View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

piChain: When a Blockchain Meets Paxos

Conrad Burchert[1] and Roger Wattenhofer[2]

1 ETH Zurich, Switzerland
bconrad@ethz.ch
2 ETH Zurich, Switzerland
wattenhofer@ethz.ch

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.2017.2

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.

© Conrad Burchert and Roger Wattenhofer;
licensed under Creative Commons License CC-BY

- 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.

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.

[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.

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.

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:

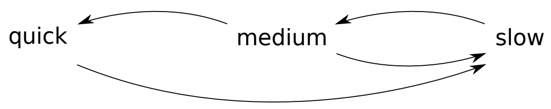
$$\begin{tikzcd}
\text{quick} \arrow[rr, bend right, ""] & \text{medium} \arrow[l, bend right] \arrow[r, bend right] & \text{slow} \arrow[l, bend right]
\end{tikzcd}$$

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

[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.
[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                    $b_{\text{max}} = \perp$
    $b_{\text{com}} = \perp$                                $b_{\text{prop}} = \perp$
                                                 $b_{\text{supp}} = \perp$

    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 $b_{\text{max}}$ then
                                              4:    $b_{\text{max}} = b_{\text{new}}$
                                              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($b_{\text{com}}$, $b_{\text{new}}$) mes-
                                                 sage:
                                             14: if $b_{\text{new}} = b_{\text{max}}$ 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
```

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 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_{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 [14], Spanner [7], 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 [12, 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]

[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.

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 [8, 10, 13].

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 [9]. This is implemented in various cryptocurrencies, e.g., the masternode system of Dash [11]. 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:

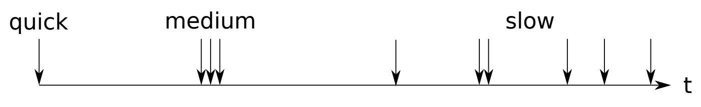

Figure 2 Waiting time of nodes in different states: A quick node creates a block instantly when seeing a new transaction. A medium node waits long enough to ensure not to compete with a quick node. A slow node waits long enough to ensure to compete neither with a quick nor a medium node. The figure represents 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 $4+2\epsilon+r$ 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

[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.

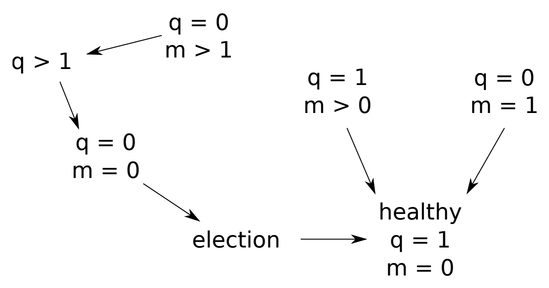

Figure 3 Recovery of the network to the healthy state. The nodes of this graph represent all possible states of the whole system: how many quick (q) nodes do we have and if relevant how many medium (m) nodes. The election state is special, as it is characterized by possibly multiple medium nodes, which 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.

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 Figure 3.

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).

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.

[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.

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 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.

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

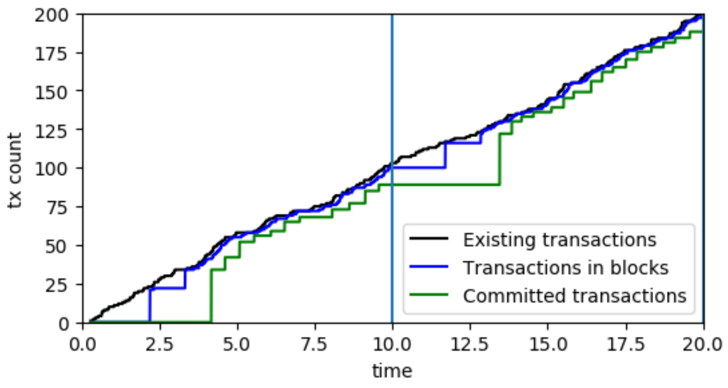

Figure 4 piChain with $n = 20$. After 10s the single quick node crashes. No new blocks are created until a slow node creates a block and becomes medium. A bit later, the same node becomes quick, and blocks are again created and committed with a fast pace.

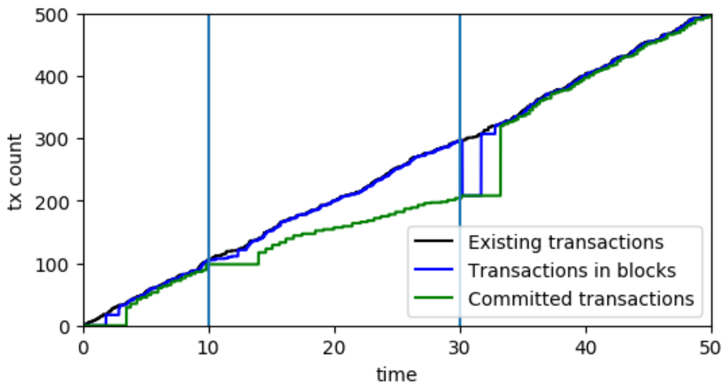

Figure 5 The network is partitioned into 8 resp. 12 nodes after 10s. Each side of the partition quickly finds a single quick node that is adding blocks, but only the majority side can commit its blocks. When the partition is resolved after 30s, the minority side temporarily loses its blocks because of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

Figure 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.

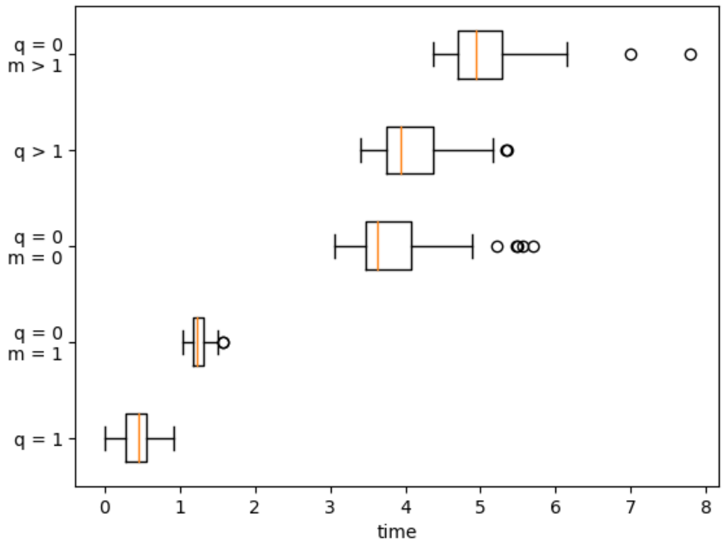

Figure 6 The time piChain needs to become healthy when being started in a random state. The start states are clustered according to Figure 3, e.g., when a third of the nodes is initially in each category quick, medium, and slow, we add a measurement to class $q > 1$.

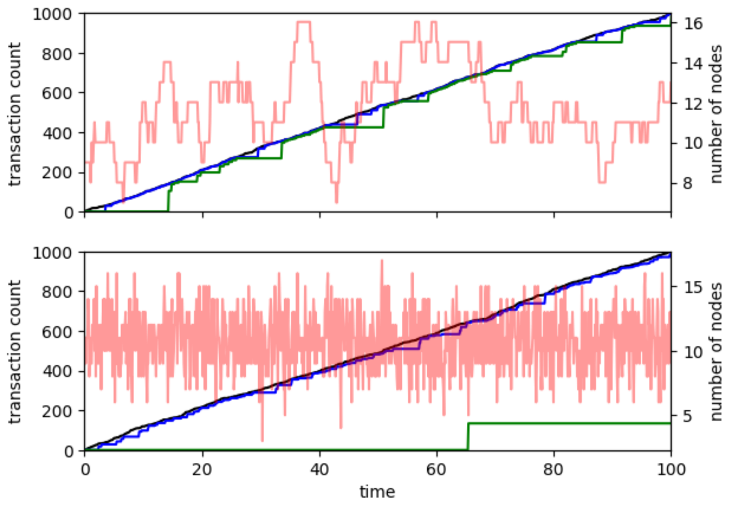

Figure 7 piChain with unstable nodes that crash and recover. The red curves show the number of currently working nodes. In the upper plot crashes last for an average of 20s, in the lower plot for 0.2s, but both plots have the same average number of 11 working nodes. One can see that blocks are generated 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.

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 James C. Corbett, Jeffrey Dean, 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, and Dale Woodford. Spanner: Google's globally-distributed database, 2012.

8 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.

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

10 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.

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

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

13 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.

14 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.

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.