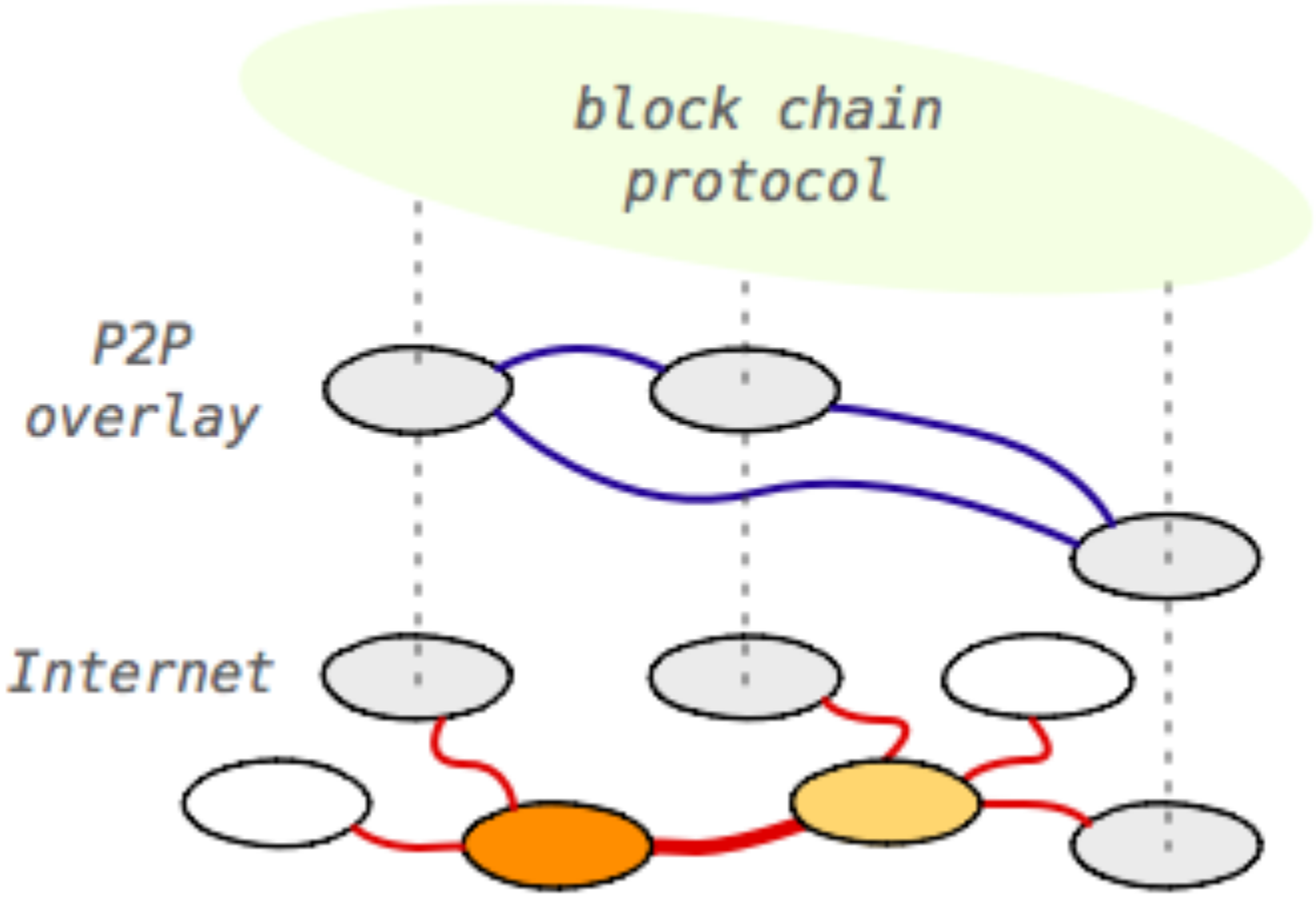
Minimum Viable Block Chain

*By Ilya Grigorik (/) on **May 05, 2014***



Cryptocurrencies, and Bitcoin in particular, have been getting a lot of attention from just about every angle: regulation, governance, taxation, technology, product innovation, and the list goes on. The very concept of a "peer-to-peer (decentralized) electronic cash system" turns many of our previously held assumptions about money and finance on their head.

That said, putting the digital currency aspects aside, an arguably even more interesting and far-reaching innovation is the underlying block chain technology. Regardless of what you think of Bitcoin, or its altcoin derivatives (https://en.bitcoin.it/wiki/List_of_alternative_cryptocurrencies), as a currency and a store of value, behind the scenes they are all operating on the same basic block chain principles outlined by Satoshi Nakamoto (https://bitcoin.org/bitcoin.pdf):

> *We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power... The network itself requires minimal structure.*

The block chain is agnostic to any "currency". In fact, it can (and will) be adapted to power many other use cases. As a result, it pays to understand the how and the why behind the "minimum viable block

chain":

- **What follows is not** an analysis of the Bitcoin block chain. In fact, I intentionally omit mentioning both the currency aspects, and the many additional features that the Bitcoin block chain is using in production today.
- **What follows is** an attempt to explain, from the ground up, why the particular pieces (digital signatures, proof-of-work, transaction blocks) are needed, and how they all come together to form the "minimum viable block chain" with all of its remarkable properties.

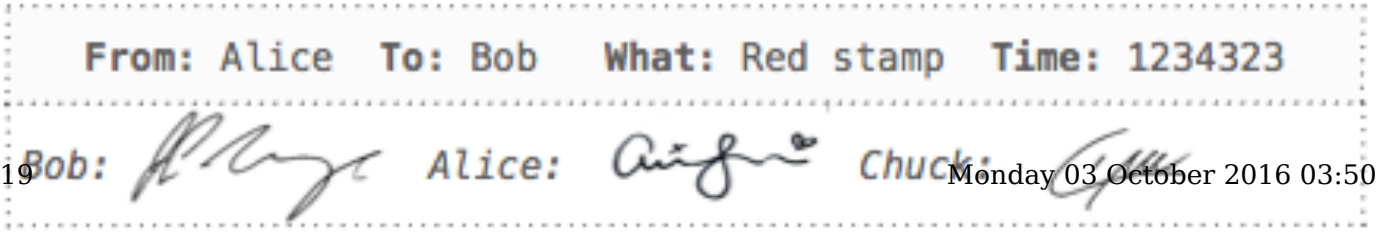
I have learned long ago that writing helps me refine my own sloppy thinking, hence this document. Primarily written for my own benefit, but hopefully helpful to someone else as well. Feedback is always welcome, leave a comment below!

Securing transactions with triple-entry bookkeeping # ↑

Alice and Bob are stamp collectors. It's nothing serious, and they're mostly in it for the social aspects of meeting others, exchanging stories, and doing an occasional trade. If both parties see something they like, they negotiate right there and then and complete the swap. In other words, it's a simple barter system (https://en.wikipedia.org/wiki/Barter).

Then, one day Bob shows up with a stamp that Alice feels she absolutely must have in her collection. Except there is a problem, because Bob is not particularly interested in anything that Alice has to offer. Distraught, Alice continues negotiating with Bob and they arrive at a solution: they'll do a one-sided transaction where Bob will give Alice his stamp and Alice will promise to repay him in the future.

Both Bob and Alice have known each other for a while, but to ensure that both live up to their promise (well, mostly Alice), they agree to get their transaction "notarized" by their friend Chuck.



They make three copies (one for each party) of the above transaction receipt indicating that Bob gave Alice a "Red stamp". Both Bob and Alice can use their receipts to keep account of their trade(s), and Chuck stores his copy as evidence of the transaction. Simple setup but also one with a number of great properties:

1. **Chuck can authenticate both Alice and Bob** to ensure that a malicious party is not attempting to fake a transaction without their knowledge.
2. **The presence of the receipt in Chuck's books is proof of the transaction.** If Alice claims the transaction never took place then Bob can go to Chuck and ask for his receipt to disprove Alice's claim.
3. **The absence of the receipt in Chuck's books is proof that the transaction never took place.** Neither Alice nor Bob can fake a transaction. They may be able to fake their copy of the receipt and claim that the other party is lying, but once again, they can go to Chuck and check his books.
4. **Neither Alice nor Bob can tamper with an existing transaction.** If either of them does, they can go to Chuck and verify their copies against the one stored in his books.

What we have above is an implementation of "triple-entry bookkeeping", which is simple to implement and offers good protection for both participants. Except, of course you've already spotted the weakness, right? We've placed a lot of trust in an intermediary. If Chuck decides to collude with either party, then the entire system falls apart.

Moral of the story? **Be (very) careful about your choice of the intermediary!**

Securing transactions with PKI

↑

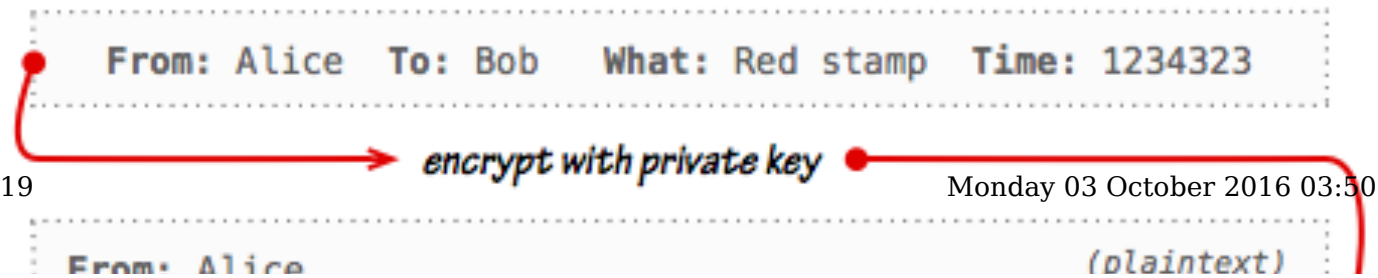
Dissatisfied with the dangers of using a "reliable intermediary", Bob decides to do some research and discovers that public key cryptography can eliminate the need for an intermediary! This warrants some explanation...

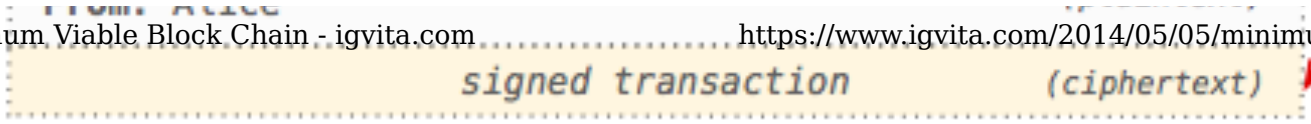
> *Public-key cryptography, also known as asymmetric cryptography, refers to a cryptographic algorithm which requires two separate keys, one of which is secret (or private) and one of which is public. Although different, the two parts of this key pair are mathematically linked. The public key is used to encrypt plaintext or to verify a digital signature; whereas the private key is used to decrypt ciphertext or to create a digital signature.*
>
> *https://en.wikipedia.org/wiki/Public-key_cryptography*

The original intent behind using a third party (Chuck) was to ensure three properties:

- **Authentication:** a malicious party can't masquerade as someone else.
- **Non-repudiation:** participants can't claim that the transaction did not happen after the fact.
- **Integrity:** the transaction receipt can't be modified after the fact.

Turns out, public key cryptography can satisfy all of the above requirements. Briefly, the workflow is as follows:





1. Both Alice and Bob generate a set public-private keypairs.
2. Both Alice and Bob publish their public keys to the world.
3. Alice writes a transaction receipt in plaintext.
4. Alice encrypts the plaintext of the transaction using her private key.
5. Alice prepends a plaintext "signed by" note to the ciphertext.
6. Both Alice and Bob store the resulting output.

Note that step #5 is only required when many parties are involved: if you don't know who signed the message then you don't know whose public key you should be using to decrypt it. This will become relevant very soon...

This may seem like a lot of work for no particular reason, but let's examine the properties of our new receipt:

1. Bob doesn't know Alice's private key, but that doesn't matter because he can look up her public key (which is shared with the world) and use it to decrypt the ciphertext of the transaction.
2. Alice is not really "encrypting" the contents of the transaction. Instead, by using her private key to encode the transaction she is "signing it": anyone can decrypt the ciphertext by using her public key, and because she is the only one in possession of the private key this mechanism guarantees that only she could have generated the ciphertext of the transaction.

How does Bob, or anyone else for that matter, get Alice's public key? There are many ways to handle distribution of public keys - e.g. Alice publishes it it on her website. We'll assume that some such mechanism is in place.

As a result, the use of public key infrastructure (PKI) fulfills all of our earlier requirements:

1. Bob can use Alice's public key to authenticate signed transaction by decrypting the ciphertext.
2. Only Alice knows her private key, hence Alice can't deny that the transaction took place - she signed it.
3. Neither Bob nor anyone else can fake or modify a transaction without access to Alice's private key.

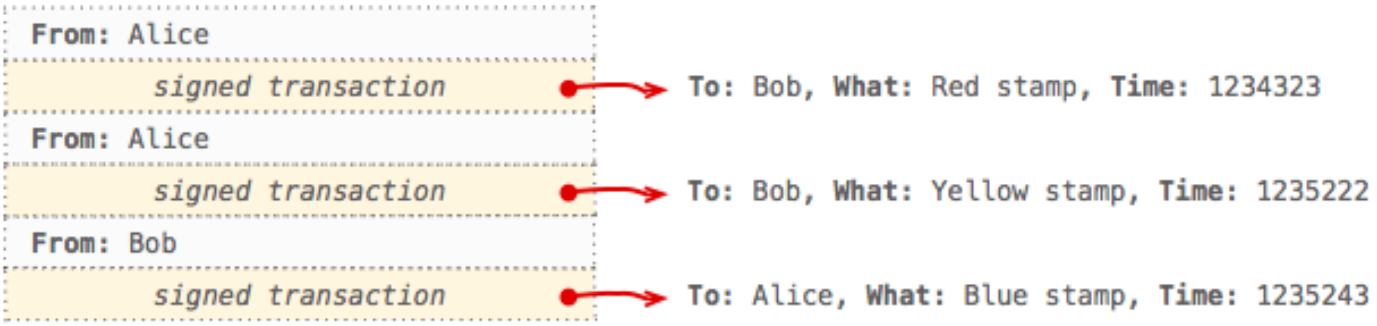
Note that for #2, Alice can deny that she is the true owner of the public-private keypair in question - i.e. someone is faking her identity in #1. The keypair to identity association is something your key distribution mechanisms needs to account for.

Both Alice and Bob simply store a copy of the signed transaction and the need for an intermediary is eliminated. The "magic" of public key cryptography is a perfect match for their two-party barter system.

Balance = Σ(receipts) # ↑

With the PKI infrastructure in place, Bob and Alice complete a few additional trades: Alice acquires another stamp from Bob and Bob picks up a stamp from Alice. They each follow the same steps as before to

generate signed transactions and append them to their respective ledgers.



The records are secure, but there is a small problem: it's not clear if either party has an outstanding balance. Previously, with just one transaction, it was clear who owed whom (Alice owed Bob) and how much (one red stamp), but with multiple transactions the picture gets really murky. Are all stamps of equal value? If so, then Alice has a negative balance. If not, then it's anyone's guess! To resolve this, Alice and Bob agree on the following:

- Yellow stamp is worth twice the value of a red stamp.
- Blue stamp is equal in value to a red stamp.

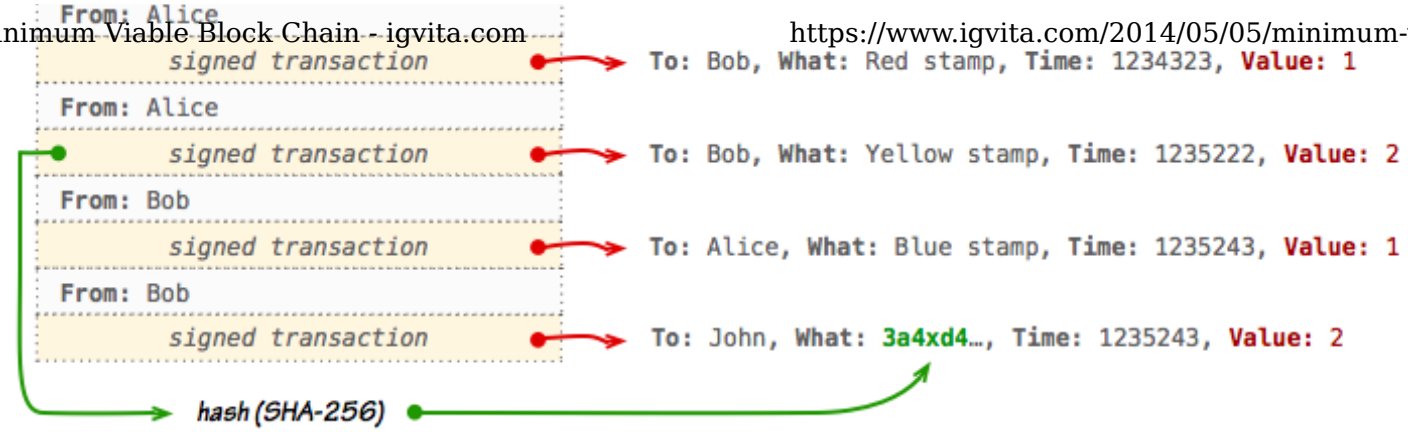
Finally, to ensure that their new agreement is secure they regenerate their ledgers by updating each transaction with its relative value. Their new ledgers now look as follows:

With that, computing the final balance is now a simple matter of iterating through all of the transactions and applying the appropriate debits and credits to each side. The net result is that Alice owes Bob 2... *units of value*. What's a "unit of value"? It's an arbitrary medium of exchange (https://en.wikipedia.org/wiki/Medium_of_exchange) that Alice and Bob have agreed on. Further, since "unit of value" doesn't roll off the tongue, Alice and Bob agree to call 1 unit of value as 1 *chroma* (plural: *chroms*).

All of the above seems trivial, but **the fact that the balance of each party is a function of all of the receipts in the ledger has an important consequence: anyone can compute everyone's balance.** There is no need for any trusted intermediaries and the system is trivial to audit. Anyone can traverse the full ledger, verify the trades, and figure out the outstanding balances of each party.

Multi-party transfers & verification # ↑

Next, Bob stumbles across a stamp owned by John that he really likes. He tells John about the secure ledger he is using with Alice and asks him if he would be willing to do a trade where Bob transfers his balance with Alice as a method of payment - i.e. Bob gets the stamp from John, and Alice would owe John the amount she previously owed Bob. John agrees, but now they have dilemma. How exactly does Bob "transfer" his balance to John in a secure and verifiable manner? After some deliberation, they arrive at an ingenious plan:



Bob creates a new transaction by following the same procedure as previously, except that he first computes the SHA-256 checksum (a unique fingerprint) of the encrypted transaction he wants to transfer and then inserts the checksum in the "What" field of the new receipt. In effect, he is linking the new transfer receipt to his previous transaction with Alice, and by doing so, transfers its value to John.

> *To keep things simple, we'll assume that all transfers "spend" full value of the transaction being transfered. It's not too hard to extend this system to allow fractional transfers, but that's unnecessary complexity at this point.*

With the new transaction in place, John makes a copy of the encrypted ledger for his safekeeping (now there are three copies) and runs some checks to verify its integrity:

1. John fetches Alice's and Bob's public keys and verifies the first three transactions.
2. John verifies that Bob is transferring a "valid" transaction:
    - The transaction that is being transferred is addressed to Bob.
    - Bob has not previously transfered the same transaction to anyone else.

If all the checks pass, they complete the exchange and we can compute the new balances by traversing the ledger: Bob has a net zero balance, Alice has a debit of 2 chroms, and John has a credit of 2 chroms (courtesy of Alice). Further, John can now take his new ledger to Alice and ask her for payment, and even though Alice wasn't present for their transaction, that's not a problem:

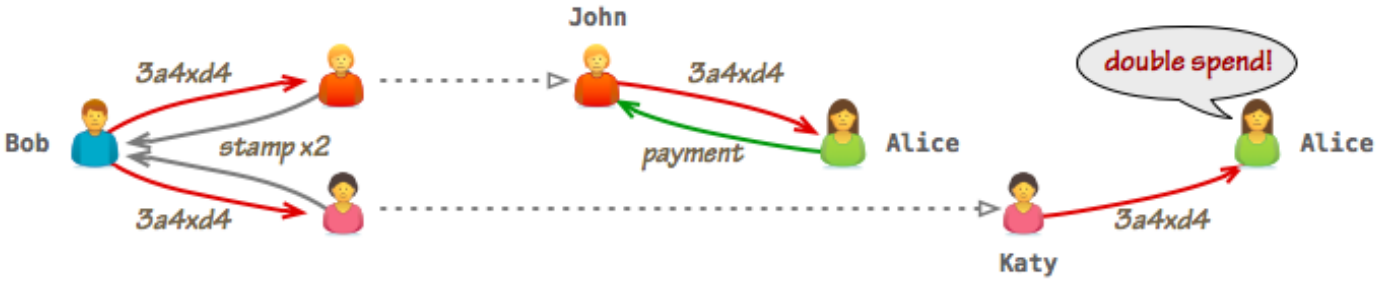
- Alice can verify the signature of the new transfer transaction using Bob's public key.
- Alice can verify that the transfer transaction is referencing one of her own valid transactions with Bob.

> *The above transfer and verification process is a pretty remarkable property of the system! Note that to make it all work, we need two enabling technologies: (a) use of PKI, which enables digital signature verification, and (b) the receipt ledger, which enables us to look at the full transaction history to verify balances and to link previous transactions to enable the transfer.*

Satisfied with their ingenuity John and Bob part ways: Bob goes home with a new stamp and John with a new ledger. On the surface, everything looks great, but **they've just exposed themselves to a challenging security problem... Can you spot it?**

Double-spending and distributed consensus

Shortly after completing the transaction with John, Bob realizes that they have just introduced a critical flaw into their system and one that he could exploit to his advantage if he acts quickly: both Bob and John have updated their ledgers to include the new transaction, but neither Alice nor anyone else is aware that it has taken place. As a result, **there is nothing stopping Bob from approaching other individuals in his network and presenting them with an old copy of the ledger that omits his transaction with John!** If he convinces them to do a transaction, just as he did with John, then he can "double-spend (https://en.wikipedia.org/wiki/Double-spending)" the same transaction as many times as he wants!



Of course, once multiple people show up with their new ledgers and ask Alice for payment, the fraud *will* be detected, but that is of little consolation - Bob has already run away with his loot!

The double-spend attack was not possible when we only had two participants since in order to complete the transaction you'd verify and update both sets of books simultaneously. As a result, all ledgers were always in sync. However, the moment we added an extra participant we introduced the possibility of incomplete and inconsistent ledgers between all the participants, which is why the double-spend is now possible.

In CS speak, a two-party ledger provides "strong consistency", and growing the ledger beyond two parties requires some form of distributed consensus (https://en.wikipedia.org/wiki/Consensus_(computer_science)) to mitigate double-spend.

The simplest possible solution to this problem is to require that all parties listed in the ledger must be present at the time when each new transaction is made, such that everyone can update their books simultaneously. An effective strategy for a small-sized group, but also not a scalable one for a large number of participants.

Requirements for a distributed consensus network # ↑

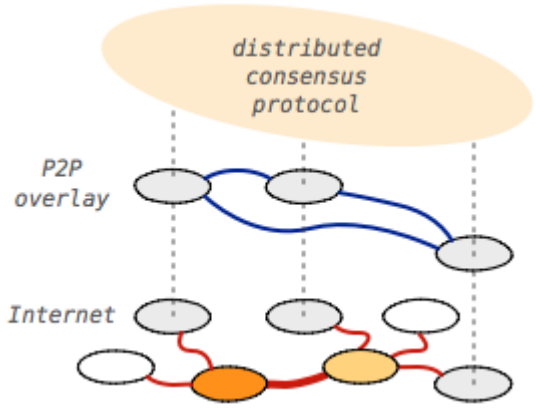
Let's imagine that we want to scale our ledger to all stamp collectors around the globe such that anyone can trade their favorite stamps in a secure manner. Obviously, requiring that every participant must be present to register each transaction would never work due to geography, timezones, and other limitations. Can we build a system where we don't need everyone's presence and approval?

1. Geography is not really an issue: we can move communication online.
2. Timezone problems can be solved with software: we don't need each individual to manually update their ledgers. Instead, we can build software that can run on each participant's computer and automatically receive, approve, and add transactions to the ledger on their behalf.

In effect, we could build a peer-to-peer (P2P) network that would be responsible for distributing new transactions and getting everyone's approval! Except, unfortunately that's easier said than done in practice. For example, **while a P2P network can resolve our geography and timezone problems, what happens when even just one of the participants goes offline?** Do we block all transactions until they're back

online?

Note that the "how" of building a P2P network is a massive subject in its own right: protocols and signaling, traversing firewalls and NATs, bootstrapping, optimizing how updates are propagated, security, and so on. That said, the low-level mechanics of building such a network are out of scope of our discussion... we'll leave that as an exercise for the reader.



Turns out, distributed consensus (https://en.wikipedia.org/wiki/Consensus_(computer_science)) is a well studied problem in computer science, and one that offers some promising solutions. For example, two-phase commit (https://en.wikipedia.org/wiki/Two-phase_commit_protocol) (2PC) and Paxos (https://en.wikipedia.org/wiki/Paxos_(computer_science)) both enable a mechanism where we only need the majority quorum (50%+) of participants to be present to safely commit a new transaction: as long as the majority has accepted the transaction the remainder of the group is guaranteed to eventually converge on the same transaction history.

That said, neither 2PC nor Paxos are sufficient on their own. For example, how would either 2PC or Paxos know the total number of participants in our P2P stamp-collector network when new participants are joining on a daily basis and others are disappearing without notice? If one of the prior participants is offline, are they offline temporarily, or have they permanently left the network? Similarly, there is another and an even more challenging "Sybil attack (https://en.wikipedia.org/wiki/Sybil_attack)" that we must account for: there is nothing stopping a malicious participant from creating many profiles to gain an unfair share of voting power within our P2P network.

If the number of participants in the system was fixed and their identities were authenticated and verified (i.e. a trusted network), then both 2PC and Paxos would work really well. Alas, that is simply not the case in our ever changing stamp collector P2P network. Have we arrived at a dead end? Well, not quite...

One obvious solution to solve our dilemma is to eliminate the "distributed" part from the problem statement. Instead of building a P2P distributed system we could, instead, build a global registry of all stamp collectors that would record their account information, authenticate them and (try to) ensure that nobody is cheating by creating multiple identities, and most importantly, keep one shared copy of the ledger! Concretely, **we could build a website where these trades can take place, and the website would then take care of ensuring the integrity and correct ordering of all transactions by recording them in its centralized database.**

The above is a practical solution but, let's admit it, an unsatisfying one since it forces us to forfeit the peer-to-peer nature of our ledger system. It places all of the trust in a single centralized system and opens up an entirely new set of questions: what is the uptime, security and redundancy of the system; who maintains the system and what are their incentives; who has administrative access, and so on. **Centralization brings its own set of challenges.**

Let's rewind and revisit some of the problems we've encountered with our P2P design:

- Ensuring that every participant is always up to date (strongly consistent system) imposes high coordination costs and affects availability: if a single peer is unreachable the system cannot commit new transactions.

- In practice we don't know the global status of the P2P network: number of participants, whether individuals are temporarily offline or decided to leave the network, etc.
- Assuming we can resolve the above constraints, the system is still open to a Sybil attack where a malicious user can create many fake identities and exercise unfair voting power.

Unfortunately, resolving all of the above constraints is impossible unless we relax some of the requirements: the CAP theorem (https://en.wikipedia.org/wiki/CAP_theorem) tells us that our distributed system can't have strong consistency, availability, and partition tolerance. As a result, in practice our P2P system must operate under the assumption of weak(er) consistency (http://www.igvita.com/2010/06/24/weak-consistency-and-cap-implications/) and deal with its implications:

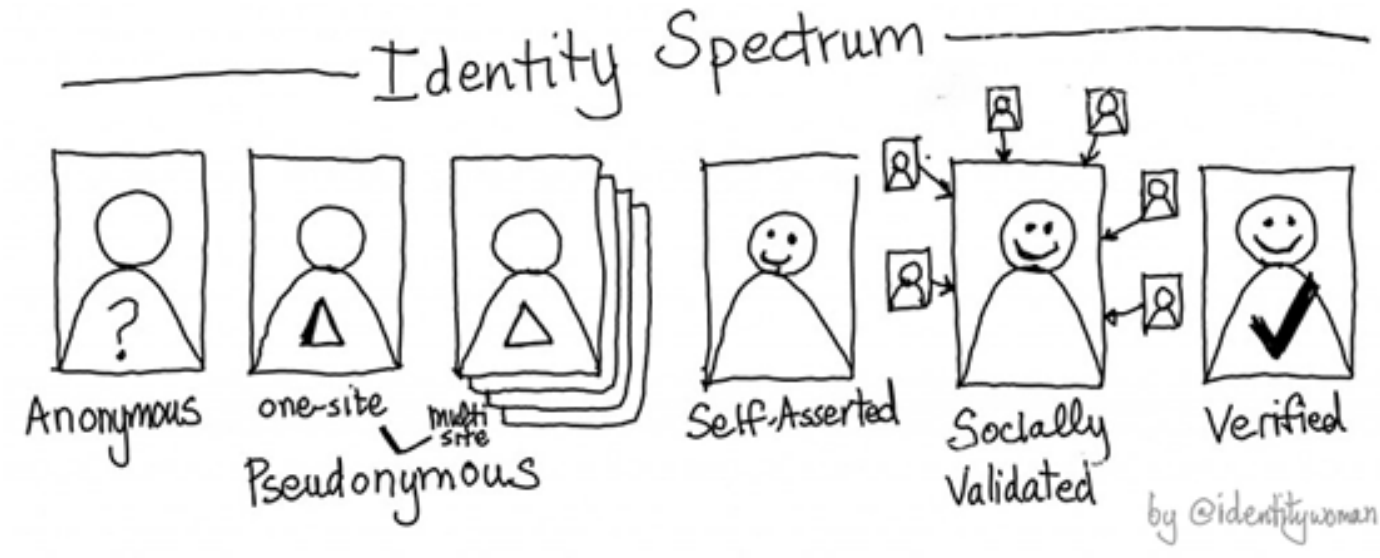
- We must accept that some ledgers will be out of sync (at least temporarily).
- The system must eventually converge on a global ordering (linearizability) of all transactions.
- The system must resolve ledger conflicts in a predictable manner.
- The system must enforce global invariants - e.g. no double-spends.
- The system should be secure against Sybil and similar attacks.

Protecting the network from Sybil attacks

↑

Achieving consensus in a distributed system, say by counting votes of each participant, opens up many questions about the "voting power" of each peer: who is allowed to participate, do certain peers have more voting power, is everyone equal, and how do we enforce these rules?

To keep things simple, let's say everyone's vote is equal. As a first step, we could require that each participant sign their vote with their private key, just as they would a transaction receipt, and circulate it to their peers - signing a vote ensures that someone else can't submit a vote on their behalf. Further, we could make a rule that only one vote is allowed to be submitted. If multiple votes are signed by the same key then all of them are discarded - make up your mind already! So far so good, and now the hard part...



(http://www.identitywoman.net/the-identity-spectrum)

How do we know if any particular peer is allowed to participate in the first place? If all that's needed is just a unique private key to sign a vote, then a malicious user could simply generate an unlimited number of new keys and flood the network. The root problem is that **when forged identities are cheap to generate and use, any voting system is easily subverted.**

To solve this problem we need to make the process of submitting the vote "expensive". Either the

cost of generating a new identity must be raised, or the very process of submitting a vote must incur sufficiently high costs. To make this concrete, consider some real-world examples:

- When you show up to vote in your local government election, you are asked to present an ID (e.g. a passport) that is (hopefully) expensive to fake. In theory, nothing stops you from generating multiple fake IDs, but if the costs are high enough (monetary costs of making a fake, risk of being caught, etc), than the cost of running a Sybil attack will outweigh its benefits.
- Alternatively, imagine that you had to incur some other cost (e.g. pay a fee) to submit a vote. If the cost is high enough, then once again, the barrier to running a large-scale Sybil attack is increased.

Note that neither of the above examples "solves" the Sybil attack completely, but they also don't need to: as long as we raise the cost of the attack to be larger than the value gained by successfully subverting the system, then the system is secure and behaves as intended.

> *Note that we're using a loose definition of "secure". The system is still open for manipulation, and the exact vote count is affected, but the point is that a malicious participant doesn't affect the final outcome.*

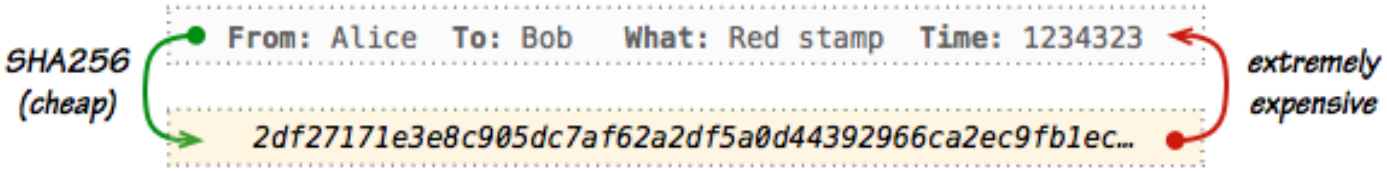
Proof-of-work as a participation requirement # ↑

Any user can easily (and cheaply) generate a new "identity" in our P2P network by generating a new private-public keypair. Similarly, any user can sign a vote with their private key and send it into the P2P network - that's also cheap, as the abundance of spam email in our inboxes clearly illustrates. Hence, submitting new votes is cheap and a malicious user can easily flood the network with as many votes as they wish.

However, what if we made one of the steps above expensive such that you had to expend significantly more effort, time, or money? That's the core idea behind requiring a proof-of-work (https://en.wikipedia.org/wiki/Proof-of-work_system):

1. The proof-of-work step should be "expensive" for the sender.
2. The proof-of-work step should be "cheap" to verify by everyone else.

There are many possible implementations of such a method, but for our purposes we can re-use the properties provided by the cryptographic hash functions we encountered earlier:



1. It is easy to compute the hash value for any given message.
2. It is expensive to generate a message that has a given hash value.

We can impose a new rule in our system requiring that every signed vote must have a hash value that begins with a particular substring - i.e. require a partial hash collision of, say, two zero prefix. If this seems completely arbitrary, that's because it is - stay with me. Let's walk through the steps to see how this works:

1. Let's say a valid vote statement is a simple string: *"I vote for Bob"*.
2. We can use the same SHA-256 algorithm to generate a hash value (http://www.xorbin.com/tools

/sha256-hash-calculator) for our vote.
    - `sha256("I vote for Bob") → b28bfa35bcd071a321589fb3a95cac...`
3. The resulting hash value is invalid because it does not start with our required substring of two zeros.
4. We modify the vote statement by appending an arbitrary string and try again:
    - `sha256("I vote for Bob - hash attempt #2") → 7305f4c1b1e7...`
5. The resulting hash value does not satisfy our condition either. We update the value and try again, and again, and... 155 attempts later we finally get:
    - `sha256("I vote for Bob - hash attempt #155") → 008d08b8fe...`

The critical property of the above workflow is that the output of the cryptographic hash function (SHA-256 in this case) is completely different every time we modify the input: the hash value of the previous attempt does not tell us anything about what the hash value of the next attempt when we increment our counter. **As a result, generating a valid vote is not just "hard problem", but also one better described as a lottery where each new attempt gives you a random output.** Also, we can adjust the odds of the lottery by changing the length of the required prefix:

1. Each character of the SHA-256 checksum has 16 possible values: *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f.*
2. In order to generate a hash with a valid two zero prefix the sender will need 256 (16^2) attempts on average.
3. Bumping the requirement to 5 zeros will require more than 1,000,000 (16^5) attempts on average... Point being, we can easily increase the cost and make the sender spend more CPU cycles to find a valid hash.

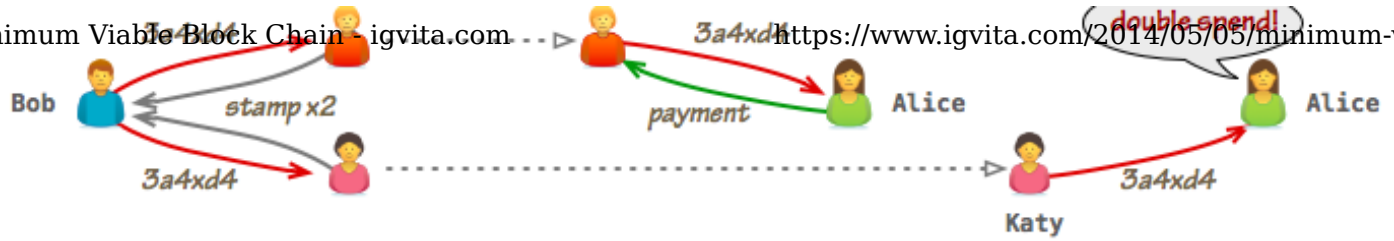
> *How many SHA256 checksums can we compute on a modern CPU? The cost depends on the size of the message, CPU architecture, and other variables. If you're curious, open up your console and run a benchmark: `$> openssl speed sha` .*

The net result is that generating a valid vote is "expensive" for the sender, but is still trivial to verify for the receiver: the receiver hashes the transaction (one operation) and verifies that the checksum contains the required hash collision prefix... Great, so how is this useful for our P2P system? **The above proof-of-work mechanism allows us to adjust the cost of submitting a vote such that the total cost of subverting the system (i.e. spoofing enough valid votes to guarantee a certain outcome) is higher than the value gained by attacking the system.**

> *Note that the "high cost to generate a message" is a useful property in many other contexts. For example, email spam works precisely because it is incredibly cheap to generate a message. If we could raise the cost of sending an email message - say, by requiring a proof-of-work signature - then we could break the spam business model by raising costs to be higher than profits.*

Building the minimum viable block chain # ↑

We've covered a lot of ground. Before we discuss how the block chain can help us build a secure distributed ledger, let's quickly recap the setup, the properties, and the unsolved challenges within of our network:



1. Alice and Bob complete a transaction and record it in their respective ledgers.
    1. Once done, Bob has a PKI-protected IOU from Alice.
2. Bob completes a transaction with John where he transfers Alice's IOU to John. Both Bob and John update their ledgers, but Alice doesn't know about the transaction... yet.
    1. **Happy scenario:** John asks Alice to redeem his new IOU; Alice verifies his transaction by fetching Bob's public key; if the transaction is valid she pays John the required amount.
    2. **Not so happy scenario:** Bob uses his old ledger that omits his transaction with John to create a double-spend transaction with Katy. Next, both Katy and John show up at Alice's doorstep and realize that only one of them will get paid.

The double-spend is possible due to the "weak consistency" of the distributed ledger: neither Alice nor Katy know about John and Bob's transaction, which allows Bob to exploit this inconsistency to his advantage. Solution? If the network is small and all participants are known, we can require that each transaction must be "accepted" by the network before it is deemed valid:

- **Unanimous consensus:** whenever a transaction takes place the two parties contact all other participants, tell them about the transaction, and then wait for their "OK" before they commit the transaction. As a result, all of the ledgers are updated simultaneously and double-spend is no longer possible.
- **Quorum consensus:** to improve processing speed and availability of the network (i.e. if someone is offline, we can still process the transaction) we can relax the above condition of unanimous consensus to quorum consensus (50% of the network).

Either of the above strategies would solve our immediate problem for a small network of known and verified participants. However, neither strategy scales to a larger, dynamic network where neither the total number of participants is known at any point in time, nor their identity:

1. We don't know how many people to contact to get their approval.
2. We don't know whom to contact to get their approval.
3. We don't know whom we are calling.

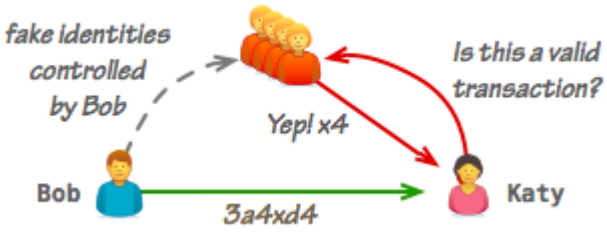
> *Note that we can use any means of communication to satisfy above worfklow: in person, internet, avian carriers (http://www.ietf.org/rfc/rfc1149.txt), etc!*

Lacking identity and global knowledge of all the participants in the network we have to relax our constraints. **While we can't guarantee that any particular transaction is valid, that doesn't stop us from making a statement about the probability of a transaction being accepted as valid:**

- **Zero-confirmation transaction:** we can accept a transaction without contacting any other participants. This places full trust on the integrity of the payer of the transaction - i.e. they won't double-spend.
- **N-confirmation transaction:** we can contact some subset of the (known) participants in the

network and get them to verify our transaction. The more peers we contact, the higher the probability that we will catch malicious parties attempting to defraud us.

What is a good value for "N"? The answer depends on the amount being transferred and your trust and relationship with the opposite party. If the amount is small, you may be willing to accept a higher level of risk, or, you may adjust your risk tolerance based on what you know about the other party. Alternatively, you will have to do some extra work to contact other participants to validate your transaction. In either case, there is a tradeoff between the speed with which the transaction is processed (zero-confirmation is instant), the extra work, and the risk of that transaction being invalid.



So far, so good. Except, there is an additional complication that we must consider: our system relies on transaction confirmations from other peers, but nothing stops a malicious user from generating as many fake identities as needed (recall that an "identity" in our system is simply a public-private keypair, which is trivial to generate) to satisfy Katy's acceptance criteria.

Whether Bob decides to execute the attack is a matter of simple economics: if the gain is higher than the cost then he should consider running the attack. Conversely, if Katy can make the cost of running the attack higher than the value of the transaction, then she should be safe (unless Bob has a personal vendetta and/or is willing to lose money on the transaction... but that's out of scope). To make it concrete, let's assume the following:

- Bob is transferring 10 chroms to Katy.
- The cost of generating a fake identity and transaction response is 0.001 chroms: energy costs to keep the computer running, paying for internet connectivity, etc.

If Katy asks for 1001 confirmations, then it no longer makes (economic) sense for Bob to run the attack. Alternatively, **we could add a proof-of-work requirement for each confirmation and raise the cost for each valid response from 0.001 chroms to 1:** finding a valid hash will take CPU time, which translates to a higher energy bill. As a result, Katy would only need to ask for 11 confirmations to get the same guarantee.

Note that Katy also incurs some costs while requesting each confirmation: she has to expend effort to send out the requests and then validate the responses. Further, if the cost of generating a confirmation and verifying it is one-to-one, then Katy will incur the same total cost to verify the transaction as its value... which, of course, makes no economic sense.

This is why the asymmetry of proof-of-work is critical. Katy incurs low cost to send out requests and validate the responses, but the party generating the confirmation needs to expend significantly more effort to generate a valid response.

Great, problem solved, right? Sort of... in the process we've created another economic dilemma. **Our network now incurs a cost to validate each transaction that is of equal or higher value than the transaction itself.** While this acts as an economic deterrent against malicious participants, why would any legitimate participant be willing to incur any costs for someone else? A rational participant simply wouldn't, it doesn't make sense. Doh.

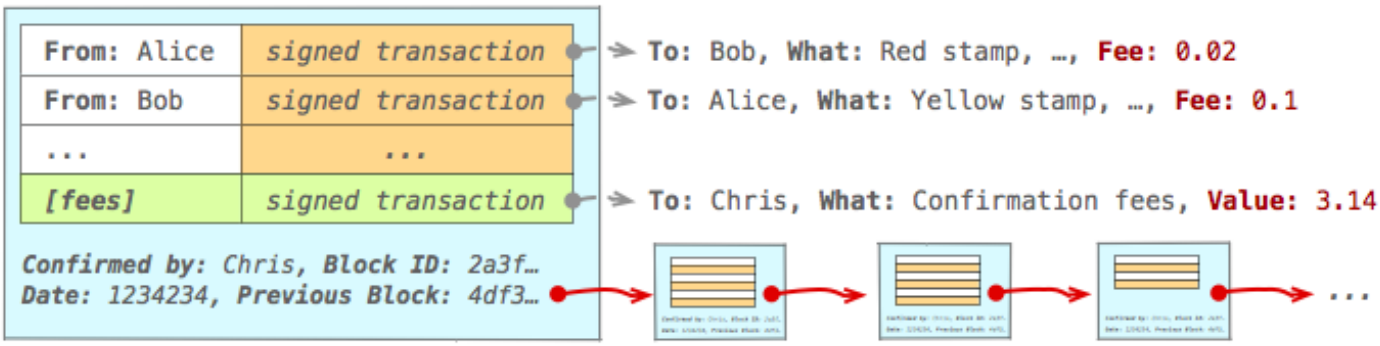
Adding "blocks" & transaction fee incentives

If participants in the network must incur a cost to validate each other's transactions, then we must provide an economic incentive for them to do so. In fact, at a minimum we need to offset their costs, because otherwise an "idle" participant (anyone who is not submitting their own transactions) would continue accruing costs on behalf of the network — that wouldn't work. Also a couple of other problems that we need address:

1. If the cost of verifying the transaction is equal to or higher than the value of the transaction itself (to deter malicious participants), than the total transaction value is net-zero, or negative! E.g. Bob transfers 10 chroms to Katy; Katy spends 10 chroms to compensate other peers to validate the transaction; Katy is sad.
2. How does Katy pay for confirmations? If that's its own transaction then we have a recursive problem.

Let's start with the obvious: the transaction fee can't be as high as the value of the transaction itself. Of course, Katy doesn't have to spend the exact value to confirm the transaction (e.g. she can allocate half the value for confirmations), but then it becomes a question of margins: if the remaining margin (value of transaction - verification fees) is high enough, than there is still incentive for fraud. Instead, ideally we would like to incur the lowest possible transaction fees and still provide a strong deterrent against malicious participants. Solution?

We can incentivize participants in the network to confirm transactions by allowing them to pool and confirm multiple transactions at once - i.e. confirm a "block" of transactions. Doing so would also allow them to aggregate transaction fees, thereby lowering the validation costs for each individual transaction.



A block is simply a collection (one or more) of valid transactions - think of it as the equivalent of a page in a physical ledger. In turn, each block contains a reference to a previous block (previous page) of transactions, and the full ledger is a linked sequence of blocks. Hence, **block chain**. Consider the example above:

1. Alice and Bob generate new transactions and announces them to the network.
2. Chris is listening for new transaction notifications, each of which contains a transaction fee that the sender is willing to pay to get it validated and confirmed by the network:
    1. Chris aggregates unconfirmed transactions until he has a direct financial incentive (sum of transaction fees > his cost) to perform the necessary work to validate the pending transactions.
    2. Once over the threshold, Chris first validates each pending transaction by checking that none of the inputs are double-spent.
    3. Once all transactions are validated Chris adds an extra transaction to the pending list (indicated in green in the diagram above) that transfers the sum of advertised transaction fees to himself.
    4. Chris generates a block that contains the list of pending transactions, a reference to the

previous block (such that we can traverse the blocks and see the full ledger) and performs the proof-of-work challenge to generate a block hash value that conforms to accepted rules of the network - e.g. partial hash collision of *N* leading zeros.
5. Finally, once Chris finds a valid block, he distributes it to all other participants.

3. Both Alice and Bob are listening for new block announcements and look for their transaction in the list:
    1. Alice and Bob verify integrity of the block - i.e. verify proof-of-work and contained transactions.
    2. If the block is valid and their transaction is in the list, then the transaction has been confirmed!

We made a big leap here. Previously we've only had one type of record in our network - the signed transaction. Now we have signed transactions and blocks. The former is generated by the individuals engaging in trade, and the latter is generated by parties interested in collecting fees by validating and confirming transactions.

Also, note that the above scheme requires some minimum volume of transactions in the system to sustain the incentives for individuals creating the blocks: the more transactions there are, the lower the fees have to be for any single transaction.

Phew, ok, Alice has announced a new transaction and received a valid block from Chris confirming it. That's one confirmation, what about the rest? Also, Chris is (hopefully) not the only participant who is incentivized to work on generating the blocks. What if someone else generates a different block at the same time, and which of those blocks is "valid"? This is where it gets interesting...

Racing to claim the transaction fees # ↑

The remarkable part about introducing the ability to aggregate fees by verifying a block of transactions is that it creates a role for a new participant in the network who now has a direct financial incentive to secure it. You can now make a profit by validating transactions, and where there is profit to be made, competition follows, which only strengthens the network - a virtuous cycle and a clever piece of social engineering!

That said, the incentive to compete to validate transactions creates another interesting dilemma: how do we coordinate this block generation work in our distributed network? The short answer is, as you may have already guessed, we don't. Let's add some additional rules into our system and examine how they resolve this problem:

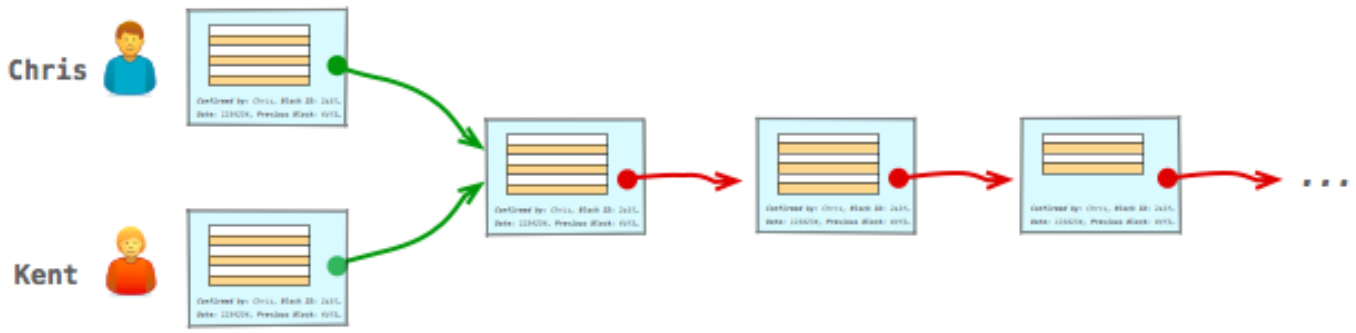
1. Any number of participants is allowed to participate ("race") to create a valid block. There is no coordination. Instead, each interested participant listens for new transactions and decides whether and when they want to try to generate a valid block and claim the transaction fees.
2. When a valid block is generated, it is immediately broadcast into the network.
    1. Other peers check the validity of the block (check each transaction and validity of the block itself), and if valid, add it to their ledgers and then finally rebroadcast it to other peers in the network.
    2. Once added, the new block becomes the "topmost block" of their ledger. As a result, if that same peer was also working on generating a block, then they need to abort their previous work and start over: they now need to update their reference to the latest block and also remove any transactions from their unconfirmed list that are contained in the latest block.

3. Once the above steps are complete, they start working on a new block, with the hope that they'll be the first ones to discover the next valid block, which would allow them to claim the transaction fees.

3. ... repeat the above process until the heat death of the universe.

The lack of coordination between all the participants working on generating the blocks means there will be duplicate work in the network, and that's OK! **While no single participant is guaranteed to claim any particular block, as long as the expected value (probability of claiming the block times the expected payout, minus the costs) of participating in the network is positive, then the system is self-sustaining.**

> *Note that there is also no consensus amongst the peers on which transactions should be validated next. Each participant aggregates their own list and can use different strategies to optimize their expected payoff. Also, due to the nature of our proof-of-work function (finding a partial hash collision for a SHA-256 checksum of the block), the only way to increase the probability of claiming a block is to expend more CPU cycles.*

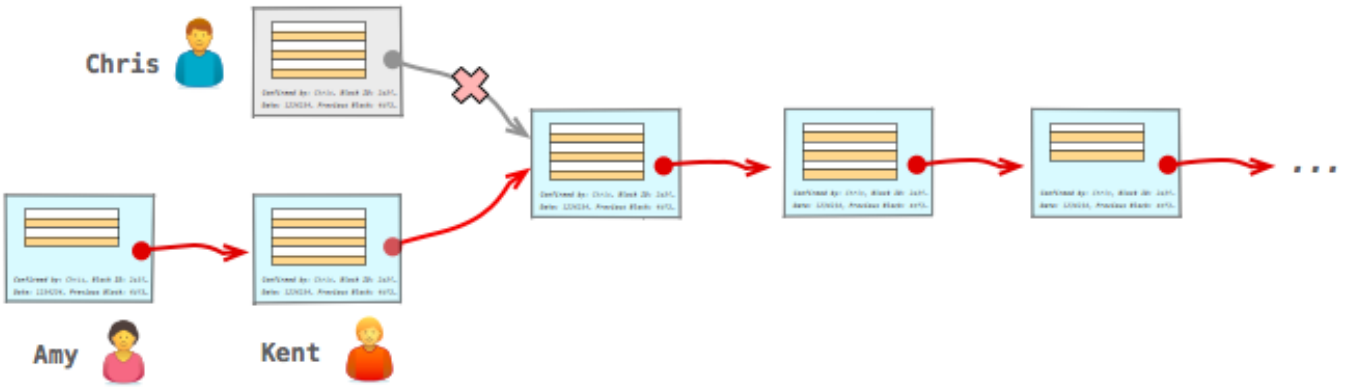


There is one more caveat that we need to deal with: it's possible that two peers will find a valid block at about the same time and begin propagating through the network - e.g. Kent and Chris in the diagram above. As a result, some fraction of the network may end up accepting Kent's block as topmost block, while the rest will take Chris's block. Now what?

Resolving chain conflicts

↑

Once again, we're going to take a hands-off approach and let the random nature of the block generation process resolve the conflict, albeit with one additional rule: if multiple chains are detected, the participants should immediately switch to and build on top of the longest chain. Let's see how this works in practice:



1. Some peers will start building new blocks on top of Kent's block, others on top of Chris's block.
2. At some point, someone will find a new block and begin propagating it through the network.

1. When other peers receive the new block, the part of the network that was working with a different topmost block will detect that there is now a longer alternative chain, which means that they need to switch to it - e.g. in the above example, the peers who were working with Chris's block stop their work, drop Chris's block, and switch to the longer (Amy + Kent's) chain.
2. Any transactions that are part of the discarded block but that are not yet confirmed are placed in the pending list and the process starts over.

It's possible that the race condition can persist for multiple blocks, but eventually one branch will race ahead of the other and the rest of the network will converge on the same longest chain.

Great, we now have a strategy to resolve conflicts between different chains in the network. Specifically, the network promises linearizability of transactions by recording them in a linked list of blocks. But, crucially, it makes no promises about an individual block "guaranteeing" the state of any transaction. Consider the example above:

- Alice sends out her transaction into the network.
- Chris generates a valid block that confirms her transaction.

Except, there is a fork in the chain and Chris's block is later "removed" as the network converges on Kent's branch of the chain. As a result, even when Alice receives a block with her transaction, she can't be sure that this block won't be undone in the future!

Blocks are never "final" # ↑

No block is "final", ever. Any block can be "undone" if a longer chain is detected. In practice, forks should be detected fairly quickly, but there is still always the possibility of an alternative chain. Instead, **the only claim we can make is that the "deeper" any particular block is in the chain, the less likely it is that it will undone.** Consequently, no transaction can ever be treated as "final" either, we can only make statements about the probability of it being undone.

1. **0-confirmation transaction:** exchange is performed without waiting for any block to include the transaction.
2. **1-confirmation transaction:** latest valid block includes the transaction.
3. **N-confirmation transaction:** there is a valid block that includes the transactions, and there are *N-1* blocks that have since been built on top of that block.

If you are willing to accept the risk, you always have the option to go with a 0-confirmation transaction: no transaction fees, no need to wait for confirmations. However, you also place a lot of trust in the opposite party.

Alternatively, if you want to lower your risk, then you should wait for one or more blocks to be built on top of the block that includes your transaction. **The longer you wait, the more blocks will be built on top of the block that contains your transaction, the lower the probability of an alternative chain that may undo your transaction.**

By "undo" we mean any scenario where one of the participants can make the network accept an alternative transaction transferring funds to any account other than yours - e.g. you complete the transaction, hand over the widget and get a receipt, but the attacker then injects a transaction that double spends those

same funds to an alternative account.

Why does the length of the block chain act as a good proxy for "safety" of a transaction? If an attacker wanted to undo a particular transaction, then they will need to build a chain that begins at a block prior to the one where that transaction is listed, and then build a chain of other blocks that is longer than the one currently used by the network. As a result, the deeper the block, the higher the amount of computational effort that would be required to replace it by creating an alternative chain. The longer the chain the more expensive it is to execute an attack.

How many blocks should you wait for before accepting a transaction? There is no one number, the answer depends on the properties of the network (time to generate each block, propagation latency of the transactions and blocks, size of the network, etc), and the transaction itself: it's value, what you know about the other party, your risk profile, and so on.

Properties of the (minimum viable) block chain# ↑

1. **Individual transactions are secured by PKI.**
    - Transactions are authenticated: a malicious party can't masquerade as someone else and sign a transaction on their behalf.
        - Authentication is only with respect to the public-private keypair. There is no requirement for "strong authentication" that links the keypair to any other data about the participants. In fact, a single participant can generate and use multiple keypairs! In this sense, the network allows anonymous transactions.
    - Non-repudiation: participants can't claim that the transaction did not happen after the fact.
    - Integrity: transactions can't be modified after the fact.
2. **Once created, transactions are broadcast into the P2P network.**
    - Participants form a network where transactions and blocks are relayed amongst all the participating peers. There no central authority.
3. **One or more transactions are aggregated into a "block".**
    - A block validates one or more transactions and claims the transaction fees.
        - This allow the transaction fees to remain small relative to the value of each transaction.
    - A valid block must have a valid proof-of-work solution.
        - Valid proof-of-work output is hard to generate and cheap to verify.
        - Proof-of-work is used to raise the cost of generating a valid block to impose a higher cost on running an attack against the network.
    - Any peer is allowed to work on generating a valid block, and once a valid block is generated, it is broadcast into the network.
        - Any number of peers can compete to generate a valid block. There is no coordination. When a fork is detected, it is resolved by automatically switching to the longest chain.
    - Each block contains a link to the previous valid block, allowing us to traverse the full history of all recorded transactions in the network.

4. **Peers listen for new block announcements and merge them into their ledgers.**
    - Inclusion of the transaction in a block acts as a "confirmation" of that transaction, but that fact alone does not "finalize" any transaction. Instead, we rely on the length of the chain as a proxy for "safety" of the transaction. Each participant can choose their own level of risk tolerance, ranging from 0-confirmation transactions to waiting for any arbitrary number of blocks.

The combination of all of the above rules and infrastructure provides a decentralized, peer-to-peer block chain for achieving distributed consensus of ordering of signed transactions. That's a mouthful, I know, but it's also an ingenious solution to a very hard problem. The individual pieces of the block-chain (accounting, cryptography, networking, proof-of-work), are not new, but the emergent properties of the system when all of them are combined are pretty remarkable.

Ilya Grigorik *is a web performance engineer at Google, co-chair of the W3C Web Performance working group, and author of High Performance Browser Networking (O'Reilly) book — follow on Twitter (https://twitter.com/igrigorik), Google+ (https://plus.google.com/+IlyaGrigorik).*

