piChain: When a Blockchain meets Paxos

Abstract

We present a new fault-tolerant distributed state machine to inherit the best features of its "parents in spirit": Paxos, providing strong consistency, and a blockchain, providing simplicity and availability. Our proposal is simple as it does not include any heavy weight distributed failure handling protocols such as leader election. In addition, our proposal has a few other valuable features, e.g., it is responsive, it scales well, and it does not send any overhead messages.

1998 ACM Subject Classification C.4 Performance of Systems

Keywords and phrases Consensus, Crash Failures, Availability, Network Partition, Consistency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.0

1 Introduction

Two fundamental philosophies to build fault-tolerant distributed state machines exist. One is mentioned in every other newspaper: The *blockchain* [22] is a fault-tolerant data structure to organize transactions. Its main advantage is its simplicity; the main disadvantage is that a blockchain is only eventually consistent.

On the other hand, we have the various consensus and agreement protocols designed by the distributed systems community, e.g., *Paxos* [16]. These protocols usually provide strong consistency, but have scalability issues. The idea of this paper is to marry the two worlds in a natural way, inheriting the best features of both.

We present *piChain*, a fault-tolerant distributed state machine (also known as repeated consensus, agreement, ledger, log, history, event sourcing) based on a blockchain, with integrated strong consistency. Our proposal is:

- Fault-Tolerant: piChain can handle various types of faults, e.g., crashes, crash-recoveries, message omissions, network partitions. It *cannot* handle arbitrarily malicious ("byzantine") faults, as we believe that there is a performance penalty many applications are not willing to pay.
- Fast: The basic functionality has no overhead; transactions can be created as fast as they can be sent and received by the network. Strong consistency will usually be achieved in one message round-trip time.
- Quiet: If no new transactions are created, no messages need to be sent. In other words, piChain needs no heartbeat.
- Scalable: piChain works with just a few nodes as well as hundreds of nodes, in a single location as well as distributed around the globe. There are no subroutines that produce a quadratic number of messages. If shooting for scalability, each node needs to send and receive only a few messages per transaction.
- Light: In comparison to other protocols, piChain is a light protocol, e.g., it does not have an explicit leader election subroutine. As such, the piChain architecture should be simple to understand and modify.
- Available and Consistent: piChain provides *both* availability and strong consistency. As such it will continue to try to process transactions even if the network is partitioned, and only short intervals of connectivity of a majority of nodes are enough to commit again to a globally consistent state. As such piChain works in harsh networking environments.

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão; Article No. 0; pp. 0:1–0:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are many applications that may benefit to have both availability and strong consistency. For example, take an airline reservation system with two data centers. For each flight, each data center may initially have an allowance of half the available seats. Even if the two data centers are temporarily not connected because of a network partition, both data centers can decide to sell seats based on their local allowance. When the whole system is reconnected, the data centers can regain strong consistency, e.g., re-balance their allowance.

In the next section we present the architecture of piChain. Section 3 explains how piChain differs from previous protocols. In Section 4 we explain why piChain is correct and efficient, and in Section 5 we evaluate our implementation.

2 Architecture

In this section we describe the three ingredients of the piChain architecture, and how they interact with each other.

2.1 Transactions and Blocks

We want to order and store *transactions*. Transactions can be, e.g., executable commands in a storage system, or financial transactions. Apart from its content, each transaction includes a unique ID, which is a combination of the unique ID of the node that created the transaction, and a sequence number.[1] Transactions are being sent to all the nodes in the system.[2]

Transactions are grouped in *blocks*, a block may contain many or just a single transaction. Blocks are created by arbitrary nodes, like transactions they contain an ID, which is again a combination of the ID of the creator of the block and a sequence number.

In addition, each block contains a pointer to a parent block.[3] The parent of a newly created block is the deepest block the creator has seen. What is the *deepest* block? Each block contains a depth field, which is the total number of transactions the block and all its ancestor blocks contain.[4] If two blocks have exactly the same depth, we will use the unique block ID to break ties.

The root block is already available when the system is initialized; the root block has no transactions, no parent, and its depth is 0. The transactions of a block may not contradict the transactions of the blocks on the path to the root.[5] After its creation a block is sent to all nodes in the system.

In piChain, any node can create a block, the only question is *when*.[6]

2.2 Node States

Each node is in one of three states: quick, medium, or slow. This state describes the node's eagerness to create a block.

[1] The sequence number is simply how many transactions that node already created. These sequence numbers help nodes to recover missing messages, e.g. if a node v has seen a transaction with ID (u,7) by node u but not transaction (u,6), node v can ask node u or any other node about the missing transaction.

[2] If the system consists of just a few nodes, all messages are sent directly. If the system consists of hundreds of nodes, one should rather use an overlay, and send all messages between neighbors using flooding. Thanks to flooding, each node must only transmit or receive a constant number of messages per transaction.

[3] We use the usual family relations to describe relations between blocks, in particular parent, ancestor, and descendent.

[4] In other words, the depth of a block b is the sum of the depth of b's parent and the number of transactions block b contains.

[5] E.g., a transaction that commands to move a file cannot be included in a block if its parent block included a transaction that deleted the file.

[6] In contrast to the Bitcoin blockchain, nodes do not have to perform a proof of work in order to generate a new block.

If a node is *quick*, it will create blocks quickly: whenever a quick node sees a new transaction that it has not yet seen included in a block, it will instantly create a block with this transaction.[7]

A *slow* node will only create a block if it has waited for a considerable time after seeing a new transaction that is not yet in a block. The earliest time t for a slow node to consider creating a block is when it should have received the block already by either a quick or a medium node. In addition to this earliest time t, slow nodes will also randomly wait even longer, long enough that only one (the fastest) slow node s will create a block in expectation, as other slow nodes will see that block by s before they will create a block.

In contrast to quick and slow nodes, *medium* nodes only exist transiently. After learning about a new transaction, a medium node waits until it should have received a block with this transaction by a quick node.

Each node will upgrade its state as follows:

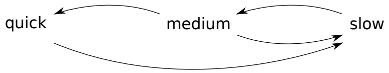


Figure 1 The state transitions: A node promotes itself to the next faster state when it creates a block. A node demotes itself to slow if it sees a block b by some other node, and either the creator of b is quick or b is the new deepest block.

2.3 Strong Consistency

Whenever a quick node creates a block and is not already in the process of committing another block, it may decide to commit this block. Committing a block commits all the transactions in that block, and all the transactions in all the blocks on the path from the root to that block.[8] Committed blocks (transactions) are final and cannot be uncommitted again. A committable block must be a descendant of all previously committed blocks, i.e., the committed blocks are totally ordered in tree of blocks. The first committed block is the root block; the root block is the *precursor* of the second committed block, which in turn is again the precursor of the third committed block. In other words, every committed block except the root block have the previous committed block as their precursor.

In order to commit a block, a quick node must convince a majority of nodes twice. This works trivially if there is no competition, i.e., if there is a single quick node. If no node is quick, blocks will transiently not be committed, even though new blocks are still created. If there are multiple quick nodes (e.g. after a network partition), our protocol still works, as it is a variant of Paxos, formally described in Algorithm 1.

In Algorithm 1, each node stores a list of already committed blocks, initially only the root block is committed. For *every* committed block b, each node stores three variables $b_{\max}$ (the deepest block seen in round 1), b_{prop} (a proposed block) and b_{supp} (a block supporting the proposed block). These are initially $\perp$ for every block, including the root. In addition, a quick node q that initiates a the commit protocol temporarily also stores b_{com} (a compromise block).

Every committed block (but the root block) has a precursor block. In order to commit a new block b_{new}, a quick node needs to refer the last already committed block b, the precursor block of the block to be committed. In order to avoid notational clutter in Algorithm 1, we omitted all the precursor

[7] A quick node may also wait a bit to accumulate several transactions; however, other nodes must know about such an intentional offset and adapt their timings accordingly.
[8] Many of which may already be committed.

Algorithm 1 Committing: Paxos with Blocks

```
Quick Node                                   All Nodes

$b_{\text{prop}} =$ deepest block            $b_{\max} = \perp$
$b_{\text{com}} = \perp$                     $b_{\text{prop}} = \perp$
                                             $b_{\text{supp}} = \perp$

Phase 1 ......................................................................
 1: Send try($b_{\text{new}}$) to all nodes
                                              2: On receiving a try($b_{\text{new}}$) message:
                                              3: if $b_{\text{new}}$ deeper than $b_{\max}$ then
                                              4:     $b_{\max} = b_{\text{new}}$
                                              5:     Answer with ok($b_{\text{prop}}, b_{\text{supp}}$)
                                              6: end if
Phase 2 ......................................................................
 7: Majority responded with ok($b_{\text{prop}}, b_{\text{supp}}$):
 8: $b_{\text{com}} = b_{\text{new}}$
 9: if some response included $b_{\text{prop}} \neq \perp$ then
10:     $b_{\text{com}} = b_{\text{prop}}$ with deepest $b_{\text{supp}}$
11: end if
12: Send propose($b_{\text{com}}, b_{\text{new}}$) to all nodes
                                             13: On receiving a propose($b_{\text{com}}, b_{\text{new}}$) message:
                                             14: if $b_{\text{new}} = b_{\max}$ then
                                             15:     $b_{\text{prop}} = b_{\text{com}}$
                                             16:     $b_{\text{supp}} = b_{\text{new}}$
                                             17:     Answer with ack($b_{\text{com}}$)
                                             18: end if
Phase 3 ......................................................................
19: Majority responded with ack($b_{\text{com}}$):
20: Send commit($b_{\text{com}}$) to all nodes
```

information. When we write that a node sends b_{x}, we mean that the node sends both the ID of its precursor block b (for reference) and the value of b_{x}.

Moreover, Algorithm 1 can be pipelined: A quick node that passed phase two can already start committing the next block. This becomes even more powerful with an implicit phase 1. Every propose for a block can implicitly be a try($\perp$) with an empty block of depth 0. This way a successful phase 2 for a block always includes a successful phase 1 for the next round and the quick node can directly propose a successor block again. In the regular case the quick node is only sending one message type, a pipelined/truncated combination of a phase 3 (line 20) and phase 2 (line 12) message: "commit(b) and propose($b_{\text{new}}, \perp$)". If some other node intervenes with its own try message and reaches a majority in the first phase, the quick node's propose will fail in line 14 and the quick node has do to an explicit new phase 1.

After a block b is committed, there might be blocks which are neither a descendant of b nor on the path from the root block to b. These blocks are removed; the transactions inside these blocks may however be salvaged (if they do not contradict the transactions of the committed blocks), by simply creating a new block with these transactions as a descendent of b. Committing a block is also an opportunity to compact the log up to block b.

Finally, we could also use commits to implement node membership changes; we simply add a new node u with a transaction t, and node u will be included as a voting node in Algorithm 1 as soon as transaction t has been committed.

3 Related Work

There is no lack of protocols that try to solve the same problem as we do, e.g., Chubby [4], Zookeeper [13], Spanner [14], CORFU [2]. The most recent heavy weight champion of this class is probably Raft [23]. Indeed, Raft has been designed with a similar agenda in mind (simplicity first, strong consistency, explicit timing, no byzantine failures). The main argument in favor of piChain's simplicity is its "lightness": Raft (and its competitors) uses leaders, and these leaders have their epochs (or terms) when they are ruling. Whenever a leader is not responsive anymore, other nodes have to notice this first, then they have to agree that they want a new leader, at which point a leader election algorithm is started.[9] In piChain, each node just decides by itself that something needs to be done: it directly promotes or demotes itself without communication. Moreover, piChain is quiet in the sense that it does not send any messages if there are no new transactions, whereas Raft needs some kind of heartbeat. Finally, piChain provides scalability and availability out of the box.

The only heavy part of piChain is Algorithm 1, a blockchain version of Paxos [11, 16]. In our opinion, Paxos and blockchains are a natural fit, since a blockchain orders transactions which is then augmented to strong consistency with Paxos. In contrast to Paxos, piChain introduces a new way to prevent proposers from interrupting ongoing commits. Lamport's Paxos cared about *correctness* rather than efficiency. Paxos is a truly seminal protocol that is hard to get around these days.[10] One may claim that Lamport's original publication [16] left message timing as a exercise to the reader, and that piChain is just an explanation how to implement a repeated version of Paxos also known as multi-Paxos. Paxos was studied in various dimensions over the years, e.g., [6, 17, 19, 21, 25]. We do not use any of these improvements, just the original.

Depending on the application, commits do not have to be done often but just from time to time, to shorten the blockchain. If the application depends on fast commits, we can also commit each and every block. As we will discuss in Section 4, thanks to majority-only voting, pipelining and truncating, piChain will be faster than three-phase commit [26].

In addition, piChain cares about availability, something Paxos did not bother with. After Brewer [3] explained that consistency is not everything and availability should not be forgotten, the pendulum is swinging back and forth between strong consistency and high availability. We believe that there is value in having both, as many applications may want to implement decisions even if they are not final.

Speaking of high availability: "Satoshi Nakamoto" introduced the concept of a blockchain in a byzantine setting [22]. While his proof-of-work based system can tolerate byzantine failures with anonymous nodes, generating blocks cannot be fast because of forks, and blocks are never really committed [7, 9, 12].

Previous work tried to prevent forks of the Bitcoin blockchain by using strongly consistent byzantine agreement to agree on blocks before adding them to the blockchain [8]. This is implemented in various cryptocurrencies, e.g., the masternode system of Dash [10]. While the byzantine setting is more difficult than piChain, those systems cannot recover from a state of too many crashed nodes, as the block creators are themselves elected in blocks. The system uses strong consistency to append to the blockchain. We believe that piChain is more natural, ordering by the blockchain first and strong consistency later.

In the last 20 years we have seen an army of new protocols that try to cope with byzantine (arbitrary, malicious) failures, e.g., PBFT [5], Farsite [1], Zyzzyva [15] and a byzantine version of Paxos [18]. These protocols build on the earlier theoretical work, among others again by Lamport [20, 24]. In many applications one must be able to tolerate byzantine behaviour. However, Google or Amazon have developed their fault-tolerant distributed systems to handle crash failures only. We believe that the cost of fully byzantine protocols may be prohibitive for applications which need to be efficient. Nevertheless, one can add byzantine tolerant elements to piChain, e.g. when creating a new transaction, nodes could be asked to cryptographically authenticate the transaction.

4 Analysis

In this section we discuss why piChain is correct and efficient.

4.1 Transactions and Blocks

Let us start with some basics about transactions and blocks. First note that the blocks form a tree: Every block has a parent, which is another block that has been created earlier. By induction following the parent pointers brings us to earlier and earlier blocks, and ultimately to the root block.

Also, every transaction will be in a block: Even if it is a slow node, the creator of the transaction will include it in a block after it has not seen it in a block for long enough. If a transaction is in a block b that is discarded because of committing a block which is neither an ancestor nor a descendent of b, then the transaction is again on the market. If the transaction does not contradict an already committed transaction, the nodes will put it in a block again.

4.2 Node States

The waiting times of quick, medium, and slow nodes until they create a block are crucial to the performance of piChain:

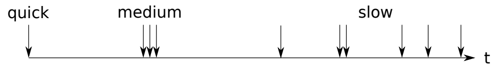


Figure 2 Waiting time of nodes in different states: A quick node creates a block instantly when seeing a new transaction. A medium node waits long enough to ensure not to compete with a quick node. A slow node waits long enough to ensure to compete neither with a quick nor a medium node. The figure represents a situation with one quick, three medium, and some slow nodes.

The waiting times depend on the network characteristics. To understand this aspect better, let us first discuss a simplistic network model by assuming that the time to deliver a message between any two nodes is always exactly 1 time unit, in both directions. Quick nodes always immediately create a block when learning about a new transaction, also in this simplified model. After learning about a new transaction, a medium node u should wait time $1 + \epsilon$, unless the transaction was created by u, in which case u should wait time $2 + \epsilon$ (as this is enough time to send the transaction to the quick node and send a block with the transaction back). A slow node should wait time $3 + 2\epsilon + r$ (or

[9] We would argue that the leader election algorithm of Raft is similar to the first Paxos round.

[10] When working on piChain we tried to deviate from Paxos more radically, without success.

$4 + 2\epsilon + r$ for a self-created transaction), where the parameter r is a random time chosen uniformly in the interval $[0, n + 1]$, where n is the number of nodes. In this simplified model, these timings are the shortest possible timings allowed by the description in Section 2.2, as they allow the faster nodes to deliver a block before a slower node will create one.

For example, let us assume that slow node s produces a transaction at time t, sending it to the other nodes. If there is a medium node m, node m will learn about this transaction at time $t + 1$, create a block at time $t + 3 + \epsilon$, which will arrive back at slow node s at time $t + 4 + \epsilon$, right before the earliest possible time s may create a block. Arguing all other cases is similar, we know that a quick node will not be challenged by a medium node, and a medium node will not be challenged by a slow node.

In absence of quick and medium nodes, the random time r for the slow nodes makes sure that there is a good chance only a single slow node creates a block. Moreover, using a standard Chernoff bound one can show that with high probability at most $O(\log n)$ slow nodes create a block within one time unit.

If there is exactly one quick node, and all others are slow, we call the system *healthy*.[11] Our system should almost always be healthy. The healthy state is stable, since the timings guarantee that the quick node will not be challenged by the other nodes. But what if something does not work as anticipated, e.g. the single quick node crashes?

If the single quick node crashes (or the network has problems and does not deliver the messages of the quick node), we have only slow nodes. If so, a slow node will jump in and produce a block. There might be several such slow nodes, but there will be few. If there is more than one slow node challenging the crashed quick node, they will all send out their blocks more or less concurrently (otherwise they would had seen the other blocks before creating one themselves). According to Figure 1, all of these fastest slow nodes become medium nodes, but (unless there are further crashes or message omissions) all see all their blocks before they produce a next block. Only one of these blocks is the deepest (even if several have the same number of transactions we know that ties are broken by block ID), so only one now-medium node will create a second block and become quick. All other medium nodes see a deeper block and go back to slow again. This summarizes our implicit *election*.[12] This is probably the most regular way to get back to the healthy state, but others exist, as summarized in this figure:

So what if message delays are more realistic? The idea is to use previous measurements, like in TCP. If a non-quick node v learns about a transaction created by node u, node v considers the current message delays in the system. As described in Section 2.2, node v gives the faster nodes enough time to deliver their blocks, based on previous delivery times. If a node does not know anything about these times (e.g., the system just started), it can assume a very crude upper bound on message delivery time. We would suggest for a slow node v to wait for $2R + r \cdot 0.5R + 2\epsilon$, where r is again a random value in $[0, n + 1]$, and R is the absolute worst round-trip time any node has seen (or can imagine) when the network was not faulty. This way, we will be back in the healthy state in expected time $3.5R$ (we have $2.5R$ time until the first slow node s creates a block,[13], plus R time until node s creates a second block).

[11] Because of this, one might consider naming quick nodes "leaders", slow nodes "followers", and medium nodes "candidates". However, the term leader usually implies that the system *guarantees* to have at most one. Our piChain architecture does not attempt to have such a guarantee, it does not even have an explicit leader election subroutine. Instead nodes just update their state (quick, medium, slow) locally without interacting with other nodes, so the classic terms seem inaccurate.

[12] One might consider to"vaccinate" the healthy state: When we are in a healthy state, the single quick node q may consider choosing another node as its "heir"; this heir would then assume a medium state, and as such automatically be the first to create a block after the quick node crashed.

[13] If n nodes choose a random value in $[0, n + 1]$, the expected lowest value is 1.

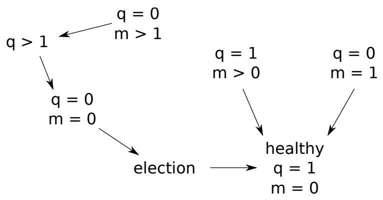


Figure 3 Recovery of the network to the healthy state. The nodes of this graph represent all possible states of the whole system: how many quick (q) nodes do we have and if relevant how many medium (m) nodes. The election state is special, as it is characterized by possibly multiple medium nodes, which have in-flight blocks. The in-flight blocks will demote all medium nodes except for the one that created the deepest block, so only that node will become quick.

We could optimize R a bit more aggressively if we have previous timing measurements. Premature block creations will be handled gracefully in piChain, so they are no big deal really. On the other hand, these slow waiting times will happen so rarely that they barely matter for our overall system performance, so just choosing a high R is also fine. After all, the system will be mostly in the healthy state, where the single quick node does not wait at all.

4.3 Strong Consistency

This brings us to the third part of the architecture, the commits. First of all, if the system is healthy, Algorithm 1 reduces to a simple message ping pong: try-ok-propose-ack-commit not unlike three-phase-commit (3PC). One may argue that Algorithm 1 is faster than 3PC because the quick node does not have to wait for replies from *all* nodes, but merely a majority. Also, Algorithm 1 can be pipelined, and a stable quick node can directly enter round 2, i.e. in a healthy system, Algorithm 1 is truncated to lines 12–22.

If there are no quick nodes, we do not even attempt to commit a block. So the only interesting remaining cases are multiple competing quick nodes (after a partition), or if there are so many errors (crashes, message omissions) in the system that the quick node cannot easily finish its commit because it does not get the needed majorities.

The principle why the algorithm works is built around the implication of accepting a block proposal in round 2. A node u which answers ok in round 2 may have just been part of a successful commit.[14] To make sure only one block can be committed as a successor of a previous commit, node u must not accept a proposal for any other block until the node can be convinced that the commitment was unsuccessful. Luckily, any arriving propose message that proposes a different block with a deeper support proves that the last proposal was unsuccessful and the node can therefore safely support this new proposal. So in order to get a successful commit, we need a uninterrupted "double whammy" of rounds 1 and 2.

We can formally proof this intuition. Let $p = (b_{\text{com}}, b_{\text{new}})$ be a successful successor for committed block b, i.e., p was accepted by a majority of the nodes in round 2. Let p' be the first subsequent proposal. In order to not be ignored in round 1, p''s support needs to be deeper than p. Both p and p' are seen by a majority of nodes, so there is a node which has seen both p and p'. This node will report p to the quick node at the end of round 1. Therefore the quick node leading the proposal had both

[14] Node u may not know that, though.

to choose from in line 10. If p' was for a block different than p that block must have been proposed with a deeper support block, as otherwise the quick node leading the commitment process would not have chosen it as the compromise block. However this contradicts the assumption that p' was the first proposal with a deeper support after p, therefore there can only be one successful proposal and thus a node which receives a new proposal with deeper support can assume all previous proposals failed.

This proof also applies to the truncated version, as the order of message arrivals is exactly the same. Every node simulates the arrival of a try message for the next round after every propose message is accepted, this way the quick node does not violate the protocol by omitting the first phase.

5 Evaluation

To test our implementation, we run it in a discrete event simulation to get accurate timings not affected by distributed time measuring difficulties. Most failures have no effect on the system, so we only tested a few rare but harsh scenarios. We have $n = 20$ nodes embedded randomly in a square with diagonal 0.5, with distance being message delay, so the maximum possible message delay is 0.5s. Transactions are created by random nodes at random times with an average rate of 10 transactions per second. The nodes use a worst-case round-trip time of $R = 1$s.

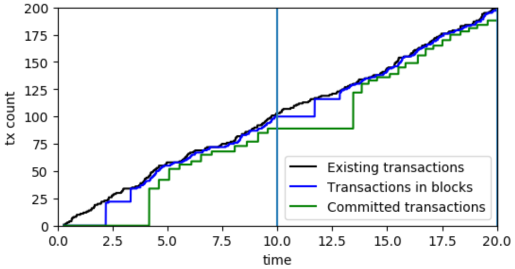


Figure 4 piChain with $n = 20$. After 10s the single quick node crashes. No new blocks are created until a slow node creates a block and becomes medium. A bit later, the same node becomes quick, and blocks are again created and committed with a fast pace.

Figure 4 shows an example how the network recovers to healthy after the quick node crashed. Averaged over 100 runs of such crashes, the time until we are back to the healthy state is 3.67s on average, with a standard deviation of 0.49s. This conforms with our expectation of $3.5R$. The variations can be explained by how we measure. In particular, to get the protocol going, we need a transaction, and transactions first have to be created and delivered. Further randomness is introduced by different waiting times of slow nodes. Figure 5 presents the effects of a network partition.

In Section 4.2 we claimed that we always get back to the healthy state quickly. Figure 6 shows the actual times of the state chart of Figure 3.

In another experiment we analyze the behaviour of the algorithm in a highly unstable situation, where nodes repeatedly crash and recover; while being crashed all messages of a node are dropped. Two of the resulting plots are shown in Figure 7.

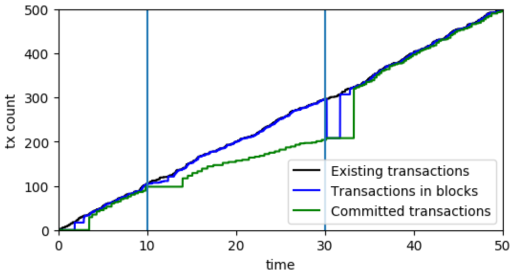


Figure 5 The network is partitioned into 8 resp. 12 nodes after 10s. Each side of the partition quickly finds a single quick node that is adding blocks, but only the majority side can commit its blocks. When the partition is resolved after 30s, the minority side temporarily loses its blocks because of the committed blocks on the majority side (see discussion in second paragraph of Section 4.1).

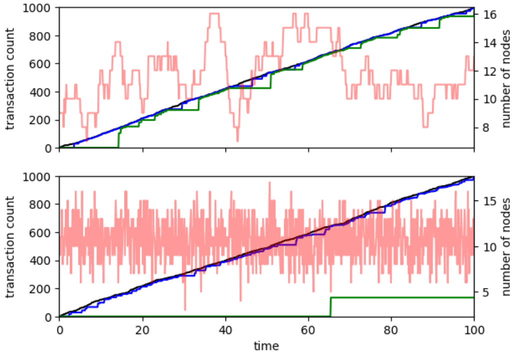


Figure 7 piChain with unstable nodes that crash and recover. The red curves show the number of currently working nodes. In the upper plot crashes last for an average of 20s, in the lower plot for 0.2s, but both plots have the same average number of 11 working nodes. One can see that blocks are generated quickly in both plots, but committing takes more time in the lower plot, since Algorithm 1 is often interrupted by crashes.

References

1 Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review (OSR)*, 2002.

2 Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

3 Eric A Brewer. Towards robust distributed systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

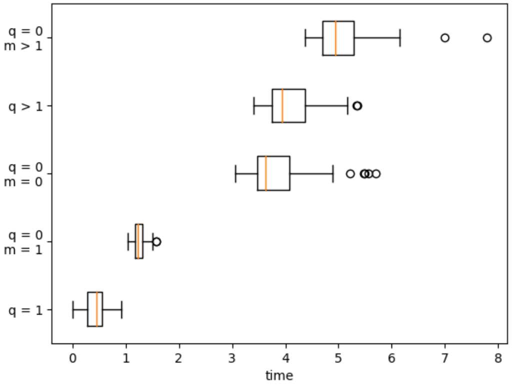


Figure 6 The time piChain needs to become healthy when being started in a random state. The start states are clustered according to Figure 3, e.g., when a third of the nodes is initially in each category quick, medium, and slow, we add a measurement to class $q > 1$.

4 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating systems design and implementation (OSDI)*, 2006.

5 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

6 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

7 Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gun Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In *3rd Workshop on Bitcoin Research (BITCOIN)*, 2016.

8 Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2016.

9 Christian Decker and Roger Wattenhofer. Information Propagation in the Bitcoin Network. In *13th IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

10 Evan Duffield and Daniel Diaz. Dash: A privacy-centric crypto-currency, 2014.

11 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

12 Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

13 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference (USENIX ATC)*, 2010.

14 Michael Epstein Andrew Fikes Christopher Frost JJ Furman Sanjay Ghemawat Andrey Gubarev Christopher Heiser Peter Hochschild Wilson Hsieh Sebastian Kanthak Eugene Kogan Hongyi Li Alexander Lloyd Sergey Melnik David Mwaura David Nagle Sean Quinlan Rajesh Rao Lindsay Rolig Yasushi Saito Michal Szymaniak Christopher Taylor Ruth Wang James C. Corbett, Jeffrey Dean and Dale Woodford. Spanner: Google's globally-distributed database, 2012.

15 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review (OSR)*, 2007.

16 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

17 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

18 Leslie Lamport. Byzantizing paxos by refinement. In *International Symposium on Distributed Computing (DISC)*, 2011.

19 Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks (DSN)*, 2004.

20 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982.

21 Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *ACM Symposium on Cloud Computing*, 2014.

22 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

23 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

24 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 1980.

25 Jun Rao, Eugene J Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. In *International Conference on Very Large Data Bases (VLDB)*, 2011.

26 Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, 1983.

Ethereum White Paper

A NEXT GENERATION SMART CONTRACT & DECENTRALIZED APPLICATION PLATFORM

By Vitalik Buterin

When Satoshi Nakamoto first set the Bitcoin blockchain into motion in January 2009, he was simultaneously introducing two radical and untested concepts. The first is the "bitcoin", a decentralized peer-to-peer online currency that maintains a value without any backing, intrinsic value or central issuer. So far, the "bitcoin" as a currency unit has taken up the bulk of the public attention, both in terms of the political aspects of a currency without a central bank and its extreme upward and downward volatility in price. However, there is also another, equally important, part to Satoshi's grand experiment: the concept of a proof of work-based blockchain to allow for public agreement on the order of transactions. Bitcoin as an application can be described as a first-to-file system: if one entity has 50 BTC, and simultaneously sends the same 50 BTC to A and to B, only the transaction that gets confirmed first will process. There is no intrinsic way of determining from two transactions which came earlier, and for decades this stymied the development of decentralized digital currency. Satoshi's blockchain was the first credible decentralized solution. And now, attention is rapidly starting to shift toward this second part of Bitcoin's technology, and how the blockchain concept can be used for more than just money.

Commonly cited applications include using on-blockchain digital assets to represent custom currencies and financial instruments ("colored coins"), the ownership of an underlying physical device ("smart property"), non-fungible assets such as domain names ("Namecoin") as well as more advanced applications such as decentralized exchange, financial derivatives, peer-to-peer gambling and on-blockchain identity and reputation systems. Another important area of inquiry is "smart contracts" - systems which automatically move digital assets according to arbitrary pre-specified rules. For example, one might have a treasury contract of the form "A can withdraw up to X currency units per day, B can withdraw up to Y per day, A and B together can withdraw anything, and A can shut off B's ability to withdraw". The logical extension of this is decentralized autonomous organizations (DAOs) - long-term smart contracts that contain the assets and encode the bylaws of an entire organization. What Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions, allowing users to create any of the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code.

Table of contents

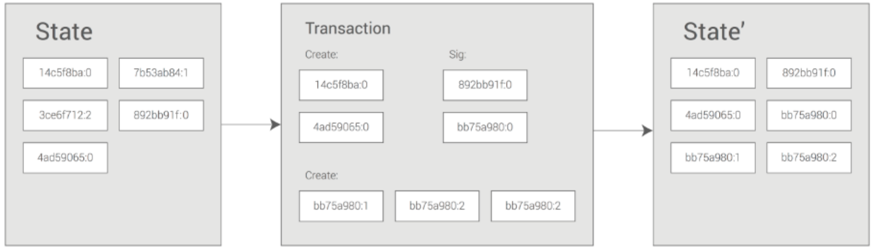
History

The concept of decentralized digital currency, as well as alternative applications like property registries, has been around for decades. The anonymous e-cash protocols of the 1980s and the 1990s, mostly reliant on a cryptographic primitive known as Chaumian blinding, provided a currency with a high degree of privacy, but the protocols largely failed to gain traction because of their reliance on a centralized intermediary. In 1998, Wei Dai's b-money became the first proposal to introduce the idea of creating money through solving computational puzzles as well as decentralized consensus, but the proposal was scant on details as to how decentralized consensus could actually be implemented. In 2005, Hal Finney introduced a concept of "reusable proofs of work", a system which uses ideas from b-money together with Adam Back's computationally difficult Hashcash puzzles to create a concept for a cryptocurrency, but once again fell short of the ideal by relying on trusted computing as a backend.

Because currency is a first-to-file application, where the order of transactions is often of critical importance, decentralized currencies require a solution to decentralized consensus. The main roadblock that all pre-Bitcoin currency protocols faced is the fact that, while there had been plenty of research on creating secure Byzantine-fault-tolerant multiparty consensus systems for many years, all of the protocols described were solving only half of the problem. The protocols assumed that all participants in the system were known, and produced security margins of the form "if N parties participate, then the system can tolerate up to N/4 malicious actors". The problem is, however, that in an anonymous setting such security margins are vulnerable to sybil attacks, where a single attacker creates thousands of simulated nodes on a server or botnet and uses these nodes to unilaterally secure a majority share.

The innovation provided by Satoshi is the idea of combining a very simple decentralized consensus protocol, based on nodes combining transactions into a "block" every ten minutes creating an ever-growing blockchain, with proof of work as a mechanism through which nodes gain the right to participate in the system. While nodes with a large amount of computational power do have proportionately greater influence, coming up with more computational power than the entire network combined is much harder than simulating a million nodes. Despite the Bitcoin blockchain model's crudeness and simplicity, it has proven to be good enough, and would over the next five years become the bedrock of over two hundred currencies and protocols around the world.

Bitcoin As A State Transition System



From a technical standpoint, the Bitcoin ledger can be thought of as a state transition system, where there is a "state" consisting of the ownership status of all existing bitcoins and a "state transition function" that takes a state and a transaction and outputs a new state which is the result. In a standard banking system, for example, the state is a balance sheet, a transaction is a request to move $X from A to B, and the state transition function reduces the value in A's account by $X and increases the value in B's account by $X. If A's account has less than $X in the first place, the state transition function returns an error. Hence, one can formally define:

```
APPLY(S,TX) -> S' or ERROR
```

In the banking system defined above:

```
APPLY({ Alice: $50, Bob: $50 },"send $20 from Alice to Bob") = { Alice: $30,
Bob: $70 }
```

But:

```
APPLY({ Alice: $50, Bob: $50 },"send $70 from Alice to Bob") = ERROR
```

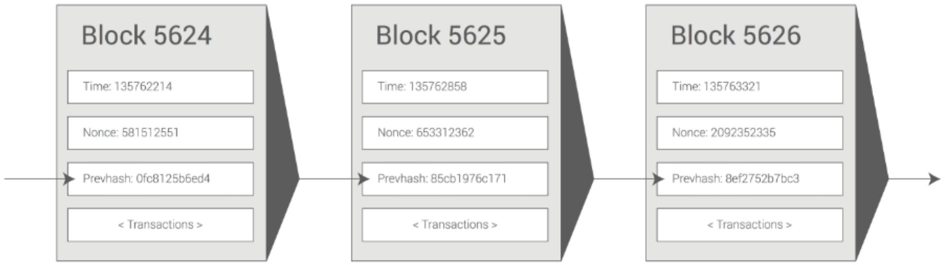
The "state" in Bitcoin is the collection of all coins (technically, "unspent transaction outputs" or UTXO) that have been minted and not yet spent, with each UTXO having a denomination and an owner (defined by a 20-byte address which is essentially a cryptographic public key[1]). A transaction contains one or more inputs, with each input containing a reference to an existing UTXO and a cryptographic signature produced by the private key associated with the owner's address, and one or more outputs, with each output containing a new UTXO to be added to the state.

The state transition function `APPLY(S,TX) -> S'` can be defined roughly as follows:

1. For each input in TX:
    i. If the referenced UTXO is not in S, return an error.
    ii. If the provided signature does not match the owner of the UTXO, return an error.
2. If the sum of the denominations of all input UTXO is less than the sum of the denominations of all output UTXO, return an error.
3. Return S with all input UTXO removed and all output UTXO added.

The first half of the first step prevents transaction senders from spending coins that do not exist, the second half of the first step prevents transaction senders from spending other people's coins, and the second step enforces conservation of value. In order to use this for payment, the protocol is as follows. Suppose Alice wants to send 11.7 BTC to Bob. First, Alice will look for a set of available UTXO that she owns that totals up to at least 11.7 BTC. Realistically, Alice will not be able to get exactly 11.7 BTC; say that the smallest she can get is 6+4+2=12. She then creates a transaction with those three inputs and two outputs. The first output will be 11.7 BTC with Bob's address as its owner, and the second output will be the remaining 0.3 BTC "change", with the owner being Alice herself.

Mining



If we had access to a trustworthy centralized service, this system would be trivial to implement; it could simply be coded exactly as described. However, with Bitcoin we are trying to build a decentralized currency system, so we will need to combine the state transition system with a consensus system in order to ensure that everyone agrees on the order of transactions. Bitcoin's decentralized consensus process requires nodes in the network to continuously attempt to produce packages of transactions called "blocks". The network is intended to produce roughly one block every ten minutes, with each block containing a timestamp, a nonce, a reference to (ie. hash of) the previous block and a list of all of the transactions that have taken place since the previous block. Over time, this creates a persistent, ever-growing, "blockchain" that constantly updates to represent the latest state of the Bitcoin ledger.

The algorithm for checking if a block is valid, expressed in this paradigm, is as follows:

1. Check if the previous block referenced by the block exists and is valid
2. Check that the timestamp of the block is greater than that of the previous block[2] and less than 2 hours into the future.
3. Check that the proof of work on the block is valid.
4. Let S[0] be the state at the end of the previous block.
5. Suppose TX is the block's transaction list with n transactions. For all i in 0...n-1, setS[i+1] = APPLY(S[i],TX[i]) If any application returns an error, exit and return false.
6. Return true, and register S[n] as the state at the end of this block

Essentially, each transaction in the block must provide a state transition that is valid. Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be (securely) computed for any block by starting from the genesis state and sequentially applying every transaction in every block. Additionally, note that the order in which the miner includes transactions into the block matters; if there are two transactions A and B in a block such that B spends a UTXO created by A, then the block will be valid if A comes before B but not otherwise.

The interesting part of the block validation algorithm is the concept of "proof of work": the condition is that the SHA256 hash of every block, treated as a 256-bit number, must be less than a dynamically adjusted target, which as of the time of this writing is approximately 2^{190}. The purpose of this is to make block creation computationally "hard", thereby preventing sybil attackers from remaking the entire blockchain in their favor. Because SHA256 is designed to be a completely unpredictable pseudorandom function, the only way to create a valid block is simply trial and error, repeatedly incrementing the nonce and seeing if the new hash matches. At the current target of 2192, this means an average of 264 tries; in general, the target is recalibrated by the network every 2016 blocks so that on average a new block is produced by some node in the network every ten minutes. In order to compensate miners for this computational work, the miner of every block is entitled to include a transaction giving themselves 25 BTC out of nowhere. Additionally, if any transaction has a higher total denomination in its inputs than in its outputs, the difference also goes to the miner as a "transaction fee". Incidentally, this is also the only mechanism by which BTC are issued; the genesis state contained no coins at all.

In order to better understand the purpose of mining, let us examine what happens in the event of a malicious attacker. Since Bitcoin's underlying cryptography is known to be secure, the attacker will target the one part of the Bitcoin system that is not protected by cryptography directly: the order of transactions. The attacker's strategy is simple:

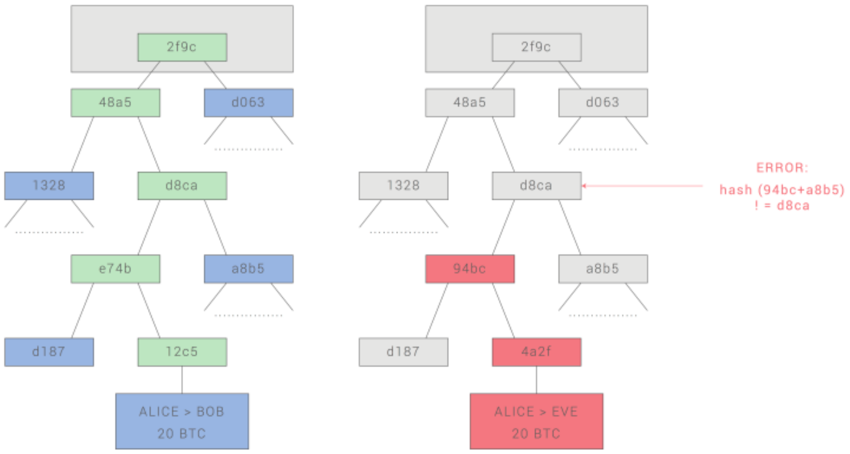
1. Send 100 BTC to a merchant in exchange for some product (preferably a rapid-delivery digital good)
2. Wait for the delivery of the product
3. Produce another transaction sending the same 100 BTC to himself
4. Try to convince the network that his transaction to himself was the one that came first.

Once step (1) has taken place, after a few minutes some miner will include the transaction in a block, say block number 270000. After about one hour, five more blocks will have been added to the chain after that block, with each of those blocks indirectly pointing to the transaction and thus "confirming" it. At this point, the merchant will accept the payment as finalized and deliver the product; since we are assuming this is a digital good, delivery is instant. Now, the attacker creates another transaction sending the 100 BTC to himself. If the attacker simply releases it into the wild, the transaction will not be processed; miners will attempt to run APPLY(S,TX) and notice that TX consumes a UTXO which is no longer in the state. So instead, the attacker creates a "fork" of the blockchain, starting by mining another version of block 270000 pointing to the same block 269999 as a parent but with the new transaction in place of the old one. Because the block data is different, this requires redoing the proof of work. Furthermore, the attacker's new version of block 270000 has a different hash, so the original blocks 270001 to 270005 do not "point" to it; thus, the original chain and the attacker's new chain are completely separate. The rule is that in a fork the longest blockchain (ie. the one backed by the largest quantity of proof of work) is taken to be the truth, and so legitimate miners will work on the 270005 chain while the attacker alone is working on the 270000 chain. In order for the attacker to make his blockchain the longest, he would need to have more computational power than the rest of the network combined in order to catch up (hence, "51% attack").

Merkle Trees

Left: it suffices to present only a small number of nodes in a Merkle tree to give a proof of the validity of a branch.

Right: any attempt to change any part of the Merkle tree will eventually lead to an inconsistency somewhere up the chain.



An important scalability feature of Bitcoin is that the block is stored in a multi-level data structure. The "hash" of a block is actually only the hash of the block header, a roughly 200-byte piece of data that contains the timestamp, nonce, previous block hash and the root hash of a data structure called the Merkle tree storing all transactions in the block.

A Merkle tree is a type of binary tree, composed of a set of nodes with a large number of leaf nodes at the bottom of the tree containing the underlying data, a set of intermediate nodes where each node is the hash of its two children, and finally a single root node, also formed from the hash of its two children, representing the "top" of the tree. The purpose of the Merkle tree is to allow the data in a block to be delivered piecemeal: a node can download only the header of a block from one source, the small part of the tree relevant to them from another source, and still be assured that all of the data is correct. The reason why this works is that hashes propagate upward: if a malicious user attempts to swap in a fake transaction into the bottom of a Merkle tree, this change will cause a change in the node above, and then a change in the node above that, finally changing the root of the tree and therefore the hash of the block, causing the protocol to register it as a completely different block (almost certainly with an invalid proof of work).

The Merkle tree protocol is arguably essential to long-term sustainability. A "full node" in the Bitcoin network, one that stores and processes the entirety of every block, takes up about 15 GB of disk space in the Bitcoin network as of April 2014, and is growing by over a gigabyte per month. Currently, this is viable for some desktop computers and not phones, and later on in the future only businesses and hobbyists will be able to participate. A protocol known as "simplified payment verification" (SPV) allows for another class of nodes to exist, called "light nodes", which download the block headers, verify the proof of work on the block headers, and then download only the "branches" associated with transactions that are relevant to them. This allows light nodes to determine with a strong guarantee of security what the status of any Bitcoin transaction, and their current balance, is while downloading only a very small portion of the entire blockchain.

Alternative Blockchain Applications

The idea of taking the underlying blockchain idea and applying it to other concepts also has a long history. In 2005, Nick Szabo came out with the concept of "secure property titles with owner authority", a document describing how "new advances in replicated database technology" will allow for a blockchain-based system for storing a registry of who owns what land, creating an elaborate framework including concepts such as homesteading, adverse possession and Georgian land tax. However, there was unfortunately no effective replicated database system available at the time, and so the protocol was never implemented in practice. After 2009, however, once Bitcoin's decentralized consensus was developed a number of alternative applications rapidly began to emerge:

- **Namecoin** - created in 2010, Namecoin is best described as a decentralized name registration database. In decentralized protocols like Tor, Bitcoin and BitMessage, there needs to be some way of identifying accounts so that other people can interact with them, but in all existing solutions the only kind of identifier available is a pseudorandom hash like1LW79wp5ZBqaHW1jL5TCiBCrhQYtHagUWy. Ideally, one would like to be able to have an account with a name like "george". However, the problem is that if one person can create an account named "george" then someone else can use the same process to register "george" for themselves as well and impersonate them. The only solution is a first-to-file paradigm, where the first registrant succeeds and the second fails - a problem perfectly suited for the Bitcoin consensus protocol. Namecoin is the oldest, and most successful, implementation of a name registration system using such an idea.
- **Colored coins** - the purpose of colored coins is to serve as a protocol to allow people to create their own digital currencies - or, in the important trivial case of a currency with one unit, digital tokens,

on the Bitcoin blockchain. In the colored coins protocol, one "issues" a new currency by publicly assigning a color to a specific Bitcoin UTXO, and the protocol recursively defines the color of other UTXO to be the same as the color of the inputs that the transaction creating them spent (some special rules apply in the case of mixed-color inputs). This allows users to maintain wallets containing only UTXO of a specific color and send them around much like regular bitcoins, backtracking through the blockchain to determine the color of any UTXO that they receive.

- **Metacoins** - the idea behind a metacoin is to have a protocol that lives on top of Bitcoin, using Bitcoin transactions to store metacoin transactions but having a different state transition function, APPLY'. Because the metacoin protocol cannot prevent invalid metacoin transactions from appearing in the Bitcoin blockchain, a rule is added that if APPLY'(S,TX) returns an error, the protocol defaults to APPLY'(S,TX) = S. This provides an easy mechanism for creating an arbitrary cryptocurrency protocol, potentially with advanced features that cannot be implemented inside of Bitcoin itself, but with a very low development cost since the complexities of mining and networking are already handled by the Bitcoin protocol.

Thus, in general, there are two approaches toward building a consensus protocol: building an independent network, and building a protocol on top of Bitcoin. The former approach, while reasonably successful in the case of applications like Namecoin, is difficult to implement; each individual implementation needs to bootstrap an independent blockchain, as well as building and testing all of the necessary state transition and networking code. Additionally, we predict that the set of applications for decentralized consensus technology will follow a power law distribution where the vast majority of applications would be too small to warrant their own blockchain, and we note that there exist large classes of decentralized applications, particularly decentralized autonomous organizations, that need to interact with each other.

The Bitcoin-based approach, on the other hand, has the flaw that it does not inherit the simplified payment verification features of Bitcoin. SPV works for Bitcoin because it can use blockchain depth as a proxy for validity; at some point, once the ancestors of a transaction go far enough back, it is safe to say that they were legitimately part of the state. Blockchain-based meta-protocols, on the other hand, cannot force the blockchain not to include transactions that are not valid within the context of their own protocols. Hence, a fully secure SPV meta-protocol implementation would need to backward scan all the way to the beginning of the Bitcoin blockchain to determine whether or not certain transactions are valid. Currently, all "light" implementations of Bitcoin-based meta-protocols rely on a trusted server to provide the data, arguably a highly suboptimal result especially when one of the primary purposes of a cryptocurrency is to eliminate the need for trust.

Scripting

Even without any extensions, the Bitcoin protocol actually does facilitate a weak version of a concept of "smart contracts". UTXO in Bitcoin can be owned not just by a public key, but also by a more complicated script expressed in a simple stack-based programming language. In this paradigm, a transaction spending that UTXO must provide data that satisfies the script. Indeed, even the basic public key ownership mechanism is

implemented via a script: the script takes an elliptic curve signature as input, verifies it against the transaction and the address that owns the UTXO, and returns 1 if the verification is successful and 0 otherwise. Other, more complicated, scripts exist for various additional use cases. For example, one can construct a script that requires signatures from two out of a given three private keys to validate ("multisig"), a setup useful for corporate accounts, secure savings accounts and some merchant escrow situations. Scripts can also be used to pay bounties for solutions to computational problems, and one can even construct a script that says something like "this Bitcoin UTXO is yours if you can provide an SPV proof that you sent a Dogecoin transaction of this denomination to me", essentially allowing decentralized cross-cryptocurrency exchange.

However, the scripting language as implemented in Bitcoin has several important limitations:

- **Lack of Turing-completeness** - that is to say, while there is a large subset of computation that the Bitcoin scripting language supports, it does not nearly support everything. The main category that is missing is loops. This is done to avoid infinite loops during transaction verification; theoretically it is a surmountable obstacle for script programmers, since any loop can be simulated by simply repeating the underlying code many times with an if statement, but it does lead to scripts that are very space-inefficient. For example, implementing an alternative elliptic curve signature algorithm would likely require 256 repeated multiplication rounds all individually included in the code.
- **Value-blindness** - there is no way for a UTXO script to provide fine-grained control over the amount that can be withdrawn. For example, one powerful use case of an oracle contract would be a hedging contract, where A and B put in $1000 worth of BTC and after 30 days the script sends $1000 worth of BTC to A and the rest to B. This would require an oracle to determine the value of 1 BTC in USD, but even then it is a massive improvement in terms of trust and infrastructure requirement over the fully centralized solutions that are available now. However, because UTXO are all-or-nothing, the only way to achieve this is through the very inefficient hack of having many UTXO of varying denominations (eg. one UTXO of 2^k for every k up to 30) and having the oracle pick which UTXO to send to A and which to B.
- **Lack of state** - UTXO can either be spent or unspent; there is no opportunity for multi-stage contracts or scripts which keep any other internal state beyond that. This makes it hard to make multi-stage options contracts, decentralized exchange offers or two-stage cryptographic commitment protocols (necessary for secure computational bounties). It also means that UTXO can only be used to build simple, one-off contracts and not more complex "stateful" contracts such as decentralized organizations, and makes meta-protocols difficult to implement. Binary state combined with value-blindness also mean that another important application, withdrawal limits, is impossible.
- **Blockchain-blindness** - UTXO are blind to blockchain data such as the nonce and previous hash. This severely limits applications in gambling, and several other categories, by depriving the scripting language of a potentially valuable source of randomness.

Thus, we see three approaches to building advanced applications on top of cryptocurrency: building a new blockchain, using scripting on top of Bitcoin, and building a meta-protocol on top of Bitcoin. Building a new blockchain allows for unlimited freedom in building a feature set, but at the cost of development time and bootstrapping effort. Using scripting is easy to implement and standardize, but is very limited in its capabilities, and meta-protocols, while easy, suffer from faults in scalability. With Ethereum, we intend to build a generalized framework that can provide the advantages of all three paradigms at the same time.

Ethereum

The intent of Ethereum is to merge together and improve upon the concepts of scripting, altcoins and on-chain meta-protocols, and allow developers to create arbitrary consensus-based applications that have the scalability, standardization, feature-completeness, ease of development and interoperability offered by these different paradigms all at the same time. Ethereum does this by building what is essentially the ultimate abstract foundational layer: a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. A bare-bones version of Namecoin can be written in two lines of code, and other protocols like currencies and reputation systems can be built in under twenty. Smart contracts, cryptographic "boxes" that contain value and only unlock it if certain conditions are met, can also be built on top of our platform, with vastly more power than that offered by Bitcoin scripting because of the added powers of Turing-completeness, value-awareness, blockchain-awareness and state.

Ethereum Accounts

In Ethereum, the state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. An Ethereum account contains four fields:

- The **nonce**, a counter used to make sure each transaction can only be processed once
- The account's current **ether balance**
- The account's **contract code**, if present
- The account's **storage** (empty by default)

"Ether" is the main internal crypto-fuel of Ethereum, and is used to pay transaction fees. In general, there are two types of accounts: externally owned accounts, controlled by private keys, and contract accounts, controlled by their contract code. An externally owned account has no code, and one can send messages from an externally owned account by creating and signing a transaction; in a contract account, every time the contract account receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn.

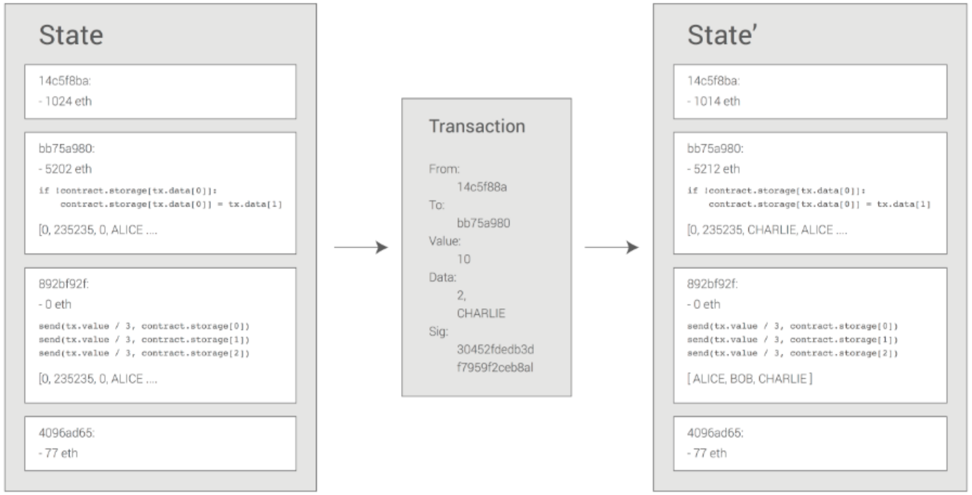
Messages and Transactions

"Messages" in Ethereum are somewhat similar to “transactions” in Bitcoin, but with three important differences. First, an Ethereum message can be created either by an external entity or a contract, whereas a Bitcoin transaction can only be created externally. Second, there is an explicit option for Ethereum messages to contain data. Finally, the recipient of an Ethereum message, if it is a contract account, has the option to return a response; this means that Ethereum messages also encompass the concept of functions.

The term "transaction" is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account. Transactions contain the recipient of the message, a signature identifying the sender, the amount of ether and the data to send, as well as two values called STARTGAS and GASPRICE. In order to prevent exponential blowup and infinite loops in code, each transaction is required to set a limit to how many computational steps of code execution it can spawn, including both the initial message and any additional messages that get spawned during execution. STARTGAS is this limit, and GASPRICE is the fee to pay to the miner per computational step. If transaction execution "runs out of gas", all state changes revert - except for the payment of the fees, and if transaction execution halts with some gas remaining then the remaining portion of the fees is refunded to the sender. There is also a separate transaction type, and corresponding message type, for creating a contract; the address of a contract is calculated based on the hash of the account nonce and transaction data.

An important consequence of the message mechanism is the "first class citizen" property of Ethereum - the idea that contracts have equivalent powers to external accounts, including the ability to send message and create other contracts. This allows contracts to simultaneously serve many different roles: for example, one might have a member of a decentralized organization (a contract) be an escrow account (another contract) between an paranoid individual employing custom quantum-proof Lamport signatures (a third contract) and a co-signing entity which itself uses an account with five keys for security (a fourth contract). The strength of the Ethereum platform is that the decentralized organization and the escrow contract do not need to care about what kind of account each party to the contract is.

Ethereum State Transition Function



```
if !contract.storage[tx.data[0]]:
    contract.storage[tx.data[0]] = tx.data[1]
```

```
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
```

```
if !contract.storage[tx.data[0]]:
    contract.storage[tx.data[0]] = tx.data[1]
```

```
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
```

The Ethereum state transition function, APPLY(S,TX) -> S' can be defined as follows:

1. Check if the transaction is well-formed (ie. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.
2. Calculate the transaction fee as STARTGAS * GASPRICE, and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.
3. Initialize GAS = STARTGAS, and take off a certain quantity of gas per byte to pay for the bytes in the transaction.
4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.
5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

For example, suppose that the contract's code is:

```
if !contract.storage[msg.data[0]]:
  contract.storage[msg.data[0]] = msg.data[1]
```

Note that in reality the contract code is written in the low-level EVM code; this example is written in Serpent, our high-level language, for clarity, and can be compiled down to EVM code. Suppose that the contract's storage starts off empty, and a transaction is sent with 10 ether value, 2000 gas, 0.001 ether gasprice, and two data fields: [2, 'CHARLIE'][3]. The process for the state transition function in this case is as follows:

1. Check that the transaction is valid and well formed.
2. Check that the transaction sender has at least 2000 * 0.001 = 2 ether. If it is, then subtract 2 ether from the sender's account.
3. Initialize gas = 2000; assuming the transaction is 170 bytes long and the byte-fee is 5, subtract 850 so that there is 1150 gas left.
4. Subtract 10 more ether from the sender's account, and add it to the contract's account.
5. Run the code. In this case, this is simple: it checks if the contract's storage at index 2 is used, notices that it is not, and so it sets the storage at index 2 to the value CHARLIE. Suppose this takes 187 gas, so the remaining amount of gas is 1150 - 187 = 963
6. Add 963 * 0.001 = 0.963 ether back to the sender's account, and return the resulting state.

If there was no contract at the receiving end of the transaction, then the total transaction fee would simply be equal to the provided GASPRICE multiplied by the length of the transaction in bytes, and the data sent alongside the transaction would be irrelevant. Additionally, note that contract-initiated messages can assign a gas limit to the computation that they spawn, and if the sub-computation runs out of gas it gets reverted only to the point of the message call. Hence, just like transactions, contracts can secure their limited computational resources by setting strict limits on the sub-computations that they spawn.

Code Execution

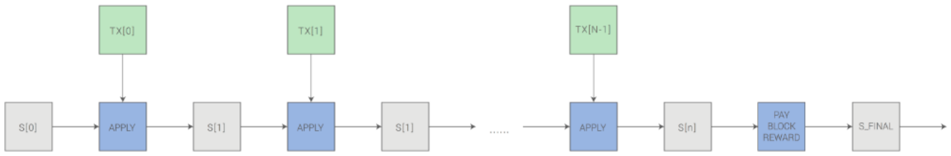
The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte represents an operation. In general, code execution is an infinite loop that consists of repeatedly carrying out the operation at the current program counter (which begins at zero) and then incrementing the program counter by one, until the end of the code is reached or an error or STOP or RETURN instruction is detected. The operations have access to three types of space in which to store data:

- The **stack**, a last-in-first-out container to which 32-byte values can be pushed and popped
- **Memory**, an infinitely expandable byte array
- The contract's long-term **storage**, a key/value store where keys and values are both 32 bytes. Unlike stack and memory, which reset after computation ends, storage persists for the long term.

The code can also access the value, sender and data of the incoming message, as well as block header data, and the code can also return a byte array of data as an output.

The formal execution model of EVM code is surprisingly simple. While the Ethereum virtual machine is running, its full computational state can be defined by the tuple (block_state, transaction, message, code, memory, stack, pc, gas), where block_state is the global state containing all accounts and includes balances and storage. Every round of execution, the current instruction is found by taking the pc-th byte of code, and each instruction has its own definition in terms of how it affects the tuple. For example, ADD pops two items off the stack and pushes their sum, reduces gas by 1 and increments pc by 1, and SSTORE pushes the top two items off the stack and inserts the second item into the contract's storage at the index specified by the first item, as well as reducing gas by up to 200 and incrementing pc by 1. Although there are many ways to optimize Ethereum via just-in-time compilation, a basic implementation of Ethereum can be done in a few hundred lines of code.

Blockchain and Mining



The Ethereum blockchain is in many ways similar to the Bitcoin blockchain, although it does have some differences. The main difference between Ethereum and Bitcoin with regard to the blockchain architecture is that, unlike Bitcoin, Ethereum blocks contain a copy of both the transaction list and the most recent state. Aside from that, two other values, the block number and the difficulty, are also stored in the block. The block validation algorithm in Ethereum is as follows:

1. Check if the previous block referenced exists and is valid.
2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level Ethereum-specific concepts) are valid.
4. Check that the proof of work on the block is valid.
5. Let S[0] be the STATE_ROOT of the previous block.
6. Let TX be the block's transaction list, with n transactions. For all in in 0...n-1, setS[i+1] = APPLY(S[i],TX[i]). If any applications returns an error, or if the total gas consumed in the block up until this point exceeds the GASLIMIT, return an error.
7. Let S_FINAL be S[n], but adding the block reward paid to the miner.
8. Check if S_FINAL is the same as the STATE_ROOT. If it is, the block is valid; otherwise, it is not valid.

The approach may seem highly inefficient at first glance, because it needs to store the entire state with each block, but in reality efficiency should be comparable to that of Bitcoin. The reason is that the state is stored in the tree structure, and after every block only a small part of the tree needs to be changed. Thus, in general, between two adjacent blocks the vast majority of the tree should be the same, and therefore the data can be stored once and referenced twice using pointers (ie. hashes of subtrees). A special kind of tree known as a "Patricia tree" is used to accomplish this, including a modification to the Merkle tree concept that allows for nodes to be inserted and deleted, and not just changed, efficiently. Additionally, because all of the state information is part of the last block, there is no need to store the entire blockchain history - a strategy which, if it could be applied to Bitcoin, can be calculated to provide 5-20x savings in space.

Applications

In general, there are three types of applications on top of Ethereum. The first category is financial applications, providing users with more powerful ways of managing and entering into contracts using their money. This includes sub-currencies, financial derivatives, hedging contracts, savings wallets, wills, and ultimately even some classes of full-scale employment contracts. The second category is semi-financial applications, where money is involved but there is also a heavy non-monetary side to what is being done; a perfect example is self-enforcing bounties for solutions to computational problems. Finally, there are applications such as online voting and decentralized governance that are not financial at all.

Token Systems

On-blockchain token systems have many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization. Token systems are surprisingly easy to implement in Ethereum. The key point to understand is that all a currency, or token systen, fundamentally is is a database with one operation: subtract X units from A and give X units to B, with the proviso that (i) X had at least X units before the transaction and (2) the transaction is approved by A. All that it takes to implement a token system is to implement this logic into a contract.

The basic code for implementing a token system in Serpent looks as follows:

```
from = msg.sender
to = msg.data[0]
value = msg.data[1]

if contract.storage[from] >= value:
    contract.storage[from] = contract.storage[from] - value
    contract.storage[to] = contract.storage[to] + value
```

This is essentially a literal implementation of the "banking system" state transition function described further above in this document. A few extra lines of code need to be added to provide for the initial step of distributing the currency units in the first place and a few other edge cases, and ideally a function would be added to let other contracts query for the balance of an address. But that's all there is to it. Theoretically, Ethereum-based token systems acting as sub-currencies can potentially include another important feature that on-chain Bitcoin-based meta-currencies lack: the ability to pay transaction fees directly in that currency. The way this would be implemented is that the contract would maintain an ether balance with which it would refund ether used to pay fees to the sender, and it would refill this balance by collecting the internal currency units that it takes in fees and reselling them in a constant running auction. Users would thus need to "activate" their accounts with ether, but once the ether is there it would be reusable because the contract would refund it each time.

Financial derivatives and Stable-Value Currencies

Financial derivatives are the most common application of a "smart contract", and one of the simplest to implement in code. The main challenge in implementing financial contracts is that the majority of them require reference to an external price ticker; for example, a very desirable application is a smart contract that hedges against the volatility of ether (or another cryptocurrency) with respect to the US dollar, but doing this requires the contract to know what the value of ETH/USD is. The simplest way to do this is through a "data feed" contract maintained by a specific party (eg. NASDAQ) designed so that that party has the ability to update the contract as needed, and providing an interface that allows other contracts to send a message to that contract and get back a response that provides the price.

Given that critical ingredient, the hedging contract would look as follows:

1. Wait for party A to input 1000 ether.
2. Wait for party B to input 1000 ether.
3. Record the USD value of 1000 ether, calculated by querying the data feed contract, in storage, say this is $x.
4. After 30 days, allow A or B to "ping" the contract in order to send $x worth of ether (calculated by querying the data feed contract again to get the new price) to A and the rest to B.

Such a contract would have significant potential in crypto-commerce. One of the main problems cited about cryptocurrency is the fact that it's volatile; although many users and merchants may want the security and convenience of dealing with cryptographic assets, they many not wish to face that prospect of losing 23% of the value of their funds in a single day. Up until now, the most commonly proposed solution has been issuer-backed assets; the idea is that an issuer creates a sub-currency in which they have the right to issue and revoke units, and provide one unit of the currency to anyone who provides them (offline) with one unit of a specified underlying asset (eg. gold, USD). The issuer then promises to provide one unit of the underlying asset to anyone who sends back one unit of the crypto-asset. This mechanism allows any non-cryptographic asset to be "uplifted" into a cryptographic asset, provided that the issuer can be trusted.

In practice, however, issuers are not always trustworthy, and in some cases the banking infrastructure is too weak, or too hostile, for such services to exist. Financial derivatives provide an alternative. Here, instead of a single issuer providing the funds to back up an asset, a decentralized market of speculators, betting that the price of a cryptographic reference asset will go up, plays that role. Unlike issuers, speculators have no option to default on their side of the bargain because the hedging contract holds their funds in escrow. Note that this approach is not fully decentralized, because a trusted source is still needed to provide the price ticker, although arguably even still this is a massive improvement in terms of reducing infrastructure requirements (unlike being an issuer, issuing a price feed requires no licenses and can likely be categorized as free speech) and reducing the potential for fraud.

Identity and Reputation Systems

The earliest alternative cryptocurrency of all, Namecoin, attempted to use a Bitcoin-like blockchain to provide a name registration system, where users can register their names in a public database alongside other data. The major cited use case is for a DNS system, mapping domain names like "bitcoin.org" (or, in Namecoin's case, "bitcoin.bit") to an IP address. Other use cases include email authentication and potentially more advanced reputation systems. Here is the basic contract to provide a Namecoin-like name registration system on Ethereum:

```
if !contract.storage[tx.data[0]]:
    contract.storage[tx.data[0]] = tx.data[1]
```

The contract is very simple; all it is is a database inside the Ethereum network that can be added to, but not modified or removed from. Anyone can register a name with some value, and that registration then sticks forever. A more sophisticated name registration contract will also have a "function clause" allowing other contracts to query it, as well as a mechanism for the "owner" (ie. the first registerer) of a name to change the data or transfer ownership. One can even add reputation and web-of-trust functionality on top.

Decentralized File Storage

Over the past few years, there have emerged a number of popular online file storage startups, the most prominent being Dropbox, seeking to allow users to upload a backup of their hard drive and have the service store the backup and allow the user to access it in exchange for a monthly fee. However, at this point the file storage market is at times relatively inefficient; a cursory look at various existing solutions shows that, particularly at the "uncanny valley" 20-200 GB level at which neither free quotas nor enterprise-level discounts kick in, monthly prices for mainstream file storage costs are such that you are paying for more than the cost of the entire hard drive in a single month. Ethereum contracts can allow for the development of a decentralized file storage ecosystem, where individual users can earn small quantities of money by renting out their own hard drives and unused space can be used to further drive down the costs of file storage.

The key underpinning piece of such a device would be what we have termed the "decentralized Dropbox contract". This contract works as follows. First, one splits the desired data up into blocks, encrypting each block for privacy, and builds a Merkle tree out of it. One then makes a contract with the rule that, every N blocks, the contract would pick a random index in the Merkle tree (using the previous block hash, accessible from contract code, as a source of randomness), and give X ether to the first entity to supply a transaction with a simplified payment verification-like proof of ownership of the block at that particular index in the tree. When a user wants to re-download their file, they can use a micropayment channel protocol (eg. pay 1 szabo per 32 kilobytes) to recover the file; the most fee-efficient approach is for the payer not to publish the transaction until the end, instead replacing the transaction with a slightly more lucrative one with the same nonce after every 32 kilobytes.

An important feature of the protocol is that, although it may seem like one is trusting many random nodes not to decide to forget the file, one can reduce that risk down to near-zero by splitting the file into many pieces via secret sharing, and watching the contracts to see each piece is still in some node's possession. If a contract is still paying out money, that provides a cryptographic proof that someone out there is still storing the file.

Decentralized Autonomous Organizations

The general concept of a "decentralized organization" is that of a virtual entity that has a certain set of members or shareholders which, perhaps with a 67% majority, have the right to spend the entity's funds and modify its code. The members would collectively decide on how the organization should allocate its funds. Methods for allocating a DAO's funds could range from bounties, salaries to even more exotic mechanisms such as an internal currency to reward work. This essentially replicates the legal trappings of a traditional company or nonprofit but using only cryptographic blockchain technology for enforcement. So far much of the talk around DAOs has been around the "capitalist" model of a "decentralized autonomous corporation" (DAC) with dividend-receiving shareholders and tradable shares; an alternative, perhaps described as a "decentralized autonomous community", would have all members have an equal share in the decision making and require 67% of existing members to agree to add or remove a member. The requirement that one person can only have one membership would then need to be enforced collectively by the group.

A general outline for how to code a DO is as follows. The simplest design is simply a piece of self-modifying code that changes if two thirds of members agree on a change. Although code is theoretically immutable, one can easily get around this and have de-facto mutability by having chunks of the code in separate contracts, and having the address of which contracts to call stored in the modifiable storage. In a simple implementation of such a DAO contract, there would be three transaction types, distinguished by the data provided in the transaction:

- `[0,i,K,V]` to register a proposal with index i to change the address at storage index K to value V
- `[0,i]` to register a vote in favor of proposal i
- `[2,i]` to finalize proposal i if enough votes have been made

The contract would then have clauses for each of these. It would maintain a record of all open storage changes, along with a list of who voted for them. It would also have a list of all members. When any storage

change gets to two thirds of members voting for it, a finalizing transaction could execute the change. A more sophisticated skeleton would also have built-in voting ability for features like sending a transaction, adding members and removing members, and may even provide for Liquid Democracy-style vote delegation (ie. anyone can assign someone to vote for them, and assignment is transitive so if A assigns B and B assigns C then C determines A's vote). This design would allow the DO to grow organically as a decentralized community, allowing people to eventually delegate the task of filtering out who is a member to specialists, although unlike in the "current system" specialists can easily pop in and out of existence over time as individual community members change their alignments.

An alternative model is for a decentralized corporation, where any account can have zero or more shares, and two thirds of the shares are required to make a decision. A complete skeleton would involve asset management functionality, the ability to make an offer to buy or sell shares, and the ability to accept offers (preferably with an order-matching mechanism inside the contract). Delegation would also exist Liquid Democracy-style, generalizing the concept of a "board of directors".

In the future, more advanced mechanisms for organizational governance may be implemented; it is at this point that a decentralized organization (DO) can start to be described as a decentralized autonomous organization (DAO). The difference between a DO and a DAO is fuzzy, but the general dividing line is whether the governance is generally carried out via a political-like process or an "automatic" process; a good intuitive test is the "no common language" criterion: can the organization still function if no two members spoke the same language? Clearly, a simple traditional shareholder-style corporation would fail, whereas something like the Bitcoin protocol would be much more likely to succeed. Robin Hanson's futarchy, a mechanism for organizational governance via prediction markets, is a good example of what truly "autonomous" governance might look like. Note that one should not necessarily assume that all DAOs are superior to all DOs; automation is simply a paradigm that is likely to have have very large benefits in certain particular places and may not be practical in others, and many semi-DAOs are also likely to exist.

Further Applications

1. Savings wallets. Suppose that Alice wants to keep her funds safe, but is worried that she will lose or someone will hack her private key. She puts ether into a contract with Bob, a bank, as follows:

- Alice alone can withdraw a maximum of 1% of the funds per day.
- Bob alone can withdraw a maximum of 1% of the funds per day, but Alice has the ability to make a transaction with her key shutting off this ability.
- Alice and Bob together can withdraw anything.

Normally, 1% per day is enough for Alice, and if Alice wants to withdraw more she can contact Bob for help. If Alice's key gets hacked, she runs to Bob to move the funds to a new contract. If she loses her key, Bob will get the funds out eventually. If Bob turns out to be malicious, then she can turn off his ability to withdraw.

2. Crop insurance. One can easily make a financial derivatives contract but using a data feed of the weather instead of any price index. If a farmer in Iowa purchases a derivative that pays out inversely based on the precipitation in Iowa, then if there is a drought, the farmer will automatically receive money and if there is enough rain the farmer will be happy because their crops would do well.

3. A decentralized data feed. For financial contracts for difference, it may actually be possible to decentralize the data feed via a protocol called "SchellingCoin". SchellingCoin basically works as follows: N parties all put into the system the value of a given datum (eg. the ETH/USD price), the values are sorted, and everyone between the 25th and 75th percentile gets one token as a reward. Everyone has the incentive to provide the answer that everyone else will provide, and the only value that a large number of players can realistically agree on is the obvious default: the truth. This creates a decentralized protocol that can theoretically provide any number of values, including the ETH/USD price, the temperature in Berlin or even the result of a particular hard computation.

4. Smart multi-signature escrow. Bitcoin allows multisignature transaction contracts where, for example, three out of a given five keys can spend the funds. Ethereum allows for more granularity; for example, four out of five can spend everything, three out of five can spend up to 10% per day, and two out of five can spend up to 0.5% per day. Additionally, Ethereum multisig is asynchronous - two parties can register their signatures on the blockchain at different times and the last signature will automatically send the transaction.

5. Cloud computing. The EVM technology can also be used to create a verifiable computing environment, allowing users to ask others to carry out computations and then optionally ask for proofs that computations at certain randomly selected checkpoints were done correctly. This allows for the creation of a cloud computing market where any user can participate with their desktop, laptop or specialized server, and spot-checking together with security deposits can be used to ensure that the system is trustworthy (ie. nodes cannot profitably cheat). Although such a system may not be suitable for all tasks; tasks that require a high level of inter-process communication, for example, cannot easily be done on a large cloud of nodes. Other tasks, however, are much easier to parallelize; projects like SETI@home, folding@home and genetic algorithms can easily be implemented on top of such a platform.

6. Peer-to-peer gambling. Any number of peer-to-peer gambling protocols, such as Frank Stajano and Richard Clayton's Cyberdice, can be implemented on the Ethereum blockchain. The simplest gambling protocol is actually simply a contract for difference on the next block hash, and more advanced protocols can be built up from there, creating gambling services with near-zero fees that have no ability to cheat.

7. Prediction markets. Provided an oracle or SchellingCoin, prediction markets are also easy to implement, and prediction markets together with SchellingCoin may prove to be the first mainstream application of futarchy as a governance protocol for decentralized organizations.

8. On-chain decentralized marketplaces, using the identity and reputation system as a base.

Miscellanea And Concerns

Modified GHOST Implementation

The "Greedy Heaviest Observed Subtree" (GHOST) protocol is an innovation first introduced by Yonatan Sompolinsky and Aviv Zohar in December 2013. The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate - because blocks take a certain time to propagate through the network, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security. Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing a stale block 70% of the time (since the other 30% of the time A produced the last block and so will get mining data immediately) whereas B will have a risk of producing a stale block 90% of the time. Thus, if the block interval is short enough for the stale rate to be high, A will be substantially more efficient simply by virtue of its size. With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network hashpower to have de facto control over the mining process.

As described by Sompolinsky and Zohar, GHOST solves the first issue of network security loss by including stale blocks in the calculation of which chain is the "longest"; that is to say, not just the parent and further ancestors of a block, but also the stale children of the block's ancestors (in Ethereum jargon, "uncles") are added to the calculation of which block has the largest total proof of work backing it. To solve the second issue of centralization bias, we go beyond the protocol described by Sompolinsky and Zohar, and also allow stales to be registered into the main chain to receive a block reward: a stale block receives 93.75% of its base reward, and the nephew that includes the stale block receives the remaining 6.25%. Transaction fees, however, are not awarded to uncles.

Ethereum implements a simplified version of GHOST which only goes down five levels. Specifically, a stale block can only be included as an uncle by the 2nd to 5th generation child of its parent, and not any block with a more distant relation (eg. 6th generation child of a parent, or 3rd generation child of a grandparent). This was done for several reasons. First, unlimited GHOST would include too many complications into the calculation of which uncles for a given block are valid. Second, unlimited GHOST with compensation as used in Ethereum removes the incentive for a miner to mine on the main chain and not the chain of a public attacker. Finally, calculations show that five-level GHOST with incentivization is over 95% efficient even with a 15s block time, and miners with 25% hashpower show centralization gains of less than 3%.

Fees

Because every transaction published into the blockchain imposes on the network the cost of needing to download and verify it, there is a need for some regulatory mechanism, typically involving transaction fees, to prevent abuse. The default approach, used in Bitcoin, is to have purely voluntary fees, relying on miners to act as the gatekeepers and set dynamic minimums. This approach has been received very favorably in the Bitcoin community particularly because it is "market-based", allowing supply and demand between miners and transaction senders determine the price. The problem with this line of reasoning is, however, that transaction processing is not a market; although it is intuitively attractive to construe transaction processing as a service that the miner is offering to the sender, in reality every transaction that a miner includes will need to be processed by every node in the network, so the vast majority of the cost of transaction processing is borne by third parties and not the miner that is making the decision of whether or not to include it. Hence, tragedy-of-the-commons problems are very likely to occur.

However, as it turns out this flaw in the market-based mechanism, when given a particular inaccurate simplifying assumption, magically cancels itself out. The argument is as follows. Suppose that:

1. A transaction leads to k operations, offering the reward kR to any miner that includes it where R is set by the sender and k and R are (roughly) visible to the miner beforehand.
2. An operation has a processing cost of C to any node (ie. all nodes have equal efficiency)
3. There are N mining nodes, each with exactly equal processing power (ie. 1/N of total)
4. No non-mining full nodes exist.

A miner would be willing to process a transaction if the expected reward is greater than the cost. Thus, the expected reward is kR/N since the miner has a 1/N chance of processing the next block, and the processing cost for the miner is simply kC. Hence, miners will include transactions where kR/N > kC, or R > NC. Note that R is the per-operation fee provided by the sender, and is thus a lower bound on the benefit that the sender derives from the transaction, and NC is the cost to the entire network together of processing an operation. Hence, miners have the incentive to include only those transactions for which the total utilitarian benefit exceeds the cost.

However, there are several important deviations from those assumptions in reality:

1. The miner does pay a higher cost to process the transaction than the other verifying nodes, since the extra verification time delays block propagation and thus increases the chance the block will become a stale.
2. There do exist non-mining full nodes.

3. The mining power distribution may end up radically inegalitarian in practice.
4. Speculators, political enemies and crazies whose utility function includes causing harm to the network do exist, and they can cleverly set up contracts whose cost is much lower than the cost paid by other verifying nodes.

Point 1 above provides a tendency for the miner to include fewer transactions, and point 2 increases NC; hence, these two effects at least partially cancel each other out. Points 3 and 4 are the major issue; to solve them we simply institute a floating cap: no block can have more operations than BLK_LIMIT_FACTOR times the long-term exponential moving average. Specifically:

```
blk.oplimit = floor((blk.parent.oplimit * (EMAFACTOR - 1) + floor(parent.opcount * BLK_LIMIT_FACTOR)) /
EMA_FACTOR)
```

BLK_LIMIT_FACTOR and EMA_FACTOR are constants that will be set to 65536 and 1.5 for the time being, but will likely be changed after further analysis.

Computation And Turing-Completeness

An important note is that the Ethereum virtual machine is Turing-complete; this means that EVM code can encode any computation that can be conceivably carried out, including infinite loops. EVM code allows looping in two ways. First, there is a JUMP instruction that allows the program to jump back to a previous spot in the code, and a JUMPI instruction to do conditional jumping, allowing for statements like while x < 27: x = x * 2. Second, contracts can call other contracts, potentially allowing for looping through recursion. This naturally leads to a problem: can malicious users essentially shut miners and full nodes down by forcing them to enter into an infinite loop? The issue arises because of a problem in computer science known as the halting problem: there is no way to tell, in the general case, whether or not a given program will ever halt.

As described in the state transition section, our solution works by requiring a transaction to set a maximum number of computational steps that it is allowed to take, and if execution takes longer computation is reverted but fees are still paid. Messages work in the same way. To show the motivation behind our solution, consider the following examples:

- An attacker creates a contract which runs an infinite loop, and then sends a transaction activating that loop to the miner. The miner will process the transaction, running the infinite loop, and wait for it to run out of gas. Even though the execution runs out of gas and stops halfway through, the transaction is still valid and the miner still claims the fee from the attacker for each computational step.
- An attacker creates a very long infinite loop with the intent of forcing the miner to keep computing for such a long time that by the time computation finishes a few more blocks will have come out and it will not be possible for the miner to include the transaction to claim the fee. However,

the attacker will be required to submit a value for STARTGAS limiting the number of computational steps that execution can take, so the miner will know ahead of time that the computation will take an excessively large number of steps.

- An attacker sees a contract with code of some form like send(A,contract.storage[A]); contract.storage[A] = 0, and sends a transaction with just enough gas to run the first step but not the second (ie. making a withdrawal but not letting the balance go down). The contract author does not need to worry about protecting against such attacks, because if execution stops halfway through the changes get reverted.
- A financial contract works by taking the median of nine proprietary data feeds in order to minimize risk. An attacker takes over one of the data feeds, which is designed to be modifiable via the variable-address-call mechanism described in the section on DAOs, and converts it to run an infinite loop, thereby attempting to force any attempts to claim funds from the financial contract to run out of gas. However, the financial contract can set a gas limit on the message to prevent this problem.

The alternative to Turing-completeness is Turing-incompleteness, where JUMP and JUMPI do not exist and only one copy of each contract is allowed to exist in the call stack at any given time. With this system, the fee system described and the uncertainties around the effectiveness of our solution might not be necessary, as the cost of executing a contract would be bounded above by its size. Additionally, Turing-incompleteness is not even that big a limitation; out of all the contract examples we have conceived internally, so far only one required a loop, and even that loop could be removed by making 26 repetitions of a one-line piece of code. Given the serious implications of Turing-completeness, and the limited benefit, why not simply have a Turing-incomplete language? In reality, however, Turing-incompleteness is far from a neat solution to the problem. To see why, consider the following contracts:

```
C0: call(C1); call(C1);
C1: call(C2); call(C2);
C2: call(C3); call(C3);
...
C49: call(C50); call(C50);
C50: (run one step of a program and record the change in storage)
```

Now, send a transaction to A. Thus, in 51 transactions, we have a contract that takes up 2^{50} computational steps. Miners could try to detect such logic bombs ahead of time by maintaining a value alongside each contract specifying the maximum number of computational steps that it can take, and calculating this for contracts calling other contracts recursively, but that would require miners to forbid contracts that create other contracts (since the creation and execution of all 50 contracts above could easily be rolled into a single contract). Another problematic point is that the address field of a message is a variable, so in general it may not even be possible to tell which other contracts a given contract will call ahead of time. Hence, all in all, we have a surprising conclusion: Turing-completeness is surprisingly easy to manage, and the lack of

Turing-completeness is equally surprisingly difficult to manage unless the exact same controls are in place - but in that case why not just let the protocol be Turing-complete?

Currency And Issuance

The Ethereum network includes its own built-in currency, ether, which serves the dual purpose of providing a primary liquidity layer to allow for efficient exchange between various types of digital assets and, more importantly, of providing a mechanism for paying transaction fees. For convenience and to avoid future argument (see the current mBTC/uBTC/satoshi debate in Bitcoin), the denominations will be pre-labelled:

- 1: wei
- 10^12: szabo
- 10^15: finney
- 10^18: ether

This should be taken as an expanded version of the concept of "dollars" and "cents" or "BTC" and "satoshi". In the near future, we expect "ether" to be used for ordinary transactions, "finney" for microtransactions and "szabo" and "wei" for technical discussions around fees and protocol implementation.

The issuance model will be as follows:

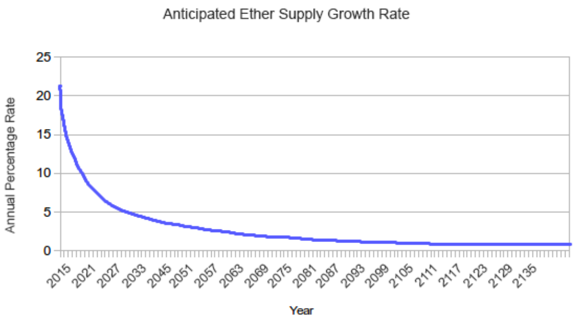
- Ether will be released in a currency sale at the price of 1337-2000 ether per BTC, a mechanism intended to fund the Ethereum organization and pay for development that has been used with success by a number of other cryptographic platforms. Earlier buyers will benefit from larger discounts. The BTC received from the sale will be used entirely to pay salaries and bounties to developers, researchers and projects in the cryptocurrency ecosystem.
- 0.099x the total amount sold will be allocated to early contributors who participated in development before BTC funding or certainty of funding was available, and another 0.099x will be allocated to long-term research projects.
- 0.26x the total amount sold will be allocated to miners per year forever after that point.

Issuance Breakdown

The permanent linear supply growth model reduces the risk of what some see as excessive wealth concentration in Bitcoin, and gives individuals living in present and future eras a fair chance to acquire currency units, while at the same time discouraging depreciation of ether because the "supply growth rate" as a percentage still tends to zero over time. We also theorize that because coins are always lost over time due to carelessness, death, etc, and coin loss can be modeled as a percentage of the total supply per year, that the total currency supply in circulation will in fact eventually stabilize at a value equal to the annual issuance divided by the loss rate (eg. at a loss rate of 1%, once the supply reaches 26X then 0.26X will be mined and 0.26X lost every year, creating an equilibrium).

Group	At launch	After 1 year	After 5 years
Currency units	1.198X	1.458X	2.498X
Purchasers	83.5%	68.6%	40.0%
Early contributor distribution	8.26%	6.79%	3.96%
Long-term endowment	8.26%	6.79%	3.96%
Miners	0%	17.8%	52.0%

Despite the linear currency issuance, just like with Bitcoin over time the supply growth rate nevertheless tends to zero.



Mining Centralization

The Bitcoin mining algorithm basically works by having miners compute SHA256 on slightly modified versions of the block header millions of times over and over again, until eventually one node comes up with a version whose hash is less than the target (currently around 2^{190}). However, this mining algorithm is vulnerable to two forms of centralization. First, the mining ecosystem has come to be dominated by ASICs (application-specific integrated circuits), computer chips designed for, and therefore thousands of times more efficient at, the specific task of Bitcoin mining. This means that Bitcoin mining is no longer a highly decentralized and egalitarian pursuit, requiring millions of dollars of capital to effectively participate in. Second, most Bitcoin miners do not actually perform block validation locally; instead, they rely on a centralized mining pool to provide the block headers. This problem is arguably worse: as of the time of this writing, the top two mining pools indirectly control roughly 50% of processing power in the Bitcoin network, although this is mitigated by the fact that miners can switch to other mining pools if a pool or coalition attempts a 51% attack.

The current intent at Ethereum is to use a mining algorithm based on randomly generating a unique hash function for every 1000 nonces, using a sufficiently broad range of computation to remove the benefit of specialized hardware. Such a strategy will certainly not reduce the gain of centralization to zero, but it does not need to. Note that each individual user, on their private laptop or desktop, can perform a certain quantity of mining activity almost for free, paying only electricity costs, but after the point of 100% CPU utilization of their computer additional mining will require them to pay for both electricity and hardware. ASIC mining companies need to pay for electricity and hardware starting from the first hash. Hence, if the centralization gain can be kept to below this ratio, (E + H) / E, then even if ASICs are made there will still be room for ordinary miners.

Additionally, we intend to design the mining algorithm so that mining requires access to the entire blockchain, forcing miners to store the entire blockchain and at least be capable of verifying every transaction. This removes the need for centralized mining pools; although mining pools can still serve the legitimate role of evening out the randomness of reward distribution, this function can be served equally well by peer-to-peer pools with no central control. It additionally helps fight centralization, by increasing the number of full nodes in the network so that the network remains reasonably decentralized even if most ordinary users prefer light clients.

Scalability

One common concern about Ethereum is the issue of scalability. Like Bitcoin, Ethereum suffers from the flaw that every transaction needs to be processed by every node in the network. With Bitcoin, the size of the current blockchain rests at about 20 GB, growing by about 1 MB per hour. If the Bitcoin network were to process Visa's 2000 transactions per second, it would grow by 1 MB per three seconds (1 GB per hour, 8 TB per year). Ethereum is likely to suffer a similar growth pattern, worsened by the fact that there will be many applications on top of the Ethereum blockchain instead of just a currency as is the case with Bitcoin, but ameliorated by the fact that Ethereum full nodes need to store just the state instead of the entire blockchain history.

The problem with such a large blockchain size is centralization risk. If the blockchain size increases to, say, 100 TB, then the likely scenario would be that only a very small number of large businesses would run full nodes, with all regular users using light SPV nodes. In such a situation, there arises the potential concern that the full nodes could band together and all agree to cheat in some profitable fashion (eg. change the block reward, give themselves BTC). Light nodes would have no way of detecting this immediately. Of course, at least one honest full node would likely exist, and after a few hours information about the fraud would trickle out through channels like Reddit, but at that point it would be too late: it would be up to the ordinary users to organize an effort to blacklist the given blocks, a massive and likely infeasible coordination problem on a similar scale as that of pulling off a successful 51% attack. In the case of Bitcoin, this is currently a problem, but there exists a blockchain modification suggested by Peter Todd which will alleviate this issue.

In the near term, Ethereum will use two additional strategies to cope with this problem. First, because of the blockchain-based mining algorithms, at least every miner will be forced to be a full node, creating a lower bound on the number of full nodes. Second and more importantly, however, we will include an intermediate state tree root in the blockchain after processing each transaction. Even if block validation is centralized, as long as one honest verifying node exists, the centralization problem can be circumvented via a verification protocol. If a miner publishes an invalid block, that block must either be badly formatted, or the state S[n] is incorrect. Since S[0] is known to be correct, there must be some first state S[i] that is incorrect where S[i-1] is correct. The verifying node would provide the index i, along with a "proof of invalidity" consisting of the subset of Patricia tree nodes needing to process APPLY(S[i-1],TX[i]) -> S[i]. Nodes would be able to use those nodes to run that part of the computation, and see that the S[i] generated does not match the S[i] provided.

Another, more sophisticated, attack would involve the malicious miners publishing incomplete blocks, so the full information does not even exist to determine whether or not blocks are valid. The solution to this is a challenge-response protocol: verification nodes issue "challenges" in the form of target transaction indices, and upon receiving a node a light node treats the block as untrusted until another node, whether the miner or another verifier, provides a subset of Patricia nodes as a proof of validity.

Putting It All Together: Decentralized Applications

The contract mechanism described above allows anyone to build what is essentially a command line application run on a virtual machine that is executed by consensus across the entire network, allowing it to modify a globally accessible state as its “hard drive”. However, for most people, the command line interface that is the transaction sending mechanism is not sufficiently user-friendly to make decentralization an attractive mainstream alternative. To this end, a complete “decentralized application” should consist of both low-level business-logic components, whether implemented entirely on Ethereum, using a combination of Ethereum and other systems (eg. a P2P messaging layer, one of which is currently planned to be put into the Ethereum clients) or other systems entirely, and high-level graphical user interface components. The Ethereum client’s design is to serve as a web browser, but include support for a “eth” Javascript API object, which specialized web pages viewed in the client will be able to use to interact with the Ethereum blockchain. From the point of view of the “traditional” web, these web pages are entirely static content, since the blockchain and other decentralized protocols will serve as a complete replacement for the server for the purpose of handling user-initiated requests. Eventually, decentralized protocols, hopefully themselves in some fashion using Ethereum, may be used to store the web pages themselves.

Conclusion

The Ethereum protocol was originally conceived as an upgraded version of a cryptocurrency, providing advanced features such as on-blockchain escrow, withdrawal limits and financial contracts, gambling markets and the like via a highly generalized programming language. The Ethereum protocol would not "support" any of the applications directly, but the existence of a Turing-complete programming language means that arbitrary contracts can theoretically be created for any transaction type or application. What is more interesting about Ethereum, however, is that the Ethereum protocol moves far beyond just currency. Protocols and decentralized applications around decentralized file storage, decentralized computation and decentralized prediction markets, among dozens of other such concepts, have the potential to substantially increase the efficiency of the computational industry, and provide a massive boost to other peer-to-peer protocols by adding for the first time an economic layer. Finally, there is also a substantial array of applications that have nothing to do with money at all.

The concept of an arbitrary state transition function as implemented by the Ethereum protocol provides for a platform with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, Ethereum is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come.

Notes and Further Reading

Notes

1. A sophisticated reader may notice that in fact a Bitcoin address is the hash of the elliptic curve public key, and not the public key itself. However, it is in fact perfectly legitimate cryptographic terminology to refer to the pubkey hash as a public key itself. This is because Bitcoin's cryptography can be considered to be a custom digital signature algorithm, where the public key consists of the hash of the ECC pubkey, the signature consists of the ECC pubkey concatenated with the ECC signature, and the verification algorithm involves checking the ECC pubkey in the signature against the ECC pubkey hash provided as a public key and then verifying the ECC signature against the ECC pubkey.
2. Technically, the median of the 11 previous blocks.
3. Internally, 2 and "CHARLIE" are both numbers, with the latter being in big-endian base 256 representation. Numbers can be at least 0 and at most 2^256-1.

Further Reading

1. Intrinsic value: https://tinyurl.com/BitcoinMag-IntrinsicValue
2. Smart property: https://en.bitcoin.it/wiki/Smart_Property
3. Smart contracts: https://en.bitcoin.it/wiki/Contracts
4. B-money: http://www.weidai.com/bmoney.txt
5. Reusable proofs of work: http://www.finney.org/~hal/rpow/
6. Secure property titles with owner authority: http://szabo.best.vwh.net/securetitle.html
7. Bitcoin whitepaper: http://bitcoin.org/bitcoin.pdf
8. Namecoin: https://namecoin.org/
9. Zooko's triangle: http://en.wikipedia.org/wiki/Zooko's_triangle
10. Colored coins whitepaper: https://tinyurl.com/coloredcoin-whitepaper
11. Mastercoin whitepaper: https://github.com/mastercoin-MSC/spec
12. Decentralized autonomous corporations, Bitcoin Magazine: https://tinyurl.com/Bootstrapping-DACs
13. Simplified payment verification:https://en.bitcoin.it/wiki/Scalability#Simplifiedpaymentverification
14. Merkle trees: http://en.wikipedia.org/wiki/Merkle_tree
15. Patricia trees: http://en.wikipedia.org/wiki/Patricia_tree
16. GHOST: http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf
17. StorJ and Autonomous Agents, Jeff Garzik: https://tinyurl.com/storj-agents
18. Mike Hearn on Smart Property at Turing Festival: http://www.youtube.com/watch?v=Pu4PAMFPo5Y

19. Ethereum RLP: https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-RLP
20. Ethereum Merkle Patricia trees: https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-Patricia-Tree
21. Peter Todd on Merkle sum trees:http://sourceforge.net/p/bitcoin/mailman/message/31709140/