ComChain: A Blockchain with Byzantine Fault Tolerant Reconfiguration*

Guillaume Vizier
Ecole Polytechnique, France

Vincent Gramoli
University of Sydney, CSIRO Data61

Abstract

Selecting which blockchain participants can decide upon a new block is a difficult problem. Consortium blockchains need the participants to be predetermined while public blockchains incentivize all participants to waste their resources to decide every block.

In this paper, we introduce the *community blockchain* that allows potentially all participants to decide upon "some" block while restricting the set of participants deciding upon "one" block. To this end, we propose a blockchain reconfiguration, a Byzantine consensus protocol that allows to dynamically change the set of blockchain participants deciding upon the upcoming blocks. The resulting blockchain, called *ComChain*, is resilience optimal, and transitions through different configurations of participants recorded in dedicated blocks, so that each configuration decides upon its subsequent transaction blocks. We evaluate an implementation that adds reconfiguration to the Red Belly Blockchain and demonstrates its practical performance in a distributed system.

Keywords. Community blockchain, reconfiguration, Byzantine consensus

1 Introduction

A blockchain is an abstraction representing a linked list of transaction blocks implemented in a distributed system of nodes whose roles can be classified as follows: (i) *clients* that can read (audit) and write (transact) the data and (ii) *deciders* that run a consensus algorithm to decide upon the new block of transactions at a given index of the chain. There exist various types of blockchains that differ mostly in the way their nodes have permissions to play specific roles, and in the consensus algorithm that deciders run to agree upon new blocks.

In public blockchains [31, 36], all nodes are potentially deciders and can participate in the creation of new blocks [40]. To cope with Sybil attacks, they typically rely on proof-of-work challenges that restrict the power of a malicious user to its resources. Provided that the malicious user does not own a large portion of the resources of the system, it cannot impose its decision to the others. As all nodes can be deciders and rewards are in place to incentivise nodes to become deciders, these systems typically consume a large amount of resources. For example, Bitcoin and Ethereum consume individually as much electricity as a reasonably-sized country.[1]

In consortium and private blockchains, proof-of-work and other synchronous alternatives are rarely used because they expose these blockchains to known network attacks [32, 17, 18]. Instead, the permission for

*The short version of this paper appeared in the proceedings of IEEE Blockchain 2018 [41]. The current version extends the other with the proofs of correctness and termination, as well as new experiments.

[1]The electricity consumed by Bitcoin and Ethereum are respectively estimated to 24 and 9 terawatts per hour according to `http://bitcoinenergyconsumption.com` and `http://ethereumenergyconsumption.com`.

some nodes to act as deciders is hardcoded [5, 26, 38]. The drawback of hardcoding these permissions is that the membership is static: each time a decider must be added or removed, then the whole blockchain must be stopped and restarted [39, 42]. This lack of dynamism is a major issue in long-lived blockchain where hardware components eventually fail and consortia naturally evolve.[2] As a result, such systems are often considered too "centralized" due to the inalterable power they offer to their deciders.

In this paper we cope with these two issues, by offering a *community blockchain* model that bridges the gap between public and consortium/private blockchains. In particular, a community blockchain (i) inherently copes with Sybil attacks by identifying its dynamic set of deciders and (ii) allows any participants to become a decider without wasting the resources of all participants.

To this end, the community blockchain relies on a new type of block: the *configuration block*. Its role is to define among all participants a subset of deciders, or community, responsible for deciding the upcoming transaction blocks. More precisely, each configuration block lists a *configuration* as a set of decider nodes identified by their public key. These n decider nodes keep adding new transaction blocks. After some time, these decider nodes propose new configurations, as any set of nodes, to each other, and, despite $t < \frac{n}{3}$ misbehaving or Byzantine nodes, $n-t$ correct must reach a consensus on one configuration. These deciders have the responsibility of selecting a configuration that is *acceptable* according to application-specific rules (e.g., new deciders must be sufficiently representative of the participants, they must be owned by k different companies and hosted in ℓ different jurisdictions, and must not have been held accountable for misbehaviors in the past [8]).

We also propose a partially synchronous implementation of this community blockchain, called *ComChain*. ComChain builds upon the Red Belly Blockchain [24, 13], the fastest blockchain we know of. In ComChain, the genesis block stores the initial configuration as the set of deciders. Upon reception of transactions, these deciders validate them and agree to append a new block of validated transactions. Once these deciders reach an agreement on a new acceptable configuration, they sign the configuration and store this signed configuration into a new configuration block. From this point on, the new configuration defines the new set of deciders that will append the next transaction blocks until the next reconfiguration and so on.

The community blockchain model allows potentially all participants to decide upon "some" block while restricting the set of participants deciding upon "one" block. As opposed to the consortium blockchain model that has a predetermined set of deciders [40], the community blockchain model is dynamic. As opposed to the public blockchain model that incentivizes all participants to be deciders of the same block, our model limits the waste of resources. We prove that our algorithm terminates and does not impact the resilience of the consensus.

We also deploy ComChain on 12 physical machines to evaluate the performance of the reconfiguration and its impact on the blockchain service when executed concurrently to transaction invocations. These experiments demonstrate empirically two things: *(i)* Performance decreases with the number of nodes to add but increases with the number of nodes to remove. *(ii)* While the reconfiguration affects the throughput of the blockchain, it disrupts the blockchain service.

To the best of our knowledge, no deterministic blockchain implementation support a dynamic membership. Traditional ideas consist of picking the block of the fastest node to solve a random cryptopuzzle through proof-of-work [31, 36] or selecting a random set of nodes through sortition [20]. All these ideas fundamentally rely on randomness to prevent any group of nodes from controlling the outcome of the selection. For the sake of security, however, the set of deciders has to be frequently renewed so that many uncontrollable random processes get executed in a long-lived system. Repeating these random selections will eventually select nodes prone to attacks (e.g., belonging to partnering companies who can build a coalition or too few jurisdictions so that judges can stop the service). This randomness is in contradiction with the concept of community blockchain that trusts the current configuration to choose an acceptable new

[2] R3 had 11 participants in 2016 and includes now more than 70 participants.

configuration.

Section 2 presents an overview of the related work. Section 3 introduces our model. Section 4 presents an overview of our solution at a high level. Section 5 specifies a detailed implementation, called ComChain, and proves it correct. Section 6 illustrates how our reconfiguration mechanism performs empirically. Section 7 discusses some of our assumptions. Section 8 concludes.

2 Related work

In this section, we present the related work, both in the areas of blockchain systems and of distributed systems reconfiguration.

2.1 Reconfiguring distributed systems

Reconfiguration of distributed systems have been extensively studied in the past as it finds applications in cluster membership changes [27], atomic storage [7, 21, 35], file systems [15], replicated state machines [30, 1] and rolling upgrades [23]. The most common types of reconfigurable services are atomic storage that supports reads and writes [21, 22] and replicated state machine that supports generic commands, like transactions [30, 1]. Blockchains resemble replicated state machines in that they also support transactions, however, each new decided transaction block depends on the previous block as opposed to the replicated state machine commands that are decided regardless of one another [10].

Paxos and Vertical Paxos [29] were proposed originally as crash-tolerant consensus algorithms that can be used for reconfiguration. Byzantine fault tolerant variants of both algorithms exist. Like typical Byzantine fault tolerant consensus algorithms [6], they rely on a leader to solve consensus in order to reconfigure and prevent blockchains to scale. In other papers [35, 9, 2], the service providing the information about the configuration of the system is distributed across several nodes, however, the configuration changes require an authorised trusted party to be decided. Just like a permissionless blockchain model, the community model has to scale to many nodes and cannot rely on any trusted entity or a single leader.

The Byzantine reconfiguration of a service is usually a non trivial algorithm as it must cohabit with the service itself. Unfortunately, some solutions are individual components of a larger system that is not described [15, 38]. Often, the way the new configuration is decided is described, but not the way to actually change the set of nodes of the considered system, which can lead to unexpected problems.

2.2 Blockchains

Byzantine consensus algorithms recently gained in popularity in blockchain systems for their ability to cope with malicious behaviors without resorting to an energy-greedy proof-of-work [10, 39, 24, 13]. Little work on this topic has been formalized and most of the available documentation is presented informally [43].

Hybrid mechanisms were recently proposed to combine Byzantine fault tolerance (BFT) with proof-of-work [34] in order to elect a committee of decider nodes that will then run consensus only to process transactions. As the authors explain, however, this protocol is not guaranteed to be secure and may fork in what they call their *snailchain*. ByzCoin [28] also relies on proof-of-work to select a committee but relies on a leader to solve consensus. As it is impossible to elect a correct leader in a Byzantine environment, such an approach may suffer performance degradation.

Similar trust assumptions as the ones we used in Section 4.5 were formalized in proof-of-stake blockchains [14]. Proof-of-stake protocols naturally imply reconfiguration as the voting power evolves with the amount of money owned by a node, and thus with time.

Algorand [20] relies on a new BFT algorithm. Their algorithm uses reconfiguration in the sense that only a fraction of all available users actively contribute to the consensus algorithm at each step. Their tolerance

model is stronger than ours as they resist an attacker that can corrupt nodes instantaneously but the number of nodes an adversary can corrupt is bounded in the same way as ours: not more than a third of participating nodes can be corrupted.

Unfortunately, all the aforementioned solutions cannot work in a partially synchronous environment: all the messages they exchange have to be delivered in less than a predefined amount of time for the algorithms to work.

To the best of our knowledge, no deterministic blockchain implementation working in a partially synchronous environment supports a dynamic membership. Probabilistic alternatives revert typically to proof-of-* [31, 36], randomized consensus [16] or sortition [20]. This randomness is in contradiction with the concept of community blockchain that trusts the current configuration to choose an acceptable new configuration deterministically.

Deterministic reconfiguration was however suggested online in some Byzantine fault tolerant extensions of existing blockchain projects [39, 42]. Hyperledger Fabric aims at supporting membership changes without compromising the network (cf. `https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#dynamic-membership`), however, as of May 30, 2019, it "requires that the peer or orderer process is restarted"[3]. In addition, as of May 30, 2019, Hyperledger Fabric is not Byzantine Fault Tolerant, as it uses Kafka that can only tolerate crash faults. An orderer was made BFT using BFTS-mart but accepts only timestamps, and not transactions. Tendermint mentions validator set changes [42], however, as of May 30, 2019 as well, this requires an external application that handles those reconfigurations (cf. Section 7.1, second paragraph of `https://atrium.lib.uoguelph.ca/xmlui/bitstream/handle/10214/9769/Buchman_Ethan_201606_MAsc.pdf?sequence=7&isAllowed=y`). In contrast, our Byzantine fault tolerant reconfiguration is non-disruptive and the main feature of ComChain.

3 Model

The system is made up of a set π of k asynchronous sequential processes, namely $\Pi = \{p_1, \ldots, p_k\}$; i is called the "index" of p_i. "Asynchronous" means that each process proceeds at its own speed, which can vary with time and remains unknown to the other processes. "Sequential" means that a process executes one step at a time. Fisher and al. have proven [19] that the consensus problem cannot be solved when nodes can possibly fail. Thus, we need to make additional assumptions on the network. Several major running blockchains [31, 36, 33] assume synchrony in the network: in other words, they assume that every message arrives after at most a predefined time period. Even recent papers [20] make the same assumption. But, this assumption, no matter how the predefined time period is, does not seem realistic: how can anyone ensure that a split in the connection between two actors of the systems will always last for less than that duration? Hence, we make a weaker assumption: all messages eventually arrive. In other words, we do not know a priori for how long we will wait, but every message will someday arrive at its destination. This assumption is the *partial synchrony*.

The classic failure model considers that a node is either *correct* in that it follows its specification or *Byzantine*, in which case it behaves arbitrarily. But the status of a node never changes, which is unlikely if the blockchain is supposed to run during a long time. Thus, we define a dynamic failure model. A node is said to be *correct during a time frame* Δt if it follows its specification during that time interval. The aforementioned time period Δt is not defined with a fixed duration. A node is said to be *Byzantine during a time frame* Δt if it stops following its specification at any time during that time interval. A node is said to be *correct (respectively Byzantine) within the* i^{th} *consensus instance* if it is correct (respectively Byzantine) for the time frame between its proposal and its decision for this consensus instance.

[3] `https://hyperledger-fabric.readthedocs.io/en/latest/msp.html#msp-setup-on-the-peer-orderer-side`, accessed on May 30. 2019.

A *consensus instance* is one execution of a consensus algorithm. Consensus instances occur sequentially. It is launched by nodes proposing a value. $n-t$ nodes need to propose a value in order to launch a consensus instance. The nodes taking part in the consensus propose regularly, and thus launch consensus instances.

Definition 1. *A* configuration *is a set* Π *of n nodes, fulfilling the following requirements :*

- ***Network****: we have an asynchronous point-to-point reliable network for our n nodes*[4][5]*,*
- ***Awareness****: all correct nodes have knowledge of the full blockchain,*
- ***Correctness threshold****: for all instances of consensus I decided by this configuration, the inequality* $\frac{2n}{3} < C_I$ *is verified, where* C_I *is the number of* ***correct*** *nodes in* Π *for consensus I.*

We assume that the nodes always have enough storage capacities to store the blockchain-related information and the non-committed transactions issued by the clients.

In the following, n, n', ... represent the number of nodes participating in the consensus, and t, t', ... are upper-bounds on the number of Byzantine nodes among them.

Definition 2. *A* new configuration*, with regard to a configuration* Π*, is a configuration* Π' *of* n' *nodes, fulfilling the following additional requirements :*

- ***Network****: we have an asynchronous point-to-point reliable network for all correct nodes in* $\Pi \cup \Pi'$*,*
- ***Listening****: all correct nodes of* Π' *expect to receive information from the current configuration, and possess the code to handle this information correctly. Receiving information from the DNS service introduced below belongs to this* ***Listening*** *requirement.*

Note that we do not make any assumption on the behavior of the nodes that are not part of the current configuration: once they accomplished their duty and have been removed from the set of decider nodes, they can become Byzantine, crash, etc.

To ensure the consistency of the information present in our blockchain, nodes sign the new configurations they propose, using an asymmetric encryption scheme. Thus, each node possesses a couple of private-public keys, and each node knows the public key of every other node taking part in the consensus. We assume that no private key can be forged or stolen.

For new nodes to know the current configuration, we need to make an additional trust assumption. For clients to know the current configuration without needing to examine the whole blockchain, we assume to have a correct DNS service keeping track of the latest configuration. For new incoming nodes, they also need to examine the whole blockchain to prevent corruption of the first blocks a posteriori. This service can be distributed, in order to be Byzantine fault tolerant, but we consider it as a whole throughout our paper, as it is out of the scope of this paper. Similarly to Bitcoin [31] that hardcodes few DNS seeds, the location of this server has to be static for clients and nodes to be able to contact it without any further information.

4 The Community Blockchain

In this section, we introduce *community blockchains* that bridge the gap between public and consortium blockchains. Community blockchains differ from public blockchains by constraining the set of deciders for a particular block and differ from consortium or private blockchains by letting all nodes decide upon some block. To this end, the community blockchain reconfigures periodically the set of deciders by replacing the current configuration by a new configuration that is valid.

[4] Point-to-point reliable channel can be implemented with secure channels in a partially synchronous environment.

[5] We actually only need a network between correct nodes, hence the difference in Definition 2.

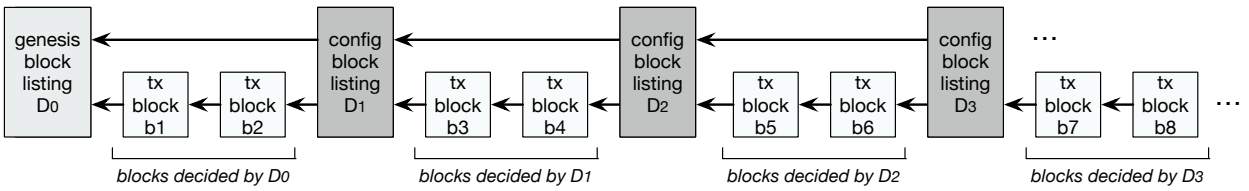


Figure 1: A community blockchain where each transaction block is decided by the deciders listed in the last preceding configuration block (starting from the genesis block on the left and where each block points backward towards its predecessor block(s))

4.1 Two lists of blocks

The community blockchain contains two types of blocks: the classic or *transaction blocks* that contain transactions and the *configuration blocks* (cf. Definition 3) that contain a list of deciders. As indicated in Figure 1 these two types of blocks are linked so that the blockchain maintains a structure similar to a skip list. At the bottom, all transaction and configuration blocks are chained together with a linked list. This guarantees that the configurations and the set of transactions are totally ordered. At the top, the configuration blocks are chained together with another linked list. This helps verify that the current configuration is properly signed by the deciders of the previous configuration.

Each block is linked to the previous block, no matter their respective type: we keep the general structure of a chain of blocks. *In addition*, each **configuration block** is linked to the previous configuration block.

The number of *transaction blocks* between two configuration blocks is not fixed. We will come back on the details about blocks in Section 5.8.

Definition 3 (Configuration block)**.** *A* configuration block *is a block containing:*

1. *Information of the decider nodes of the new configuration: IP addresses, ports and public keys.*
2. *The signatures of this new configuration by at least $t+1$ deciders of the previous configuration.*
3. *The hash and index of the last configuration block (indicating the previous configuration).*
4. *The hash and index of the last transaction block (indicating the last committed transactions).*

Note that the genesis block is a special kind of configuration block that does not contain the hash and index of previous blocks, but simply the information of the decider nodes of the initial configuration.

4.2 Deciding upon a new configuration

The idea of reconfiguration simply consists of replacing the current configuration in use by the system by a new configuration. In a community blockchain system, a reconfiguration must guarantee for example that the system can keep serving the transaction requests while the set of decider nodes is being reconfigured. This implies redirecting transactions from a configuration to a new one as soon as the reconfiguration occurs (and the new configuration decider nodes are aware of it) but not earlier. We present the goal of the reconfiguration a community blockchain should ensure below.

Definition 4 (Reconfiguration)**.** *A* reconfiguration *replaces a configuration c by a different (possibly overlapping) configuration c' such that:*

1. *c' is proposed by a node;*
2. *all correct nodes of c agree upon c';*

3. *c' verifies a validity predicate (c' is valid);*

4. *there is no service interruption;*[6]

5. *there is no data loss;*

6. *data integrity is preserved;*

7. *the reconfiguration takes a finite amount of time.*

The requirements (1-2) and (7) are guaranteed by a consensus algorithm. The data integrity (6) is preserved by the community blockchain. We explain how a configuration can be valid (3) and how the system stays uninterrupted (4) while data is kept (5) below.

4.3 Verifying that a configuration is valid

Depending on the application, some configurations are not acceptable. This could be because all deciders belong to the same geographical region, because they are managed by the same company, because they are not sufficiently random or simply because the number of deciders is not acceptable. In a community blockchain, the correct nodes of the current configuration have the responsibility of assessing whether a proposed configuration is acceptable. As mentioned in the Introduction, these decider nodes might also have the responsibility of proposing suitable configurations for the application-dependent context (e.g., nodes hosted in different continents and countries, representing different institutions, etc). Although there is no way to distinguish a Byzantine node from a correct node, especially if the Byzantine send correct messages, a recent cryptographic protocol, Polygraph [8], identifies deterministically Byzantine nodes that are responsible for a disagreement. Future work could exploit the techniques of such a protocol to remove responsible nodes from a valid configuration. Provided that every correct node has a way to identify an acceptable configuration, we propose a mechanism to guarantee that only an acceptable configuration can be included in a configuration block.

We assume in the following that we have at our disposal a predicate valid_configuration that allows us to decide whether a configuration is valid or not. This predicate is application-dependent, and its implementation is out of the scope of this paper.

To this end, we use a public-key cryptosystem and list in each configuration block, starting from the genesis block, the public key (in addition to their IP address and port number) of the decider nodes of the corresponding configuration. This allows decider nodes to sign (using their private key) the new configuration that they decide. This signature is easily verifiable using the decider public key available in the configuration block.

To propose a new configuration, a node must first send this configuration to the other $n-1$ nodes and launch a binary Byzantine consensus. Each node receiving this configuration takes part in this binary Byzantine consensus instance and checks whether the configuration is acceptable by proposing 1 (if acceptable) or 0 (if not). By definition of the binary Byzantine consensus, the value that is decided has to be proposed by a correct, hence the consensus decides 1 (meaning that a consensus is reached regarding the acceptability of this configuration) only if at least one correct node found it acceptable. If the consensus decides 1, then every correct node signs the configuration and sends it to the requester.

We write that some information is *signed by a configuration c* if it is signed by at least $t+1$ nodes among the n decider nodes of this configuration c. As t is the maximum number of Byzantine nodes, $t+1$ signatures guarantee that at least one signature comes from a correct node. Given a current configuration c, we write that a newly decided configuration c' is *valid* and that $\mathsf{valid}(c')$ returns `true` if the configuration c' has been

[6] Recall that we assumed that the memory of our nodes is always sufficient to receive the proposed transactions.

signed by the current configuration c, and contains the hashes and indexes of both the last configuration block and last transaction block.

Note that, for the sake of simplicity, we assume that if a correct node proposes a *configuration block*, every correct node proposes a configuration block too. The configurations proposed by different nodes can be identical or different: this detail does not matter in our algorithms. As precised in Section 5.4, if several valid configurations are proposed, the consensus algorithm of the underlying blockchain will decide which configuration will be the next one.

4.4 Transferring blocks and pending transactions

Once a new configuration is decided, we need to distinguish whether new decider nodes absent from the old configuration are added to the new configuration, whether decider nodes are removed, or both new decider nodes are added and old decider nodes are removed. In particular, adding decider nodes requires to transfer the blockchain to the new node whereas removing decider nodes requires to have these removed nodes stop accepting new transactions and transfer their pending transactions to the deciders of the new configuration.

Adding nodes If we add nodes, without removing any of the former participants, to keep our threshold of strictly more than two third of correct nodes, we need to be sure that the newly added nodes are aware of their new role, and possess enough information to perform their task. Thus, the old nodes send to all new nodes a copy of the up-to-date blockchain, and the new nodes need to have the full knowledge of this blockchain, by verifying its chain of hashes, before taking part in the consensus. Note that as our model is partially synchronous, we can safely launch the next consensus instance without waiting for the new nodes to be up-to-date.

Removing nodes If we only remove nodes, the data that might be lost is not the blockchain, or the already committed transactions, but the pending transactions. Thus, in our reconfiguration mechanism, the nodes-to-be-removed have to transfer all their transactions to nodes of the new configuration, which will then propose them to the new consensus instance. To ensure that they eventually transfer all of them, they stop acknowledging and taking into account arriving transactions after the new configuration has been decided. Each node can thus stop working when it has no more transaction to send.

Adding and removing nodes To handle the general case, we just use two steps: we first add the needed nodes, and once these are ready to perform consensus because they have caught up with the latest state of the blockchain, we remove the unneeded ones.

4.5 Catching up with the most up-to-date information

In this section, we explain how a new node aware of some old configuration can retrieve the most up-to-date configuration and block information. The difficulty is that do not impose an old configuration of n nodes to remain correct, hence by the time the new node tries to catch up, the old configuration it knows is not guaranteed to have strictly more than $\frac{2n}{3}$ correct nodes anymore.

4.5.1 With the latest configuration

A correct node part of the current configuration keeps receiving information regarding the ongoing reconfiguration. As new nodes may join and leave, they need a mechanism to retrieve the latest configuration to know where to send their transaction and balance requests. In particular, by the time a new node joins the system, the system may have moved to the k^{th} configuration where the $k-1$ preceding configurations may

be faulty in that they no longer have less than a third of Byzantine nodes. To cope with this issue, every new correct node bootstraps by contacting the DNS service that provides them with the latest configuration.

More specifically, there are two cases where we need to be able to verify that a configuration has actually been decided by the consensus:

- When a configuration has just been decided.
- When a node connects to the blockchain for the first time, or rejoins after at least one configuration change.

In the first case, a node already knows for sure the current configuration. Thus it can know, when receiving a configuration, if it has been signed by at least $t+1$ nodes of the current configuration (cf. Section 4.3). Moreover, it can wait for receiving $t+1$ identical configurations, sent by the current configuration, which then are correct. In the second case, we have to trust the DNS service to give the current configuration. Then we connect to the current configuration as indicated by the DNS service, and listen for the next configuration changes.

4.5.2 With the latest state of the blockchain

A node receiving a new valid configuration block looks into the list of decider nodes of the new configuration, and sees whether it is part of the new configuration or not.

If the node has to become a decider node, it needs to know about the latest state of the blockchain, in order to have the same validity predicate for transactions as all other correct nodes. Thus, all decider nodes of the old configuration send the whole chain of blocks to new nodes. Upon reception of $t+1$ identical blockchains from different decider nodes of the old configuration, the new node knows that at least one has been sent by a correct node, and thus represents the current state of the blockchain. The new node then knows the whole history of the blockchain, and is able to perform its duty.

Note that we cannot use the simple distinction of light vs. full nodes used in classic blockchain because our goal is to tolerate Byzantine failures deterministically for critical applications. In particular, even a light node cannot trust another full node as it can behave maliciously. This is reason why a new node (that could be viewed initially as light) needs to contact at least $t+1$ nodes. Note that the same kind of technique is used in the Red Belly Blockchain to prevent a wallet that cannot store the entire blockchain from taking into account false information from a decider node [13].

5 Putting the Community to Work

In this section, we propose a community blockchain implementation, called *ComChain*, that builds upon the Red Belly Blockchain [24]. The Red Belly Blockchain features the Democratic Byzantine Fault Tolerant (DBFT) consensus algorithm [11] to solve the Blockchain Consensus problem, where nodes cooperate to select a block [12] as restated in Section 5.2. Its particularity is its absence of leader that allows it to reach an unprecedented throughput in terms of the number of transactions it can commit per second [24].

We consider a system where a predetermined set of decider nodes belong to an initial configuration stored in the genesis block. These nodes use a reliable multicast described in Section 5.1 to reliably exchange messages among the decider nodes of a configuration. The choice of the initial configuration is left to future work but, depending on the application, could require nodes to belong to different geographical regions, different institutions, and having distinct hardware and system specifications for the sake of security.

5.1 Reliable multicast

In order to exchange information between the nodes of a configuration, we define two reliable multicast primitives based on the classic definition of reliable broadcast, presented by Bracha [3, 4] and reused in DBFT [10]. Note that the term "multicast" is used here to indicate that the information is propagated to subsets of decider nodes represented by configurations, as opposed to the classic "broadcast" that propagates to all nodes without distinction. The reliable broadcast is a communication primitive among n nodes where at most $t < \frac{n}{3}$ can be Byzantine [3]. This abstraction provides two primitives, RB_broadcast and RB_deliver. With p a node, in one instance of the broadcast protocol, this abstraction has the following properties:

- *Validity*. If p is correct and a correct node RB-delivers a message m from p, then p RB-broadcast m.
- *Unicity*. A correct node RB-delivers at most one message from p (whether p is correct or not).
- *Termination-1*. If p is correct and RB-broadcasts a message m, all the correct nodes eventually RB-deliver m from p.
- *Termination-2*. If a correct node RB-delivers a message m from p (possibly Byzantine) then all the correct nodes eventually RB-deliver the same message m from p.

As we need to make a distinction between messages containing proposals for transactions, for configurations or agreement on the validity of a configuration, we define new multicast primitives, that precise the set of nodes it is sent to.

Definition 5. *We define two operations* RM_broadcast_new *and* RM_deliver_new*, being the* RB_broadcast *(respectively the* RB_deliver*) operation from a node to all nodes of the new configuration not being part of the old configuration.*

Definition 6. *We define two operations* RM_broadcast_old *and* RM_deliver_old*, being the* RB_broadcast *(respectively the* RB_deliver*) operation from a node to all nodes of the old configuration.*

5.2 Blockchain consensus

The blockchain consensus consists for a set of nodes to collaboratively decide upon a new block [10, 12]. This problem is different from the proof-of-work Blockchain problem [31, 36] and the classic Byzantine agreement problem [2, 5, 38] where all nodes compete in trying to force others to decide the block they propose. We restate below the definition of blockchain consensus as originally referred to as *Validity Predicate-based Byzantine Consensus* [10, 12, 11].

Definition 7 (Blockchain Consensus)**.** *With the assumption that every correct process proposes a value to the consensus, each correct process decides on a value while satisfying Termination, Agreement and Blockchain validity:*

- ***Termination:*** *Every correct process decides after a finite amount of time.*
- ***Agreement:*** *Two correct processes decide on the same value.*
- ***Blockchain validity:*** *The value decided by a correct process verifies a predefined predicate* valid().

Note that the Blockchain validity allows nodes to collaboratively decide a value that results from all the proposed values, as opposed to the validity of the classic Byzantine agreement where the value decided can only be one of the proposed value.

Algorithm 1 Collaborative signature of configurations

```
function HANDLESIGNATUREREQUESTS
    when RB_deliver signing request for c from p_i do
        BinaryConsensus(valid_configuration(c), GETID(c, p_i))
    end
    when BinaryConsensus{id'} returns 1 do
        ⟨c', p_j⟩ ← GETITEMSFROMID(id')
        c' ← SIGN(c')
        Send c' to p_j
    end
    when RB_deliver signed configuration c'' from p_k do
        if EQUALCONFIGURATIONS(c'', self.sc) then
            MERGESIGNATURES(c'')
        end if
    end
end function
```

Crain et al. [10] propose a solution to the blockchain consensus problem where each correct node proposes a set of transactions to be committed. It uses a reliable broadcast primitives to exchange these proposals among correct nodes until these nodes obtain an array of $n-t$ proposals. Then every correct node spawns n binary consensus instances and fill a bitmask with the n corresponding decisions. Finally, they apply the bitmask to the array of proposals to extract multiple sets of transactions. The decided block is built from all the extracted transactions whose signatures are correct and that do not conflict with each other. An *instance* of consensus is one execution of a consensus algorithm on a set of proposals.

5.3 Collaboratively signing a configuration

As described in Section 4.3 , before proposing a configuration, a node has to gather $t+1$ ECDSA[7] signatures of nodes of the current configuration. Therefore, a correct node broadcasts the new configuration for signature by the current configuration. Upon reception of signature requests from other nodes, it executes Algorithm 1. For the sake of symmetry, a node broadcasts also the new configuration to sign to itself. We assume that `true` and 1 are equivalent, as well as `false` and 0. As validity is application dependent (cf. Section 4.3), we consider that each correct node is equipped with a predicate valid_configuration that, given a configuration c, returns `true` only if c is valid according to the current application.

As binary consensus is needed to decide whether a configuration is valid, let us introduce two notations for the binary consensus instance associated with a particular decider node *id* and the configuration it proposes v.

Definition 8. BinaryConsensus(v, id) *represents the proposal of the (binary) value v to the instance of the binary consensus uniquely identified with id.* BinaryConsensus$\{id'\}$ *represents the binary consensus instance having identifier* id'*. We say that* BinaryConsensus$\{id'\}$ *returns when a value has been decided for the binary consensus instance of identifier* id'*.*

We also define two additional methods to turn this pair of $\langle v, id\rangle$ (where v is a binary value or boolean and *id* is a node identifier) associated with a binary consensus instance, into the identifier id' of this binary consensus instance and to reverse the operation. GETID is defined in Algorithm 2. GETITEMS is the associated getter. We use $self.sc.c$ to refer to the configuration (if any) that the node wants other participants to sign. A configuration as stored by a node is a tuple:

[7] The Elliptic Curve Digital Signature Algorithm (ECDSA) is the asymmetric crypto-library used by Bitcoin [31].

Algorithm 2 From a signature request to a binary consensus

```
function GETID(c, p_i)
    h ← sha256(c.i)          // with '.' the concatenation
    self.consensus_ids[h] ← (c, i)
    return hash
end function
```

- c, the configuration as a set of node identifiers, initially $\emptyset$ and
- *signatures_set*, an array of signatures indexed by node identifiers, initially $\perp$ (undefined) at all indices

These functions are implemented with a dictionary-like structure that maintains the computed identifiers for the launched binary consensus instances. This structure allows us to retrieve the configuration and the process identifier from the identifier of a binary consensus instance.

The MERGESIGNATURES function groups the signatures present in the incoming signed configuration with these already gathered for the next-to-propose configuration. It is described below in Algorithm 3.

Algorithm 3 Merging two signatures of the same configuration

```
function MERGESIGNATURES(c)
    h ← sha256(self.sc.c);
    for ns in c.signature_set do
        already_signed ← false;
        for signature in self.sc.signatures_set do
            if signature.signer = ns.signer then
                already_signed ← true;
            end if
        end for
        if not already_signed then
            if ns.signer in current_configuration and
                    verify(ns.signer, ns.hash) = h then
                self.sc.signatures_set.append(ns)
            end if
        end if
    end for
end function
```

5.4 Deciding upon a new configuration

To decide on the next configuration, we launch a *multivalue* consensus algorithm among all nodes, whose proposed and decided values are not necessarily binary values. To this end, we use the partially synchronous DBFT algorithm [10] as a black box, with the serialized signed configuration as input and the function HASENOUGHSIGNATURES, presented in Algorithm 4, as valid predicate. In this algorithm, *cc* refers to the current configuration.

More precisely, the DBFT algorithm returns the value decided by all correct processes. We will use this value to fulfill the configuration change by inputing it in the suitable functions described in the next section. To decide whether we have to execute the AddNodes function, the RemoveNodes function or both, we simply check the inclusion of one configuration in the other: (i) if the new configuration is strictly included in the old configuration then we remove nodes; (ii) if the old configuration is strictly included in the new one, then we add nodes; (iii) if none of them is included in the other, then we run both AddNodes

Algorithm 4 The valid predicate for configurations

```
function HASENOUGHSIGNATURES(c)
    h ← sha256(c.configuration);
    counter ← 0;
    for s in c.signatures_set do
        if s.signer ∈ cc ∧ verify(s.signer, s.hash) = h then
            counter ← counter + 1;
        end if
    end for
    return counter ≥ t + 1
end function
```

and RemoveNodes; (iv) if each of them is included in the other, then we do not do anything as the new configuration is the old one.

The set of signatures being finite, our valid predicate terminates in finite time. Thus, as the consensus algorithm presented in [10] terminates in finite time, a decision about a potential new configuration is made in finite time.

5.5 Transition from the current to the new configuration

The functions we use above return a configuration proposed by a node that verifies validity conditions (modeled by the valid_configuration predicate) and on which all correct nodes agree. The behavior after having chosen such a new configuration differs from the one after having decided a block of transactions.

As previously indicated in Section 5.4 , after the consensus returns a value, we need to process this value with the functions previously described in Section 4.

5.5.1 Adding nodes

If the list of nodes of the current configuration is strictly included in the new list of nodes, then we are only adding new nodes to our community.

For old nodes to add new nodes, they use the protocol presented in Algorithm 5.

Algorithm 5 Participation to the consensus when adding nodes

```
function DECIDECONFIGURATIONWHENADDING(c)
    ADDCONFIGURATIONBLOCK(c)
    RM_broadcast_new blockchain
    RM_broadcast_old the last block index
    Run consensus on the new configuration
    when (RM_deliver_old last index of n' − t' nodes) do
        Inform the DNS service that transactions should be
        sent to the new configuration
        // clients should send their txs to the new config
    end
end function
```

The function ADDCONFIGURATIONBLOCK used in Algorithm 5 simply adds a new configuration block to the blockchain. We do not need to wait for the new nodes to be ready, because according to their specification, described below in Section 5.6 , they will not take part in the consensus before being up-to-date. Thus, if these new nodes are needed as correct nodes to reach consensus, the consensus algorithm will wait

for them to be up-to-date before deciding anything. We could implement a waiting mechanism to wait for them before launching the next consensus instance, which would only reduce the performance.

Provided that new decider nodes RM_broadcast_old the last index of the blockchain in finite time (shown in Lemma 4), the node receives the last index of the blockchain from all correct nodes of the new configuration in finite time. Thus it sends a request to update the DNS in finite time.

5.5.2 Removing nodes

If the new list of participants is strictly included in the list of nodes representing the current configuration, then we are removing nodes. We do so by using the protocol described in Algorithm 6 where *self* depicts the node running the function.

Algorithm 6 Participation to the consensus when removing nodes

```
function DECIDECONFIGURATIONWHENREMOVING(c)
    Inform the DNS service that txs should be sent to the new config;
    // clients send their txs to the new config.
    if self in the nodes to remove then
        stop acknowledging transactions
        start transmitting the queued txs to the new config
        when the node has no transaction left do
            shut down
        end
    else
        perform consensus with the new configuration
    end if
end function
```

As a node only has a finite number of transactions to process, and as a node-to-be-removed stops accepting transactions, such a node shuts down in finite time.

5.5.3 Adding and removing nodes

Replacing an old configuration by a new configuration that is independent can be achieved by both adding and removing nodes from the old configuration. With the old configuration *old_configuration*, and the new configuration c, we use the first case above (adding nodes) to switch from *old_configuration* to *old_configuration* $\cup\, c$, then the second case (removing nodes) to switch from *old_configuration* $\cup\, c$ to c.

Algorithm 7 depicts how to merge these two cases. It requires every node to have already stored both c, the next configuration, and $c \cup$ *old_configuration*, with at least $t+1$ signatures. Thus, we need:

- either each node to sign both the requested configuration c and the union of this configuration c with the current one *old_configuration* (in Algorithm 1) and to transmit this information to every other node,
- or to ask every node to request the signature of $c \cup$ *old_configuration* before creating the configuration block.

The former approach adds complexity in the storage and retrieval of signed configurations, but limits the exchanges on the network. The latter approach is less efficient, because each node will launch n binary consensus instances in addition to the normal complexity, in order for each of them to get the union of configurations signed.

Algorithm 7 Behavior of a node participating to the consensus

```
function DECIDECONFIGURATIONGENERALCASE(c)
    addConfigurationBlock(c ∪ old_configuration)
    run consensus on the union of old and new configs.
    when (RM_deliver_old last index of n' − t' nodes) do
        Inform the DNS service of the new config.
        // clients send their txs to the new config.
        if self not in c then
            stop acknowledging transactions
        end if
    end
    if self not in c then
        when self has no transactions left do
            RM_broadcast_new "no transactions left";
        end
    else
        RM_broadcast_new "no transactions left";
    end if
    when RM_deliver_old n − t times "no transaction left" do
        ProposeConfiguration(c, change);
    end
end function
```

5.6 Behavior of a new node

So far, we only focused on the behavior of nodes that took part in the blockchain. We now focus on the behavior of newly joining nodes. Algorithm 8 presents the way new nodes take part in the consensus.

Algorithm 8 Protocol used by the nodes of the new configuration to prepare themselves for running the consensus

```
function PREPARECONFIGURATION
    when RM_deliver_new t + 1 identical blockchains do
        store this blockchain as the valid one
        RM_broadcast_old index of the last block
        start behaving like a decider node
    end
end function
```

5.7 Safety

In this section, we prove that our algorithm decides a new configuration using a *Blockchain Consensus* algorithm (see Definition 7). We also prove that the threshold used in the definition of a **new** configuration (see Definition 2) is correct.

5.7.1 Blockchain validity

For configurations, the validity predicate we consider is the valid_configuration predicate. Thus we have to prove that a configuration decided by a correct node verifies that predicate.

Lemma 1. *A configuration decided by a correct node verifies the* valid_configuration() *predicate.*

Proof. At line 2 of Algorithm 1 , we see that each correct node will propose the result of the valid_configuration predicate on a configuration to the binary consensus instance associated with this configuration. Thus, the binary consensus instance related to a configuration that does not satisfy the valid_configuration predicate will return 0: a binary consensus instance can only return a value proposed by a majority of nodes, and we have at most t Byzantine nodes. Hence if every correct node proposes $0 = \mathsf{valid\_configuration}(\mathit{invalid\_configuration})$ at line 2 of Algorithm 1 , at least $2t+1$, thus a majority of, nodes propose 0, which forces the consensus algorithm to return 0.

At line 6 of Algorithm 1 , we see that only Byzantine nodes can sign an invalid configuration, which is at most t: a correct node signs only configurations for which the binary consensus instance returned 1, and thus are valid with respect to our validity predicate. As the valid predicate of the underlying consensus algorithm is the HASENOUGHSIGNATURES function presented in Algorithm 4 , only configurations signed by at least $t+1$ nodes of the current configuration can be decided. Among these $t+1$ nodes, at most t are Byzantine, so at least one correct node signed the configuration. Thus, a configuration verifying the valid predicate of the underlying consensus algorithm verifies the valid_configuration predicate.

As the underlying consensus algorithm ensures that the values decided by correct nodes verify its valid predicate, a configuration decided by a correct node verifies the valid_configuration predicate. □

We proved that all configurations decided by correct nodes are valid.

5.7.2 Agreement

In this section, we prove that the decision on a new configuration respects the *Agreement* property of the Blockchain consensus (see Definition 7).

Lemma 2. *Two correct decider nodes decide on the same configuration.*

Proof. As the configuration decided by a correct node is the result of a multivalued instance of a consensus algorithm verifying the *Blockchain consensus* (see Section 5.4), the Agreement property holds. □

We proved that all correct nodes will decide upon the same new configuration for every reconfiguration.

5.7.3 Termination

In this section we prove that we can always change the configuration in a finite amount of time. We do not prove that every single function terminates, but focus on the parts that could last forever. More precisely, by relying on the termination proof of DBFT [10], we show that we can have a configuration signed within a finite amount of time (Lemma 3), and that new nodes are up-to-date after a finite amount of time.

Lemma 3. *A node having requested signatures for a valid configuration receives this configuration signed by the current configuration within a finite amount of time.*

Proof. The RB-broadcast abstraction ensures that every correct node receives the configuration to sign after a finite amount of time. Thus, with p_i the process proposing the configuration c, every correct node calls $\mathsf{BinaryConsensus}(\mathsf{valid\_configuration}(c), \textsc{GetId}(c, p_i))$, where $\mathsf{valid\_configuration}(c)$ should return 1.

Crain et al. [10] prove that the binary consensus we use terminates, and signing occurs in a finite amount of time. Thus, the node having requested signatures for the above mentioned configuration receives at least $2t+1$ signed configurations in finite time.

Merging the signatures is achieved through two loops over finite sets, and thus terminates. Hence, a node having requested signatures for a valid configuration has this configuration, signed by the current configuration in a finite amount of time. □

We proved that a node is able to propose a valid configuration to the consensus in finite time.

The decision upon a new configuration is given by the underlying consensus algorithm, thus takes a finite amount of time (see Definition 7).

We now prove that new nodes contribute to the consensus after a finite time period.

Lemma 4. *A correct new node participates to the consensus after a finite timeframe.*

Proof. Crain et al. [10] ensure that we reach consensus in finite time, thus a new node receives at least $n-t \geq 2t+1$ up-to-date blockchains in finite time, thanks to the RB-broadcast abstract presented in Section 5.1 , and the partial synchrony assumption.

Thus, a new node receives at least $t+1$ identical blockchains in finite time, one of which has been sent by a correct decider node, and thus is correct, and is then up-to-date with the latest history of the blockchain. □

We proved that new nodes will not be considered Byzantine because they are not up-to-date, and that they will indeed be eventually up-to-date.

We now prove that the DNS knows the decided configuration in finite time.

Lemma 5. *The DNS service is updated after a finite amount of time.*

Proof. To be Byzantine fault tolerant, the DNS has to wait for an identical update message from at least $t+1$ different decider nodes of the current configuration.

As precised at the end of Section 5.5.1 , and using Lemma 4 , when adding nodes, all correct nodes from the previous configuration send an update message to the DNS in finite time.

When removing nodes, updating the DNS is the first step of the algorithm, thus the update message is also sent in finite time to the DNS by all correct nodes.

Thus, in every situation, the DNS receives at least $t+1$ identical update messages in finite time, thanks to the partial synchrony assumption. Hence, the DNS is updated after a finite timeframe. □

We proved that clients and newly joining nodes will have a reliable way of knowing the current configuration in a finite amount of time after the change actually occurred.

5.7.4 Threshold for the new configuration

In this section, we prove that the resilience of the consensus algorithm does not change between two configurations.

Lemma 6. *To be able to run an instance of a consensus algorithm with a threshold of Byzantine tolerance $n > f(t)$, using the new configuration, we need to verify :*

$$n' > f(t')$$

where n' is the number of nodes in the new configuration, and t' the number of Byzantine nodes in this same configuration and for this particular instance of consensus.

Proof. When we change the nodes taking part to this consensus algorithm, the newly added nodes need to update their knowledge of the current blockchain before taking part to the consensus.

Lemma 4 ensures that new correct nodes will contribute to the consensus after a finite time. As we do not use the bound on the delay messages can suffer on the network in our consensus algorithm (it only needs the partial synchrony assumption), we do not need to wait for the nodes to be up-to-date before launching the next consensus instance on all up-to-date nodes of the proposed configuration.

Thus we can include all the new nodes in the next consensus instance, and as the new configuration respects the same threshold, they will eventually be enough correct nodes to perform consensus. □

Lemma 4 was proven using the DBFT algorithm [10], but the result we use from this paper is a liveness result. Hence, Lemma 6 still holds with consensus algorithms having the same liveness properties.

In addition, as algorithms needing $f(x) = 2x + 1$ exist [10], our threshold for the ratio for the number of correct nodes in a new reconfiguration is valid.

5.8 A resilience optimal blockchain service

In this section, we explain how ComChain piggybacks transactions and configurations within the same block for the sake of efficiency and demonstrate that it is resilience optimal.

5.8.1 Piggybacking transactions and configurations

So far, we have handled only configuration blocks, assuming that for a given consensus instance, either only configuration blocks are proposed, or only transaction blocks are proposed. This is also an easy way to dissociate the processing of the configurations and transactions.

The assumption we made is unrealistic. For an implementation, we merge the definitions of configuration block and transaction block: every block contains both a configuration (possibly empty or null) and a batch of transactions (possibly empty). We then have to adapt the way we validate a block: we split the block in the two aforementioned types of blocks, validate each one separately, and say that a general block is valid if both the transaction block and the configuration block we built from are valid. Naturally, we have to consider an empty transaction block or an empty configuration block as valid.

5.8.2 Resilience optimality of ComChain

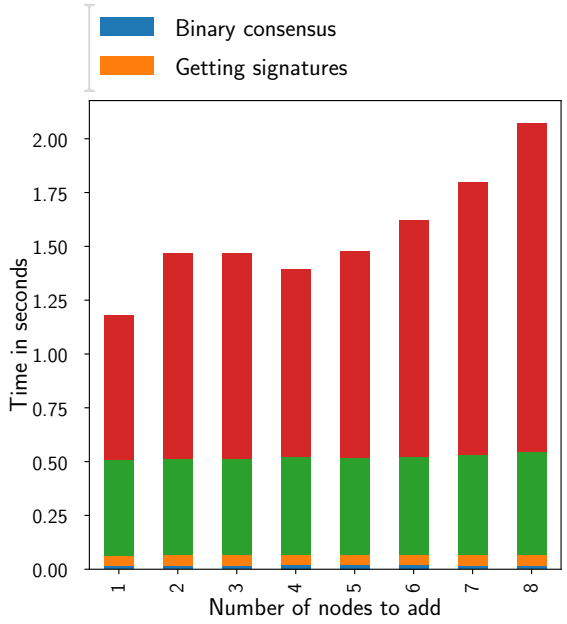
A blockchain service requires consensus to work. For the sake of efficiency, ComChain relies on a partially synchronous signature-free consensus algorithm DBFT [11] that is resilience optimal. The resilience optimality of ComChain thus follows from the fact that it tolerates as many Byzantine failures as DBFT.

Theorem 1. *The ComChain service is resilience optimal.*

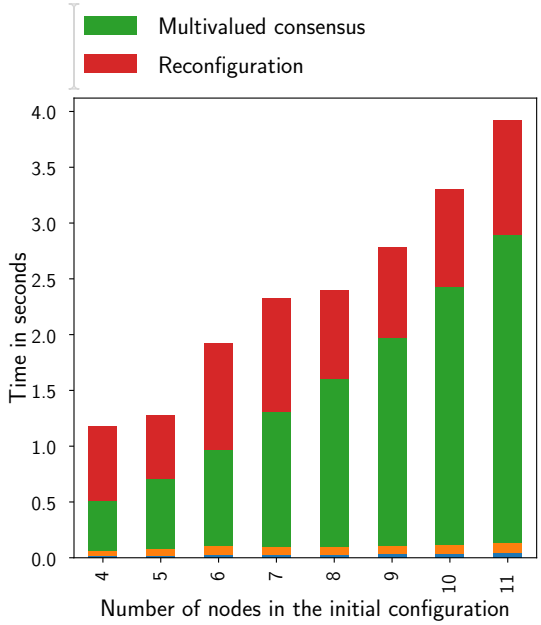
Proof. ComChain offers a blockchain service where distinct distributed servers may include in blocks, the transactions requested by potentially overlapping sets of clients. Hence ComChain requires a n-shared asset transfer object that is known to require consensus number n [25], and cannot be implemented without consensus. In addition, the consensus protocol of ComChain, DBFT [11], is known to be resilience optimal while tolerating $t < n/3$ Byzantine failures [10]. It follows that the ComChain service is resilience optimal as well, as it requires consensus to work and it tolerates the same number $t < n/3$ of Byzantine failures. □

6 Evaluation

In this section, we deploy ComChain on a distributed set of machines to measure the latency and throughput of reconfiguration between and transaction invocations. The experiments are distributed on 12 independent physical machines and the transactions use the Unspent Transactions Output (UTXO) model signed using the Elliptic Curve Digital Signature Algorithm (ECDSA) of Bitcoin [31] also used in the Red Belly Blockchain [11].



(a) Time in seconds to make a decision and to perform reconfiguration when starting with 4 nodes, depending on the number of nodes we want to add.



(b) Time in seconds to make a decision and to perform reconfiguration when adding 1 node, depending on the number of nodes of the initial configuration.

Figure 2: Evaluation of the reconfiguration when adding nodes.

6.1 Experimental settings

We deployed our experiments on a distributed set of physical machines using the Emulab platform[8]. We ran the experiments on up to 12 machines, each with two 64-bit Xeon processors running a total 8 cores at 2.4 GHz with 2 GB of memory and Ubuntu 14.04. We launched all nodes at the beginning of an experiment, one virtual machine per physical machine to make sure all communications went through physical links, with the DNS service running on the same physical machine as the first of these nodes.

6.2 Time to add new nodes

As presented in Figure 2, we focused on the impact of the size of the initial and final configurations on the time needed to perform one reconfiguration.

We observe in Figure 2a that increasing the number of nodes of the final configuration increases only slightly the time needed to perform the (binary and multivalue) consensus part: such an increase was expected, but as it is only due to the increase of the size of the configuration block, the impact is relatively small. The impact on the reconfiguration part is however more important. The overall trend is an increase, due to the number of nodes that need to update themselves. The lack of a steady trend can be explained by the $\frac{2}{3}$ ratio of correct nodes over total nodes: we evolve here at the limit of the threshold, thus the number of nodes we have to wait for, or the ratio of nodes we have to wait for over the total number of new nodes, does not evolve linearly.

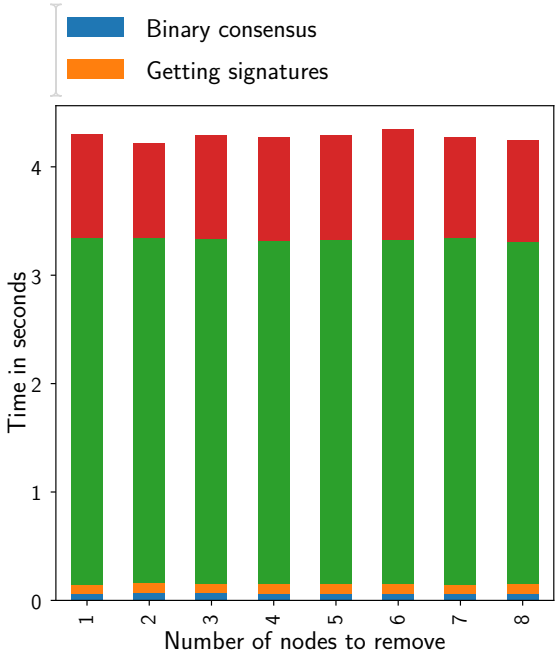
In Figure 2b, we depict the impact that the number of nodes in the initial configuration has on the time needed to perform consensus. We observe that the time to reconfigure increases only slightly, which was to

[8]"*Emulab is a network testbed, giving researchers a wide range of environments in which to develop, debug, and evaluate their systems.*" We used the datacenter installation located at the University of Utah (see `www.emulab.net`).

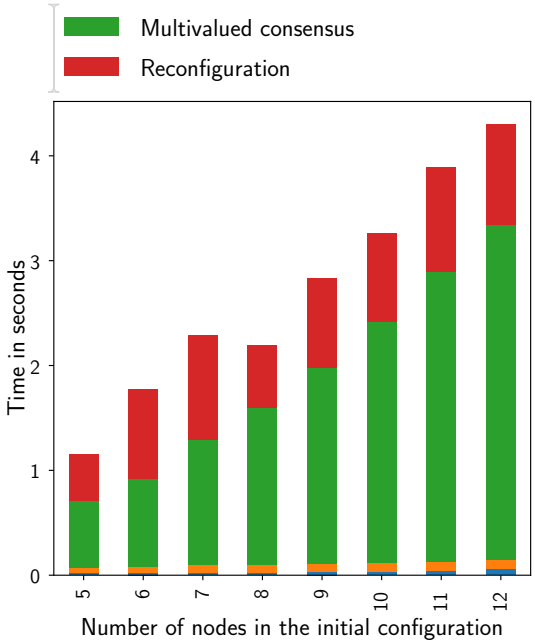
be expected: as we waited for all nodes to be up-to-date, the overhead due to the reconfiguration increases only slightly throughout the experiment. Our reconfiguration mechanism is thus scalable with the number of added nodes.

6.3 Time to remove existing nodes

We carried out similar experiments for the nodes removal, whose results are presented in Figure 3. We first observe on Figure 3a that the number of nodes we remove does not impact the time needed to perform the reconfiguration. The time needed to remove nodes varies actually from around one millisecond depending on the number of nodes we remove. This confirms that the impact of the size of the new configuration on our reconfiguration mechanism is negligible.



(a) Time in seconds to make a decision and to perform reconfiguration when starting with 12 nodes, depending on the number of nodes we want to remove.

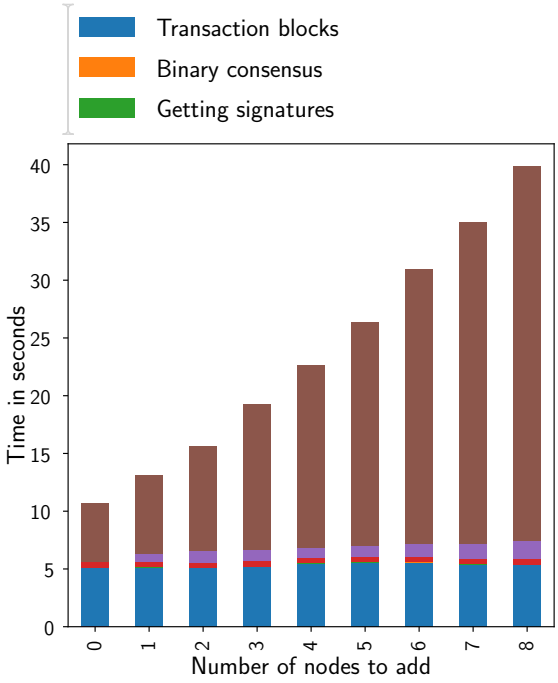


(b) Time in seconds to make a decision and to perform reconfiguration when removing 1 node, depending on the number of nodes of the initial configuration.

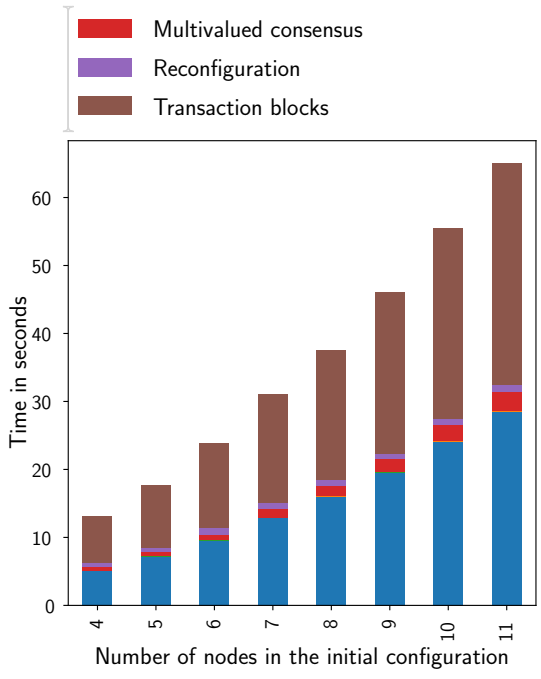
Figure 3: Evaluation of the reconfiguration when removing nodes.

Then, considering the impact of the number of nodes in the initial configuration (see Figure 3b), we can make the same remarks than before: the time spent for reaching an agreement increases with the initial number of deciders, but the actual reconfiguration period increases only slightly and irregularly due to our ratio.

Though it slightly slows down the blockchain, the reconfiguration itself takes less time than an instance of consensus. In addition, we expect the time needed to reconfigure not to increase significantly with the number of nodes with the current implementation, and thus that its impact on a real-world implementation should be negligible. However, our implementation does not currently transfer the blockchain from one configuration to the following.



(a) Time in seconds to make a decision and to perform reconfiguration and process 20 transaction blocks when starting with 4 nodes, depending on the number of nodes we want to add.



(b) Time in seconds to make a decision and to perform reconfiguration and process 20 transaction blocks when adding 1 node, depending on the number of nodes of the initial configuration.

Figure 4: Evaluation of the reconfiguration and transactions processing when adding nodes.

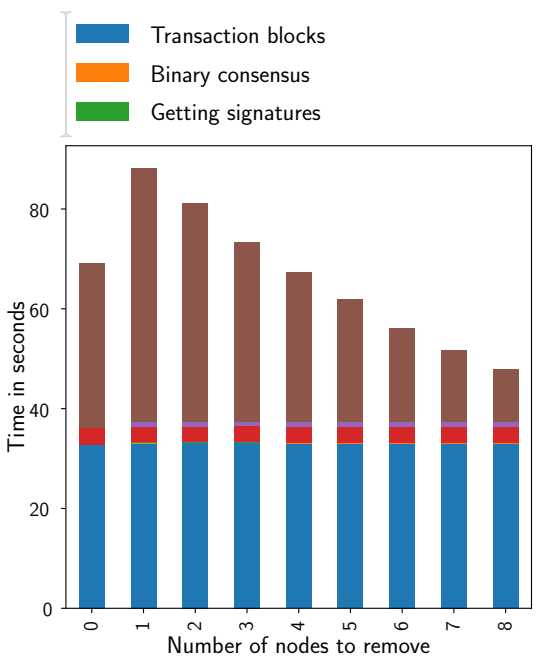
6.4 Mixing reconfiguration and transaction requests

On Figures 4 and 5, we display the total time needed to process 10 transaction blocks (in blue), one reconfiguration, then 10 other transaction blocks (in brown) using the new configuration. As expected, adding nodes increases the time to process the same number of blocks: as the latency of the underlying Red Belly Blockchain (without reconfiguration) tends to decrease with the number of nodes [24], adding nodes will slow down the processing of our 21 blocks.

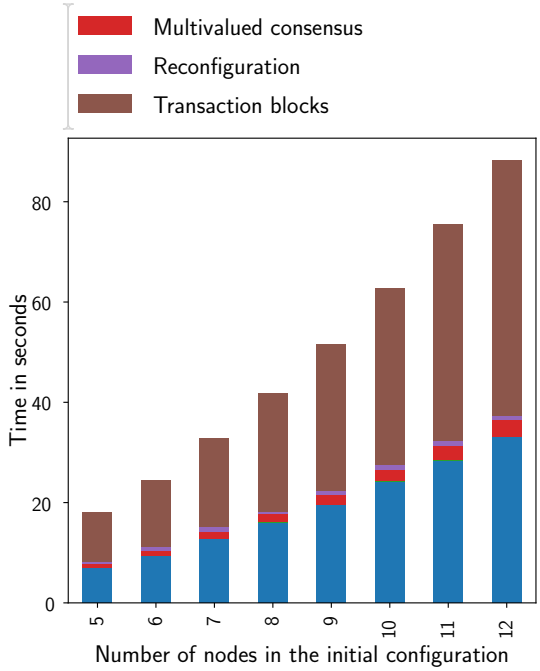
A more interesting result is well-illustrated on Figure 5a: though reducing the number of nodes should speed up the second sequence of transaction blocks, we see that the time needed to process these increases. It is most likely due to the slowest node trying to catch up with the fastest ones, as each node has been temporarily unaware of the other's messages during its reconfiguration. This behavior is less striking but also present on Figure 5b.

6.5 Impact of the reconfiguration on the transaction throughput

In order to measure the impact of the reconfiguration on the transaction throughput of the blockchain, we measured the throughput during a minute and a half while issuing a reconfiguration. The results are depicted in Figure 6. The throughput is computed as the number of transactions committed within a transaction block divided by the time it takes to append this transaction block in the blockchain. In this experiment the initial configuration contains 15 decider nodes. The reconfiguration triggered at around 32 seconds adds 5 nodes to these 15 nodes to obtain a new configuration with 20 decider nodes. We observe that the throughput decreases drastically from 35 transactions per second to 20 transactions per second. This is due



(a) Time in seconds to make a decision and to perform reconfiguration and process 20 transaction blocks when starting with 12 nodes, depending on the number of nodes we want to remove.



(b) Time in seconds to make a decision and to perform reconfiguration and process 20 transaction blocks when removing 1 node, depending on the number of nodes of the initial configuration.

Figure 5: Evaluation of the reconfiguration and transactions processing when removing nodes.

to the reconfiguration delaying the transaction processing as explained in Section 4.4: transactions must be handled by the new configuration as soon as the new configuration is installed as transaction blocks and configuration blocks must be totally ordered. Despite the slow down, the throughput is never null and recovers quickly after the reconfiguration. As the configuration resulting from the reconfiguration is larger the throughput of the system remains lower after the reconfiguration than before the reconfiguration.

7 Discussion

In this section, we discuss the independence of our algorithm from the presented consensus algorithm and the way one can define a configuration.

7.1 Independence from the consensus algorithm

The presented algorithms require a pre-existing reliable multicast abstraction similar to the one by G. Bracha [3], the partial synchrony assumption, a blockchain like the Red Belly Blockchain and our DNS service. The last two requirements are common to all implemented public blockchains. One assumption hidden in the definition of a configuration 1 is the threshold ($\forall I, \frac{2n}{3} < C_I$) needed to ensure that consensus can be reached using the algorithms proposed in DBFT [10]. Note that this threshold can be replaced by the minimal number of correct nodes needed per consensus instance by any Byzantine fault tolerant consensus algorithm.

In other words, our algorithms rely essentially on the reliable multicast abstraction, the partial synchrony

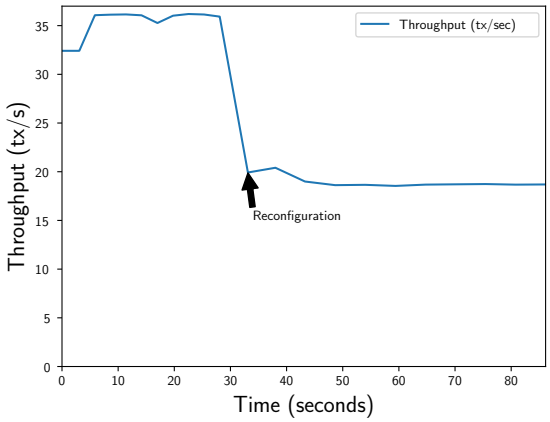


Figure 6: Evolution of the throughput during and after a reconfiguration, in transactions per second. The first configuration contains 15 decider nodes while the second one contains 20 decider nodes

assumption and some validity-based consensus algorithm. Yet, validity-based consensus algorithms are a superset of consensus algorithms as formalized in the distributed computing literature. Every algorithm solving the consensus problem can be adapted into a validity-based consensus algorithm, which would need exactly the same assumptions and suffer from the same limitations.

7.2 Flexibility with respect to the proposed configurations

We left the definition of the new configurations to future work because it depends on the environment and the targeted application goals. In particular, in Algorithm 1, we only described how to act if a node requests a signature for a configuration. Depending on the use-case of the community blockchain, the source of configurations can be adapted. For a private or consortium blockchain, a system administrator could have an interface to change the servers used for the consensus (useful when one of them crashes), or to add a new member of the current configuration. In a public environment, designing a source of configurations that keeps the blockchain safe for malicious behaviors is much harder. Several proposals have been made to ensure that a given threshold is met [20, 37], but no deterministic approach can be considered due to the predictability of such paradigms, which could widen the time frame for attackers to subvert participants to the consensus.

8 Conclusion

In this paper we proposed the community blockchain model that bridges the gap between public blockchains, which aim at dedicating all resources to find each block, and consortium blockchains, which restrict the block decision to a small "elite". The community blockchain model offers a deterministic solution that replaces nodes by others through a Byzantine consensus-based reconfiguration while transferring pending transactions from the old configuration to the new one.

While Byzantine reconfiguration of distributed systems has been studied in the past, the advent of blockchains has brought new opportunities, which have remained unexplored until now. Consortium blockchains have not been considered for public use due to their static membership. This is why we believe community blockchains will open new interesting research challenges in the context of blockchain and large-scale reconfiguration.

Finally, we implemented ComChain, a community blockchain that uses Byzantine reconfiguration to alternate between different configurations of nodes to decide subsequent blocks. Our ComChain experimentation on a distributed set of physical machines demonstrates the feasibility of using Byzantine reconfiguration in a blockchain. While latency increases with the number of added nodes, it reduces with the number of removed nodes. Finally, these experiments show that reconfiguration may affect the throughput but can proceed at runtime without disrupting the transaction service.

Acknowledgments. This research is in part supported under Australian Research Council Discovery Projects funding scheme (project number 180104030) entitled "Taipan: A Blockchain with Democratic Consensus and Validated Contracts" and Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things". Vincent Gramoli is a Future Fellow of the Australian Research Council.

References

[1] Eduardo Alchieri, Fernando Luís Dotti, Odorico Machado Mendizabal, Fernando Pedone. Reconfiguring Parallel State Machine Replication. *SRDS 2017*: 104-113.

[2] Alysson Bessani, João Sousa, and Eduardo EP Alchieri, State machine replication for the masses with BFT-SMaRt, *Dependable Systems and Networks (DSN)*, 2014 44th Annual IEEE/IFIP International Conference on. IEEE, 2014.

[3] Gabriel Bracha, Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143 (1987).

[4] Gabriel Bracha and Sam Toueg, Asynchronous consensus and broadcast protocols, *Journal of the ACM*, 32(4):824-840 (1985).

[5] Christian Cachin. Architecture of the Hyperledger Blockchain Fabric. Workshop on Distributed Cryptocurrencies and Consensus Ledgers. July 2016.

[6] Miguel Castro, and Barbara Liskov, Practical Byzantine fault tolerance, *OSDI*, Vol. 99, 1999.

[7] Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial and Alexander A. Shvartsman, Reconfigurable Distributed Storage for Dynamic Networks, *Journal of Parallel and Distributed Computing (JPDC)*, 69(1):100-116 Elsevier jan 2009

[8] Pierre Civit, Seth Gilbert, Vincent Gramoli. Polygraph: Accountable Byzantine Agreement. 2019. Cryptology ePrint Archive, Report 2019/587. `https://eprint.iacr.org/2019/587`

[9] James A. Cowling, Dan R. K. Ports, Barbara Liskov, Raluca Ada Popa, Abhijeet Gaikwad. Census: Location-aware membership management for large-scale distributed systems, USENIX ATC, 2009.

[10] Tyler Crain, Vincent Gramoli, Mikel Larrea and Michel Raynal. (Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains, Arxiv Technical Report, `https://arxiv.org/pdf/1702.03068.pdf`

[11] Tyler Crain, Vincent Gramoli, Mikel Larrea and Michel Raynal. DBFT: Efficient Leaderless Byzantine Consensus and its Applications to Blockchains, Proceedings of the 17th IEEE International Symposium on Network Computing and Applications, 2018.

[12] Tyler Crain, Vincent Gramoli, Mikel Larrea, Michel Raynal. Blockchain Consensus, *ALGOTEL 2017 - 19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, May 2017, Quiberon, France.

[13] Tyler Crain, Chris Natoli, Vincent Gramoli. Evaluating the Red Belly Blockchain. Technical report arXiv 1812.11747, 2018.

[14] Phil Daian, Rafael Pass and Elaine Shi, Snow White: Robustly reconfigurable consensus and applications to provably secure proofs of stake, *Cryptology ePrint Archive*, Report 2016/919, 2017.

[15] Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration, *OPODIS*, 2014.

[16] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. *NSDI* 2016.

[17] Parinya Ekparinya, Vincent Gramoli, Guillaume Jourjon. Impact of Man-In-The-Middle Attacks on Ethereum. *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[18] Parinya Ekparinya, Vincent Gramoli, Guillaume Jourjon. The Attack of the Clones against Proof-of-Authority. *Community Ethereum Development Conference (EDCON)* 2019.

[19] M.J. Fischer, N.A. Lynch, and M.S. Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM*, 32(2):374-382 (1985)

[20] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos and Nickolai Zeldovich, Algorand: Scaling Byzantine Agreements for Cryptocurrencies, *Cryptology ePrint Archive*, Report 2017/454, May 2017

[21] Seth Gilbert, Nancy A. Lynch, Alexander A. Shvartsman: Rambo: a robust, reconfigurable atomic memory service for dynamic networks. Distributed Computing 23(4): 225-272 (2010)

[22] Gregory V. Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, Alexander A. Shvartsman: Reconfigurable distributed storage for dynamic networks. J. Parallel Distrib. Comput. 69(1): 100-116 (2009)

[23] Vincent Gramoli, Len Bass, Alan Fekete and Daniel Sun, Rollup: Non-Disruptive Rolling Upgrade with Fast Consensus-Based Dynamic Reconfigurations, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(9):2711-2724 Sep 2016.

[24] Vincent Gramoli. The Red Belly Blockchain. Invited talk. MIT, MA, USA. June 2017. `https://hades.it.usyd.edu.au`

[25] Guerraoui, Kuznetsov, Monti, Pavlovic, Seredinschi. AT2: Asynchronous Trustworthy Transfers. Arxiv Technical Report, `https://arxiv.org/pdf/1812.10844.pdf`

[26] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn, Corda: An Introduction, White paper, 2016.

[27] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, Benjamin Reed: ZooKeeper: Wait-free Coordination for Internet-scale Systems. USENIX Annual Technical Conference 2010.

[28] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser and Bryan Ford, Enhancing bitcoin security and performance with strong consistency via collective signing, *CoRR*, abs/1602.06997, 2016.

[29] Leslie Lamport, Dahlia Malkhi and Lidong Zhou, Vertical Paxos and Primary-Backup Replication, *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009

[30] Leslie Lamport, Dahlia Malkhi and Lidong Zhou. Reconguring a State Machine. *SIGACT News* 41(1): 63-73 (2010)

[31] Satoshi Nakamoto, Bitcoin: a peer-to-peer electronic cash system. http://www.bitcoin.org (2008)

[32] Chris Natoli and Vincent Gramoli, The balance attack against proof-of-work blockchains: The R3 testbed as an example, *DSN*, 2017.

[33] Chris Natoli and Vincent Gramoli, The blockchain anomaly, *Proc. 5th IEEE Int'l Symposium on Network Computing and Applications (NCA'16)*, IEEE ComputerPress, pp. 310-317 (2016)

[34] Rafael Pass and Elaine Shi, Hybrid Consensus: Efficient Consensus in the Permissionless Model, *Cryptology ePrint Archive*, Report 2016/917, 2016. `http://eprint.iacr.org/2016/917.pdf`

[35] Rodrigo Rodrigues, Barbara Liskov, Member, IEEE, Kathryn Chen, Moses Liskov and David Schultz, Automatic reconfiguration for large-scale reliable storage systems, *IEEE Transactions on Dependable and Secure Computing*, 9.2 (2010): 145–158.

[36] Wood Gavin, Ethereum: A secure decentralized generalized transaction ledger. *White paper* (2015)

[37] Jiangshan Yu, David Kozhaya, Jrmie Decouchant and Paulo Esteves-Verissimo, RepuCoin: Your reputation is your power, *unpublished draft*.

[38] Jepsen, Tendermint 0.10.2. Unknown author. White paper, Sept. 2017. https://jepsen.io/analyses

[39] João Sousa and Alysson Bessani and Marko Vukolić, A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. *Dependable Systems and Networks (DSN)*, 2014 44th Annual IEEE/IFIP International Conference on. IEEE, 2018. arXiv:1709.06921.

[40] https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

[41] Guillaume Vizier and Vincent Gramoli, ComChain: Bridging the Gap Between Public and Consortium Blockchains. IEEE Blockchain, DOI:10.1109/Cybermatics_2018.2018.00249 pp.1449-1474 (2018)

[42] Tendermint documentation. https://tendermint.com/docs/tendermint-core/using-tendermint.html#adding-a-validator, Accessed 30-05-2019

[43] EOS documentation. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md#consensus-algorithm-bft-dpos, Accessed 30-05-2019.