ComChain: Bridging the Gap Between Public and Consortium Blockchains

Conference Paper · July 2018

DOI: 10.1109/Cybermatics_2018.2018.00249

2 authors, including:

Vincent Gramoli
Data61-CSIRO and University of Sydney
122 PUBLICATIONS **954** CITATIONS

SEE PROFILE

ComChain: Bridging the Gap Between Public and Consortium Blockchains

Guillaume Vizier
École Polytechnique
guillaume.vizier@polytechnique.edu

Vincent Gramoli
University of Sydney
vincent.gramoli@sydney.edu.au

***Abstract*—As an alternative to the energy greedy proof-of-work, new blockchains constrain the set of participants whose selection is debatable. These blockchains typically allow a fixed consortium of machines to decide upon new transaction blocks.**

In this paper, we introduce the *community blockchain* that bridges the gap between these public blockchains and constrained blockchains. The idea is to allow potentially all participants to decide upon "some" block while restricting the set of participants deciding upon "one" block.

We also propose an implementation called *ComChain* that builds upon the Red Belly Blockchain, the fastest blockchain we are aware of. It runs a consensus among the existing community to elect a new community. This reconfiguration speeds up as the number of removed nodes increases.

***Index Terms*—Community blockchain, reconfiguration**

I. Introduction

A blockchain is an abstraction representing a linked list of blocks implemented in a distributed system of nodes who can have two different roles: (i) *clients* that can read and write the data and (ii) *deciders* that run a consensus algorithm to decide upon the block at a given index of the chain. Various types of blockchains exist, that differ mostly in the way their nodes have permissions to play specific roles, and in the consensus algorithm deciders run to agree upon new blocks.

In public blockchains [18], [22], all nodes are potentially deciders and can participate in the creation of new blocks [25]. To cope with Sybil attacks, they typically restrict the power of a user to its resources (e.g., computational power or amount of coins). Provided that the malicious users do not own a large portion of the resources of the system, they cannot impose their decision to the others. In consortium and private blockchains, synchronous implementations are rarely used due to their weakness against known network attacks [19]. Instead, the permission for some nodes to act as deciders is hardcoded [6], [24]. Thus, each time the set of deciders changes, the whole blockchain must be stopped and restarted.

This lack of dynamism is a major issue in long-lived blockchains where hardware components fail and consortia evolve. Hence, 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 participant to become a decider.

To this end, the community blockchain relies on a new type of blocks: the *configuration block*. Its role is to define among all participants a subset of deciders responsible for deciding the upcoming transaction blocks. More precisely, each configuration block lists a *configuration* as a set of deciders identified by their public key. These n nodes keep adding new transaction blocks. These nodes can also propose new configurations to each other, and, despite $t < \frac{n}{3}$ Byzantine nodes, $n - t$ correct must reach a consensus on one configuration. These nodes have the responsibility of selecting a configuration that is *acceptable* according to **application-specific** rules.

We also propose an implementation of this community blockchain, called *ComChain*, that does not require synchrony. ComChain builds upon the Red Belly Blockchain [15]. 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 it into a new configuration block. From this point on, the new configuration defines the new set of deciders for the blockchain until the next reconfiguration.Our distributed experiments confirm that the reconfiguration can occur at runtime without impacting ongoing transaction requests.

Section II presents an overview of the related work. Section III introduces our model. Section IV presents our solution at a high level. Section V specifies a detailed implementation, called ComChain. Section VI illustrates how our reconfiguration mechanism performs empirically and Section VII concludes. The proofs are deferred to a longer paper.

II. Related Work

To the best of our knowledge, no deterministic blockchain implementation supports a dynamic membership. Probabilistic alternatives revert typically to proof-of-* [18], [22], [20], [1], randomized consensus [10] or sortition [12]. 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 several blockchain projects. Hyperledger Fabric aims at supporting membership changes without compromising the

network[1], however, it "requires that the peer or orderer process is restarted"[2]. Tendermint mentions validator set changes [26], however, this requires an external application that handles those reconfigurations.[3] In contrast, our reconfiguration is non-disruptive and the main feature of ComChain.

Until now, researchers have designed reconfiguration algorithms for cluster membership changes [16], rolling upgrades [14], atomic storage [13], [21] or replicated state machines [17], [2], but not blockchains. 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 independence of replicated state machine commands.

III. Model

The system is made up of a set $\Pi = \{p_1, \ldots, p_n\}$ of n processes, i is called the *index* of p_i. These processes proceed at their own speed (*asynchronous*) and execute one step at a time (*sequential*). As consensus cannot be solved with asynchronous communication in the presence of failures [11], we assume that communication is *partially synchronous* in that messages get delivered in a bounded amount of time but the bound is unknown. Note that this assumption is weaker than the one made by mainstream blockchains [18], [22].

We consider a dynamic Byzantine failure model. A node is said to be *correct during a time frame* Δ if it follows its specification during that time frame. A node is said to be *Byzantine during a time frame* Δ if it stops following its specification at any time during that time frame. A node is said to be *correct (resp. Byzantine) within the* i^{th} *consensus instance* if it is correct (resp. Byzantine) for the time frame between its proposal and its decision for this consensus instance.

A *consensus instance* is one execution of a consensus algorithm. Each *consensus instance* can replace the set of n deciders, among which $t < \frac{n}{3}$ can be Byzantine, by another set n' of deciders where $t' < \frac{n'}{3}$ can be Byzantine. More generally, n, n', ... represent the number of deciders, and t, t', ... are upper-bounds on the number of Byzantine nodes among them. Each consensus instance occurs one at a time and $n - t$ nodes have to propose a value to launch a consensus instance among n deciders. Although not specified here, we assume that correct deciders propose periodically as part of their protocol, so that consecutive consensus instances get launched.

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

- ***Network****: we have a partially synchronous point-to-point reliable network for all correct nodes in* Π[4]*,*
- ***Awareness****: all correct nodes have knowledge of the full blockchain*[5]*,*

[1] https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#dynamic-membership.

[2] https://hyperledger-fabric.readthedocs.io/en/latest/msp.html#msp-setup-on-the-peer-orderer-side.

[3] Section 7.1, second paragraph of https://atrium.lib.uoguelph.ca/xmlui/bitstream/handle/10214/9769/Buchman_Ethan_201606_MAsc.pdf.

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

[5] They have in memory every block of the blockchain.

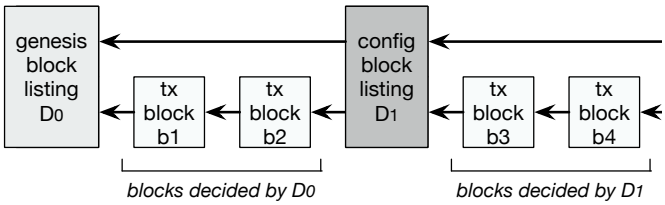


Fig. 1: A community blockchain

- ***Correctness threshold****: for all instances of consensus* I*, with* C_I *the number of* ***correct*** *nodes in* Π *for consensus* I*,* $\frac{2n}{3} < C_I$.

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

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

- ***Network****: we have a partially synchronous 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.*[6]

To guarantee integrity, nodes sign their proposed configurations. We thus assume an asymmetric crypto-system where each node has a private key associated with a publicly known key so that no existing private key can be forged or stolen. To allow new blockchain participants to join the correct service with certainty, we assume a correct DNS service keeping track of the latest configuration. Similarly, Bitcoin [18] uses hard-coded DNS seeds that are assumed trustworthy. An additional trust assumption has been proven necessary in [9].

IV. The Community Blockchain

In this section, we introduce the *community blockchain* paradigm to bridge the gap between public and consortium blockchains. Community blockchains constrain the set of deciders for a particular block but let potentially all nodes decide upon some block, by periodically replacing the set of deciders by a new valid configuration.

A. Two lists of blocks

To store the current configuration, the community blockchain contains two types of blocks: the classic or *transaction blocks* with transactions and the *configuration blocks* (cf. Definition 3) with a set of deciders. The resulting structure resembles a skip list depicted in Fig. 1. At the bottom, all blocks are chained together with a linked list. This guarantees that all blocks are totally ordered. At the top, the configuration blocks are chained together with another linked list, hence allowing to verify that the current configuration is properly signed by the deciders of the previous configuration.

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

[6] Receiving information from the DNS service introduced below belongs to this **Listening** requirement.

1) *Information on the deciders of the new configuration*[7].
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 *and* transaction *block (indicating the* previous configuration *and* the last committed transactions*).*

Note that the genesis block is a special configuration block without the hash and index of previous blocks.

B. Deciding upon a new configuration

Reconfiguration consists of replacing the current configuration in use by a new configuration. In a community blockchain, a reconfiguration must guarantee that the system can keep serving the transaction requests while the set of decider nodes is being reconfigured, hence the Definition 4.

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

1) *all correct nodes of* c *agree upon* c'*, proposed by a node;*
2) c' *verifies a validity predicate (*c' *is valid);*
3) *there is no service interruption;*
4) *there is no data loss;*
5) *data integrity is preserved;*
6) *the reconfiguration takes a finite amount of time.*

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

C. Verifying that a configuration is valid

Configurations of the community blockchain must fulfill some requirements defined by the application. A trivial example is that they cannot be empty. But specific applications may also require that the deciders are not all part of the same jurisdiction, etc. Based on application-specific criteria, each node can decide whether a configuration is acceptable. We explain below how we guarantee that the community blockchain only uses *valid configurations*.

Initially, we list the public key, the IP address and the port number of the decider nodes of the initial configuration in the genesis block. To propose a new configuration c, a decider node sends c to the other $n-1$ decider nodes. Each node receiving c proposes 1 if it considers c acceptable, 0 otherwise, to a new consensus instance. Upon agreement, if 1 is decided, then every correct decider node signs configuration c (with their private key) and sends c along with their signature to the requester.

We write that a newly decided configuration c' is *valid* and that $\mathsf{valid}(c')$ returns `true` if the configuration c' has been signed by at least $t+1$ nodes of the current configuration c[8], and contains the hashes and indexes of the previous configuration block and transaction block.

[7] IP address, port and public key.

[8] As t is the maximum number of Byzantine nodes, $t+1$ signatures guarantee that at least one signature comes from a correct node.

D. Reconfiguration

We distinguish two cases: (i) *adding* nodes to and (ii) *removing* nodes from a configuration. To handle the general case, we first add the new nodes, and once these are ready to perform consensus, we remove the unneeded ones.

When *adding* nodes, new deciders must have knowledge of the most up-to-date blockchain (and configuration). To this end, each old node sends a copy of the up-to-date blockchain to new nodes. A new node waits for having received $t+1$ identical blockchains, before taking part to 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.

When *removing* nodes, we have to ensure that the already acknowledged transactions do not get lost. Thus, the removed nodes transfer all the enqueued and not-processed transactions to the nodes of the next configuration. To ensure that the removed nodes eventually transfer all of them, they stop acknowledging and taking into account arriving transactions after the new configuration has been decided. Each removed node can thus safely stop when it has no more pending transaction.

E. Catching up with the most up-to-date information

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 progressed to the k^{th} configuration where the $k-1$ preceding configurations have become faulty. To cope with this issue, every new correct node bootstraps by contacting the DNS service that provides the latest configuration. As mentioned previously, this type of assumption is needed by existing blockchains [9], [18].

V. Putting the Community to Work

In this section, we implement a community blockchain, called *ComChain*, that builds upon the Red Belly Blockchain [15]. This blockchain features a leaderless consensus algorithm [7] to solve the Blockchain Consensus problem, where nodes cooperate to select a block [8] (see Section V-B).

A. 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 [4], [5] and reused in [7]. The reliable broadcast is a communication primitive among n nodes where at most $t < \frac{n}{3}$ can be Byzantine [4]. 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 two multicast primitives, RB_broadcast_new and RB_broadcast_old, that precise the set of nodes it is sent to.

Definition 5. *We define four operations* RB_broadcast_new *and* RB_deliver_new *(resp.* RB_broadcast_old *and* RB_deliver_old*), 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 (resp. to all nodes of the old configuration).*

B. Blockchain consensus

The blockchain consensus, originally referred to as *Validity Predicate-based Byzantine Consensus*, consists for a set of nodes to collaboratively decide upon a new block [7], [8] (see Definition 6). This problem is different from the proof-of-work Blockchain problem [18], [22] and the classic Byzantine agreement problem [3], [6], [24].

Definition 6 (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:*** *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().

Crain et al. [7] propose a solution to the blockchain consensus problem where each correct decider proposes a value. The deciders then agree on a set of accepted proposals from which is forged the next block. An *instance* of consensus is one execution of a consensus algorithm on a set of proposals.

C. Collaboratively signing a configuration

As described in Section IV-C, before proposing a configuration, a node has to gather $t+1$ 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. As validity is application dependent (cf. Section IV-C), 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.

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))
    when BinaryConsensus{id'} returns 1 do
        ⟨c', p_j⟩ ← GetItemsFromId(id')
        c' ← sign(c')
        Send c' to p_j
    when RB_deliver signed configuration c'' from p_k do
        if EqualConfigurations(c'', self.sc) then
            MergeSignatures(c'')
```

Definition 7. BinaryConsensus(v, id) *represents the proposal of the (binary) value v to the instance of 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 methods, GetId and GetItems, to turn $\langle v, id\rangle$ (where v is a boolean and id is a node identifier) into the identifier id' of a binary consensus instance and to reverse the operation. These functions are implemented via a dictionary-like structure that maintains the identifiers for the launched binary consensus instances. We use $self.sc.c$ to refer to the configuration that the node wants other participants to sign. A configuration as stored by a node is a tuple:

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

The MergeSignatures function groups the signatures present in the incoming signed configuration with these already gathered for the next-to-propose configuration.

D. Deciding upon a new configuration

To decide on the next configuration, we launch a *multivalue* consensus algorithm among all nodes. To this end, we use the partially synchronous DBFT algorithm [7] as a black box, with the serialized signed configuration as input and a function checking the configuration has been signed by at least $t+1$ nodes of the current configuration as valid predicate.

More precisely, the DBFT algorithm returns the value decided by all correct processes. This value will be input in the 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: if the new configuration is included in the old configuration then we remove nodes; etc.

E. Transition from the current to the new configuration

The functions we use above return a configuration proposed by a node that verifies validity conditions and on which all correct nodes agree. As previously indicated in Section V-D, after the consensus returns a value, we need to process this value with the functions previously described in Section IV.

Algorithm 2 Participation to the consensus when adding nodes

```
function DECIDECONFIGURATION(c)
    ADDCONFIGURATIONBLOCK(c)
    RB_broadcast_new blockchain
    RB_broadcast_old the last block index
    Run consensus on the new configuration
    when (RB_deliver_old last index of n' − t' nodes) do
        Inform the DNS service that transactions should be
        sent to the new configuration
```

Algorithm 3 Participation to the consensus when removing nodes

```
function DECIDECONFIGURATION(c)
    ADDCONFIGURATIONBLOCK(c)
    Inform the DNS service that txs should be sent 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
    else
        perform consensus with the new configuration
```

1) Adding nodes: For old nodes to add new nodes, they use the protocol presented in Algorithm 2.

The function ADDCONFIGURATIONBLOCK used in Algorithm 2 adds a new configuration block to the blockchain. As precised in Section IV-D, we do not need to wait for the new nodes to be ready.

2) Removing nodes: We remove nodes by using the protocol described in Algorithm 3 where *self* depicts the node running the function.

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 oc, and the new configuration c, we use the first case above to switch from oc to $oc \cup c$, then the second to switch from $oc \cup c$ to c.

F. Blocks

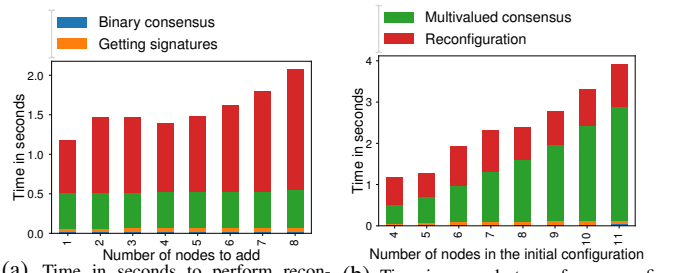
In this whole section, we have handled only configuration blocks, assuming that for a given consensus instance, either only configuration blocks are proposed, or only transaction blocks are proposed. This is also an easy way to dissociate the processing of the configurations and transactions.

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). To validate a general block, we split it 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.

VI. EVALUATION

In this section, we deploy ComChain on a distributed set of machines to measure the latency and throughput of reconfiguration between and during UTXO[9] transaction invocations.

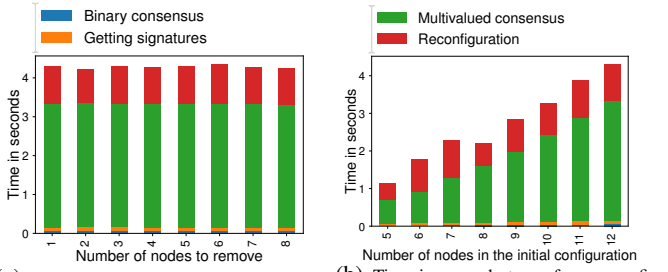
[9]Unspent Transaction Outputs (UTXOs) is the model of transactions used by Bitcoin [18].



(a) Time in seconds to perform reconfiguration when starting with 4 deciders, depending on the number of nodes to add.

(b) Time in seconds to perform reconfiguration when adding 1 node, depending on the number of initial deciders.

Fig. 2: Evaluation of the reconfiguration when adding nodes.



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

(b) Time in seconds to perform reconfiguration when removing 1 node, depending on the number of deciders.

Fig. 3: Evaluation of the reconfiguration when removing nodes.

Experimental settings. We launched experiments on a distributed set of physical machines using the Emulab platform[10]. 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 the nodes.

Time to add new nodes. As presented in Figure 2, we focused on the impact of the size of the initial or the final configuration on the time needed to perform one reconfiguration.

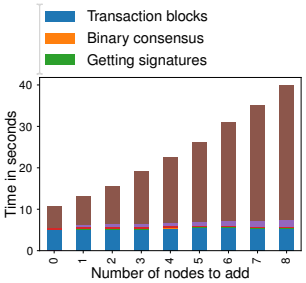
We observe in Figure 2a that increasing the number of nodes of the final configuration does not impacts the time needed to perform the (binary and multivalue) consensus part. For the reconfiguration time, we observe a significant increase, due to the nodes that need to update themselves. The lack of a steady trend can be explained by the $\frac{2}{3}$ ratio of correct nodes over total nodes: we evolve at the limit of the threshold, thus the number of nodes we have to wait for does not evolve linearly.

In Figure 2b, we depict the impact of the initial number of deciders on the time needed to perform reconfiguration. We observe that the reconfiguration time increases only sightly.

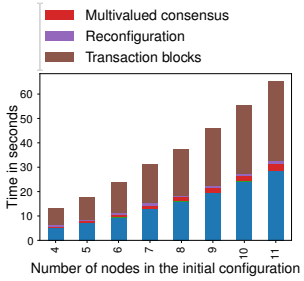
Time to remove existing nodes. We carried out similar experiments for the nodes removal, whose results are presented in Figure 3. We first observe on Figure 3a that the number of nodes we remove does not impact the reconfiguration time.

Then, as for adding nodes, the time spent for reaching an

[10]"*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).

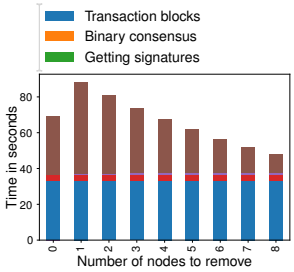


(a) Time in seconds to perform reconfiguration and process 20 transaction blocks when starting with 4 nodes, depending on the number of added nodes.

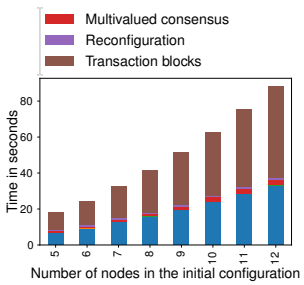


(b) Time in seconds to perform reconfiguration and process 20 transaction blocks when adding 1 node, depending on the initial number of deciders.

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



(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 to remove.



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

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

agreement increases with the initial number of deciders[11], 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. However, our implementation does not currently transfer the blockchain from one configuration to the next one.

Mixing reconfiguration and transaction requests. On Figures 4 and 5, we display the 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.

Though reducing the number of nodes should speed up the second sequence of transaction blocks, we see on Figure 5a that the time needed to process these increases when we start removing nodes. It is most likely due to the slowest node trying to catch up with the fastest ones, as each node has been unaware of the others' messages during its reconfiguration. This behavior is also present on Figure 5b.

VII. Conclusion

In this paper we proposed community blockchain, a paradigm to change nodes on-the-fly with a consensus-based reconfiguration without disrupting the transaction service. Our ComChain implementation and its experiments on a distributed set of physical machines confirm the feasibility and non-disruption of this approach. As future work, we plan to measure the information transfer time from old nodes to new nodes depending on the blockchain size.

[11]The latency of the underlying Red Belly Blockchain tends to increase with the number of nodes [15].

Acknowledgments

This research is in part supported under Australian Research Council's Discovery Projects funding scheme (project number 180104030) entitled "Taipan: A Blockchain with Democratic Consensus and Validated Contracts".

References

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, Alexander Spiegelman: Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. *OPODIS 2017*: 1-19.

[2] E. Alchieri, F. L. Dotti, O. M. Mendizabal, F. Pedone. Reconfiguring Parallel State Machine Replication. *SRDS 2017*: 104-113.

[3] A. Bessani, J. Sousa, and E. Alchieri, State machine replication for the masses with BFT-SMaRt, *DSN*, 2014.

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

[5] G. Bracha and S. Toueg, Asynchronous consensus and broadcast protocols, *JACM*, 32(4):824-840 (1985).

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

[7] T. Crain, V. Gramoli, M. Larrea and M. Raynal, (Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains, https://arxiv.org/pdf/1702.03068.pdf

[8] T. Crain, V. Gramoli, M. Larrea and M. Raynal, Blockchain Consensus, *ALGOTEL*, 2017.

[9] P. Daian, R. Pass and E. Shi, Snow White: Robustly reconfigurable consensus and applications to provably secure proofs of stake, *Cryptology ePrint Archive*, Report 2016/919, 2017.

[10] I. Eyal, A.E. Gencer, E.G. Sirer, and R. van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. *NSDI* 2016.

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

[12] Y. Gilad, R. Hemo, S. Micali, G. Vlachos and N. Zeldovich, Algorand: Scaling Byzantine Agreements for Cryptocurrencies, *Cryptology ePrint Archive*, Report 2017/454, 2017.

[13] S. Gilbert, N.A. Lynch, A.A. Shvartsman: Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing* 23(4):225-272, 2010.

[14] V. Gramoli, L. Bass, A. Fekete and D. Sun, Rollup: Non-Disruptive Rolling Upgrade with Fast Consensus-Based Dynamic Reconfigurations, *IEEE TPDS*, 27(9):2711-2724, 2016.

[15] V. Gramoli. The Red Belly Blockchain. Invited talk. MIT, MA, USA. June 2017. http://gramoli.redbellyblockchain.io/web/doc/talks/facebook.pdf

[16] P. Hunt, M. Konar, F.P. Junqueira, B. Reed: ZooKeeper: Wait-free Coordination for Internet-scale Systems. *ATC*, 2010.

[17] L. Lamport, D. Malkhi and L. Zhou, Reconguring a State Machine. *SIGACT News* 41(1): 63-73 (2010)

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

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

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

[21] R. Rodrigues, B. Liskov, K. Chen, M. Liskov and D. Schultz, Automatic reconfiguration for large-scale reliable storage systems, *IEEE TDSC* 9.2:145--158, 2010.

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

[23] J. Yu, D. Kozhaya, J. Decouchant and P. Esteves-Verissimo, RepuCoin: Your reputation is your power, *unpublished draft*.

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

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

[26] http://tendermint.readthedocs.io/projects/tools/en/master/using-tendermint.html#adding-a-validator