees are not sufficient:

```
Invoice         <=====[inv]========    (Out of band)

Request         ---[req]---->
Cancel          <--[cancel]--

                B --- C --- D --- E
```

- E (The seller) did not recognize the provided invoiceId in the Request message:

```
Invoice         <=====[inv]========    (Out of band)

Request         ------[req]------->
Cancel          <-----[cancel]-----

                B --- C --- D --- E
```

- E (The seller) waited too long[1] for a Commit message, so it decided to cancel:

```
Invoice         <=====[inv]========    (Out of band)

Request         ------[req]------->
Response        <-----[resp]-------
```

(continues on next page)

[1] Offset's Mutual credit protocol does not contain any built in timeouts. Timeouts can be set up by the user, externally.

(continued from previous page)

```
Cancel          <-----[cancel]-----

                B --- C --- D --- E
```

- D unfriends E, and cancels a pending request:

```
Invoice         <=====[inv]========    (Out of band)

Request         ------[req]------->
                B --- C --- D --- E
Unfriend
Cancel          <--[cancel]--
                B --- C --- D     E
```

In the figure above: a request was sent from B to E. Next, D unfriends E before E manages to send the response message. In that case D sends a Cancel message for this transaction all the way back to B, and the transaction credits are unfrozen.

- Cancellation in Response period that happens due to unfriending nodes:

```
Invoice         <=====[inv]========    (Out of band)

Request         ------[req]------->
Response        <-----[resp]-------
                B --- C --- D --- E
Unfriend
Cancel          <--[cancel]--
                B --- C --- D     E
```

10.5.2 Cancellation responsibility

Consider cancellation from the point of view of Offset users.

Recall that a Cancel message is sent backwards (In the direction from the seller to the buyer). A Cancel message can be sent by one of the nodes along the route from the buyer to the seller, or by the seller itself. The buyer can not issue a Cancel message.

From the point of view of an Offset user, the seller can cancel a transaction as long as the funds where not yet collected (A Collect message was not yet sent). The buyer can cancel a transaction as long as a Commit was not given to the seller.

Internally, even if the buyer has not provided a Commit message to the seller, the credits for the transaction are still frozen. The credits will be unfrozen only when the seller decides to cancel the transaction, or until two consequent nodes along the route unfriend each other.

The frozen credits block available capacity both for the buyer and the seller, so we assume that the seller has the incentive to eventually cancel the transaction.

10.5.3 Delayed funds collection

The seller does not have to send a Collect message immediately after he receives the Commit message. This allows to implement something like "return policy", where the buyer keeps the money frozen for a certain period of time.

If during this period of time the buyer wants to return the good, the seller can send a Cancel message, cancelling the transaction.

The main advantage of using this method (over creating a new payment from the seller to the buyer to return the money) is that the seller nor the buyer have to repay the transaction fees. When the transaction is cancelled, the transaction fees are fully returned to the seller.

Illustration:

```
Invoice         <=====[inv]========    (Out of band)

Request         ------[req]------->
Response        <-----[resp]-------

Commit          ======[commit]====>    (Out of band)
Goods           <=====[goods]======    (Out of band)

Return          ======[goods]=====>    (Out of band)
Cancel          <-----[cancel]----
                B --- C --- D --- E
```

10.6 Atomicity

Atomicity is guaranteed by using a hash lock created by the buyer: `srcHashedLock`.

Assume that the node E issued an invoice and handed it to B.

B wants to pay the invoice. The payment begins by sending a Request message along the path from B to E. The payment is considered successful when B hands a Commit message to E.

We should examine the possibility of B waiting indefinitely during the sending of Request and Response messages along the route.

During this time (Request + Response period), B can discard the transaction by walking away. E will not be able to make progress because in order to send the Collect message, the correct srcPlainLock is required, but E does not know it before B sends the Commit message.

If B sends a valid Commit message to E, the transaction is considered successful, and B can not reverse it. This happens because B reveals srcPlainLock at the Commit message sent to E.

All the above said, there is no escaping the fact that there must exist some trust between the buyer and the seller during the transaction, as a network protocol can not secure the passage of goods in the real world.

Some examples:

- Upon receiving a valid Commit from the buyer, a seller can run away with the goods.
- After a buyer provides a seller with a Commit message, the seller can show a screenshot on his phone claiming that the provided Commit is not valid, and refuse to hand the buyer the goods. Later when the buyer leaves, the seller can send a Collect message. The buyer will eventually have a Receipt, but by the time it happens, the buyer might be too far away to deal with the fraud.

10.7 Receipt verifiability

A Receipt is a proof that a certain invoice was paid. It can be verified by anyone that possesses:

- The invoice (`invoiceId` + public key of seller)
- The Receipt

Verification is performed by checking the signature (See description of signature at the Receipt definition).

In order to make sure the buyer can not have a valid Receipt before the payment actually completed, we use a hash lock that is issued by the payment destination: `destHashedLock`.

When the buyer receives a Response message it can not yet create a valid Receipt, because the buyer doesn't yet know `destPlainLock`. This value is revealed only at the Collect message, when the payment is considered to be successful.

An alternative solution could be to let the seller sign a new signature over the Collect message, but instead we chose to use a hash lock, which is a less expensive cryptographic operation. Using a hash lock also does not require access to the identity of the seller.

This leaves the whole protocol with only one cryptographic signature over the Response message, signed by the seller.

10.8 Multi-Route transactions

Sometimes it might not be possible to send a payment along a single route. In such cases it is useful to send the transaction along multiple routes. The protocol allows sending a payment along multiple routes atomically. This is done as follows:

1. Buyer gets an Invoice from the seller for a certain amount of credits.
2. Buyer sends a Request message along a route.
3. A Response or a Cance message is returned.
4. Go back to (2) until the wanted amount of credits is acheived (for paying the invoice), and a Response message with `isComplete=true` is received.
5. Buyer creates a Commit message and hands it to the seller.
6. Seller verifies the Commit message. If valid, the payment is accepted and the goods are handed to the buyer.
7. The Seller sends back Collect messages for all requests. Each Collect message is sent along the reversed route of the corresponding request.
8. Only the Collect message corresponding to the Response message with `isComplete=true` can be used to compose a valid Receipt.

10.9 Future work

This is not a final protocol specification. The Offset protocol is expected to change in the future. If you have ideas, suggestions or questions, don't hesitate to open an issue at the project's repository.

CHAPTER

ELEVEN

INITIAL SETUP

Hey, welcome to the advanced usage tutorial for Offset. Going through this tutorial requires minimal familiarity using the command line interface on your computer.

You will also need a working desktop computer. Offset should work correctly on Windows, Linux and macOS.

11.1 Download

To use this tutorial, you first need to download an Offset release suitable for your system. You can download Offset from the releases page.

The downloaded file is a compressed file. Extract the files. You should now have a directory tree similar to this:

```
.
├── bin
│   ├── stcompact
│   ├── stctrl
│   ├── stindex
│   ├── stmgr
│   ├── stnode
│   └── strelay
├── LICENSE-AGPL3
├── LICENSE-APACHE
├── LICENSE-MIT
└── README.md
```

11.2 Installation

No special installation is required. Copy the resulting directory to any placed you would like on your system. For your convenience, you are recommended to add the `bin` directory to your working path.

To make sure that everything works correctly, run `stmgr -V` in your terminal. You should see output like this:

```
$ stmgr -V
stmgr 0.1.0
```

CHAPTER

TWELVE

NODE

An Offset node is roughly the equivalent of an Offset card.

An Offset node that lives inside a mobile device has a major disadvantage: when the mobile device enters "sleep mode", or loses Internet connection, the node becomes offline. As a result, transactions can not be forwarded using the node.

This document describes how to set up a node on your desktop computer. You can use the same instructions to run your Offset node in the cloud.

Recommended prior reading:

- *Network structure*
- *Initial setup*

12.1 Running your own node

We begin by presenting all the commands required to have a basic node running. We will then explain what the commands do.

```
# Public facing node address
NODE_ADDRESS="www.mynode.com"

# Public facing node port
NODE_PORT=9876

# Create a directory for the node's data
mkdir node

# Create a subdirectory for the node's trusted applications.
mkdir node/trusted

# generate node's identity
stmgr gen-ident --output node/node.ident

# Init node database
stmgr init-node-db --idfile node/node.ident --output node/node.db

# Generate node tickets:
stmgr node-ticket  --address $NODE_ADDRESS:$NODE_PORT \
        --idfile node/node.ident \
        --output node/node.ticket
```

(continues on next page)

(continued from previous page)

```
# Create a directory for the app's data
mkdir app

# Generate app's identity
stmgr gen-ident --output app/app.ident

# Generate app ticket:
stmgr app-ticket --idfile app/app.ident \
        --pbuyer --pconfig --proutes --pseller \
        --output app/app.ticket

# Fill in apps tickets at the corresponding nodes' "trusted" directories
cp app/app.ticket node/trusted/

# Create node entries:
stmgr node-entry \
        --address $NODE_ADDRESS:$NODE_PORT \
        --appid app/app.ident \
        --nodeid node/node.ident \
        --output node/node.rcard

# Start node:
stnode --database node/node.db \
        --idfile node/node.ident \
        --laddr 0.0.0.0:$NODE_PORT \
        --trusted node/trusted &
```

This is all your need to do to have your own node up and running. To connect your mobile phone to your node, you need to transfer the `node/node.rcard` file to your mobile phone (You can also scan it as a QR code).

12.2 Node commands explained

Let's begin explaining the above commands.

`NODE_ADDRESS` is the public facing address of the node. Applications will connect to the node through this address.

If the scope of your experiment is your local machine, you can use `127.0.0.1`.

If you are running a test at home that includes your mobile phone, you might be able to use your address in your private LAN network.

- Linux: `ifconfig` or `ip addr`
- Windows: `ipconfig`
- macOS: `ifconfig`

Your local IP address will usually look like `10.100.101.2` or `192.168.0.5`, but it might be something else.

If you deploy your node to the cloud, you should set `NODE_ADDRESS` to your public facing IP, or possibly to your domain name. If you are using a domain name, note that you do not need to obtain a certificate, because Offset uses a different type of authentication.

`NODE_PORT` is the public facing listening port of the node. Usually you should be able to select any port that you like that is larger or equal to 1024[1].

Next, we create a directory for the node's data, and invoke:

[1] In most operating systems, ports below 1024 are usually reserved, and require administrator priviledges to use.

```
stmgr gen-ident --output node/node.ident
```

This command creates a new identity to be associated with the node. An identity is a key pair: A private key and a public key. All transactions issued through this node will be signed using this identity.

Next, we create an initial database for the node:

```
stmgr init-node-db --idfile node/node.ident --output node/node.db
```

The node's database contains the full state of the node. It contains, for example, all the current balances, configured friends, configured relay servers and index servers. The command above will create an empty new node database.

Next, we create a node ticket:

```
stmgr node-ticket  --address $NODE_ADDRESS:$NODE_PORT \
        --idfile node/node.ident \
        --output node/node.ticket
```

A node ticket is a file containing the node's public address and public key. This information allows Application to securely connect to the node.

We continue to create an Application. We first create the directory `app`, which is going to contain all of the application's files.

As for the node, we begin by generating an identity file for the application:

```
stmgr gen-ident --output app/app.ident
```

Next, we create an application ticket:

```
stmgr app-ticket --idfile app/app.ident \
        --pbuyer --pconfig --proutes --pseller \
        --output app/app.ticket
```

The application's ticket contains the the application's public key, and permissions. In the command above we gave the application all the possible permissions: buying, configuration, routes query and selling.

The application's ticket is then stored at the node's trusted directory:

```
cp app/app.ticket node/trusted/
```

By storing the application's ticket in this directory, we register the application with the node. If we skip this step, the node will not be willing to communicate with the application.

Next, we create a node entry, also known as a "remote node" file:

```
stmgr node-entry \
        --address $NODE_ADDRESS:$NODE_PORT \
        --appid app/app.ident \
        --nodeid node/node.ident \
        --output node/node.rcard
```

The remote node file allows an Offset mobile app to connect to this as an Offset application.

The node is not running yet. To run the node, we invoke:

```
stnode --database node/node.db \
        --idfile node/node.ident \
        --laddr 0.0.0.0:$NODE_PORT \
        --trusted node/trusted &
```

The *&* sign at the end of the command means that the command will run at the background. If this is not what you want, you may omit the sign.

12.3 Resulting files tree

These are the files you should have after running the above commands:

```
app/
├── app.ident
└── app.ticket

node/
├── node.db
├── node.ident
├── node.rcard
├── node.ticket
└── trusted
    └── app.ticket
```

CHAPTER

THIRTEEN

RELAY

A relay server is used to relay communication between Offset nodes. We use relays because most user devices are behind NATs, and therefore do not own a public address. As a result, it is usually very difficult for two user devices to communicate directly.

Recommended prior reading:

- *Network structure*
- *Initial setup*

13.1 Running your own relay

To set up a relay, run:

```
# Public facing relay address
RELAY_ADDRESS="www.myrelay.com"

# Public facing relay port
RELAY_PORT=4567

# Create a directory for the relay's data
mkdir relay

# Generate a new identity for the relay
stmgr gen-ident --output relay/relay.ident

# Generate relay ticket
stmgr relay-ticket \
        --address $RELAY_ADDRESS:$RELAY_PORT \
        --idfile relay/relay.ident \
        --output relay/relay.relay

# Start the relay:
strelay --idfile relay/relay.ident --laddr 0.0.0.0:$RELAY_PORT &
```

13.2 Relay commands explained

We first set `RELAY_ADDRESS` as a public facing address for the relay. `RELAY_ADDRESS` is the public facing address of the relays. Nodes will connect to the relay through this address.

If the scope of your experiment is your local machine, you can use `127.0.0.1`. If you are running a test at home that includes your mobile phone, you might be able to use your address in your private LAN network.

If you want to deploy your relay to the cloud, you should set `NODE_ADDRESS` to your public facing IP, or possibly to your domain name. If you are using a domain name, note that you do not need to obtain a certificate, because Offset uses a different type of authentication.

`RELAY_PORT` is the public facing port of the relay. You can pick any port number that you want that is at least 1024[1].

Next, we create a directory named `relay`, that is going to hold all of our relay's data.

The following command:

```
stmgr gen-ident --output relay/relay.ident
```

Generates a new identity for the relay. An identity is a pair of private and public keys. When nodes connect to the relay, the relay will use his private key to prove his identity.

Next, we generate a relay ticket:

```
stmgr relay-ticket \
        --address $RELAY_ADDRESS:$RELAY_PORT \
        --idfile relay/relay.ident \
        --output relay/relay.relay
```

A relay ticket is a file containig the relay's public address and public key. The ticket file allows nodes to connect to the relay. A relay ticket file can also be used inside the Offset mobile app, to configure a new relay.

Finally, the last command:

```
strelay --idfile relay/relay.ident --laddr 0.0.0.0:$RELAY_PORT &
```

Starts the relay. The `&` sign at the end of the command means that the command will run in the background. If this is not what you want, you can omit it.

[1] In most operating systems, ports below 1024 are usually reserved, and require administrator priviledges to use.

CHAPTER

FOURTEEN

INDEX

Index servers have the role of indexing all the nodes in an Offset network. Upon request, an Index server can return routes between nodes with a requested amount of credit capacity.

Anyone can run an index server instance. However, index servers are only useful if they share information with each other. We say that the **index servers form a federation**.

In the current implementation of Offset, every node reports his current status to only one index server. This index server than forwards the information about this node to all the other index servers he knows about.

If you run your own index server but do not register it with other index servers, your index server will only have partial information about the nodes in the network, and therefore your index server will not be useful for finding routes between any pair of nodes.

Index servers only share data if they were configured to do so. For two index servers A and B to share data, the following must hold:

- A registered B as a trusted index server
- B registered A as a trusted index server

This document describes how to set up an index server on your desktop computer. You can use the same instructions to run your index server in the cloud.

Recommended prior reading:

- *Network structure*
- *Initial setup*

14.1 Running your own index server

We begin by presenting all the commands required to configure and run an index server:

```
# Public facing index server's address
INDEX_ADDRESS="www.myindex.com"

# Public facing index server's port, used by nodes
INDEX_PORT_CLIENT=3456

# Public facing index server's port, used to federate with other index
# servers
INDEX_PORT_SERVER=6543

# Create a directory for the index server's data
mkdir index
```

(continues on next page)

(continued from previous page)

```
# Create a directory for trusted index servers.
mkdir index/trusted

# Create an identity for the index server
stmgr gen-ident --output index/index.ident

# Generate index ticket to be used by nodes
stmgr index-ticket \
        --address $INDEX_ADDRESS:$INDEX_PORT_CLIENT \
        --idfile index/index.ident \
        --output index/index_client.index

# Generate index ticket to be used by other index servers
stmgr index-ticket \
        --address $INDEX_ADDRESS:$INDEX_PORT_SERVER \
        --idfile index/index.ident \
        --output index/index_server.index

# Register other trusted index servers
# Note that those servers also have to register
# our index server tickets as # trusted
cp other_index1.index index/trusted/
cp other_index2.index index/trusted/

# Start index server
stindex --idfile index/index.ident \
      --lclient $INDEX_ADDRESS:$INDEX_PORT_CLIENT \
      --lserver $INDEX_ADDRESS:$INDEX_PORT_SERVER \
      --trusted index/trusted &
```

To be able to federate with another index server, you and the other index server's operator should exchange index servers tickets. Your ticket is found at `index/index_server.index`.

Assuming that you received the file `other_index1.index` from the other index server operator, you should copy this file to your `index/trusted` directory. It is crucial that both you and the other index server operator add each other's ticket to the `index/trusted` directory.

To configure nodes to use your index server, provide them with index server ticket for node clients: `index/index_client.index`. This file can also be added using the Offset app.

14.2 Index commands explained

Selecting public listening address and ports:

```
INDEX_ADDRESS="www.myindex.com"
INDEX_PORT_CLIENT=3456
INDEX_PORT_SERVER=6543
```

`INDEX_ADDRESS` is the index server's public address. You may use either IP address or a domain name. If you choose to use a domain name, note that you do not need to register a certificate, as Offset has its own authentication mechanism.

`INDEX_PORT_CLIENT` is the index's public listening port for clients. In other words, nodes will connect to `INDEX_ADDRESS:INDEX_PORT_CLIENT`. You can pick any port number that you want that is at least 1024[1].

[1] In most operating systems, ports below 1024 are usually reserved, and require administrator priviledges to use.

INDEX_PORT_SERVER is the index's publick listening port for servers. This means other index servers federating with this index server will connect to INDEX_ADDRESS:INDEX_PORT_SERVER.

Next, we create directories to store the data required for the index server. We also create a subdirectory called trusted to keep a list of trusted index servers that this index server will federate with:

```
mkdir index
mkdir index/trusted
```

We generate a identity for the index server:

```
stmgr gen-ident --output index/index.ident
```

An identity is a key pair: A private key and a public key. The identity is used for authentication during communication with other nodes and other index servers.

We then create two tickets for this index server. A ticket contains the index's public address and public key. The first ticket will be used by nodes to connect to this index server:

```
stmgr index-ticket \
        --address $INDEX_ADDRESS:$INDEX_PORT_CLIENT \
        --idfile index/index.ident \
        --output index/index_client.index
```

The second ticket will be used by other index servers to connect to this index server:

```
stmgr index-ticket \
        --address $INDEX_ADDRESS:$INDEX_PORT_SERVER \
        --idfile index/index.ident \
        --output index/index_server.index
```

The next step is to set up trusted index servers for this index server. In the snippet below, we add two index servers tickets to our trusted directory:

```
cp other_index1.index index/trusted/
cp other_index2.index index/trusted/
```

To make federation work, we also have to provide our index server's ticket to the owners of those index servers. Federation between index servers only works if both index servers configured it.

Finally, we start the index server:

```
stindex --idfile index/index.ident \
      --lclient $INDEX_ADDRESS:$INDEX_PORT_CLIENT \
      --lserver $INDEX_ADDRESS:$INDEX_PORT_SERVER \
      --trusted index/trusted &
```

The & sign at the end of the command means that the command will run at the background. If this is not what you want, you may omit the sign.

CHAPTER

FIFTEEN

ROADMAP

Some rough ideas about the future of Offset

15.1 Protocol

- Allow payment using a single out of band file exchange (invoice file), instead of two files exchange (invoice + commit).
- Allow removing currency after it was added. Might require a zero balance.
- Currently the buyer pays fees. Maybe this is a wrong model and the seller should pay fees instead? Check incentives and safety issues around this.
- **Extend/rethink index server protocol.**
  - Add cyclic routes search for rebalancing?
  - Strategy for unfriending: creating a zero balance with a friend by rebalancing.
- Rethink numbering of token channel messages and increments during inconsistencies. Can this mechanism be cheated?
- Log money gained through commisions? Might be required for tax reports? How to do this elegantly and securely?
- Allow payment without an invoice, like a donation?

15.2 Economics

- Decide on the single currency vs multiple currency debate
- Dealing with regulations: KYC (Know Your Customer), AML (Anti Money Laundering) and filing taxes?

15.3 Offset core

- Update compiler and dependencies
- Implement bidirectional connection attempt between nodes, along relays. Currently only one node attempts to connect to the other node, according to the order of public keys.
- Solve relay/index server stall bug? Seems to be solved when servers are restarted.
- Serialization: Should we change `capnp` communication serialization to something else? Possibly `protobuf`?
- Solve issue of running `stcompact` on very old android devices (FiatJaf + Blackhole bug)
- Implement better searching algorithm for index servers. [Related issue]
- Allow seeing relays connectivity status (Online/Offline). Currently only possible for index servers.
- Implement Offset application permissions.
- Encrypt private key on disk: stnode + stcompact
- **Use a better database [Issue]**
  - for node's database
  - For stcompact (Mobile app database)
  - Possibly use sqlite3?
- **Transactions interface**
  - Should different node's applications know about each other's transactions?
  - Should there be an option to list all pending incoming/outgoing transactions? If so, how to do this efficiently?
- Safe erasure of secret data [Issue]
- Exponential backoff for all retrying connectors [Issue]
- **Enhance signature security**
  - Should add textual constant prefix to all signatures?
  - Is the signature malleable? Is this an issue? What can be done to solve?
- Replace `ring` cryptographic library with something else, due to testing issues. [Issue] **Closed by** [PR 300]
- Add automatic tool to check security issues with Rust dependencies (cargo-audit) [Issue], **Closed by** [PR 301].

15.4 Offset mobile app

- Update compiler and dependencies
- Better colors for the Offset mobile app (Should be consistent with website)
- **Choose better defaults**
  - Card should be enabled when created?
  - Friend should be open when created?
  - Currency should be open when created?
- Allow sending Offset data as text? (Currently can send as files or QR codes)

- Merge incoming/outgoing transactions into one screen (Requested by awpcrypto)
- Keep purchase history (Incoming/Outgoing) forever.
- Add transaction time and date (stcompact)
- Password for locking/unlocking the Offset app.
- **Support Importing/Exporting**
  - Nodes (Private key + database)
  - Transactions history?
- Make it easier to add friends, while still keeping it safe?
- Implement bill splitting using Offset? [Issue]
- Include different assets for different architectures during build in an elegant way (Currently done using a bash script).
- Add mobile app to F-Droid store [Issue]

15.5 Documentation

- High quality video tutorials
- Document multi-currency feature.
- Document security considerations: What makes Offset secure?
- Add FAQ page

15.6 Community

- Add a discussion group / bulletin board for the project.
- Create a dummy website to check payments with a dummy currency.

CHAPTER

SIXTEEN

CONTRIBUTING

The Offset project needs your support!

16.1 Project scope

Offset is an open source project, currently split between a few repositories:

- Offset core, containing the core protocol and logic behind Offset.
- Offset mobile app, containing mostly UI code for the Offset mobile app.
- Offset inside docker, dockerized versions of Offset entities.
- Offset documentation, containing Offset documentation (The document you are reading right now)

16.2 How can I help?

- Development: Offset is far from complete. There are many features to add and many bugs to fix. Check out the issues pages on the Offset repositories and see if there is any issue you would like to help solve.
- Bug reports: If you find any issue with Offset, or have an interesting idea, don't hesitate to open an issue here at the relevant repository.
- Documentation: If you find a mistake in the documentation, or have something to add, don't hesitate to fork the documentation repository and send us a pull request.
- Design: The Offset website and the Offset app user interface are still a work in progress. If you have the skills to make it look better, don't hesitate to contact us.
- Spread the message: Show the Offset app to your friends, and teach them how to use it.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search