4th International Symposium on Foundations and Applications of Blockchain 2021

FAB 2021, May 7, 2021, University of California, Davis, California, USA (Virtual Conference)

Edited by

Vincent Gramoli

Mohammad Sadoghi

Editors

Vincent Gramoli
University of Sydney and EPFL, Lausanne, Switzerland
vincent.gramoli@sydney.edu.au

Mohammad Sadoghi
University of California, Davis, USA
msadoghi@ucdavis.edu

ACM Classification 2012
Theory of computation → Distributed algorithms; Computer systems organization → Dependable and fault-tolerant systems and networks; Applied computing → Digital cash; Applied computing → Online banking

ISBN 978-3-95977-196-2

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-196-2.

Publication date
June, 2021

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.FAB.2021.0

ISBN 978-3-95977-196-2 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

4th International Symposium on Foundations and Applications of Blockchain 2021 (FAB 2021).
Editors: Vincent Gramoli and Mohammad Sadoghi
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The goal of 4th International Symposium on Foundations and Applications of Blockchain 2021 (FAB'21) is to bring researchers and practitioners of blockchain – the technology behind Bitcoin – together to share and exchange results. The program of FAB'21 features two keynote speakers and seven presentations of scientific papers, followed by a student session. The program committee selected five papers for publication in the proceedings out of twelve submissions. Prof. Rachid Guerraoui's keynote's speech is about the road to a universal internet machine; Prof. Elaine Shi's keynote talk is about game-theoretically secure protocols inspired by blockchains.

The scientific papers published in these proceedings cover topics ranging from new distributed problem formalizations to solutions to decentralized finance problems. Aștefănoaei et al. formalize the notion of Dynamic Repeated Consensus for blockchain applications by offering Tenderbake that improves over the one-shot consensus protocol Tendermint. Cholvi et al. combine recent results on the cryptocurrency object and conflict-free replicated data structures to introduce the Distributed-Grow-only Set object that supports an atomic append operation. They propose an eventually consistent and Byzantine fault tolerant implementation of it that does not need consensus. Bhat et al. propose the DAISIM open-source simulator for the DAI stable coin offered by the MakerDAO project. They model investors with a portfolio of four assets to investigate when investors choose to mint or burn DAI, and determine the DAI price. Krishnamachari et al. propose a new approach to construct the Automated Market Makers that maintain a liquidity pool of assets related mathematically; this approach eliminates arbitrage opportunities. Mitra et al. introduce a consistent time metric for blockchain distributed systems by generating blocks regularly and inserting timestamps in each block. They illustrate their approach with an implementation in Hyperledger Fabric.

The program also features two additional presentations about blockchain applications. Sguerra et al. present a short paper on the performance of auctions running on Ethereum and Tezos whereas Kahya and Krishnamachari present EcoTrojan to incentivize environment-friendly on-campus behaviors.

To promote and support undergraduate research, we have introduced a unique student session in collaboration with Blockchain Acceleration Foundation (FAB), a nonprofit student-run organization that fosters blockchain research and education tailored to undergraduates. The session covers exciting and promising projects such as (1) BAF Wallet supported by NEAR Foundation, (2) Alchedemia for Next Generation Digital Learning Environment (NGDLE) supported by the Algorand Foundation, (3) Ethereum Teacher Training Program supported by the Ethereum Foundation, (4) Token Delegation and DeFi Governance supported by Blockchain Clubs at UCLA, (5) Solace DeFi supported by Blockchain at UCSB, and (6) ZUZ Exchangeable Credit by Blockchain at CMU.

4th International Symposium on Foundations and Applications of Blockchain 2021 (FAB 2021).
Editors: Vincent Gramoli and Mohammad Sadoghi
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Tenderbake – A Solution to Dynamic Repeated Consensus for Blockchains

Lăcrămioara Aştefănoaei
Nomadic Labs, Paris, France

Pierre Chambart
Nomadic Labs, Paris, France

Antonella Del Pozzo
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Thibault Rieutord
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Sara Tucci-Piergiovanni
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Eugen Zălinescu
Nomadic Labs, Paris, France

Abstract

First-generation blockchains provide probabilistic finality: a block can be revoked, albeit the probability decreases as the block "sinks" deeper into the chain. Recent proposals revisited committee-based BFT consensus to provide deterministic finality: as soon as a block is validated, it is never revoked. A distinguishing characteristic of these second-generation blockchains over classical BFT protocols is that committees change over time as the participation and the blockchain state evolve. In this paper, we push forward in this direction by proposing a formalization of the Dynamic Repeated Consensus problem and by providing generic procedures to solve it in the context of blockchains.

Our approach is modular in that one can plug in different synchronizers and single-shot consensus. To offer a complete solution, we provide a concrete instantiation, called Tenderbake, and present a blockchain synchronizer and a single-shot consensus algorithm, working in a Byzantine and partially synchronous system model with eventually synchronous clocks. In contrast to recent proposals, our methodology is driven by the need to *bound* the message buffers. This is essential in preventing spamming and run-time memory errors. Moreover, Tenderbake processes can synchronize with each other *without* exchanging messages, leveraging instead the information stored in the blockchain.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Blockchain, BFT-Consensus, Dynamic Repeated Consensus

Digital Object Identifier 10.4230/OASIcs.FAB.2021.1

Related Version *Full Version*: `https://arxiv.org/abs/2001.11965`

Acknowledgements We thank Philippe Bidinger for feedback on a previous version of this paper.

1 Introduction

Besides raising public interest, blockchains have also recently gained traction in the scientific community. The underlying technology combines advances in several domains, most notably from distributed computing, cryptography, and economics, in order to provide novel solutions for achieving trust in decentralized and dynamic environments.

Our work has been initially motivated by Tezos [18, 1], a blockchain platform that distinguishes itself through its self-amendment mechanism: protocol changes are proposed and voted upon. This feature makes Tezos especially appealing as a testbed for experimenting

with different consensus algorithms to understand their strengths and suitability in the blockchain context. Tezos relies upon a consensus mechanism build on top of a liquid proof-of-stake system, meaning that block production and voting rights are given to participants in proportion to their stake and that participants can delegate their rights to other stakeholders. As Nakamoto consensus [21, 17], Tezos' current consensus algorithm [24] achieves only probabilistic finality assuming an attacker with at most half of the total stake, and relying on a synchrony assumption.

The initial goal of this work was to strengthen the resilience of Tezos through a BFT consensus protocol to achieve deterministic finality while relaxing the synchrony assumption. We had two general requirements that we found were missing in the existing BFT consensus protocols. First, for security reasons, message buffers need to be bounded: assuming unbounded buffers may lead to memory errors, which can be caused either accidentally or maliciously, through spamming for instance. Second, as previously observed [2], plugging a classical BFT consensus protocol in a blockchain setting with a proof-of-stake boils down to solve a form of repeated consensus [13], where each consensus instance (i) produces a block, i.e., the decided value, and (ii) runs among a committee of processes which are selected based on their stake. To be applicable to open blockchains, committees need to be dynamic and change frequently. Frequent committee changes is fundamental in blockchains for mainly two reasons: (i) it is not desirable to let a committee be responsible for producing blocks for too long, for neither fairness nor security; (ii) participants' stake may change frequently.

Dynamic Repeated Consensus. Typically, repeated consensus is solved with state machine replication (SMR) implementations. We, instead, propose to use a novel formalism, dynamic repeated consensus (DRC) to take into account that, in the context of *open* blockchains, participants in consensus change. To this end, we propose that the selection of participants is based upon information readily available in the blockchains.

To solve DRC, we follow the methodology initially presented in [14] and revived more recently in [29, 22, 23]: we decouple the logic for synchronizing the processes in consensus instances from the consensus logic itself. Thus, our solution uses two main generic ingredients: a synchronizer and a single-shot consensus skeleton. Our approach is modular in that one can plug in different synchronizers and single-shot consensus algorithms. Our solution works in a partially synchronous model where the bound on the message delay is *unknown*, and the communication is *lossy* before the global stabilization time (GST). We note that losing messages is a consequence of processes having bounded memory: if a message is received when the buffers are full, then it is dropped.

Blockchain-based Synchronizer. The need for and the benefits of decoupling the synchronizer from the consensus logic have already been pointed out in [29, 22, 23, 6]. Indeed, such separation of concerns allows reusability and simpler proofs. We continue this line of work and propose a synchronizer for DRC which does not exchange messages. Instead, it relies upon local clocks while leveraging information already stored in the blockchain. Our solution allows buffers to be bounded and guarantees that correct processes in the synchronous period are always in the same round, except for negligeable periods of time due to clock drifts. Thus, processes can discard all the messages not associated with their current or next round. This is similar to the communication-closed round model [10, 16] and in contrast to most existing solutions, which, in principle, need to store messages for an unbounded number of rounds.

Consensus algorithm. To complete our DRC solution, Tenderbake, we also present a single-shot consensus algorithm. Single-shot Tenderbake is inspired by Tendermint [7, 3], in turn inspired by PBFT [8] and DLS [14]. We improve Tendermint in two aspects: i) we remove the reliable broadcast requirement during the asynchronous period, and ii) we provide faster termination. Tendermint terminates once processes synchronize in the same round after GST, in the worst case, in n rounds, where n is the size of the committee. Single-shot Tenderbake terminates in $f+2$ rounds, where f is the upper bound on the number of Byzantine processes. Tenderbake departs from its closest relatives Tendermint and HotStuff [29] in that it is driven by a bounded-buffers design leveraging a synchronizer that paces protocol phases on timeouts only. However, the price for this is that Tenderbake is not optimistic responsive as HotStuff, which makes progress at the speed of the network and terminates in $f+1$ rounds, at the cost of an additional phase. As a last difference, we note that, contrary to recent pipelined algorithms [29, 9], Tenderbake lends itself better to open blockchains. Pipelined algorithms focus more on performance, however pipelining imposes restrictions on how much and how frequently committees can change [9].

We are not aware of any existing approach providing a complete, generic DRC formalization. However, several references exist for particular aspects which we touch upon. For instance, repeated consensus with bounded buffers has been studied in [13, 27] but in system models which assume crash failures only. Working solutions for implementing dynamic committees are (mostly partially) documented in [11, 20, 19, 26, 28, 25, 5]. The differences with respect to the closest relatives of single-shot Tenderbake have been discussed above.

Outline. The paper is organized as follows: Section 2 defines the system model; Section 3 formalizes the DRC problem and proposes a generic solution; Section 4 proposes a synchronizer leveraging blockchain's immutability; Sections 5 - 6 present the single-shot consensus skeleton and respectively single-shot Tenderbake, as an example of an instantiation; Section 7 discusses message complexity and gives some intuition on the upper bound on the recovery time after GST; Section 8 concludes. Appendix B contains the detailed correctness proofs of Tenderbake.

2 System Model

We consider a message-passing distributed system composed of a possibly infinite set Π of processes. Processes have access to digital signing and hashing algorithms. We assume that cryptography is perfect: digital signatures cannot be forged, and there are no hash collisions. Each process has an associated public/private key pair for signing and processes can be identified by their public keys.

Execution model. Processes repeatedly run consensus instances to decide *output* values. New output values are appended to a *chain* that processes maintain locally. Consensus instances run in *phases*. The execution of a phase consists in broadcasting some messages (possibly none), retrieving messages, and updating the process state. At the end of a phase a correct process exits the current phase and starts the next phase. We consider that message sending and state updating are instantaneous, because their execution times are negligible in comparison to message transmission delays. This means that the duration of a phase is given by the amount of time dedicated to message retrieval.

Partial synchrony. We assume a partially synchronous system, where after some unknown time τ (the global stabilization time, GST) the system becomes synchronous and channels reliable, that is, there is a finite *unknown* bound δ on the message transfer delay. Before τ the system is asynchronous and channels are lossy.

We assume that processes have access to local clocks and that after τ these clocks are loosely synchronized: at any time after τ, the difference between the real time and the local clock of a process is bounded by some constant ρ, which, as δ, is a priori unknown.

Fault model. Processes can be *correct* or *faulty*. Correct processes follow the protocol, while faulty ones exhibit Byzantine behavior by arbitrarily deviating from the protocol.

Communication primitives. We assume the presence of two communication primitives built on top of point-to-point channels, where exchanged messages are authenticated. The first primitive is a best-effort broadcast primitive used by processes participating in a consensus instance and the second is a pull primitive which can be used by any process.

Broadcasting messages is done by invoking the primitive broadcast. This primitive provides the following guarantees: (i) integrity, meaning that each message is delivered at most once and only if some process previously broadcast it; (ii) validity, meaning that after τ if a correct process broadcasts a message m at time t, then every correct process receives m by time $t + \delta$. For simplicity, we assume that processes also send messages to themselves. Processes are notified of the reception of a message with a NewMessage event.

The pullChain primitive is used by a process to retrieve output values from other processes. This primitive guarantees that, if invoked by a process p at some time $t > \tau$, then p will eventually receive all the output values that correct processes had before t. We note that the pull primitive can be implemented in such a way that the caller does not need to pull all output values, but only the ones that it misses. Furthermore, output values can be grouped and thus received as a chain of values. Processes are notified of the reception of a chain with a NewChain event.

3 Dynamic Repeated Consensus

3.1 Problem definition

Originally, repeated consensus was defined as an infinite sequence of consensus instances executed by the *same* set of processes, with processes having to agree on an infinitely growing sequence of decision values [13]. Dynamic repeated consensus, instead, considers that each consensus instance is executed by a potentially different set of n processes where n is a parameter of the problem. More precisely, given the i-th consensus instance, only n processes $\Pi_i \subseteq \Pi$ participate in the consensus instance proposing values and deciding a unique value v_i. Processes in $\Pi - \Pi_i$ can only adopt v_i. Therefore output values can be either directly decided or adopted. We assume that every correct process agrees a priori on a value v_0.

To know the committee, each process has access to a deterministic selection function committee that returns a sequence of processes based on previous output values. More precisely, the committee Π_i is given by committee$([v_0])$ for $i \leq k$ and by committee$(\bar{v}_p[..(i-k)])$ for $i > k$, where $k > 0$ is a problem parameter, $\bar{v}_p$ denotes the sequence of output values of process p, and $\bar{s}[..j]$ denotes the prefix of length $j + 1$ of the sequence $\bar{s}$. Each process calls committee with its own decided values; however since decided values are agreed upon, committee returns the same sequence when called by different correct processes. We note that the sets Π_i are potentially unrelated to each other, and any pair of subsequent committees may differ. However, we assume that in each committee, less than a third of the members are faulty. For convenience, we consider the worst case: $n = 3f + 1$, and each committee contains exactly f faulty processes.

Dynamic repeated consensus, as repeated consensus, needs to satisfy three properties: agreement, validity, and progress. Agreement and progress have the same formulation for both problems. However, validity needs to reflect the dynamic aspect of committees. To this end, we define validity employing two predicates. The first one is isLegitimateValue. When given as input a value v_i, isLegitimateValue(v_i) returns true if the value has been proposed by a legitimate process, e.g., a process in Π_i. The second predicate is isConsistentValue. When given as input two consecutive output values v_i, v_{i-1}, isConsistentValue(v_i, v_{i-1}) returns true if v_i is consistent with v_{i-1}. This predicate takes into account the fact that an output value depends on the previous one, as commonly assumed in blockchains. For instance, when output values are blocks containing transactions, a valid block must include the identifier or hash of the previous block, and transactions must not conflict with those already decided. For conciseness, we define isValidValue(v_i, v_{i-1}) as a predicate that returns true if both isLegitimateValue(v_i) and isConsistentValue(v_i, v_{i-1}) return true for $i > 0$. Note that the use of an application-defined predicate for stating validity already appears in [2, 12].

An algorithm that solves the *Dynamic Repeated Consensus* problem must satisfy the following three properties:

- (**agreement**) At any time, if $\bar{v}_p$ and $\bar{v}_q$ are the sequences of output values of two correct processes p and q, then $\bar{v}_p$ is a prefix of $\bar{v}_q$ or $\bar{v}_q$ is a prefix of $\bar{v}_p$.
- (**validity**) At any time, if $\bar{v}_p$ is the sequence of output values of a correct process p, then the predicate isValidValue($\bar{v}_p[i]$,$\bar{v}_p[i-1]$) is satisfied for any $i > 0$.
- (**progress**) For any time t, there is a later $t' > t$ such that the sequence of output values of a correct process at time t is a strict prefix of the sequence of output values at time t'.

We use $\bar{s}[i]$ to denote the $(i+1)$-th element of the sequence $\bar{s}$.

3.2 A DRC solution for blockchains

3.2.1 Preliminaries

A *blockchain* is a sequence of linked blocks. The *head* of a blockchain is the last block in the sequence. The block *level* is its position in the sequence, with the first block having level 0. We call this block *genesis*. A block has a *header* and a *content*. The content typically consists of a sequence of transactions; it is application-specific and therefore we do not model it further. The block header includes the level of the block and the hash of the previous block, among other fields detailed later.

In a nutshell, the proposed DRC algorithm works as follows. At each level, for a block b which is proposed to be appended to the blockchain, processes run a single-shot consensus algorithm to agree on the tuple (u, h), where u is the content of b and h is the hash of the predecessor of b. Therefore, we consider that the output values in $\bar{v}$ from the DRC definition in Section 3.1 are the agreed upon tuples (u, h).

Intuitively, the block content is what needs to be agreed upon at a given level. Thanks to block hashes, the agreement obtained during a single-shot consensus instance implies agreement on the whole blockchain, *except for its head*, for which there might not yet be agreement on the other fields of the header besides the predecessor hash. The possible "disagreement" comes from processes taking a decision at possibly different times and thus on different proposed blocks which, however, share the same content. Agreement on the head is obtained implicitly at the next level. For clarity, we refer to a block as being *committed* if it is not the head of the blockchain of a correct process.

```
1   proc runDRC()
2     schedule onTimeoutPull() to be executed after I
3     updateState([genesis], ∅)
4     while true
5       initConsensusInstance()
6       (chain, certificate) := runConsensusInstance()
7       updateState(chain, certificate)

8   proc updateState(chain, certificate)
9     # NB: tail of blockchain_p is a prefix of chain
10    blockchain_p := chain
11    headCertificate_p := certificate
12    ℓ_p := length(chain)
13    h_p := hash(blockchain_p[ℓ_p − 1])

14  proc onTimeoutPull()
15    pullChain
16    schedule onTimeoutPull() to be executed after I

17  proc handleEvents()
18    while not stopEventHandler() do
19      upon NewMessage(msg)
20        handleConsensusMessage(msg)
21      upon NewChain(chain, proposalOrCertificate)
22        certificate := getCertificate(proposalOrCertificate)
23        if validChain(chain, certificate) then
24          if length(chain) > ℓ_p then
25            return (chain, certificate)
26          else if length(chain) = ℓ_p ∧ betterHead(chain, proposalOrCertificate) then
27                updateState(chain, certificate)
```

Figure 1 DRC entry point and auxiliary procedures.

In order for processes to validate a chain independently of the current consensus instance, a certificate is included in the block header to justify the decision on the previous block. A *certificate* is a quorum of signatures which serves as a justification that the content of the predecessor block was agreed upon by the "right" committee. To effectively check certificates, the public keys of committee members are stored in the blockchain.

3.2.2 A DRC algorithm

Fig. 1 presents the pseudocode of a *generic* procedure to solve DRC in the context of blockchains. It is generic in that it can run with *any* single-shot consensus algorithm.
We first enumerate the state variables at any correct process p. Namely, the state of p:

- $blockchain_p$, its local copy of the blockchain;
- ℓ_p, the level at which p runs a consensus instance, which equals the blockchain's length;
- h_p, the hash of the head of $blockchain_p$, that is, of the block at level $\ell_p - 1$;
- $headCertificate_p$, the certificate which justifies the head of $blockchain_p$.

In the pseudocode, all these state variables are considered global, while variables local to a procedure are those that do not have a subscript.

Next, we refine the answer to pullChain requests, in that we consider that the pullChain primitive retrieves more than just output values. Concretely, when a correct process p at level ℓ_p answers a pullChain request, it returns a tuple $(blockchain_p, proposalOrCertificate)$ where $blockchain_p$ is its local chain and $proposalOrCertificate$ is either: (1) the block that p considers as the current proposal at level ℓ_p or; (2) in absence of a proposal, $headCertificate_p$. Here, by *proposal* we mean a proposed block.

We now proceed to describing the entry point of the DRC algorithm, that is, the procedure runDRC in Fig. 1. Processes need not start DRC at the same time. When executing runDRC, a process starts by scheduling calls to pullChain. Then, using updateState,

it initializes its local variables, namely the state variables already presented and the variables specific to the single-shot algorithm. We use the function hash to compute the hash of some input. The function length returns the length of an input sequence.

After updating its state, the process iteratively runs consensus instances and once an instance has finished, it updates its state accordingly. Normally, a consensus instance simply decides on a value, and the corresponding block is appended to the blockchain. However, a process might also be behind other processes which have already taken decisions for more than one level. In this case, as soon as the process invokes the pullChain primitive, it retrieves missed decisions and thus possibly more blocks are appended to the blockchain.

In the presence of dynamic committees, it is not enough that processes call pullChain punctually when they are behind. Indeed, assume that a process p decides at level ℓ but the others are not aware of this and have not decided, because the relevant messages were lost; also assume that p is no longer a member of the committee at level $\ell + 1$, consequently, it no longer broadcasts messages and thus the other processes cannot progress. To solve this, each process invokes pullChain regularly, every I time units, where $I > 0$ is some constant.

During the execution of a consensus instance, processes continuously handle events to update their state. The event processing loop is implemented by the handleEvents procedure in Fig. 1. The termination of the event handler is controlled by the stopEventHandler procedure, which is specific to the single-shot consensus algorithm. There are two kinds of events: message receipts, represented by the NewMessage event, and chain receipts, represented by the NewChain event. Upon receiving a new message *msg*, a process p dispatches it to the consensus instance. Upon the receipt of a new chain, p updates its state accordingly:

- If the new chain is longer, and is valid, p starts a new consensus instance for a higher level; this is because the **return** on line 25 passes the control back to the runDRC procedure in line 6.
- If the new chain has the same length but a head which is "better", in some sense that specific to the single-shot consensus algorithm, then this signals to p that it is "behind", and in this case p only updates its state while remaining at the same level. In particular, only the DRC-related state is updated, while the single-shot instance remains unchanged. A specific betterHead procedure in given in Section 6. For the moment, we note that by means of betterHead, all processes have the same reference point for synchronization.

The NewChain event has, in addition to the *chain* parameter, the *proposalOrCertificate* parameter, which serves as a justification that the head's value has indeed been agreed upon. The role of validChain(*chain*) is two-fold:

1. to check whether *chain*'s head and the certificate from *proposalOrCertificate* match; for this to be possible, we assume access to a procedure getCertificate provided at the single-shot consensus level (see Section 6.5);
2. to check whether for any level ℓ the predicate isValidValue($chain[\ell]$, $chain[\ell-1]$) is satisfied; this means that the hash field in the header of the block $chain[\ell]$ equals hash($chain[\ell-1]$) (so that the predicate isConsistentValue is satisfied), and that the value in each block is proposed by the right committee (so that the predicate isLegitimateValue is satisfied); for the latter to be possible, certificates are stored in blocks as single-shot consensus specific elements (see Section 6.2).

The DRC solution we presented is generic, one can instantiate it by providing implementations to initConsensusInstance, startConsensusInstance, getCertificate, betterHead, and stopEventHandler. We show how to concretely implement them in Sections 5 and 6.

4 A synchronizer for blockchains

We describe a synchronizer for *round*-based consensus algorithms. Round-based consensus algorithms progress in rounds, where, at each round, processes attempt to reach a decision, and if they fail, they advance to the next round to make another attempt.

In the context of round-based consensus algorithms, a standard way to achieve termination of a single consensus instance is to ensure that processes remain at the same round for a sufficiently long period of time [14, 8, 16, 10]. The synchronizer we propose realizes this by leveraging the immutability of the blockchain. One feature of our synchronizer is that it does not exchange any message, thus, it does not increase the communication complexity. Instead, it relies on rounds having the same duration for all processes. We require that rounds duration are increasing and unbounded. Concretely, the duration of a round $r > 0$ is given by $\Delta(r)$, where Δ is a function with domain $\mathbb{N} \setminus \{0\}$ such that, for any duration $d \in \mathbb{N}$, there is a round r with $\Delta(r) \geq d$. Furthermore, we assume that rounds duration are larger than the clock skew, so that rounds are not skipped in the synchrony period. Note that by using round durations, Tenderbake cannot be optimistic responsive like, for instance, [29].

▶ Remark 1. In practice, given estimates δ_{real} of the real message delay and δ_{max} of the maximum message delay δ, we would choose Δ such that: (i) $\Delta(1)$ is slightly bigger than δ_{real}, (ii) Δ increases rapidly (e.g. exponentially) till it reaches δ_{max}, and (iii) then it increases slowly (e.g. linearly) afterwards.

To determine at which round the process should be, the synchronizer relies on local clocks. Therefore, when clocks are synchronized, all processes will be at the same round. However, a prerequisite is that processes agree on the starting time of the current instance. As different processes may decide at different rounds, and therefore at different times, there is a priori no consensus about the start time of an instance. We adopt a solution based on the following observation: if the round at which a decision is taken is eventually known by all processes, then they can agree on a common global round at which a consensus instance is considered to have terminated. Indeed, a process considers that the consensus instance has ended at the smallest round at which some process has decided.

The above solution can be implemented by (1) considering that a block header stores the round at which the block is produced, and (2) using the betterHead procedure, which is called by a process p at line 27 upon receiving a new chain in response to a pullChain request. This procedure checks if some other process has already taken a decision sooner, in terms of rounds. If this is the case, betterHead signals to its caller that it is "behind" and thus that it needs to resynchronize. We postpone the concrete implementation of betterHead to Section 6.4 because it is specific to the single-shot consensus algorithm. For the moment, to illustrate the role of betterHead, Fig. 2 shows an update of the head of a process p's blockchain. Initially, the head of p's local chain is b'. Then, p sees the block b'' at level ℓ with a smaller round than b' and therefore updates the head of its local chain to b''.

Finally, we present the synchronization procedure in Fig. 3. We assume that the genesis block contains the time t_0 of its creation. To synchronize, p uses its local clock, whose value is obtained by calling now(), and the rounds of the blocks in its blockchain to find out what its current round and the time position within this round should be. Process p determines first the starting time of the current level and stores it in t. To do this, p adds the durations of all rounds for all previous levels. Once p has determined t, it finds the current round by checking incrementally, starting from round $r = 1$ whether the round r is the current round: r is the current round if there is no time left to execute a higher round. The variable t is updated to represent the time at which round r started. The difference $t' - t$ represents the offset between the beginning of the round r and the current time.

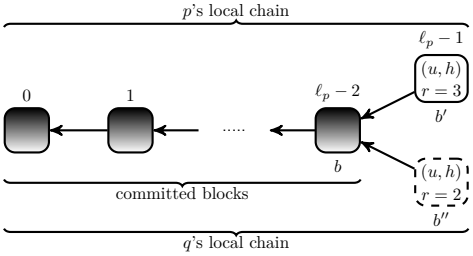


Figure 2 An update of the head of p's chain. Solid boxes represent blocks in p's chain before the update, while the dashed box represents the block that triggers the update. Block levels and labels are given above and respectively below the corresponding boxes. The hash h is that of block b.

```
28  proc synchronize()
29    t' := now()
30    t := t_0 + Σ_{ℓ=0}^{ℓ_p−1} Σ_{j=1}^{round(ℓ)} Δ(j) + Σ_{j=1}^{r'} Δ(j)
31    r := 1
32    while t + Δ(r) ≤ t' do
33      t := t + Δ(r)
34      r := r + 1
35    return (r, t' − t)
```

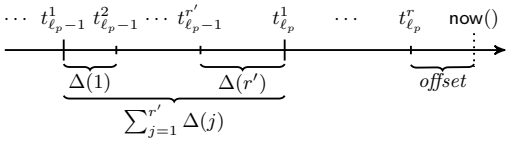


Figure 3 A round-based synchronizer and a timeline. Small/large vertical lines represent round/level boundaries, respectively.

Fig. 3 also illustrates the timeline of a process that increments its rounds using the procedure synchronize, where $t^r_{\ell_p}$ represents the starting time of the round r of level ℓ_p and r' stands for the last round of level $\ell_p - 1$. The figure also illustrates the offset $t' - t$.

5 A Single-Shot Consensus Skeleton

In this section we give a generic implementation for the procedure runConsensusInstance from Section 3.2.2. Here we make another standard assumption on the structure of the single-shot consensus algorithm, namely that each round evolves in sequential *phases*. For instance, PBFT in normal mode has 3 phases (named *pre-prepare*, *prepare*, and *commit*), Tendermint as well, DLS and Hotstuff have 4 phases, etc.

We let m denote the number of phases. As for rounds, we assume that each phase has a predetermined duration. The duration is given by the round r it belongs to, and it is denoted $\Delta'(r)$. For simplicity, we assume that $\Delta(r) = m \cdot \Delta'(r)$. We also refine the assumption on round durations, and also require that phase durations are larger than the clock skew, so that phases are not skipped in the synchrony period, i.e. $\Delta'(1) > 2\rho$.

To synchronize correctly, a process also needs to update its phase (not only its round) and to know its time position within a phase. These can be readily determined from the round and the round offset returned by synchronize. The procedure getNextPhase, presented in Fig. 4, performs this task. For the pseudocode, we consider that each phase has a label identifying it and we use *phases* to denote the sequence of phase labels.

The entry point of a single-shot consensus instance is runConsensusInstance, given in Fig. 4. As part of its state, a process p also maintains its current round r_p. A process p starts by calling synchronize in an attempt to (re)synchronize with other processes. We recall that this is just an attempt and not a guarantee because clocks are not necessarily synchronized before τ. If synchronize returns that p should be at a round in the past with respect to p's

```
36  proc runConsensusInstance()
37    (round, roundOffset) = synchronize()
38    if r_p > round then  # p is ''ahead''
39      runConsensusInstance()
40    else # p is ''behind''
41    (phase, phaseOffset) :=
42                    getNextPhase(round, roundOffset)
43      r_p := round
44      set runEventHandler timer to
45                                  Δ'(r_p) − phaseOffset
46      if p ∈ committeeAtLevel(ℓ_p) then
47        goto phase
48      else
49        goto phase-observer
```

```
50  proc getNextPhase(round, roundOffset)
51    i := roundOffset / Δ'(round)
52    phase := phases[i]
53    phaseOffset := roundOffset − i · Δ'(round)
54    return (phase, phaseOffset)

55  proc advance(decisionOption)
56    match decisionOption with
57      | Some (block, blockCertificate) →
58          c := blockchain_p ++ block
59          return (c, blockCertificate)
60      | None → # no decision
61          r_p := r_p + 1
62          filterMessages()
63          runConsensusInstance()
```

Figure 4 Entry point and progress procedures for generic single-shot consensus.

current round, then p invokes (indirectly) the synchronizer again. This active waiting loop ensures that p is ready to continue its execution as soon as it is not "ahead" anymore. We note that a jump backward to a previous round or phase may jeopardize safety. When p is "behind", it first uses the procedure getNextPhase to obtain the phase at which it should be. Next, it updates its round and the timer used to time the execution of the event handler. Concretely, through this timer, the generic procedure stopEventHandler is implemented as follows:

```
64  proc stopEventHandler()
65    return true iff timer runEventHandler expired
```

We recall this procedure is used by handleEvents at line 18 in Fig. 1.

After setting *runEventHandler*, p checks whether it is part of the committee for level ℓ_p. To this end, we assume having access to a committeeAtLevel function, which returns the committee at some given level ℓ. This function corresponds to committee$(\bar{v}_p[..(\ell - k)])$ (Section 3.1), where $\bar{v}_p$ is the sequence of output values of the caller process p. Finally, p executes the single-shot consensus algorithm according to its role and to the phase returned by getNextPhase. The determined phase is executed by means of an unconditional jump to corresponding phase label. The two **goto** statements in Fig. 4 are intentionally symmetric for committee and non-committee members to keep all processes in sync. This has the advantage of not introducing delays when they eventually become part of the committee.

Fig. 4 also shows the advance procedure, which is used by processes to handle the progress of the current consensus instance by either returning the control to runDRC when a decision can been taken; or otherwise increasing the round. In this former case, advance first prepares the updated blockchain, appending the block corresponding to the decision to its current blockchain; runDRC will then update the state accordingly, for instance increasing the level. The procedure advance has one parameter, which is optional, represented in the pseudocode as a value of an optional type (with values of the form Some x if the parameter is present or None if it is not). The parameter is present when the current consensus instance has taken a decision. In this case, the parameter is a tuple consisting of a block containing the decided value and of a certificate justifying the decision. Otherwise, when no decision is taken, the process increases its round and filters its message buffer by removing messages no longer necessary. The filtering procedure filterMessages is specific to the consensus instance.

We conclude by presenting in Fig. 5 the pseudocode capturing the behavior of the processes which are not part of a committee for a given level. We call such processes *observers*. Contrary to committee members, observers are passive in the sense that they only receive (but not send) messages and update their state accordingly.

```
66  phases[1]-observer phase:
67    handleEvents()
68  ⋮
69  phases[m − 1]-observer phase:
70    handleEvents()

71  phases[m]-observer phase:
72    handleEvents()
73    advance(getDecision())
```

Figure 5 Generic single-shot algorithm for an observer.

This observer behavior serves two purposes:

i) to keep the blockchain at each observer up to date;

ii) to check at the end of the round whether a decision was taken, and if so, whether the observer becomes a committee member at the next level.

To achieve i), the observer checks if it can adopt a proposed value. It does so by invoking the handleEvents and advance procedures, where the parameter to advance is obtained using the procedure getDecision, which is specific to the single-shot consensus algorithm. Concerning ii), when the corresponding check (line 46) is successful, the observer switches roles and acts as a committee member. We note that line 46 is reached when the observer end its round and calls advance, which in turn calls runConsensusInstance at the end.

As for the DRC solution in Section 3.2.2, the methods presented in this section are generic. One can instantiate them by providing implementations to the filterMessages and getDecision procedures. We show such concrete implementations in the next section.

6 Single-shot Tenderbake

To show the specific phase behavior of a committee member, we first introduce some terminology inspired by Tezos. Tenderbake committee members are called *bakers*. At each round, a value is proposed by the proposer whose turn comes in a round-robin fashion. Tenderbake has three types of phases: PROPOSE, PREENDORSE, and ENDORSE, each with a corresponding type of message: Propose for proposals, Preendorse for preendorsements, and Endorse for endorsements. A fourth type of message, Preendorsements, is for the retransmission of preendorsements. A baker *proposes*, *preendorses*, and *endorses* a value v (at some level and with some round) when the baker broadcasts a message of the corresponding type. Only one value per round can be proposed or (pre)endorsed. A set of at least $2f + 1$ (pre)endorsements with the same level and round and for the same value is called a *(pre)endorsement quorum certificate* (QC).

We consider that Propose messages are blocks. This is a design choice that has the advantage that values do not have to be sent again once decided.

Within a consensus instance, if a baker p receives a preendorsement QC for a value v and round r, then p keeps track of v as an *endorsable value* and of r as an *endorsable round*. Similarly, if a baker p receives a preendorsement QC for a value v and round r during the ENDORSE phase of the round r, then p locks on the value v, and it keeps track of v as a *locked value* and of r as a *locked round*. Note that the locked round stores the most recent round at which p endorsed a value, while the endorsable round stores the most recent round that p is aware of at which bakers may have endorsed a value.

The execution of a round works as follows. During the PROPOSE phase, the designated proposer proposes a value v, which can be newly generated or an endorsable value from a previous round r. During the PREENDORSE phase, a baker preendorses v if it is not locked or if it is locked on a value at a previous round than r; in particular, it does not preendorse

v if it is locked and v is newly generated. If a baker does not preendorse v, then it sends a Preendorsements message with the preendorsement QC that justifies its more recent locked round. During the ENDORSE phase, if bakers receive a preendorsement QC for v, they lock on it and endorse it. If bakers receive an endorsement QC for v, they *decide* v.

Tenderbake inherits from classical BFT solutions the two voting phases per round and the locking mechanism. Tracking endorsable values is inherited from [7]. Tenderbake distinguishes itself in a few aspects which we detail next.

Preendorsement QCs. For safety, bakers accept endorsable values only from higher rounds than their locked round. Assume a correct baker p locks and all other correct bakers locked at smaller rounds. Assume also that the messages from p are lost. To prevent p from not making progress, it is enough to include the preendorsement QC that made p lock in Endorse and Propose messages. In this way, bakers can update their endorsable values and rounds accordingly and propose values that can be accepted by any correct locked baker. Tendermint does not need such QCs as it assumes reliable communication in the asynchronous period.

The Preendorsements message. For faster termination of a consensus instance, when a baker refuses a proposal because it is locked on a higher round than the endorsable round of the proposed value, it broadcasts a Preendorsements message. This message contains a preendorsement QC justifying its higher locked round. During the next round, bakers use this QC to set their endorsable value to the one with the highest round. The consensus instance terminates with the first correct proposer. Thus, in the worst-case scenario, when the first f bakers are Byzantine, Tenderbake terminates in $f+2$ rounds after τ, assuming that processes have achieved round synchronization and that the round durations are sufficiently large.

Endorsement QCs. For processes to be able to check that blocks received by calling pullChain are already agreed upon, each block comes with an endorsement QC for the block at the previous level. Furthermore, for the same reason, in response to a pull request, a process also attaches the endorsement QC that justifies the value in the head of the blockchain.

6.1 Process state and initialization

In addition to the variables mentioned in Section 3.2.2, a process p running Tenderbake maintains its current round r_p as well as:

- $lockedValue_p$ and $lockedRound_p$ to keep track respectively of the value on which p is locked and the round during which p locked on it,
- $endorsableValue_p$ to keep track of the proposed value with a preendorsement QC (with the highest round), which can therefore be considered endorsable,
- $endorsableRound_p$ and $preendorsementQC_p$ to store the round and the preendorsement QC corresponding to an endorsable value;
- $headCertificate_p$ to store the endorsement QC for p's last decided value.

The variable $headCertificate_p$ (introduced in Section 3) is empty at level 1 (Fig. 1, line 3). The state of a process is initialized by the procedure initConsensusInstance:

```
74  proc initConsensusInstance()
75    r_p := 1
76    lockedValue_p := ⊥; lockedRound_p := 0
77    endorsableValue_p := ⊥; endorsableRound_p := 0
78    preendorsementQC_p := ∅
79    messages_p := ∅
```

where, by abuse of notation, we use $x := \perp$ to denote that x has become undefined.

6.2 Messages and blocks

We write messages using the following syntax: $type_p(\ell, r, h, payload)$, where *type* is Propose, Preendorse, Endorse, or Preendorsements, p is the process that sent the message, ℓ and r are the level and the round during which the message is generated, h is the block hash at level $\ell - 1$, and *payload* is the type specific content of the message.

The payload (eQC, u, eR, pQC) of a Propose message contains the endorsement quorum eQC that justifies the block at the previous level and the proposed value u to be agreed on. The payload also contains, in case u is a previously proposed value, the corresponding endorsable round and the preendorsement QC that justifies u. If the proposed value is new, then eR is 0 and pQC is the empty set.

Given a Propose$(\ell, r, h, (eQC, u, eR, pQC))$ message, the corresponding block has contents u, while the remaining fields, notably the hash h, are part of the block header.

The payload of a Preendorse message consists of the value to be agreed upon while the payload of an Endorse message consists of an endorsed value. The payload of a Preendorsements message consists of a preendorsement QC justifying some endorsable value and round.

6.3 Message management

The message management is designed such that message buffers are bounded. We prove this in Lemma 3 and we give some more intuition in Appendix A. In this section, we only focus on the elements needed to understand single-shot Tenderbake, namely the handleConsensusMessage and some helper procedures. The procedure handleConsensusMessage is depicted in Fig. 6. A process p adds (line 84) to its message buffer valid messages *msg* but only from the current and next round. Messages from the next round are needed in order to cater for the possible clock drift. Moreover, if a preendorsement QC is observed for a higher round than the current $endorsableRound_p$, then p updates $endorsableValue_p$, $endorsableRound_p$, and $preendorsementQC_p$ using the procedure updateEndorsable (line 85). Finally, as an optimization, if the received message is from either a higher level or from the same level but with a different hash, then p attempts to resynchronize by calling pullChain (line 87).

The procedure filterMessages() removes messages not for the current round (see Appendix A). The helper procedures used in Fig. 6 are described as follows:

- proposedValue() returns the current proposed value of the block at level ℓ;
- valueQC(qc) and roundQC(qc) return the value and respectively the round from a qc;
- pQC(*msg*) returns the preendorsement QC from a Propose or Preendorsements message *msg*; if the Propose message does not contain a preendorsement QC (because what is proposed is a new value), then pQC returns the empty set;
- proposal(), preendorsements(), and endorsements() return the proposal, preendorsements, and respectively the endorsements contained in *messages*.

6.4 Tenderbake main loop

Fig. 7 gives the execution of one round of Tenderbake by baker p, when the round's three phases are executed in sequence. We recall that the pseudocode has the same structure as that for observers, as described in Section 5. Each phase consists of a conditional broadcast followed by a call to handleEvents (described in Section 3.2.2). In addition, the ENDORSE phase calls advance (described in Section 3.2.2). In the PROPOSE phase, p checks if it is the proposer for the current level ℓ_p and round r_p (line 102). If so, p proposes:

```
80  proc handleConsensusMessage(msg)
81    let type_q(ℓ, r, h, payload) = msg
82    if ℓ = ℓ_p ∧ h = h_p ∧ (r = r_p ∨ r = r_p + 1) then
83      if isValidMessage(msg)
84        messages_p := messages_p ∪ {msg}
85        updateEndorsable(msg)
86    if (ℓ = ℓ_p ∧ h ≠ h_p) ∨ ℓ > ℓ_p then
87      pullChain

88  proc updateEndorsable(msg)
89    if |preendorsements()| ≥ 2f + 1 then
90      endorsableValue_p := proposedValue()
91      endorsableRound_p := r_p
92      preendorsementQC_p := preendorsements()
93    else if type(msg) ∈ {Propose, Preendorsements} then
94      pQC := pQC(msg)
95      if pQC ≠ ∅ ∧ roundQC(pQC) > endorsableRound_p then
96        endorsableValue_p := valueQC(pQC)
97        endorsableRound_p := roundQC(pQC)
98        preendorsementQC_p := pQC

99  proc filterMessages()
100   messages_p := messages_p \ {type(ℓ, r, h, payload) ∈ messages_p | r ≠ r_p}
```

Figure 6 Message management in Tenderbrake.

- either a new value u, returned by the procedure newValue; here it is assumed that u is consistent with respect to the value u' contained in the last block of the blockchain of the process that calls this procedure; that is, isConsistentValue(v, v') holds (see Section 3.1), where v, v' are the output values corresponding to u, u';
- or its $endorsableValue_p$ if defined; in this case, p includes in the payload of its proposal the corresponding endorsable round and the preendorsement QC that justifies it.

The payload also includes the endorsement QC to justify the decision for the previous level.

In the PREENDORSE phase, p checks if the value u from the Propose message received from the current proposer is preendorsable (lines 110-111). Namely, it checks whether one of the following conditions are satisfied:

- p is unlocked ($lockedRound_p = 0$, thus the second disjunction at line 111 is true); or
- p is locked (i.e. $lockedRound_p > 0$), u was already proposed during some previous round (i.e. $0 < eR < r_p$), and:
  - p is already locked on u itself (thus the first disjunction at line 111 is true); or
  - p is locked on $u' \neq u$ and its locked round is smaller than the endorsable round associated to u.

In the second case, there is a preendorsement QC for u and round eR, thanks to the validity check on the Propose message. If the condition holds, then p preendorses u. If p cannot preendorse u as it is locked on some value $u' \neq u$ with a higher locked round than eR, then p broadcasts the preendorsement QC that justifies v'. If received on time, this information allows the next proposer to choose a value that passes the checks at all correct bakers.

In the ENDORSE phase, p checks if it received a preendorsement QC for the proposed value u. If yes, p updates its *lockedValue* and *endorsableValue* and broadcasts its Endorse message, along with all the Preendorse messages for u (lines 117-120). Note also that in this case p has already updated its endorsable value to u and its endorsable round to r_p while executing handleEvents.

Finally, at the end of this last phase, which is also the end of the round, bakers call advance with a parameter that signals whether a decision can be taken or not. This parameter is obtained using getDecision, implemented is as follows:

```
101  PROPOSE phase:
102    if proposer(ℓ_p, r_p) = p then
103      u := if endorsableValue_p ≠ ⊥ then endorsableValue_p
104           else newValue()
105      payload := (headCertificate_p, u, endorsableRound_p, preendorsementQC_p)
106      broadcast Propose_p(ℓ_p, r_p, h_p, payload)
107    handleEvents()

108  PREENDORSE phase:
109    if ∃q, eQC, u, eR, pQC :
110        Propose_q(ℓ_p, r_p, h_p, (eQC, u, eR, pQC)) ∈ messages_p ∧
111        (lockedValue_p = u ∨ lockedRound_p < eR < r_p) then
112      broadcast Preendorse_p(ℓ_p, r_p, h_p, hash(u))
113    else if lockedValue_p ≠ ⊥ then
114      broadcast Preendorsements(ℓ_p, r_p, h_p, preendorsementQC_p)
115    handleEvents()

116  ENDORSE phase:
117    if |preendorsements()| ≥ 2f + 1 then
118      u := proposedValue()
119      lockedValue_p := u;  lockedRound_p := r_p
120      broadcast Endorse_p(ℓ_p, r_p, h_p, hash(u))
121      broadcast preendorsementQC_p
122    handleEvents()
123    advance(getDecision())
```

Figure 7 Single-shot Tenderbake for baker p.

```
124  proc getDecision()
125    if |endorsements()| ≥ 2f + 1 then
126      return Some (proposal(), endorsements())
127    else
128      return None
```

6.5 The betterHead procedure

The role of betterHead is to make processes agree on the same blockchain head; recall that they already agree on the head contents, but not necessarily on the head's header. Agreeing on the same blockchain head has in turn two roles:

- allowing agreement on the round at which a decision was taken at the previous level, which is one of the ingredients for processes to synchronize at the current level, as explained in Section 4.
- allowing agreement to take place at the current level; recall that at the current level agreement needs to be reached also on the hash of the block at the predecessor level, that is, on the hash of the head of a process' blockchain.

To reach these two goals, as suggested in Section 4, processes adopt the head with the smallest round. However, there is a caveat: if this would be the only check done by betterHead, processes might end up with a head on top of which no proposal will be accepted in case they have seen an endorsable value: indeed, the hash component of such a value may not match the new head. To avoid this situation, a process first performs an additional check in case they have seen an endorsable value. When *proposalOrCertificate* is a proposal, the check is similar to the check for preendorsing (line 111): the endorsable round of process p is smaller than the one in the received proposal (line 133). When *proposalOrCertificate* is a certificate we simply required that the process has not seen an endorsable value. The betterHead procedure implementing these checks is given next.

```
129  proc betterHead(chain, proposalOrCertificate)
130    let ⟨_, r, . . . ; ·⟩ = head(chain)
131    match proposalOrCertificate with
132    | Propose(_, _, _, (_, _, eR, _)) →
133      return endorsableRound_p < eR ∨ (endorsableRound_p = eR ∧ r < round(ℓ_p − 1))
134    | _ → # proposalOrCertificate is a certificate
135      return endorsableRound_p = 0 ∧ r < round(ℓ_p − 1)
```

In the pseudocode, $\langle \ldots ; \ldots \rangle$ denotes a block, with the part before the semicolon representing the block's header and the part after it its contents. The procedure head(*chain*) returns the head of *chain*. Also, recall that round(ℓ) returns the round contained in the header of the block at level ℓ in the caller's blockchain.
As for the implementation of getCertificate, it is a simple match on *proposalOrCertificate*:

```
136  proc getCertificate(proposalOrCertificate)
137    match proposalOrCertificate with
138      | Propose_q(_, _, _, (eQC, _, _, _)) → return eQC
139      | eQC → return eQC
```

7 Correctness and Complexity

The following theorem states that Tenderbake provides a solution to DRC. Its proof can be found in Appendix B.

▶ **Theorem 2.** *Tenderbake satisfies validity, agreement, and progress.*

Bounded memory. We assume that all values referred to by global or local variables of a process p are stored in volatile memory, except for the variable $blockchain_p$ whose value is stored on disk. We recall that the message buffer is represented by the $messages_p$ variable. The following lemma shows that a process can use fixed-sized buffers, namely of size $4n$.

▶ **Lemma 3.** *For any correct process p, at any time, $|messages_p| \leq 4n + 2$.*

Proof. Let p be some correct process. Given that in $messages_p$ only messages from the current and next round are added (line 84), and that with each new round messages from the previous round are filtered out (line 100), $messages_p$ contains at most 2 proposals, at most $2n$ preendorsements, and at most $2n$ endorsements. ◀

The following result states that a process only uses bounded memory. We assume here that the underlying implementation of the pullChain primitive does not count towards the memory usage of a process.

▶ **Theorem 4.** *At any time, the size of the volatile memory of any correct process is in $\mathcal{O}(n)$.*

Proof. A correct process maintains a constant number of variables, and except *messages*, each variable stores a primitive value or a QC. A QC contains at most n messages and each message has a constant size. The $\mathcal{O}(n)$ bound follows from these observations, and the observation concerning the *messages* variable from the proof of Lemma 3. ◀

Message and round complexity. Each round has a message complexity of $O(nm)$ due to the n-to-m broadcast, where m is the current number of processes in the system.

Concerning round complexity, it is known that consensus, in the worst case scenario, cannot be reached in less than $f + 1$ rounds [15]. In Tenderbake, after bakers synchronize and the round durations are sufficiently long (namely, at least $\delta + 2\rho$), a decision is taken in at most $f + 2$ rounds, as already mentioned in Section 6. See Lemma 15 in Appendix B for a proof. Intuitively, f rounds are needed in case the proposers of these rounds are Byzantine. Another round is needed if there is a correct process locked on a higher round than the endorsable round of the proposed value. However, in this case, the next proposer is correct and will have updated its endorsable round, and therefore its proposed value will be accepted and decided by all correct processes.

Recovery time. Finally, we discuss the time required for bakers to synchronize after τ. A worst-case scenario analysis is in our technical report [4]. Roughly, the recovery time is the maximum time between the error that the clock can experience and the time necessary for a process to fetch the missing blocks, which is at least one round-trip time: the time to ask for the current blockchain and to get the reply. We believe that in practice the time to pull a new chain (and even to pull just the last block) is considerably bigger than the maximum error clock that a process can experience during the asynchronous period. Finally, if all processes are at the same level but not at the same round, then, as the synchronizer is called at the end of every round, all processes synchronize in at most one round.

8 Conclusion

In this paper, we proposed a formalization of dynamic repeated consensus, a general approach to solve it, and a BFT solution working with bounded buffers by leveraging a blockchain-based synchronizer. We have implemented the proposed solution in a prototype[1]. A full-fledged (based on proof-of-stake and with smart contracts) implementation is being developed[2]. Experiments with running a Tenderbake testnet are underway. A Tenderbake simulator has already been implemented[3].

Besides practical aspects such as experimenting with the testnet and the simulator, as future work, we see the following exciting directions: explore the relationship between achieving asynchronous responsiveness and providing bounded buffers; improve message size and complexity by means of aggregated or threshold signatures; mechanize the proofs; and analyze Tenderbake from an economic perspective when considering rational agents.

References

1 Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the Tezos Blockchain. In *Proc. High Performance Computing and Simulation*, 2019.

2 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of Tendermint-core blockchains. In *Proc. Principles of Distributed Systems*, 2018.

3 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Dissecting Tendermint. In *Proc. Networked Systems*, 2019.

[1] `https://gitlab.com/nomadic-labs/tezos/-/tree/tenderbake_proto`
[2] `https://gitlab.com/nomadic-labs/tezos/-/blob/tenderbake-florence`
[3] `https://gitlab.com/nomadic-labs/tenderbake-simulator`

4 Lăcrămioara Aştefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci, and Eugen Zălinescu. Tenderbake – a solution to dynamic repeated consensus for blockchains, 2021. arXiv:2001.11965.

5 Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. From byzantine replication to blockchain: Consensus is only the beginning. In *Proc. International Conference on Dependable Systems and Networks*, 2020.

6 Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. In *Proc. International Symposium on Distributed Computing*, 2020.

7 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, 2018. arXiv:1807.04938.

8 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

9 T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018.

10 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Comput.*, 2009.

11 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 2019.

12 T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (Leader/Randomization/Signature)-free Byzantine consensus for consortium blockchains. *CoRR*, 2017.

13 Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, and Sam Toueg. With finite memory consensus is easier than reliable broadcast. In *Proc. Principles of Distributed Systems*, 2008.

14 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

15 Michael J Fischer and Nancy A Lynch. A lower bound for the time to assure interactive consistency. *Information processing letters*, 1982.

16 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *Proc. ACM Symposium on Principles of Distributed Computing*, 1998.

17 J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. EUROCRYPT International Conference*, 2015.

18 L.M. Goodman. Tezos – a self-amending crypto-ledger, 2014.

19 Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, 2018.

20 Jae Kwon and Ethan Buchman. Cosmos: A Network of Distributed Ledgers.

21 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

22 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *CoRR*, 2019. arXiv:1909.05204.

23 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine SMR. In *Proc. International Symposium on Distributed Computing*, 2020.

24 Nomadic Labs. Analysis of Emmy$^+$. https://blog.nomadic-labs.com/analysis-of-emmy.html, 2019.

25 Rafael Pass and Elaine Shi. Rethinking large-scale consensus. *IACR Cryptol. ePrint Arch.*, 2018.

26 Roberto Saltini. Correctness analysis of IBFT. *CoRR*, 2019.

27 Omid Shahmirzadi, Sergio Mena, and André Schiper. Relaxed atomic broadcast: State-machine replication using bounded memory. In *Proc. IEEE International Symposium on Reliable Distributed Systems*, 2009.

28 The LibraBFT Team. State machine replication in the Libra blockchain, 2019.

29 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proc. ACM Symposium on Principles of Distributed Computing*, 2019.

A Valid messages and bounded buffers

Recall handleConsensusMessage in Section 6.3. As a necessary check for message buffers to be bounded, upon the retrieval of a new message msg, a process p first checks if the level, round, and hash in msg's header match respectively p's current level, either the current round or the next round, and the hash of the block at the previous level. If yes, p then checks that the message is valid, with the procedure isValidMessage (line 83). $\mathsf{Propose}_q(\ell,r,h,(eQC, u,eR,pQC))$ is *valid* if q is the proposer for level ℓ and round r and if eQC is an endorsement QC for level $\ell-1$ with the round, hash, and value matching those in p's blockchain. In addition, either pQC is empty and eR is 0 (i.e. u is newly proposed), or the round, value, and hash from pQC match eR, u, and h_p, respectively. Messages in eQC and pQC must be valid themselves, in particular they must be generated by bakers at levels $\ell-1$ and ℓ, respectively. These validity checks ensure that the value (u,h) satisfies the isLegitimateValue predicate from Section 3.1. The validity conditions for the other types of messages are similar, and thus omitted. We note, however, that for preendorsements and endorsements it is required that the corresponding proposal has been already received, so that it can be checked that the hash included in the payload matches the proposed value.

There are three additional aspects of handleConsensusMessage in Section 6.3 that together with the validity check, ensure that buffers are bounded: (1) only valid messages are added (line 83); (2) messages for the next round are kept (line 84) to cater for the possible clock drift; (3) messages from higher levels trigger p to ask for the sender's blockchain (line 87), because such messages "from the future" suggest that p is behind; however, the sender might be lying about being ahead. Recall that the procedure advance only calls filterMessages after a round increment (line 62). Recall also that filterMessages removes messages not matching the current round (line 100). Together with the above elements, the filtering ensures that message buffers are bounded (Lemma 3).

B Correctness proof

B.1 Validity and Agreement

▶ **Theorem 5.** *Tenderbake satisfies validity.*

Proof. The local chain of a correct process p is formed by proposals and/or chains obtained by p calling pullChain. In either case, the content of each block satisfies the predicate isValidValue by the definition of either isValidMessage or validChain. ◀

▶ **Lemma 6.** *Correct bakers preendorse and endorse at most once per round at a given level.*

Proof. Preendorse and Endorse messages are sent only during the corresponding phase (line 112 and line 120, respectively). To show that there is at most one Preendorse (resp. Endorse) per round it suffices to show that a phase is executed only once per round. Firstly, phases are executed sequentially. Secondly, non-sequential jumps happen only at line 47 (resp. at line 49) in runConsensusInstance; in turn, runConsensusInstance is called by either advance (line 63), after increasing the round; or runDRC (line 6), after increasing the level (line 7) once a decision is taken (line 59) or a longer chain is received (line 25). ◀

▶ **Lemma 7.** *At most one value can have a (pre)endorsement QC per round.*

Proof. By contraction, using Lemma 6. ◀

We say a baker p is locked on a tuple (u, h) if $lockedValue_p = u$ and $h_p = h$. We define $L^{u,h}_{\ell,r}$ as the set of *correct* bakers locked on the tuple (u, h) at level ℓ and at the end of round r. We also define $\mathsf{preendos}(\ell, r, u, h)$ as the set of preendorsements generated by *correct* processes for some level ℓ, some round r, some value u, and some hash h.

▶ **Lemma 8.** *Let ℓ be a level, r a round, u a value, and h a block hash. For any round $r' \geq r$ and any tuple $(u', h') \neq (u, h)$, if $|L^{u,h}_{\ell,r}| \geq f + 1$, then $|\mathsf{preendos}_p(\ell, r', u', h')| \leq f$.*

Proof. We reason by contradiction. Suppose that $|L^{u,h}_{\ell,r}| \geq f+1$, and let $r' \geq r$ be the smallest round for which there exists a tuple $(u', h') \neq (u, h)$ such that $|\mathsf{preendos}(\ell, r', u', h')| \geq f + 1$. As $|L^{u,h}_{\ell,r}| \geq f + 1$ and $|\mathsf{preendos}(\ell, r', u', h')| \geq f + 1$, there is at least one correct process p such that $p \in L^{u,h}_{\ell,r}$ and p preendorsed (u', h') at round r'. As $p \in L^{u,h}_{\ell,r}$, we have that p is locked on (u, h) at round r. Since p preendorsed (line 112) at round r', it means that one of the two disjunctions at line 111 holds. Note that the value of r_p at line 111 is r' in this case.

Suppose the first disjunction holds, i.e., $lockedValue_p = u'$. As a process can re-lock only in the phase ENDORSE, under the condition at line 117, this means that there is a round r'' with $r \leq r'' < r'$ and at which $|\mathsf{preendorsements}()| \geq 2f + 1$. Therefore $|\mathsf{preendos}(\ell, r'', u', h')| \geq f + 1$. This contradicts the minimality of r'.

Suppose now that the second disjunction holds, that is, $lockedRound_p < r'' < r'$ where the round r'' is the endorsable round of the proposer of u'. We note that a process cannot unlock (i.e. unset *lockedRound*), but only re-lock (i.e. set *lockedRound* to a different value). Therefore $lockedRound_p \geq r$ at round r' and from this, we obtain that $r'' > r > 0$. From the validity requirements of a propose message, we obtain that it contains a preendorsement QC for (u', h'). Thus we have that $|\mathsf{preendos}(\ell, r'', u', h')| \geq f + 1$. This contradicts the minimality of r', since $r'' < r'$. ◀

▶ **Lemma 9.** *No two correct processes have two different committed blocks at the same level in their blockchain.*

Proof. We reason by contradiction. Let ℓ be some level. Assume that two different correct processes p, p' have respectively two different committed blocks b, b' at level ℓ in their blockchain, with $b \neq b'$.

By the definition of committed blocks (Section 3), as b is a committed block at ℓ, the level of the head of p's blockchain is at least $\ell + 1$. Then, as p has a block at level $\ell + 1$ in his blockchain, p has observed an endorsement QC for $(\ell + 1, r, h, u)$ for some value u and some round r, where h is the hash of block b. Similarly, p' has observed an endorsement QC for $(\ell + 1, r', h', u')$ for some value u' and some round r', where h' is the hash of block b'. As $b \neq b'$, we have that $h \neq h'$, therefore $(u, h) \neq (u', h')$. We assume without loss of generality that $r \leq r'$. Since there are at most f Byzantine processes, and by Lemma 6 correct bakers can only endorse once per round, it follows that at least $f + 1$ correct bakers endorsed (u, h) during round r at level ℓ. Before broadcasting an endorsement for (u, h) at round r (line 120) any correct process sets its *lockedValue* to u and its *lockedRound* to r (line 119), thus $|L^{u,h}_{\ell,r}| \geq f + 1$. By Lemma 8, since $|L^{u,h}_{\ell,r}| \geq f + 1$, we also have $|\mathsf{preendos}(\ell, r'', u'', h'')| \leq f$, for any round $r'' \geq r$, and any value u'' with $(u'', h'') \neq (u, h)$. This means that a correct process cannot endorse some $(u'', h'') \neq (u, h)$ at a round $r'' \geq r$. This in turn means that there cannot be $2f + 1$ endorsements for $(u'', h'') \neq (u, h)$ with round $r'' \geq r$. This contradicts the fact that there is a QC for $(\ell + 1, r', u', h')$. ◀

▶ **Theorem 10.** *Tenderbake satisfies agreement.*

Proof. By contradiction, using Lemma 9. ◀

B.2 Progress

Let *Phases* be the set of labels PROPOSE, PREENDORSE, and ENDORSE. Let $S_p : \mathbb{N}^* \times \mathbb{N}^* \times Phases \to \mathbb{R}$ be the function such that $S_p(\ell, r, phase)$ gives the starting time of the phase *phase* of round r of process p at level ℓ. We consider that the function S_p returns the real time, not the local time of process p. Note that for different processes p and q, the function S_p and S_q may return different times for the same input, because p and q determine the starting time of their phases based on their local clocks, which may be different before τ.

We say that two correct processes p and p' are *synchronized* if $\ell_p = \ell_{p'}$, $|r_p - r_{p'}| \leq 1$, and $|S_p(\ell_q, r_q, phase_q) - S_{p'}(\ell_q, r_q, phase_q)| \leq 2\rho$, where $q \in \{p, p'\}$ is the process which is "ahead". We say that q is *ahead* of q' (or that q' is *behind* q) if $S_q(\ell_q, r_q, phase_q) \leq S_{q'}(\ell_q, r_q, phase_q)$. We say that p and q are *synchronized at level ℓ and round r* if p and q are synchronized and $\ell = \ell_p = \ell_q$ and $r = \max(r_p, r_q)$. At the beginning of r one of the processes might be at round $r - 1$. However, for at least $\Delta'(r) - 2\rho$ time, the two processes are at the same round.

Let t be the last time p called getNextPhase. We denote by $levelOffset_p = now - levelStart$, where *now* is the value returned by now when called by p at t, and *levelStart* is the sum at line 30. The next lemma states that we can use level offsets to characterize process synchronization. We omit its proof, which follows from an analysis of the synchronize and getNextPhase functions.

▶ **Lemma 11.** *After τ, two correct processes p and q are synchronized iff $|levelOffset_p - levelOffset_q| \leq 2\rho$.*

▶ **Lemma 12.** *Let p and q be two correct processes. If, after τ, they remain at the same level and the head of their blockchain has the same round, then they are eventually synchronized.*

Proof. Suppose that p and q are both at the same level ℓ and that their heads have the same round. p and q have already decided at $\ell - 1$. From the agreement property, p and q agree on the output value at level $\ell - 1$, thus they agree on all blocks up to level $\ell - 2$, and on their rounds as well. Thus, the block rounds in p's and q's blockchain are respectively the same.

Next, both p and q eventually call synchronize and getNextPhase. The round returned by synchronize is eventually larger than the current round of the process, so the process eventually exits the recursion at line 39 and calls getNextPhase.

Let p be the first to call getNextPhase and let t be the time of the call. Let $t' \geq t$ be the time when q first calls getNextPhase. We first note that *levelStart* in the definition of *levelOffset* is the same for both p and q, at both times t and t'. Let $levelOffset^*_t = t - levelStart$ and $levelOffset^*_{t'} = t' - levelStart$. We consider the values of the variable $levelOffset_p$ at t and t' and denote these by (simply) $levelOffset_p$ and $levelOffset'_p$, respectively. [4] Given the bound on clock skews, $|levelOffset_p - levelOffset^*_t| \leq \rho$ and $|levelOffset_q - levelOffset^*_{t'}| \leq \rho$. By using the inequality $|a-b| \leq |a|+|b|$, we obtain that $|levelOffset_q - levelOffset_p - (t'-t)| \leq 2\rho$, that is, $|levelOffset_q - levelOffset'_p| \leq 2\rho$. By Lemma 11, p and q are synchronized at t'. ◀

▶ **Lemma 13.** *If P is a set of correct processes that are synchronized after τ at a level ℓ and a round r with $\Delta'(r) > \delta + 2\rho$, and a process $p \in P$ sends a message at the beginning of its current phase ph, then this message is received by all processes in P by the end of their phase ph.*

[4] We note that $levelOffset'_p - levelOffset_p = t' - t$, because we assume that a process measures intervals of time precisely.

Proof. Assume that p sends its message m at time $t_p = S_p(\ell, r, ph)$. Consider a process $q \in P$, and let $t_q = S_q(\ell, r, ph)$. Process q receives m at most at time $t_p + \delta$. By the synchronization hypothesis, we have that $t_p - t_q \leq 2\rho$. Then we obtain that $t_p + \delta \leq t_q + 2\rho + \delta < t_q + \Delta'(r)$, as $t_q + \Delta'(r)$ is the time of the end of the phase ph for q. If $t_p < t_q$, then q might receive m while it is still at round $r - 1$. The message m is still available to q at round r because processes keep messages from a round one unit higher than their current round. ◀

▶ **Lemma 14.** *Let ℓ be a level and r a round with $\Delta'(r) > \delta + 2\rho$. Consider that all correct bakers are synchronized at level ℓ and round r at a time after τ. Let p be the proposer at round r. If p is correct and $endorsableRound_p \geq lockedRound_q$ for any correct baker q, then all correct bakers decide at level ℓ at the end of round r.*

Proof. From Lemma 13, we obtain that the Propose message of process p is received by all correct bakers by the beginning of their phase PREENDORSE. Let eR be the value of the endorsable round field of the Propose message. Note that $eR = endorsableRound_p$. We prove next that each correct baker sends the message Preendorse(ℓ, r, h, u), where u, h are the value and the predecessor hash proposed by p. Let q be a correct baker. If q is either unlocked or locked on u, then the condition in line 111 holds, and therefore q sends its preendorsement for (u, h). If q is locked on a value different from u then by hypothesis $lockedRound_q \leq endorsableRound_p$, therefore $lockedRound_q \leq eR$. Also, $endorsableRound_p < r$, since $endorsableRound_p$ is set during the execution of handleEvents before sending the Propose message in round r. Hence, $lockedRound_q \leq eR < r$. If $lockedRound_q = eR$ then, by quorum intersection, $lockedValue_q = u$ thus the first disjunction in line 111 holds for q. If $lockedRound_q < eR < r$ then the second disjunction in line 111 holds for q (note that $r = r_p = r_q$). Thus q sends the corresponding Preendorse message. So, we have proved that all correct bakers broadcast the Preendorse(ℓ, r, h, u) messages (line 112). By Lemma 13 all these Preendorse(ℓ, r, h, u) messages are received by all correct bakers by the beginning of the phase ENDORSE. Thus, for all of them, the condition in line 117 is true, thus all correct bakers broadcast the Endorse message for (u, h) (line 120). In the next phase, for all them, the quorum condition (line 126) holds for (u, h) so they decide (u, h). ◀

▶ **Lemma 15.** *If at some time after τ all correct bakers are synchronized at some level ℓ and round r with $\Delta'(r) > \delta + 2\rho$, then all correct bakers decide at level ℓ by the end of round $r + f + 1$.*

Proof. We first remark that, after τ, thanks to synchrony, a correct baker never skips a round, and in particular never skips its turn when it is time to propose. Let $p_0, p_1, \ldots$ be the sequence of bakers in the order in which they propose starting with round r. That is, p_i is the proposer at round $r + i$, for $i \geq 0$. Let j, k be the indexes of the first and second correct bakers in this sequence. As there are at most f Byzantine processes among $\{p_0, \ldots, p_k\} \setminus \{p_j\}$, we have $j < k \leq f + 1$. We show next that all correct bakers decide by the end of round $r + k$.

Suppose first that p_j is such that $endorsableRound_{p_j} \geq lockedRound_q$, for any correct baker q. By Lemma 14, all correct bakers decide at the end of round $r + j$.

Suppose that there is a correct baker with a locked round higher than $endorsableRound_{p_j}$. Let q be the baker with the highest locked round among all correct bakers. In the round at which p_j proposes, that is, in round $r + j$, q sends a preendorsement QC that justifies its locked round in the PREENDORSE phase (line 114). By Lemma 13, this preendorsement QC is received by all correct bakers, who update in the ENDORSE phase of round $r + j + 1$ their endorsable round to q's locked round at line 97. If between rounds $r + j + 1$ and $r + k - 1$ no correct baker updates its locked round then the proposer p_k will have at round $r + k$ that

$endorsableRound_{p_k} \geq lockedRound_q$, for any correct baker q. By Lemma 14, at the end of round $r+k$ all correct bakers decide. If instead there is a correct baker that updated its locked round before round $r+k$, then let q be the baker which updates it last, at some round $r+j'$ with $j' < k$. When q changes its locked round, q has seen a prendorsement QC for round $r+j'$. This QC is sent together with the Endorse message in the phase ENDORSE, and therefore it will be received by all correct bakers at the beginning of the next phase PROPOSE. Thus every correct baker, including p_k, sets its *endorsableRound* to $r+j'$. Because j' is maximal, no correct baker changes its locked round between rounds $r+j'+1$ and $r+k-1$. Therefore, at round $r+k$, for any baker q, we have that $lockedRound_q \leq r+j' = endorsableRound_{p_k}$. Again, by Lemma 14 we conclude that at the end of round $r+k$ all correct bakers decide. ◀

▶ **Theorem 16.** *Tenderbake satisfies progress.*

Proof. We reason by contradiction. Suppose first there is a level $\ell \geq 1$ such that no correct process decides at ℓ. Clearly, ℓ is minimal with this property. We first show that eventually all correct processes are synchronized. As ℓ is minimal, we have that there is at least one correct process that has decided at $\ell - 1$.

As processes invoke pullChain at regular intervals, all correct process will eventually be at level ℓ (that is, they will have decided at $\ell - 1$). We show next that all correct processes have the same blockchain head. Let p be a correct process that has its $headCertificate_p$ for the block with the lowest round at level $\ell - 1$. Process p eventually receives a pullChain request at some point after τ and it answers. If each correct process q has $endorsableRound_q = 0$ at the time of the receipt of p's answer, then every correct process accepts p's branch, by the definition of betterHead. Suppose however that there is a process q that has $endorsableRound_q > 0$ when it receives p's answer. In this case consider a time when round durations are so big that I and Δ are very small in comparison. More precisely, there is a time period when all pullChain requests and their answers happen during a period when correct processes update their states only in response to a NewChain event, but not in response to NewMessage events. Such a period exists because regular messages are sent only at phase boundaries. This means that the chain ending with the proposal with the highest endorsable round r will be seen by all correct processes, and these processes will have their endorsable round smaller or equal to r. They will update their blockchains to this chain (if they were on a different one). Note that if two processes have the same endorsable round then they also have the same blockchain. We have this obtained that eventually all correct processes have the same blockchain (head). We can therefore apply Lemma 12 to obtain that there is a time after τ at which all correct processes are synchronized.

Now, recall that the function Δ' has the property that there is a round r such that $\Delta'(r) > \delta + 2\rho$. As Δ' is increasing, this property holds for all subsequent rounds as well. And, given that all processes are synchronized from some time on, as proved in the previous paragraph, we obtain that the hypothesis of Lemma 15 is satisfied. Therefore all correct processes decide at ℓ, which contradicts the assumption that no correct process decides at ℓ. In other words, we have proved that, for any level ℓ, there is at least one correct process that decides at ℓ.

Finally, we show that for any level ℓ, any correct process eventually decides at ℓ. Suppose that there is a correct process p that does not decide at some level $\ell \geq 1$. From the first part of the proof we obtain that there is at least one other correct process q that eventually decides at ℓ. Process q will eventually receive p's pull request, will reply, and p will therefore receive an endorsement QC for level ℓ which enables it to decide at ℓ. This contradicts the assumption, and allows us to conclude. ◀

Byzantine-Tolerant Distributed Grow-Only Sets: Specification and Applications

Vicent Cholvi
Universitat Jaume I, Castelló, Spain

Antonio Fernández Anta
IMDEA Networks Institute, Madrid, Spain

Chryssis Georgiou
University of Cyprus, Nicosia, Cyprus

Nicolas Nicolaou
Algolysis Ltd, Lemesos, Cyprus

Michel Raynal
IRISA, Rennes, France
Polytechnic University of Hong Kong, Hong Kong

Antonio Russo
IMDEA Networks Institute, Madrid, Spain

Abstract

In order to formalize Distributed Ledger Technologies and their interconnections, a recent line of research work has formulated the notion of Distributed Ledger Object (DLO), which is a concurrent object that maintains a totally ordered sequence of records, abstracting blockchains and distributed ledgers. Through DLO, the Atomic Appends problem, intended as the need of a primitive able to append multiple records to distinct ledgers in an atomic way, is studied as a basic interconnection problem among ledgers.

In this work, we propose the *Distributed Grow-only Set object* (DSO), which instead of maintaining a sequence of records, as in a DLO, maintains a set of records in an immutable way: only Add and Get operations are provided. This object is inspired by the Grow-only Set (G-Set) data type which is part of the Conflict-free Replicated Data Types. We formally specify the object and we provide a consensus-free Byzantine-tolerant implementation that guarantees eventual consistency. We then use our Byzantine-tolerant DSO (BDSO) implementation to provide consensus-free algorithmic solutions to the Atomic Appends and Atomic Adds (the analogous problem of atomic appends applied on G-Sets) problems, as well as to construct consensus-free Single-Writer BDLOs. We believe that the BDSO has applications beyond the above-mentioned problems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Grow-only Sets, Distributed Ledgers, Blockchains, Atomic appends

Digital Object Identifier 10.4230/OASIcs.FAB.2021.2

Related Version *Full Version*: `https://arxiv.org/abs/2103.08936`

Funding Partially supported by French ANR project ByBLoS (ANR-20-CE25-0002-01), Government of Madrid (CM) grant EdgeData-CM (P2018/TCS4499, cofunded by FSE & FEDER), and Spanish Ministry of Science and Innovation grant ECID (PID2019-109805RB-I00, cofunded by FEDER).

1 Introduction

Blockchains (as termed by Nakamoto in [18]) or Distributed Ledger Technologies (DLTs) (as used in [10] and [20]) became one of the most trendy data structures following the introduction of crypto-currencies [18] and their recent application in finance and token-economy. Despite their early wide adoption, little was known initially about the fundamental construction and

semantic properties of DLTs. A number of research groups attempted to provide rigorous definitions to characterise the fundamental properties of DLTs as those used in Bitcoin and beyond [1, 10, 11]. Among those, Fernández Anta et al. [10], was the first to identify and provide a formal definition of a reliable concurrent object, termed *Distributed Ledger Object* (DLO), which conveys the essential building block for many DLTs. In particular, a DLO maintains a sequence of records, and supports two basic operations: append and get. The append operation is used to add a new record at the end of the sequence, while the get operation returns the whole sequence. Implementations of DLOs under client and server crashes were proposed in [10], and under Byzantine failures in [6].

The introduction to many different DLT systems have led multiple studies [6, 9, 14, 16] to investigate the possibility of DLT interoperability, i.e., the ability for an action to be applied over a set of DLTs, rather than in a single DLT at a time. Using the DLO formalism, [9] introduced the *Atomic Appends problem*, in which several clients have a "composite" record (a set of semantically-linked "basic" records) to append. Each basic record has to be appended to a different DLO, and it must be guaranteed that either all basic records are appended to their DLOs or none of them is appended.

Consider, for example, two clients A and B, where A buys a car from B. Record r_A includes the transfer of the car's digital deed from B to A, and r_B includes the transfer from A to B of the agreed amount in some digital currency. DLO_A is a ledger maintaining digital deeds and DLO_B maintains transactions in some pre-agreed digital currency. So, while the two records are mutually dependent, they concern different DLOs. Hence, the Atomic Appends problem requires that *either* record r_A is appended in DLO_A and record r_B is appended in DLO_B, *or* no record is appended in the corresponding DLOs.

In the work presented in [9], the authors assumed that clients may fail by crashing and showed that for some cases the existence of an intermediary is necessary. They materialized such an intermediary by implementing a specialized DLT, termed *Smart* DLO (SDLO). Using the SDLO, the authors solved the Atomic Appends problem in a client competitive asynchronous environment, in which any number of clients, and up to f servers implementing the DLOs, may crash. A subsequent work solved the problem assuming Byzantine failures [6], by introducing the notion of *Byzantine Distributed Ledger Objects* (BDLO). Solutions for implementing BDLOs were presented, with each solution relying on an underlying Byzantine Total-order Broadcast Service (BToB) [7, 8, 17]. Using BToB and an intermediary SBDLO the authors demonstrated how Atomic Appends may be achieved in systems that suffer Byzantine failures. However, BToB is a strong primitive, and requires consensus to be solved. So one may ask: *Is it possible to implement Atomic Appends without solving consensus?*

It was shown in [13] that cryptocurrencies do not need consensus to be implemented. From a theoretical point of view, it was shown in [12] that, assuming one process per account, the consensus number of cryptocurrencies is 1. A non-sequential specification of money transfer was introduced in [2]. It follows that Byzantine transactional systems do not necessarily need consensus, but rather can be implemented on top of less powerful data structures. In a similar manner, in this work, we observe that intermediary S(B)DLOs and strong primitives like BToB [17], may not be necessary to allow interoperability between multiple DLOs. Note that the goal of the intermediate S(B)DLO is to collect the records to be appended atomically, so that when all the records involved are in the S(B)DLO, then the actual records are appended in their respective DLOs. It is apparent that, for *Atomic Appends*, the order of the records in the intermediary data structure is not important, but rather the membership property required redirects to a *set* data structure.

A relevant distributed set data structure was presented by Shapiro et al. in [21] with the introduction of Conflict-Free Replicated Data Types (CRDTs). A CRDT is a data structure that can be replicated in multiple network locations. CRDTs have the property that each

replica can be updated independently and concurrently, but it is always mathematically possible to resolve any inconsistencies between any pair of replicas, leading eventually all the replicas to a consistent converged value when the communication between the replica hosts is stabilized. A *Grow-Only Set* (G-Set) is such a CRDT, that supports operations add and lookup only. The add operation modifies the local state of the object by a union of the value of the set with the element we want to insert. Since add is based on union, and union is commutative, the G-Set implementation converges. In [21] (and other subsequent works), implementations of G-Sets where given in a crash-prone environment. In order to utilise a G-Set in more practical setups (like the ones in cryptocurrencies) we need to examine whether such data structure is possible when Byzantine failures are present in the system.

Chai and Zhao [5] have considered the implementation of CRDTs against Byzantine failures. In particular, they describe possible threats that clients and servers can either face or cause to CRDTs, and they show a possible solution to fulfil CRDT requirements in that failure model. Their solution relies on an external synchronization service for two main purposes: to guarantee linearizable *reads* and *writes*, and to prevent server partitions caused by Byzantine behaviour. As a consequence, multiple Byzantine failures or slow processes may lead their approach to essentially always run their "state synchronization" mechanism letting the whole data structure rely on the synchronisation service. For the implementation of the synchronisation service they either utilize a central entity, or solve consensus over a distributed set of nodes.

Contributions. In this work we examine whether G-Sets can be implemented when Byzantine processes are assumed in the system, without using consensus. We show that an implementation of an eventually consistent [22] G-Set is possible, and we demonstrate how such data structure can be used to solve Atomic Appends and other related problems. In particular, our itemized contributions are the following:

- Provide a formal definition of a Byzantine Grow-only Set Object (BDSO). [Section 2]
- Provide an implementation for an eventually consistent BDSO in an asynchronous message passing system[1]. We consider such a consistency model since, although it provides weaker guarantees than other consistency models, it is easier and more efficient to implement, while being powerful enough to be used in the type of applications we consider (described next). [Section 3]
- Use BDSOs to implement:
  - Consensus-free Byzantine Atomic Appends. [Section 4.1]
  - Consensus-free Byzantine *Atomic Adds*. This is the analogous problem of atomic appends where records must be added in an atomic way to different BDSOs. This problem could be applicable in blockchain-like systems in which the ordering of the records is not important; what is important is that the records are added in the corresponding unordered blockchains (G-Sets). An example could be a system of G-Sets that implement personal calendars, so the records in the sets are meetings. Then, fixing a two-person meeting would imply an Atomic Add of the meeting data in the calendar of both persons. [Section 4.2]
  - Consensus-free single-writer BDLOs. This data structure can be suitable to implement whatever system that requires total order among data produced by a single writer. A punch in/out system for a company is an example of such an application in which a single writer, the employee, appends records only to his/her own ledger of presences.

[1] Note that in such a system deterministic consensus can't be solved.

> A cryptocurrency can be another suitable application, with one BDLO per account, because of the need to order transactions in relation to money transfers issued by the only transaction signer. [Section 4.3]

2 The G-Set Object

In this section we provide the fundamental definition of a concurrent G-Set object.

2.1 Concurrent Objects and the G-Set Object

An *object type* T specifies (i) the set of *values* (or states) that any object O of type T can take, and (ii) the set of *operations* that a process can use to modify or access the value of O. An object O of type T is a *concurrent object* if it is a shared object accessed by multiple processes [15, 19]. Each operation on an object O consists of an *invocation* event and its unique matching *response* event, that must occur in this order. A *history* of operations on O, denoted by H_O, is the sequence of invocation and response events, starting with an invocation event. (The sequence order of a history reflects the real time ordering of the events.) We say that a history H'_O *extends* a history H_O, if H_O is a prefix of H'_O.

An operation π is *complete* in a history H_O, if H_O contains both the invocation and the matching response. A history H_O is *complete* if it contains only complete operations; otherwise it is *partial* [15, 19]. An operation π *precedes* an operation π' (or π' *succeeds* π), denoted by $\pi \rightarrow \pi'$, in H_O, if the response event of π appears before the invocation event of π' in H_O. Two operations are *concurrent* if none precedes the other. A complete history H_O is *sequential* if it contains no concurrent operations, i.e., it is an alternative sequence of matching invocation and response events, starting with an invocation and ending with a response event. A partial history is sequential, if removing its last event (that must be an invocation) makes it a complete sequential history.

A *sequential specification* of an object O, describes the behavior of O when accessed sequentially. In particular, the sequential specification of O is the set of all possible sequential histories involving solely object O [19].

A *G-Set* $\mathcal{GS}$ is a concurrent object that maintains a set $\mathcal{GS}.S$ of *records* and supports two operations (available to any process p): (i) $\mathcal{GS}.\mathsf{get}_p()$, and (ii) $\mathcal{GS}.\mathsf{add}_p(r)$. A *record* is any value drawn from an alphabet A. A process p invokes a $\mathcal{GS}.\mathsf{get}_p()$ operation to obtain the set $\mathcal{GS}.S$ of records stored in the G-Set object $\mathcal{GS}$ [2], and p invokes a $\mathcal{GS}.\mathsf{add}_p(r)$ operation to insert a new record r in $\mathcal{GS}.S$. Initially, the set $\mathcal{GS}.S$ is empty. Deleting or changing a record from $\mathcal{GS}.S$ is not possible, as our objective is for the set to be immutable with respect to record modifications of any kind.

▶ **Definition 1.** *The* sequential specification *of a G-Set $\mathcal{GS}$ over the sequential history $H_{\mathcal{GS}}$ is defined as follows. Let the initial value of $\mathcal{GS}.S = \emptyset$. If at the invocation event of an operation π in $H_{\mathcal{GS}}$ the value of the set $\mathcal{GS}.S = V$, then:*

1. *if π is a $\mathcal{GS}.\mathsf{get}_p()$ operation, then the response event of π returns V, and*
2. *if π is a $\mathcal{GS}.\mathsf{add}_p(r)$ operation, then at the response event of π, the value of the set in G-Set $\mathcal{GS}$ is $\mathcal{GS}.S = V \cup \{r\}$.*

By comparing the sequential specification of a G-Set, as defined above, with the sequential specification of a Ledger Object as defined in [10, Definition 1] (also see Appendix A), it follows that a Ledger is an *ordered* G-Set.

[2] We define only one operation to access the value of the G-Set for simplicity. In practice, other operations will also be available, like *lookup(r)* to check if a record r is in $\mathcal{GS}.S$.

2.2 Distributed G-Set Objects

We now define distributed G-Set objects, DSO for short, and the class of eventually consistent DSOs. These definitions are general and do not rely on the properties of the underlying distributed system, nor on the type of failures that may occur.

A **distributed G-Set object** (DSO) is a concurrent G-Set object that is implemented in a distributed manner. In particular, a DSO is *implemented* by a set of (possibly distinct and geographically dispersed) computing devices, that we refer as *servers*. Each server usually maintains a local copy (replica) of the DSO. We refer to the processes that invoke the get and add operations of the distributed G-Set as *clients*.

Distribution and replication intend to ensure availability and survivability of the G-Set, in case a subset of the servers fails (by crashing or acting maliciously). At the same time, they raise the challenge of maintaining *consistency* among the different views that different clients get of the DSO[3]. Consistency semantics need to be in place to precisely describe the allowed values that a get operation may return when it is executed concurrently with other get or add operations.

We now specify the properties of DSO with respect to *eventual consistency* [22]. These properties require that if an add(r) operation completes, then *eventually* all get() operations return sets that contain record r. In a similar way, other consistency guarantees such as sequential, session, causal and atomic consistencies could be formally defined.

▶ **Definition 2.** *A DSO $\mathcal{GS}$ is* **eventually consistent** *if, given any history $H_{\mathcal{GS}}$,*

(a) *EC-Safety: let S be the set of records returned by any complete operation $\pi = \mathsf{get}() \in H_{\mathcal{GS}}$. For each $r \in S$, there is an operation $\mathsf{add}(r)$ whose invocation event appears before the response event of π in $H_{\mathcal{GS}}$, and*

(b) *EC-Liveness: for every complete operation $\mathcal{GS}.\mathsf{add}(r) \in H_{\mathcal{GS}}$, there exists a history $H'_{\mathcal{GS}}$ that extends $H_{\mathcal{GS}}$ such that, for every history $H''_{\mathcal{GS}}$ that extends $H'_{\mathcal{GS}}$, every complete operation $\mathcal{GS}.\mathsf{get}()$ in $H''_{\mathcal{GS}} \setminus H'_{\mathcal{GS}}$ returns a set that contains r.*

At this point, we would like to remark that, although eventual consistency provides weaker consistency guarantees when compared, for example, with linearizability [15], it is easier and more efficient to implement, while it is powerful enough to be used in the type of applications that we later consider (see Section 4).

2.3 Distributed Setting and Byzantine-tolerant DSO

We consider a distributed setting consisting of processes (clients and servers) and an underlying communication graph in which each process can communicate with every other process.

Asynchrony. Both processing and communication are asynchronous. Therefore, each process proceeds at its own speed, which can vary arbitrarily and remains always unknown to the other processes. Message transfer delays are arbitrary but finite and remain always unknown to the processes.

Failure Model. Processes (clients and servers) can fail arbitrarily, i.e., they can be Byzantine. Specifically, we assume a *Byzantine system* in which the number of servers that can arbitrarily fail is bounded by f, and in which the total number of servers, n, is at least $3f + 1$. For clients we assume that any of them can be Byzantine. We assume reliable channels between non-Byzantine (correct) processes. Specifically, no message is lost, duplicated or modified.

[3] This tradeoff is actually captured by the well-known CAP Theorem [4].

Public and private keys. We assume that each process p (client or server) has a pair of public and private keys, and that the public keys have been distributed reliably to all the processes that may interact with each other. Hence, we discard the possibility of spurious or fake processes (there cannot be Sybil attacks). We also assume that messages sent by any process (server or client) are authenticated, so that messages corrupted or fabricated by Byzantine processes are detected and discarded by correct processes [8]. Communication channels between correct processes are reliable but asynchronous.

Byzantine-tolerant DSOs. Our first aim is to propose an algorithm that implement an eventual-consistent DSO $\mathcal{GS}$ in a Byzantine asynchronous system. Here we present the properties that a DSO should satisfy with respect to *correct processes*, given that Byzantine processes may return any arbitrary set or add any arbitrary record:

- *Byzantine Completeness (BC)*: All the get() and add() operations invoked by correct clients eventually complete.
- *Byzantine Eventual Consistency (BEC)*: This is the property of Definition 2 with respect to all operations invoked by correct clients and the add(r) operations that insert the records r returned by get() operations invoked by correct clients.

In the remainder, we say that a DSO is *Byzantine Tolerant*, denoted BDSO, and eventually consistent if it satisfies properties BC and BEC.

Byzantine Reliable Broadcast. The algorithms presented in the next section to implement BDSOs are based on an underlying Byzantine Reliable Broadcast (BRB) service [3, 20], which ensures that a message sent by a correct process is received by all correct processes, and that all correct processes eventually receive the same set of messages. The service provides two operations, BRB-broadcast and BRB-delivery; the first broadcasts a message to all processes, and the second delivers a message that was previously broadcast. The service is used by the servers, and from their point of view, the BRB service guarantees the following properties (as given in [20]):

- *Validity*: if a correct process p_i BRB-delivers a message m from a correct process p_j, then p_j BRB-broadcast m.
- *Integrity*: a message is BRB-delivered at most once by a correct server.
- *Termination 1 (local)*: if a correct process BRB-broadcasts a message, it BRB-delivers it.
- *Termination 2 (global)*: if a correct process BRB-delivers a message, all correct processes BRB-deliver it.

Validity relates outputs to inputs. Validity and integrity concern safety. Termination is on the fact that messages must be BRB-delivered; it concerns liveness. It follows (cf. [20]) that all correct processes BRB-deliver the same set of messages, which includes all the messages they BRB-broadcast.

3 Eventually Consistent BDSO Implementation

In this section we provide the implementation of eventually consistent distributed G-Sets in an asynchronous distributed system with Byzantine failures. The implementation builds on a generic deterministic Byzantine-tolerant reliable broadcast service [3, 20], which provides the properties given in the previous section. Our implementation is *optimally resilient*, in the sense that it can tolerate up to f Byzantine servers, out of $n \geq 3f + 1$ servers.

Algorithm 1 presents the code of a client process, while Algorithm 2 presents the code of a server. We now present a high level description of how the two algorithms together implement an eventually consistent BDSO.

Algorithm 1 Client API and algorithm for Eventually Consistent Byzantine-tolerant Distributed G-Set Object $\mathcal{GS}$. Code for Client p.

```
1:  Init: c ← 0
2:  function GS.get( )                                          ▷ Invocation event
3:      c ← c + 1
4:      send request GET(c, p) to 3f + 1 different servers
5:      wait responses GETRESP(c, i, S_i) from 2f + 1 different servers
6:      S ← {r : record r is in at least f + 1 sets S_i}
7:      return S                                                ▷ Response event
8:  function GS.add(r)                                          ▷ Invocation event
9:      c ← c + 1
10:     send request ADD(c, p, r) to 2f + 1 different servers
11:     wait responses ADDRESP(c, i, ACK) from f + 1 different servers
12:     return ACK                                              ▷ Response event
```

Algorithm 2 Server algorithm for Eventually Consistent Byzantine-tolerant Distributed G-Set Object. Code for Server i.

```
1:  Init: S_i ← ∅
2:  receive (GET(c, p)) from process p                          ▷ Signature of p is validated
3:      send response GETRESP(c, i, S_i) to p
4:  receive (ADD(c, p, r)) from process p                       ▷ Signature of p is validated
5:      if (r ∉ S_i) then
6:          BRB-broadcast(PROPAGATE(i, ADD(c, p, r)))
7:          wait until r ∈ S_i
8:      send response ADDRESP(c, i, ACK) to p
9:  upon (BRB-deliver(PROPAGATE(j, ADD(c, p, r)))) do           ▷ Signatures of j and p are validated
10:     if (r ∉ S_i) and (ADD(c, p, r) was received from f + 1 different servers j) then
11:         S_i ← S_i ∪ {r}
```

- When processing a $\mathcal{GS}$.add(r) operation a client sends ADD messages to a set of $2f + 1$ servers, which guarantees that at least $f + 1$ correct servers process it. These correct servers broadcast the record r to all servers using the BRB service, which leads to all correct servers i adding r to their replicas S_i of the set. When $f + 1$ acknowledgement messages are received from the servers, the operation completes.
- When processing a $\mathcal{GS}$.get() operation, a client need to ensure that the elements he returns have been received from at least 1 correct server. For this reason the client returns an element only if it was present in responses from $f+1$ different server. In order to avoid the malicious behavior of f colluding servers that never return a correctly added element, at least $2f + 1$ responses are needed out of which take the $f + 1$ consistent GETRESP containing the element. So, since $2f + 1$ are required, at least $3f + 1$ GET messages must be sent in order to always eventually get the number of needed responses.
- Every server i maintains a replica S_i of the set $\mathcal{GS}.S$. When server i receives a GET(c, p) message from a process p it returns its current set S_i to p. When i receives a message ADD(c, p, r) from p, it makes sure r has been included in its replica S_i before sending an acknowledgment. Server i adds a record r to its replica S_i only if a corresponding add request has been processed by at least one correct server. This is guaranteed by the BRB service and the requirement of receiving PROPAGATE(j, ADD(c, p, r)) from $f + 1$ different servers. This also prevents Byzantine servers from adding spurious records in the set of correct servers. The properties of the BRB service also guarantee that once a record r is delivered, then all correct servers will eventually add record r to their replicas.

We now provide the complete proof that the combination of Algorithms 1 and 2 implement an eventually consistent BDSO. In the proofs we consider that an operation π is invoked in Lines 2 or 8 of Algorithm 1, and responds in Lines 7 or 12 (resp.) of the same algorithm. Let us first show that Byzantine Completeness holds, i.e., that all operations invoked by correct processes eventually complete.

▶ **Lemma 3.** *Algorithms 1 and 2 guarantee Byzantine Completeness (BC) in a system in which at most f out of $n \geq 3f + 1$ servers are Byzantine.*

Proof. Consider an operation $\mathcal{GS}.\mathsf{get}_p()$ invoked by a correct client p. We claim that the operation eventually completes. From Algorithm 1, Line 4, p sends a request GET(c, p) to $3f + 1$ servers and waits for responses GETRESP(c, i, S_i) from $2f + 1$ different servers. From the $3f + 1$ servers to which the request is sent, at most f can be Byzantine, so at least $2f + 1$ are correct servers that will eventually receive the GET(c, p) message. These servers will immediately send the corresponding response GETRESP(c, i, S_i) to p (Line 3 of Algorithm 2). When these responses are received eventually, the waiting in Line 5 of Algorithm 1 will end. Since there is no other waiting condition, the operation will execute the return instruction and complete.

Consider now an operation $\pi = \mathcal{GS}.\mathsf{add}_p(r)$ invoked by a correct client p. Then, the request ADD(c, p, r) is sent to $2f + 1$ servers (Algorithm 1, Line 10), and p waits until responses ADDRESP(c, i, ACK) are received from $f + 1$ different servers. Since at most f servers can be Byzantine, at least $f + 1$ correct servers will receive and process the request. We prove that all these correct servers will send the corresponding response, the waiting in Line 11 will end, and operation π will complete.

Let us consider the set C of correct servers that receive request ADD(c, p, r). Assume first that there is some server $i \in C$ that has $r \in S_i$ when the request is received and processed. Then, server i sends immediately response ADDRESP(c, i, ACK) to p. Moreover, r was inserted in S_i in Line 11 of Algorithm 2, which implies that i received via BRB-deliver at least $f + 1$ messages PROPAGATE() from different servers containing ADD(c, p, r) requests. From the *Termination 2* property of the BRB service, all correct processes will receive the same $f + 1$ messages PROPAGATE(). Consider any other correct server $j \in C$ that receives request ADD(c, p, r). If $r \in S_j$ when the request is received and processed, server j sends the response ADDRESP(c, j, ACK) to p immediately. Otherwise, $r \notin S_j$ when the request is received and processed, and j waits in Line 7. From the above argument, eventually r will be inserted in S_j, the waiting will end, and j will send response ADDRESP(c, j, ACK) to p.

Assume now that no correct server $i \in C$ has $r \in S_i$ when it receives request ADD(c, p, r). Then, all the (at least $f + 1$) correct servers in C that receive and process the request invoke BRB-broadcast(PROPAGATE(i, ADD(c, p, r))) and start waiting in Line 7. From the *Termination 1* property of the BRB-service, if a correct server BRB-broadcasts a message, it also eventually BRB-delivers it. Moreover, from *Termination 2*, if it BRB-delivers the message, all correct servers also BRB-deliver it. So each correct server $i \in C$ will process in Lines 9-11 messages PROPAGATE(j, ADD(c, p, r)) from at least $f + 1$ different servers j. Hence, server i will insert r in S_i in Line 11, the waiting will end, and i will send response ADDRESP(c, i, ACK) to p. ◀

▶ **Theorem 4.** *Algorithms 1 and 2 implement an Eventually Consistent* BDSO*, in a system in which at most f out of $n \geq 3f + 1$ servers are Byzantine.*

Proof. We need to prove that Algorithms 1 and 2 guarantee Byzantine Completeness (BC) and Byzantine Eventual Consistency (BEC). BC is shown to be satisfied in Lemma 3. Regarding Byzantine Eventual Consistency, we need to demonstrate properties (a) and (b)

of Definition 2 with respect to all the operations invoked by correct clients and the $\mathsf{add}(r)$ operations that insert the records r returned in the $\mathsf{get}()$ operations invoked by correct clients. Let $H_{\mathcal{GS}}$ be any history including only invocation and response events of these operations.

Property (a): Consider a complete operation $\pi = \mathsf{get}_p() \in H_{\mathcal{GS}}$ invoked by a correct client p, let S be the set returned by π, and consider any $r \in S$. From Line 6 of Algorithm 1, r belongs to at least $f+1$ sets S_i received in responses GETRESP(c, i, S_i) from a set C of different servers. All these responses must have been sent before the response event of π (Line 7 of Algorithm 1).

Observe that C contains at least one correct server i. This mean that some correct server i had $r \in S_i$ when it sent the response GETRESP(c, i, S_i). A server i only adds a record to its local set S_i if that record was BRB-delivered in PROPAGATE$(j,$ ADD$(c', p', r))$ from $f+1$ different servers j (Line 10 of Algorithm 2). From the Validity property of the BRB service, this means that at least $f+1$ servers called $\mathsf{BRB\text{-}broadcast}$(PROPAGATE$(j,$ ADD$(c', p', r))$ in Line 6. Again, since at least one of them is correct, at least one invocation of BRB-broadcast was done by a process because it previously received a request ADD(c', p', r) from client p'. Hence the invocation of $\mathsf{add}(r)$ must have preceded the reception of this request, and by transitivity must have preceded the response event of π.

Property (b): This property holds if, for every complete operation $\mathcal{GS}.\mathsf{add}(r) \in H_{\mathcal{GS}}$, there exists a time t after which every $\mathcal{GS}.\mathsf{get}()$ operation invoked after t returns sets S that contains r. Let us first consider a complete operation $\pi = \mathcal{GS}.\mathsf{add}_p(r) \in H_{\mathcal{GS}}$ invoked by a client p (which can be correct or Byzantine). We claim that there is some correct server i that eventually adds record r to its replica S_i. This is true when p is Byzantine, since that is the requirement for an $\mathsf{add}(r)$ operation of a Byzantine client to be considered.

On the other hand, if p is correct, let us assume for contradiction that no correct server i adds record r to its replica S_i. Process p sends request ADD(c, p, r) to $2f+1$ servers, out which at least $f+1$ are correct. By assumption, $r \notin S_j$ when each of these servers j processes the request, and hence all of them execute $\mathsf{BRB\text{-}broadcast}$(PROPAGATE$(j,$ ADD$(c, p, r)))$ (Line 6 of Algorithm 2). Then, from the *Termination 1* and *Termination 2* properties of the BRB service, some correct server i will BRB-deliver at least $f+1$ messages PROPAGATE$(j,$ ADD$(c, p, r))$ from different servers j, and then record r will be added to S_i in Line 11. This is a contradiction, and some correct server i eventually adds record r to its replica S_i when client p is correct.

Hence, we have that, independently of whether p is correct, some correct server i added record r to its set S_i. Observe that a correct process i only adds records to its replica S_i, in Line 11, when BRB-deliver at least $f+1$ messages PROPAGATE$(j,$ ADD$(c, p, r))$ from different servers j. Then, if i adds r to S_i, from the *Termination 2* property all correct servers will eventually BRB-deliver at least $f+1$ messages PROPAGATE$(j,$ ADD$(c, p, r))$ from different servers j, and they will all add r to their replicas.

Let t be the first time all correct servers have r in their corresponding replica. Then, for every $\mathcal{GS}.\mathsf{get}()$ operation invoked after t, the responses from correct servers collected in Line 5 of Algorithm 1 have replicas S_i with record r. Since there at least $f+1$ responses from correct servers, in Line 6 r is included in the set S, which is then returned by $\mathcal{GS}.\mathsf{get}()$. ◀

4 Applications of BDSOs

In this section we demonstrate the usability of BDSOs by using them to provide consensus-free solutions to the Atomic Appends and Atomic Adds problems, as well as a consensus-free construction of a Single-Writer Byzantine-tolerant Distributed Ledger Object (BDLO).

4.1 The Atomic Appends Problem

The *Atomic Appends* problem was introduced in [10] as a basic interconnection problem among distributed ledgers (DLOs); see Appendix A for basic definitions with respect to DLOs. Informally, Atomic Appends requires that several records must be appended in their corresponding DLOs, so that either *all* records are appended (each in the appropriate DLO) or *none* is appended to any DLO. In [6], the problem was formulated (and solved) in the presence of Byzantine servers and clients.

Definition of the problem. For completeness, we provide the formal definition as given in [6]. A record r *depends* on a record r' if r may be appended on its intended BDLO, say $\mathcal{L}$, only if r' is appended on its intended BDLO, say $\mathcal{L}'$. Two records, r and r' are *mutually dependent* if r depends on r' and r' depends on r.

▶ **Definition 5** (2-AtomicAppends [6]). *Consider two clients, p and q, with mutually dependent records r_p and r_q. We say that records r_p and r_q are* appended atomically *in BDLO $\mathcal{L}_p$ and BDLO $\mathcal{L}_q$, respectively, when:*

- *AA-safety (AAS):* The record r_p of a correct client p is appended in $\mathcal{L}_p$ only if the record of the other client q (which may be correct or not) is also appended in $\mathcal{L}_q$.
- *AA-liveness (AAL):* If both p and q are correct, then both records are appended eventually.

Observe that it is not possible to prevent a faulty client q from appending its record r_q, even if the correct client p does not append its record. What the safety property AAS guarantees is that the opposite cannot happen. This is analogous of the property in atomic cross-chain swaps [14] that a correct process cannot end up worse than at the beginning.

We say that an algorithm *solves* the 2-AtomicAppends problem[4] under a given system, if it guarantees properties AAS and AAL of Definition 5 in every execution. Since we consider Byzantine failures, our system model with respect to the Atomic Appends problem is such that the correct processes want to proceed with the append of the records (to guarantee liveness AAL), while the Byzantine processes may try to get correct clients to append without the Byzantine clients doing so (to prevent safety AAS).

Prior solution. The solution of 2-AtomicAppends in [6], following the work in [10], uses an auxiliary, special purpose BDLO, called Smart BDLO (SBDLO) to aggregate and coordinate the append of multiple records. In a nutshell, the solution in [6] is as follows. Consider two clients, p and q, that wish to append atomically two mutually dependent records, r_p and r_q, in BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively. Then, they both send matching *atomic append requests* to the SBDLO. Once both requests are received by the SBDLO (otherwise the atomic append never takes place), the servers implementing the SBDLO proceed to append each record to the appropriate BDLOs. In particular, the servers of the SBDLO now become clients issuing the corresponding appends to the servers implementing the DBLOs $\mathcal{L}_p$ and $\mathcal{L}_q$ (each BDLO could be implemented by different servers, as these are essentially different distributed ledger systems). The whole process involves several algorithms: the algorithm run by the clients to issue the atomic append request, the algorithm run by servers to implement the SBDLO, and the algorithm run by the servers of the SBDLO (as clients) with the servers of each individual BDLO. Once both append operations are completed, the SBDLO servers acknowledge this to clients p and q. It is shown that the combination of these algorithms guarantee Properties AAS and AAL above, despite having Byzantine servers and clients.

[4] The *k-AtomicAppends* problem, for $k \geq 2$, is a generalization of the 2-AtomicAppends that can be defined in the natural way: k clients, with k mutually dependent records, to be appended to k BDLOs. To keep the presentation simple, we focus in the case of $k = 2$.

Algorithm 3 API for the 2-AtomicAppend of records r_p and r_q in ledgers $\mathcal{L}_p$ and $\mathcal{L}_q$ by clients p and q, respectively, using SBDSO $\mathcal{GS}$. Code for Client p.

```
1: function AtomicAppends(p, {p, q}, r_p, L_p, r_q)
2:     GS.add(⟨p, {p, q}, r_p, L_p, r_q⟩)
3:     return ACK
4: // Client p will know the Atomic Appends operation was completed successfully when it receives
   notifications from f + 1 different SBDSO servers. //
```

Algorithm 4 Server algorithm for Smart Byzantine-tolerant DSO. Code for Server i.

```
1: Init: S_i ← ∅
2: receive (GET(c, p)) from process p                          ▷ Signature of p is validated
3:     send response GETRESP(c, i, S_i) to p
4: receive (ADD(c, p, r)) from process p                       ▷ Signature of p is validated
5:     if (r ∉ S_i) then
6:         BRB-broadcast(PROPAGATE(i, ADD(c, p, r)))
7:         wait until r ∈ S_i
8:     send response ADDRESP(c, i, ACK) to p
9: upon (BRB-deliver(PROPAGATE(j, ADD(c, p, r)))) do     ▷ Signatures of j and p are validated
10:    if (r ∉ S_i) and (ADD(c, p, r) was received from f + 1 different servers j) then
11:        S_i ← S_i ∪ {r}
12:        if (r.v = ⟨p, {p, q}, r_p, L_p, r_q⟩) and
13:         (∃r' ∈ S_i : r'.v = ⟨q, {p, q}, r_q, L_q, r_p⟩) then
14:            L_p.append(r_p); L_q.append(r_q)
15:            Notify clients p and q that records r_p and r_q have been appended to L_p and L_q
```

Our approach. In this work we treat the part of the individual BDLOs ($\mathcal{L}_p$ and $\mathcal{L}_q$) implementations as black boxes and we focus on the auxiliary entity that is used for coordinating the atomic append requests. In [6], the SBDLO, being a Distributed Ledger object, required the use of a Byzantine Total-order Broadcast [17] service. It was shown in [10] that consensus is required for implementing a (B)DLO; this is because of the strong prefix property of (B)DLOs (see Appendix A), which requires that records must be totally ordered. Hence, atomic appends was solved using consensus to implement the SBDLO. However, one can notice that in the auxiliary entity, the atomic append requests do not need to be totally ordered. It is sufficient to only keep track whether both requests have been made. In other words, *why keeping these requests in a sequence, and not in a set?*

In this respect, we show that instead of using a special purpose BDLO as the auxiliary entity, we can simply use a special purpose eventually consistent BDSO, which we will be referring as SBDSO. As we have seen in Section 3, eventually consistent BDSOs can be implemented without consensus (instead of a Byzantine total-order broadcast service, we use only a Byzantine reliable broadcast service), yielding a *consensus-free solution to Atomic Appends* (with respect to the actual atomic append requests).

Our solution. Algorithm 3 specifies how processes p and q delegate the task of appending their records in the respective ledgers. They do so by adding in the SBDSO a description of the Atomic Appends operation to be completed. Client p uses the $\mathcal{GS}$.add operation to provide the SBDSO with the data it requires to complete the Atomic Appends, namely the participants in the Atomic Appends, the record r_p, the BDLO $\mathcal{L}_p$, and the record r_q the other client is appending. (The other client must do the same.)

For the SBDSO, it suffices to implement an eventually consistent BDSO in which up to f servers out of $n \geq 3f + 1$ are Byzantine, but that *only allows the creator of a record to add it* (signatures are used for this purpose). Algorithm 4 describes the processing of the ADD message by the SBDSO. As expected, it is very similar to the implementation of a BDSO, but with an important difference: every time a record r is added to the sequence S_i, it is checked whether a matching record r' is already there. This is the case if $r.v = \langle p, \{p, q\}, r_p, \mathcal{L}_p, r_q \rangle$, and $r'.v = \langle q, \{p, q\}, r_q, \mathcal{L}_q, r_p \rangle$. If so, the corresponding append operations are issued in the respective BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$ (the implementation of this part is the one described in [6]). So, essentially the servers implementing the SBDSO, become proxies of clients p and q, and once the above condition is met, they issue the corresponding appends. When these appends are successful, the servers implementing the ledgers $\mathcal{L}_p$ and $\mathcal{L}_q$, acknowledge the SBDSO servers. In turn, the SBDSO servers notify clients p and q that records r_p and r_q have been appended to $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively. Clients p and q will know that the Atomic Appends operations was completed successfully when they receive these notifications from at least $f + 1$ different SBDSO servers.

▶ **Theorem 6.** *The combination of Algorithms 3 and 4 solves the* 2*-AtomicAppends problem.*

The proof follows from the one in [6], taking into consideration the above discussion.

▶ Remark. Following the approach described in [6, Section IV-B], the SBDSO can be replaced by a "classical" BDSO $\mathcal{GS}$ and the use of a set of "helper" processes. The helper processes take upon themselves the task of consulting $\mathcal{GS}$ periodically in order to find new matching descriptions of and Atomic Appends operation. When such a match is found, they complete the corresponding appends (as done in Lines 13-15 of Algorithm 4).

4.2 The Atomic Adds Problem

Inspired by the Atomic Appends problem, one could define the analogous problem on BDSOs, *Atomic Adds*: several records must be added in their corresponding BDSOs, and either all records are added (each in the appropriate BDSO) or none is added. The formal definition follows that of the Atomic Appends.

▶ **Definition 7** (2-AtomicAdds)**.** *Consider two clients, p and q, with mutually dependent records*[5] *r_p and r_q. We say that records r_p and r_q are* added atomically *in BDSO $\mathcal{GS}_p$ and BDSO $\mathcal{GS}_q$, respectively, when:*

- *AAd-safety (AAdS):* The record r_p of a correct client p is added in $\mathcal{GS}_p$ only if the record of the other client q (which may be correct or not) is also added in $\mathcal{GS}_q$.
- *AAd-liveness (AAdL):* If both p and q are correct, then both records are added eventually.

The k-AtomicAdds problem can be defined in the natural way: k clients, with k mutually dependent records, to be appended to k BDSOs. It is not difficult to see that a consensus-free algorithmic solution for this problem can be derived by simple modifications of our solution to the Atomic Appends problem and the use of the BDSO implementation of Section 3.

Atomic Adds API and server code. The Atomic Adds API, shown in Algorithm 5, is very close to Algorithm 3. The main difference is the content of the data to be added (since now we have G-Sets and not ledgers). The code run by the servers of SBDSO is the same as in Algorithm 4, with the difference that Lines 12 and 13 check for matching atomic add

[5] The definition of mutually dependent records is as in the case of Atomic Appends, but for BDSOs instead of BDLOs.

Algorithm 5 API for the 2-AtomicAdds of records r_p and r_q in BDSOs $\mathcal{GS}_p$ and $\mathcal{GS}_q$ by clients p and q, respectively, using SBDSO $\mathcal{GS}$. Algorithm for Client p.

```
1: function AtomicAdds(p, {p, q}, r_p, GS_p, r_q)
2:     GS.add(⟨p, {p, q}, r_p, GS_p, r_q⟩)
3:     return ACK
4: // Client p will know the Atomic Adds operation was completed successfully when it receives
   notifications from f + 1 different SBDSO servers. //
```

Algorithm 6 Client API and algorithms for Eventually Consistent Single-Writer BDLO $\mathcal{L}$ with $n \geq 4f + 1$ and writer process w. Code for Client p.

```
1: Init: c ← 0, k ← 0
2: function L.get( )
3:     c ← c + 1
4:     send request GET(c, p) to 3f + 1 different servers
5:     wait responses GETRESP(c, i, S_i) from 2f + 1 different servers
6:     A ← {r : record r is in at least f + 1 sets S_i}
7:     S ← {r ∈ A : (r.k = 1) ∨ (∃r' ∈ A : r.k = r'.k + 1)}
8:     return sequence ⟨ρ_1, ..., ρ_m⟩, where m = |S| and r_ℓ = (ℓ, ρ_ℓ) ∈ S
9: function L.append(ρ)                          ▷ Can only be called by process w
10:    c ← c + 1, k ← k + 1
11:    r ← (k, ρ)
12:    send request ADD(c, w, r) to ⌊n/2⌋ + 2f + 1 different servers
13:    wait responses ADDRESP(c, i, ACK) from f + 1 different servers
14:    return ACK
```

requests, and once found, in Line 14 will call the corresponding add operations, $\mathcal{GS}.\mathsf{add}(r_p)$ and $\mathcal{GS}.\mathsf{add}(r_q)$, which are implemented by the algorithms in Section 3. Note that the condition in Line 10 of Algorithm 2 may have to be expanded in order to prevent the (up to f) Byzantine servers that implement the SBDSO from adding spurious records in $\mathcal{GS}_p$ and $\mathcal{GS}_q$. This may be achieved adding a record r in these DSOs only if at least $f + 1$ clients (the servers of the SBDSO) request it to be added, similarly as done in [6].

The sequence of events is now as described in the Atomic Appends solution, with the difference that no BDLOs are now involved, only BDSOs. Putting everything together, we obtain the following, whose proof details are omitted (it is essentially a restatement of the corresponding observations in the atomic appends proof in [6], and the correctness of the algorithms in Section 3):

▶ **Theorem 8.** *The combination of the API of Algorithm 1, the API of Algorithm 5, and the revised versions of Algorithms 2 and 4, yields a solution to the 2-AtomicAdds problem.*

As noted above, the SBDSO could be replaced by a "classical" BDSO and the use of a set of "helper" processes. See [6, Section IV-B] for this approach.

4.3 Consensus-free Single-Writer BDLO

The BDSO can also be used to implement a Single-Writer BDLO without relying on consensus. This is obtained with a BDSO that allows only a single writer process w to add records, in which each record has an index determining its position in the BDLO sequence, and that does not allow adding more than one record with the same index. Allowing only add operations from w is trivially achieved by validating the signature when a request is received by a server, and will not be done explicitly in our algorithms. To prove correctness we need to show that any execution of the Single-Writer BDLO $\mathcal{L}$ we implement satisfies the

Algorithm 7 Server algorithm for Eventually Consistent Single-Writer BDLO $\mathcal{L}$ with $n \geq 4f + 1$ and writer process w. Code for Server i, and Writer w.

```
1: Init: S_i ← ∅, T ← ∅
2: receive (GET(c, p)) from process p
3:     send response GETRESP(c, i, S_i) to p
4: receive (ADD(c, w, r)) from process w
5:     if (r.k ∉ T) then
6:         BRB-broadcast(PROPAGATE(i, ADD(c, w, r)))
7:         T ← T ∪ {r.k}
8:         wait until r ∈ S_i
9:     send response ADDRESP(c, i, ACK) to w
10: end receive
11: upon (BRB-deliver(PROPAGATE(j, ADD(c, w, r)))) do
12:     if (ADD(c, w, r) was received from ⌊n/2⌋ + f + 1 different servers j) then
13:         S_i ← S_i ∪ {r}
```

Byzantine Completeness and Byzantine Eventual Consistency properties, but redefined for the $\mathcal{L}$.append() and $\mathcal{L}$.get() operations, and sequences instead of sets (see Appendix A). Additionally, the Byzantine Strong Prefix property, as defined in [6], must also be satisfied.

▶ **Definition 9** (Byzantine Strong Prefix [6])**.** *If two correct clients of a BDLO $\mathcal{L}$ issue two $\mathcal{L}$.get() operations that return record sequences S and S' respectively, then either S is a prefix of S' or vice-versa.*

Algorithm 6 presents the API and the code executed by a client of the Single-Writer BDLO $\mathcal{L}$, while Algorithm 7 presents the code executed by the servers that implement it. These algorithms require that the number of servers n satisfies $n \geq 4f + 1$. As can be seen, the append operation assigns an index k to every record data d appended by w, so the record added is in fact the pair $r = (k, d)$. Observe that Algorithms 6 and 7 are very similar to Algorithms 1 and 2, but have a few differences. (1) In Algorithm 6, $\mathcal{L}$.append(d) adds an index k to each record and sends the append requests to a potentially much larger set of $\lfloor n/2 \rfloor + 2f + 1$ servers, while $\mathcal{L}$.get() filters the set to be returned so it is a sequence of records with consecutive indices. (2) Algorithm 7 avoids appending different records with the same index $r.k$ by using this field for comparisons, keeping track in T of the indices that have been BRB broadcast, and collecting at least $\lfloor n/2 \rfloor + f + 1$ messages PROPAGATE(j, ADD(c, w, r)) before adding r to the set. Observe that the requirement on n comes from the fact that the append requests are sent to $\lfloor n/2 \rfloor + 2f + 1$ servers, and hence, $f < n/4$.

▶ **Theorem 10.** *Algorithms 6 and 7 implement an eventually consistent Single-Writer BDLO $\mathcal{L}$.*

Proof. We will first show Byzantine Completeness, then Byzantine Eventual Consistency and lastly Byzantine Strong Prefix.

Byzantine Completeness: Let us consider an $\mathcal{L}$.get() operation invoked by a correct client p. Then request GET(c, p) is sent to $3f + 1$ different servers so at least $2f + 1$ correct ones will eventually send back their responses; in fact correct servers simply answer back in Line 3 of Algorithm 7 with a GETRESP(c, i, S_i) containing their local S_i. Then, the condition of the wait operation in Line 5 is eventually satisfied and the operation completes.

Let us now assume that w is correct, and consider an $\mathcal{L}$.append() operation. Then, requests ADD(c, w, r) will be sent (Line 12 of Algorithm 6) to $\lfloor n/2 \rfloor + 2f + 1$ servers, so at least $\lfloor n/2 \rfloor + f + 1$ correct ones will receive it. Since w is correct, it increments k before sending

the ADD(c, w, r) messages (Line 10 of Algorithm 6), so the same index k is not used twice. Then, every correct process that receives ADD(c, w, r) finds that $r.k \notin T$ (since T is updated in Line 7 of Algorithm 7 only after this check). Hence, the BRB-broadcast(PROPAGATE(i, ADD(c, w, r))) in Line 6 is called at least by $\lfloor n/2 \rfloor + f + 1$ correct servers. For this reason, by the Termination properties of the BRB service, the condition in Line 12 will eventually be satisfied exactly once and record r is inserted in the local set S_i (Line 13 of Algorithm 7). So the condition in Line 8 of Algorithm 7 turns true and the response is sent back to the correct client w. Since this holds for at least $\lfloor n/2 \rfloor + f + 1$ correct servers that received the request, and $\lfloor n/2 \rfloor + f + 1 > f + 1$, the condition in Line 13 of Algorithm 6 will be satisfied and the append operation will terminate.

Byzantine Eventual Consistency: In order to demonstrate Byzantine Eventual Consistency we need to demonstrate Properties (a) and (b) of Definition 2 with respect to histories $H_{\mathcal{L}}$ that contain only events of get operations by correct clients and append operations of records that are returned in those get operations. Note that $\mathcal{L}$.append(ρ) and $\mathcal{L}$.get() are considered in place of $\mathcal{GS}$.add(r) and $\mathcal{GS}$.get().

- *Property (a):* Let $\mathcal{L}$.get be a complete operation in $H_{\mathcal{L}}$. Let S be the set from where the sequence returned by $\mathcal{L}$.get is extracted. Then, from Line 7 of Algorithm 6, $\forall r \in S$ the client verified that r belongs to $f + 1$ different sets S_i (Line 6 of Algorithm 6) returned in a GETRESP(c, i, S_i) by different servers. This means that at least a correct server has $r \in S_i$. A server only adds data to its local set S_i if that data was BRB-delivered in PROPAGATE(-, ADD(-, -, r)) messages from $\lfloor n/2 \rfloor + f + 1$ different servers. Thanks to the Validity property of the BRB service, this means that at least $\lfloor n/2 \rfloor + f + 1$ servers called BRB-broadcast with that message. Again, at least $\lfloor n/2 \rfloor + 1$ of them are correct, and they called BRB-broadcast because they received ADD(c, p, r) from client w. So, $\forall r = (k, \rho) \in S$, an $\mathcal{L}$.append(ρ) invocation precedes the $\mathcal{L}$.get response.
- *Property (b):* This is equivalent to say that $\forall \rho$ such that $\mathcal{L}$.append(ρ) $\in H_{\mathcal{L}}$, eventually there exist a time t such that ρ will be included in all the sequences returned by complete $\mathcal{L}$.get $\in H_{\mathcal{L}}$ invoked after t.

  Assume w is Byzantine and consider an operation $\mathcal{L}$.append(ρ) $\in H_{\mathcal{L}}$. Then, some $\mathcal{L}$.get() operation by a correct client returned a sequence with $r = (k, \rho)$, which means that it received at least $f + 1$ messages GETRESP(c, i, S_i) in which $r \in S_i$. This means that at least one correct server i had $r \in S_i$. Then, server i BRB-delivered at least $\lfloor n/2 \rfloor + f + 1$ PROPAGATE(-, ADD(-, -, r)) messages, and by the Termination properties of the BRB service all correct servers j will do as well, and will include r in their local sets S_j. Then, any other get operation will always have $f + 1$ responses including r from correct servers.
  Assume now that w is correct. Then, it sends requests ADD(c, w, r) with $r = (k, \rho)$ to at least $\lfloor n/2 \rfloor + 2f + 1$ servers, so that at least $\lfloor n/2 \rfloor + f + 1$ correct ones will process it calling BRB-broadcast in Line 6 of Algorithm 7. From the Termination properties of the BRB service, $\lfloor n/2 \rfloor + f + 1$ PROPAGATE(-, ADD(-, -, r)) messages coming from different servers will be eventually BRB-delivered to all correct servers. Then, all correct servers will eventually add r to their local S_i because of the fulfilment of $\lfloor n/2 \rfloor + f + 1$ requirement in Line 12 of Algorithm 7. $\mathcal{L}$.get(), on its side, returns r if it was seen at least in $f + 1$ **out of** $2f + 1$ different responses. Since at most f can have Byzantine behaviour and eventually all server will include r in their local S_i, there will exist a moment in which $\mathcal{L}$.get() will always have $f + 1$ responses including r from correct servers.
  We have shown that, independently of whether w is correct, if ρ is returned in some get operation of a correct client, eventually a record $r = (k, \rho)$ will be in all the sets S_j of all correct servers j. Then, there exist a moment in which r is definitely always part of

temporary set A in Line 6 of Algorithm 6 in all get operations. Now, in order to ensure that r is part of S, and the sequence returned, we need to demonstrate that Line 7 of client Algorithm 6 does not filter it, eventually. We proceed by induction. If $r.k = 1$ then record r is included in S. If $r.k > 1$, assume the claim true for record $r' = (k-1, \rho')$. I.e., there is a time t' after which r' is always in A. Then, there is a time $t \geq t'$ in which both r and r' are always in A. After t record r will always be included in S and returned by all get operations.

Byzantine Strong Prefix: Let $S = (r_0, ..., r_a)$ and $S' = (r'_0, ..., r'_b)$ the two sets from which the sequences returned by the two $\mathcal{L}$.get() operations are extracted in Line 8. Just as a convenience in notation, we will refer $r = (k, \rho)$ as $r_k = \rho$. Line 7 of the client Algorithm 6 ensures that records in S and S' can be ordered and that there are not missing element in the sequence. If S and/or S' are empty then one is trivially prefix of the other. So let's assume they both have at least one element and, without loss of generality, that $a \leq b$. Also, let us assume by way a contradiction that the sequence extracted from S is not a prefix of the sequence from S'. This is equivalent to state that $\exists i \leq a : r_i \neq r'_i$. From Line 6 of Algorithm 6 we know that r_j and r'_j with $1 \leq j \leq a$ were returned at least by one correct server in their respective get operations. So, assuming that such an index i exists means that at least two correct servers executed Line 13 of Algorithm 7 for the two records, respectively. This implies that, for both, the condition of Line 12 was true because they received messages PROPAGATE(−, ADD(c,p,r_i)) from a set C of at least $\lfloor n/2 \rfloor + f + 1$ servers, and messages PROPAGATE(−, ADD(c,p,r'_i)) from a set C' of at least $\lfloor n/2 \rfloor + f + 1$ servers. Note that each C and C' contains at least $\lfloor n/2 \rfloor + 1$ correct servers. It is obvious that broadcasters of these PROPAGATE messages must intersect in at least one correct server j. So, from the Validity property of the BRB service, at least correct server j called both BRB-broadcast(PROPAGATE(j, ADD(c,p,r_i))) and BRB-broadcast(PROPAGATE(j, ADD(c,p,r'_i))). Line 5 of Algorithm 7 filters the received ADD(c,p,r) request, so only if $r.k \notin T$ they are propagated via the BRB-broadcast. If so, Line 7 of Algorithm 7 adds $r.k$ to T right after the BRB-broadcast. Assume, w.l.o.g., that j received ADD(c,p,r_i) before receiving ADD(c,p,r'_i). As soon as j BRB-broadcast PROPAGATE(i, ADD(c,p,r_i))), it added $r_i.k$ to T. Then, when it received ADD(c,p,r'_i) it found that $r'_i.k \in T$, and BRB-broadcast(PROPAGATE(i, ADD(c,p,r'_i))) was not executed. But this contradicts our assumption that $\exists i \leq a : r_i \neq r'_i$. Hence, the sequence extracted from S must be a prefix of the sequence from S'. ◀

5 Conclusions and Future Work

In this paper we formally define the notion of a Byzantine-tolerant Distributed G-Set Object (BDSO) and provide client and server algorithms to implement a consensus-free eventually consistent BDSO. Then we proceed with some use cases for BDSO. Building on the work in [6] and using BDSOs we provide a consensus-free solution to the Atomic Appends problem. Similarly, we provide a consensus-free solution to the Atomic Adds problem, the analogous problem that uses sets instead of ledgers. Finally, we show how a few modifications to the client and server algorithms of BDSO, enable to realise an eventual consistent Single-Writer Byzantine Distributed Ledger without solving consensus among servers but still guaranteeing the Byzantine Strong Prefix property. Single-Writer consensus-free BDLO can be suitable for many use cases, like implementing a cryptocurrency or a punch in/out system for employees of a company. These are scenarios where realising transactional systems in a Byzantine failure model through consensus may not provide reasonable performance, since the need of

updating the system global status prevents sustaining a high throughput of operations. Our future plans include implementing and experimentally evaluating the algorithms proposed in this work, as well as specifying a cryptocurrency based on single-writer BDLOs.

References

1 Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Blockchain abstract data type. In *31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 349–358. ACM, 2019. doi:10.1145/3323165.3323183.

2 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bull. EATCS*, 132:23–43, 2020. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/629.

3 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

4 Eric Brewer. Cap twelve years later: How the “rules" have changed. *Computer*, 45(2):23–29, 2012.

5 Hua Chai and Wenbing Zhao. Byzantine fault tolerance for services with commutative operations. In *IEEE International Conference on Services Computing, SCC 2014, Anchorage, AK, USA, June 27 – July 2, 2014*, pages 219–226. IEEE Computer Society, 2014. doi:10.1109/SCC.2014.37.

6 V. Cholvi, A. Fernandez Anta, C. Georgiou, N. Nicolaou, and M. Raynal. Atomic appends in asynchronous byzantine distributed ledgers. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 77–84, 2020. doi:10.1109/EDCC51268.2020.00022.

7 Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. Byzantine fault-tolerant atomic multicast. In *DSN 2018*, pages 39–50. IEEE, 2018.

8 F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

9 Antonio Fernández Anta, Chryssis Georgiou, and Nicolas Nicolaou. Atomic appends: Selling cars and coordinating armies with multiple distributed ledgers. In *International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2019, Paris, France*, pages 39–50, 2019.

10 Antonio Fernández Anta, Kishori M. Konwar, Chryssis Georgiou, and Nicolas C. Nicolaou. Formalizing and implementing distributed ledger objects. *SIGACT News*, 49(2):58–76, 2018.

11 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2015, Sofia, Bulgaria, April 26-30, 2015, Part II*, pages 281–310, 2015. doi:10.1007/978-3-662-46803-6_10.

12 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 – August 2, 2019*, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.

13 Saurabh Gupta. *A non-consensus based decentralized financial transaction processing model with support for efficient auditing.* Arizona State University, 2016.

14 Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 245–254, 2018. URL: https://dl.acm.org/citation.cfm?id=3212736.

15 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

16 T. Koens and E. Poll. Assessing interoperability solutions for distributed ledgers. *Pervasive and Mobile Computing*, 59:101079, 2019. doi:10.1016/j.pmcj.2019.101079.

17 Zarko Milosevic, Martin Hutle, and André Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In *SRDS 2011*, pages 235–244, 2011.

18 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008. [Online; accessed 22-February-2021].

19 Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013.

20 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems – An Algorithmic Approach*. Springer, 2018. `doi:10.1007/978-3-319-94141-7`.

21 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Symposium Stabilization, Safety, and Security of Distributed Systems, SSS 2011, Grenoble, France*, pages 386–400. Springer, 2011. `doi:10.1007/978-3-642-24550-3_29`.

22 Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009. `doi:10.1145/1435417.1435432`.

A DLO Definitions

For the reader's convenience, we provide the basic definitions regarding Distributed Ledger Objects [10].

A ***ledger*** $\mathcal{L}$ is a concurrent object that stores a totally ordered sequence $\mathcal{L}.S$ of records and supports two operations (available to any process p): (i) $\mathcal{L}.\mathsf{get}_p()$, and (ii) $\mathcal{L}.\mathsf{append}_p(r)$. The sequential specification of a ledger $\mathcal{L}$ is as follows:

▶ **Definition 11.** *The* sequential specification *of a ledger $\mathcal{L}$ over the sequential history $H_{\mathcal{L}}$ is defined as follows. The value of the sequence $\mathcal{L}.S$ of the ledger is initially the empty sequence. If at the invocation event of an operation π in $H_{\mathcal{L}}$ the value of the sequence in ledger $\mathcal{L}$ is $\mathcal{L}.S = V$, then:*

1. *if π is an $\mathcal{L}.\mathsf{get}_p()$ operation, then the response event of π returns V, and*
2. *if π is an $\mathcal{L}.\mathsf{append}_p(r)$ operation, then at the response event of π, the value of the sequence in ledger $\mathcal{L}$ is $\mathcal{L}.S = V \| r$ (where $\|$ is the concatenation operator).*

A ***Distributed Ledger Object***, DLO for short, is a concurrent ledger object that is implemented in a distributed manner. In particular, the ledger object is implemented by *servers*, and *clients* invoke the $\mathsf{get}()$ and $\mathsf{append}()$ operations.

▶ **Definition 12.** *A DLO $\mathcal{L}$ is* **eventually consistent** *if, given any history $H_{\mathcal{L}}$,*

(a) *Let S be the sequence of records returned by any complete operation $\pi = \mathsf{get}() \in H_{\mathcal{L}}$ and ρ_i the generic record that belongs to S. For each $\rho_i \in S$ then $H_{\mathcal{L}}$ contains $\mathsf{append}(\rho_j)$ for $j = 1...i$ whose invocation events appear before the response event of π in $H_{\mathcal{L}}$, and*

(b) *for every complete operation $\mathcal{L}.\mathsf{append}(\rho) \in H_{\mathcal{L}}$, there exists a history $H'_{\mathcal{L}}$ that extends $H_{\mathcal{L}}$ such that, for every history $H''_{\mathcal{L}}$ that extends $H'_{\mathcal{L}}$, every complete operation $\mathcal{L}.\mathsf{get}()$ in $H''_{\mathcal{L}} \setminus H'_{\mathcal{L}}$ returns a sequence that contains ρ.*

Observe that the above definition is equivalent to the one given in [10, Definition 4].

A DLO is an ***eventually consistent Byzantine-tolerant DLO*** (BDLO), if it satisfies the next three properties:

- *Byzantine Completeness (BC)*: All the $\mathsf{get}()$ and $\mathsf{append}()$ operations invoked by correct clients eventually complete.
- *Byzantine Strong Prefix (BSP)*: If two *correct clients* issue two $\mathsf{get}()$ operations that return record sequences S and S' respectively, then either S is a prefix of S' or vice-versa.

- *Byzantine Eventual Consistency (BEC)*: This is the property of Definition 12 with respect to the get() operations invoked by correct clients and the append(r) operations that append the records r returned in those get() operations.

B Acronyms table

Table 1 Meaning of acronyms.

DLT	Distributed Ledger Technologies
DLO	Distributed Ledger Object
SDLO	Smart Distributed Ledger Object
BDLO	Byzantine-tolerant Distributed Ledger Object
SBDLO	Smart Byzantine Distributed Ledger Object
G-Set	Grow-only Set
DSO	Distributed Grow-only Set Object
BDSO	Byzantine-tolerant Distributed Grow-only Set Object
BRB	Byzantine Reliable Broadcast
BToB	Byzantine Total-order Broadcast Service
CRDTs	Conflict-Free Replicated Data Type

DAISIM: A Computational Simulator for the MakerDAO Stablecoin

Shreyas Bhat
Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA

Ayten Betul Kahya
Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA

Bhaskar Krishnamachari
Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA

Rohit Kumar
Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA

Abstract

We present a computational simulation of the single-collateral DAI stablecoin launched by the MakerDAO project in 2017. At the core of the simulation is a model of cryptocurrency investors acting as rational Markowitz mean-variance portfolio optimizers, with heterogeneous risk tolerance. The simulator, called DAISIM, incorporates automated order matching and price update mechanisms to determine the DAI price. We use the simulator to evaluate how the single-collateral DAI price, as well as portfolio allocations, vary for a given population of investors as a function of exogenous parameters such as the price of ETH and various system parameters including stability rate and transaction fee. DAISIM is being made available as open-source and may be useful in evaluating other similar projects.

2012 ACM Subject Classification Applied computing → Digital cash

Keywords and phrases Stablecoin, Simulator, MakerDAO

Digital Object Identifier 10.4230/OASIcs.FAB.2021.3

Supplementary Material *Software (Source Code)*: `https://github.com/ANRGUSC/DAISIM`

Acknowledgements This work has been supported at University of Southern California (USC) in part by a gift from SovereignWallet Network.

1 Introduction

A stablecoin [11, 12] is a digital token that is designed to minimize price volatility against a peg. They are pegged to fiat currencies (most commonly the US Dollar), other assets such as gold or a basket of assets. By tying the value to an asset, stablecoins aim to mitigate the high volatility associated with other cryptocurrencies such as Bitcoin. By achieving stability, these tokens have a higher potential to be utilized as a unit of account, a store of value and a medium of exchange compared to volatile cryptocurrencies. Various methods have been developed to stabilize the value of the token. These include backing by fiat currencies, crypto-assets or using algorithmic stabilization (not backed by any asset).

One of the prominent projects is MakerDAO [6], a decentralized Stablecoin project on Ethereum blockchain launched in 2017. The Maker smart contract platform offers a crypto-asset backed Stablecoin called DAI, which has a 1:1 soft peg to the US dollar. The initial single-collateral DAI on the platform was called 'SAI' after transitioning to the new Maker Protocol with multiple collateral types. SAI officially shutdown in May 2020. Stability of single-collateral DAI was provided by Collateralized Debt Positions (CDP), Maker Governance who held the governance token called MKR and incentivized external actors.

Our goal in this paper is to develop a computational simulation framework for modeling MakerDAO, to understand how well its underlying mechanism works under different settings. The simulation model is somewhat parsimonious, trading off some loss in realism in exchange for computational tractability, insight, and ease of exposition.

The crux of our model is to focus on the population of investors and investigate whether and when they choose to mint or burn DAI, and when they choose to buy and sell ETH or DAI. We model the investors using Markowitz's Optimal Portfolio Theory [4]. Specifically, we model them as maintaining and updating a portfolio consisting of four assets USD, ETH, DAI, and cETH (collateralized-ETH, used as collateral deposit to borrow/mint DAI), as well as a debt instrument (as interest is owed on any DAI that is borrowed), accounting also for transaction fees, in order to maximize their expected return while minimizing risk. A weight-parameter characterizes the risk-tolerance of each user. Given a population of such investors and their preferred allocations, our simulator iteratively updates the price of DAI and matches buyers and sellers to determine the market clearing or settling price. It allows us to set and modify various exogenous parameters such as return and risk associated with various assets and the price of ETH as well as system parameters such as interest rate (known as stability rate in the MakerDAO ecosystem) and transaction fees, allowing us to examine how the DAI price depends on these various parameters.

The key contributions of our work are as follows:

- We show how to model the MakerDAO ecosystem using optimal portfolio theory to model investor behavior with respect to relevant assets including USD, ETH, DAI and cETH, while accounting for transaction fees and the stability rate.
- We present our design and implementation of a computational market simulator, DAISIM, that handles order matching and price updates to determine the DAI price for a given set of parameters.
- We use the simulator to study how the DAI price and DAI supply/demand and portfolio allocation is affected by various exogenous parameters (such as risk tolerance of investors, ETH price, mean and covariance of asset returns) and system parameters (such as stability rate and transaction fees).
- The simulator itself is made available as an open-source simulation tool for use by the research community online at `https://github.com/ANRGUSC/DAISIM`.

The rest of the paper is organized as follows: in section 2, we describe the basics of the MakerDAO project with a focus on the simple single-collateral DAI launched in 2017 (extension to multi-collateral DAI is the focus of our future work). In section 3, we briefly survey the relevant prior work. In section 4, we present our simulation model and how the simulator is designed. In section 5, we present some illustrative results from the simulator to show how DAI price and investor decisions are affected by various key parameters. Finally, we present our conclusions in section 6.

2 MakerDAO – Background

At the heart of the MakerDAO is an autonomous mechanism to allow users to mint the DAI token. Before the launch of multi-collateral DAI (MCD), single-collateral DAI (SAI) could only be generated through a Collateralized Debt Position (CDP), a smart contact that required the user to lock in excess collateral above a minimum ratio called Liquidation Ratio at which the collateral is subjected to forced liquidation. After MCD, CDPs are now called "Vaults" but we call them CDPs for brevity. DAI can be used for trading, borrowing, making payments and more recently, also, saving. Some key statistics of MakerDAO can be found at [10].

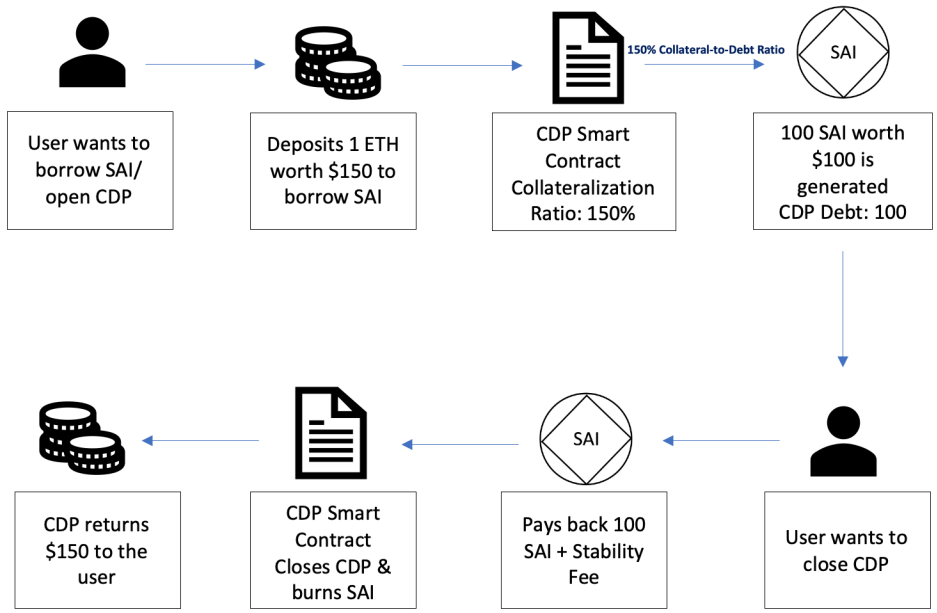


Figure 1 A sample CDP opening and closing transaction assuming that 1 ETH is equal to $150 and the minimum collateralization ratio is 150%.

Prior to the launch of multi-collateral DAI (MCD) in November 2019, DAI could only be generated through the single-collateral type, ETH[1]. MCD is now available to users at different Liquidation Ratios derived as a function of the risk pertaining the underlying collateral type determined by the Maker Risk Teams and Maker Governance.

After the user chooses a collateral-to-debt ratio (also known as the collateralization ratio) and the amount of single-collateral DAI they would like to borrow from the CDP, the smart contract deposits the collateral and returns DAI. The collateral is locked until the outstanding debt is paid in addition to the CDP Interest Rate (Stability Rate) that has accrued over time.

A CDP/Vault can be closed at any time once the debt and the CDP Interest Rate (Stability Rate) are paid. The collateralization ratio of a CDP can also be adjusted while it is active given that it is collateralized above the liquidation ratio. If a collateral becomes too risky when collateralization ratio drops to the liquidation ratio, the CDP is automatically acquired by the system and liquidated. Before MCD, liquidation was executed through a Liquidity Providing Contract whereas in MCD, an auction mechanism is used for liquidation. After the debt, Stability Rate and Liquidation Penalty have been recovered, the left-over collateral is returned to the CDP owner.

2.1 CDP and the Stability Rate

The Stability Rate acts like an interest rate on the loan. It plays a key role in maintaining stability of DAI through active governance. It is shown in DAI and paid in MKR that is removed out of circulation upon payment. When the value of DAI is below the Target Price,

[1] CDPs/Vaults can result in having more debt versus the value of collateral if there is an abrupt market crash in ETH. In this case, the collateral i.e ETH is diluted to recapitalize the system by the platform.

increasing the Stability Rate incentives users to close CDPs. Thus, it removes DAI from supply and helps restore the peg. Similarly, when the value of DAI is above the Target Price, decreasing the Stability Rate incentivizes users to open more CDPs. This increases the DAI supply and helps restore the peg. In addition to the Stability Rate, Debt Ceiling, Liquidation Ratio and Penalty Ratio are other key risk parameters for CDPs.

3 Prior work

We briefly present some relevant prior works focused on the evaluation of stablecoins, still a relatively sparse area of research. A broad survey of stablecoins is provided by Clark *et al.* [1]. Mundt and Minca [9] describe a complementary model of noncustodial stablecoins and explore different models of the liquidation structure that affects speculator decision-making and then analytically characterize the stability. Mundt and Minca [8] analyze the effects of deleveraging feedback effects that cause illiquidity during crises for non-custodial cryptocurrency-backed stable coins. Mundt et al. [7] propose a framework for relating economic mechanics of all stablecoins and formulated three classes of models for non-custodial stablecoins, for which traditional financial models are sparse. Lyons and Natraj [5] examine the efficiency and working mechanisms of stablecoins in the digital economy. They analyze how price stabilization functions in the case of stablecoins. Gudgeon *et al.* [3] investigate the feasibility of attacking the MakerDAO governance mechanism from a security perspective. Gu and Kothari [2] discuss a multiagent simulation of a generic asset-backed stablecoin with a focus on understanding demand dynamics for a stable coin in the face of exogenous price shocks.

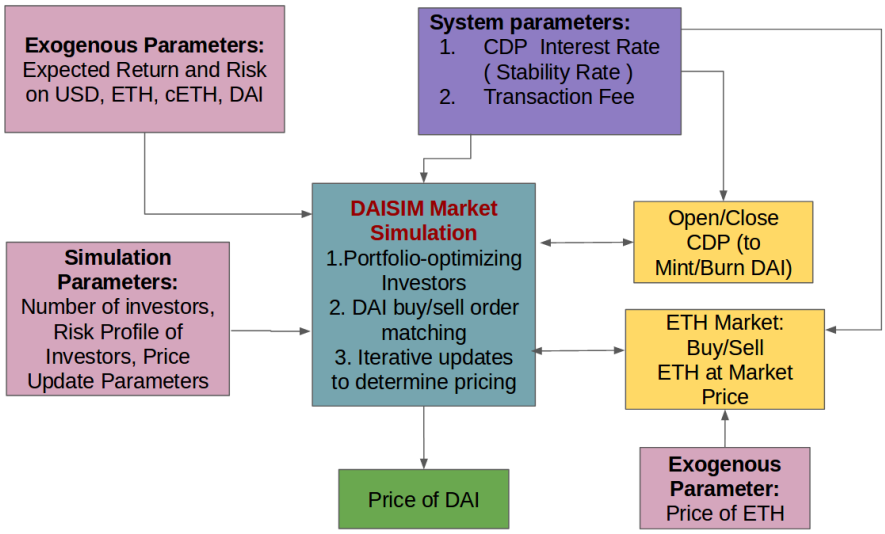


Figure 2 Overview of the DAISIM Simulator.

4 Design of the DAISIM Simulator

Considering that the Maker protocol has rapidly evolved in the last few years, this paper will assume that DAI mentioned in the subsequent sections refers to single-collateral DAI (SAI) for brevity.

4.1 System model

Our full system model for the single-collateral DAI ecosystem and our simulator is as shown in Figure 2. Exogenous inputs to the model include the price of ETH, expected return and risk (covariance) for USD, ETH, DAI and cETH. System parameters include the Stability Rate r_s and the Transaction Fees β. Additional simulation parameters include the size of the market n (number of investors), their risk profile (captured by a weight parameter λ_i for the i^{th} investor), and parameters pertaining to the price update algorithm employed in the simulation. The simulator allows investors to buy ETH on an open market as per the current ETH price P_{ETH}; it allows investors to open and close CDP's per the current stability rate; and it allows investors to buy and sell DAI from/to each other in the simulated market. All these transactions incur a constant transaction fee as specified by β. The simulator takes care of matching buy/sell orders for DAI and determining the market clearing (settling) price for DAI P_{DAI}. We describe these mechanisms in more detail below.

4.2 Price Settling Algorithm

The Price Settling Algorithm involves three steps i.e., Asset Optimization Mechanism, Order Matching Mechanism, and the Price Update Mechanism. We assume n investors each with an initial asset holdings $\mathbf{x}$ and a risk tolerance parameter λ. It is assumed that if λ is low, then the investor is risk-tolerant, and if it is high, then the investor is risk-averse. For each of these investors, we use the asset optimization mechanism to find out an optimal portfolio, x^{opt} and then use the Order Matching Mechanism to verify if all DAI Buy orders, B and the Sell Orders, S can be met. This mechanism proposes a new asset allocation of $x^M_{i,j}$ for the asset $j \in \{USD, ETH, DAI, cETH\}$ of the i^{th} investor based on $x^{opt}_{i,j}$. Then using the price update mechanism, we estimate the supply/demand of DAI based on the DAI bought/sold by the investors to achieve the optimal allocation and then update the DAI price, P_{DAI}. Table 1 can be referred for the details of different notations and their definitions used in this paper.

4.2.1 Asset Optimization Mechanism

For the i^{th} investor, consider the vector $\mathbf{x} = [x_{i,1}, x_{i,2}, x_{i,3}, x_{i,4}]$ which represents the i^{th} investor's holdings in each asset class: USD, ETH, DAI and cETH, respectively. We assume that the investor collateralizes at a constant safety ratio ρ that is well above the liquidation ratio of the protocol. Let r_s represents the stability rate. Let μ be the vector of expected return on investment in each of the four assets, and let $\mathbf{\Sigma}$ be the covariance matrix associated with the value of these assets. Let β be the transaction fee to buy or sell 1 USD worth of ETH/DAI and Ψ be the overall transaction fee incurred to reach the optimal allocation.
Let δ_{DAI} represents the current DAI debt for the investor. This debt corresponds to the amount of DAI minted from the CDP, and given the fixed collateralization ratio is assumed to be exactly equal to $\frac{x_{i,4}}{\rho}$. Then, it is easy to see that an optimal portfolio i.e., x^{opt} for the i^{th} investor with a total initial investment capital of $m = \Sigma x$ corresponds to one that maximizes:

$$\mathbf{x}^T\mu - \lambda \mathbf{x}\mathbf{\Sigma}\mathbf{x}^T - \frac{x_{i,4}}{\rho} r_s - \Psi \tag{1}$$

subject to the constraints:

$$\Psi = \mid x_{i,2} - x^{int}_{i,2} + x_{i,4} - x^{int}_{i,4} \mid \cdot \beta + \mid x_{i,3} - x^{int}_{i,3} \mid \cdot \beta \tag{2}$$

$$x_{i,1}^{int} \geq \Psi, x_{i,2}^{int} \geq 0, x_{i,3}^{int} \geq 0, x_{i,4}^{int} \geq 0 \tag{3}$$

$$\Sigma x^{int} = \Sigma x = m \tag{4}$$

Please note that $x_{i,j}^{int}$ is an intermediate allocation of the asset j for the i^{th} investor. It should also be noted that the transaction fees also apply to cETH as we need to buy ETH to deposit it as collateral. Constraints (2) and (3) ensure that the investors have enough USD holdings after choosing a portfolio allocation to cover any transaction fees incurred. After deducting the transaction fees Ψ, the updated asset holdings of the i^{th} investor are given as:

$$x_{i,1}^{opt} = x_{i,1}^{int} - \Psi,\ x_{i,2}^{opt} = x_{i,2}^{int},\ x_{i,3}^{opt} = x_{i,3}^{int},\ x_{i,4}^{opt} = x_{i,4}^{int}.$$

Table 1 Notations and Definitions.

Notations	Definitions
n	Number of investors participating in DAISIM
$\mathbf{x}$	Investor's initial asset's holdings
$x_{i,j}$	Initial investment of i^{th} investor in the asset j; $j \in \{USD, ETH, DAI, cETH\}$
λ_i	Risk tolerance parameter of i^{th} investor
$x_{i,j}^{opt}$	Optimal asset allocation for the i^{th} investor for asset j as suggested by the Asset Optimization Mechanism
B	Total outstanding buy order
S	Total outstanding sell order
$x_{i,j}^{M}$	Actual asset allocation for the i^{th} investor for asset j given by Order Matching Mechanism
ρ	Safety ratio
r_s	Stability rate
μ	Vector of expected return on investment in each of the four assets
$\mathbf{\Sigma}$	Covariance matrix associated with the value of the assets
β	Transaction fee to buy or sell 1 USD worth of ETH/DAI
δ_{DAI}	Current DAI debt for the investor
m	Initial investment capital
$x_{i,j}^{int}$	Intermediate allocation of the asset j for the i^{th} investor.
D_i^{ov}	DAI bought/sold by i^{th} investor to achieve optimal allocation
P_{ETH}	Price of ETH provided by the price oracle
P_{DAI}	Price of DAI determined by the price settling algorithm
D_i^{om}	DAI Bought by i^{th} investor in Order Matching Mechanism
D_i^{cdp}	DAI to be minted/returned by i^{th} investor to achieve optimal cETH allocation
π_j	Mean asset holdings for investors j; j $\in$ {USD, ETH, DAI, cETH}

4.2.2 Order Matching Mechanism

We assume that the market doesn't have any external source of DAI, thus we need to make sure that the total DAI in the market is fixed during the course of the Price Settling Algorithm. Let order value, $D_i^{ov} = x_{i,3}^{opt} - x_{i,3}$ denote the amount of DAI the i^{th} investor needs to buy/sell in order to reach its optimal allocation. Let B denote the total outstanding buy order, and S represents the total outstanding sell order in the market.

$$B = \sum min(D_i^{ov}, 0) \tag{5}$$

$$S = \sum max(D_i^{ov}, 0) \tag{6}$$

If $B > S$ i. e. buy orders exceed sell orders, then all the sell orders can easily be met whereas when $S > B$ all buy orders can easily be met. In case all buy and sell orders cannot be met, the buy/sell order of an i^{th} investor is achieved in proportion to their D_i^{ov} value. The different possible scenarios of the Order Matching Mechanism can be illustrated with the help of the following examples:

▶ **Example 1.** Assume we have 4 investors with the following order values.
$D_1^{ov} = 100$, $D_2^{ov} = 120$, $D_3^{ov} = -130$, $D_4^{ov} = -190$
Total Buy orders, $B = 220$
Total Sell orders, $S = 320$
Since $S > B$, We can meet all buy orders, but not all sell orders.
Total DAI Bought in Market = 220
Investor 3 sells $130/320 * 220 = 89.375$
Investor 4 sells $190/320 * 220 = 130.625$
Let D_i^{om} denotes the DAI bought by i^{th} investor in the Order Matching Mechanism. Thus,
$D_1^{om} = 100$, $D_2^{om} = 120$, $D_3^{om} = -89.375$, $D_4^{om} = -130.625$

▶ **Example 2.** Assume we have 4 investors with the following order values.
$D_1^{ov} = 500$, $D_2^{ov} = 120$, $D_3^{ov} = -130$, $D_4^{ov} = -190$
Total Buy orders, $B = 620$
Total Sell orders, $S = 320$
Since $B > S$, We can meet all sell orders.
Total DAI Sold in Market = 320
Investor 1 buys $500/620 * 320 = 258.06$
Investor 2 buys $120/620 * 320 = 61.93$
$D_1^{om} = 258.06$, $D_2^{om} = 61.93$, $D_3^{om} = -130$, $D_4^{om} = -190$

At the end of the Order Matching Mechanism, we have $x_{i,1}^M = x_{i,1} - \Psi$, $x_{i,2}^M = x_{i,2}^{opt}$, $x_{i,3}^M = x_{i,3} + D_3^{om}$, $x_{i,4}^M = x_{i,4}^{opt}$. The transaction fees in this step of the algorithm is,

$$\Psi = \mid x_{i,2}^M - x_{i,2} + x_{i,4}^M - x_{i,4} \mid \cdot \beta + \mid x_{i,3}^M - x_{i,3} \mid \cdot \beta$$

4.2.3 Price Update Mechanism

It is evident from the above discussion that the Asset Optimization Mechanism estimates the market's demand for DAI whereas the Order Matching Mechanism tries to fulfill the demand keeping total DAI in the market constant. The Price Update Mechanism updates P_{DAI} based on the supply and demand of DAI in the market. We assume that DAI minted by CDPs as another indicator for P_{DAI}. Let $D_i^{cdp} = \frac{(x_{i,4}^{opt} - x_{i,4}) * P_{ETH}}{P_{DAI} * \rho}$ be the DAI to be minted/returned by i^{th} investor to achieve optimal cETH allocation.

$$\sum_{i=1}^{n} (D_i^{ov} - D_i^{cdp}) \geq 0 \Rightarrow High\ Demand$$

$$\sum_{i=1}^{n} (D_i^{ov} - D_i^{cdp}) < 0 \Rightarrow High\ Supply$$

If we are in a high demand zone, we raise the price, else we reduce it. At the end of each iteration we fix the asset allocation $x_{i,j} = x_{i,j}^M$.

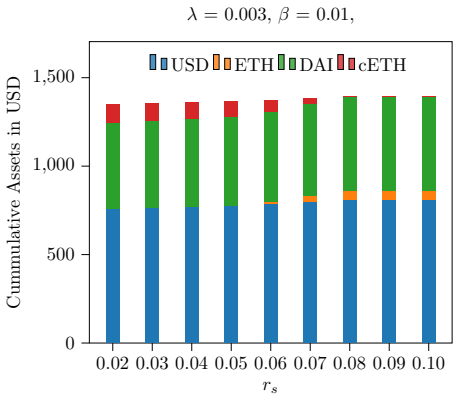


Figure 3 Investor Asset Holdings vs. r_s.

5 Simulation Results

We present here some results illustrating the DAISIM Market simulation model which allows us to show the impact of various parameters on the investors' optimal portfolio. Purely as an illustrative example, we assume that the return rates on the four assets i.e., [USD, ETH,DAI, cETH] are given by $\mu = [0.08, 0.22, 0.18, 0.16]$ and that their covariance matrix is given as follows:

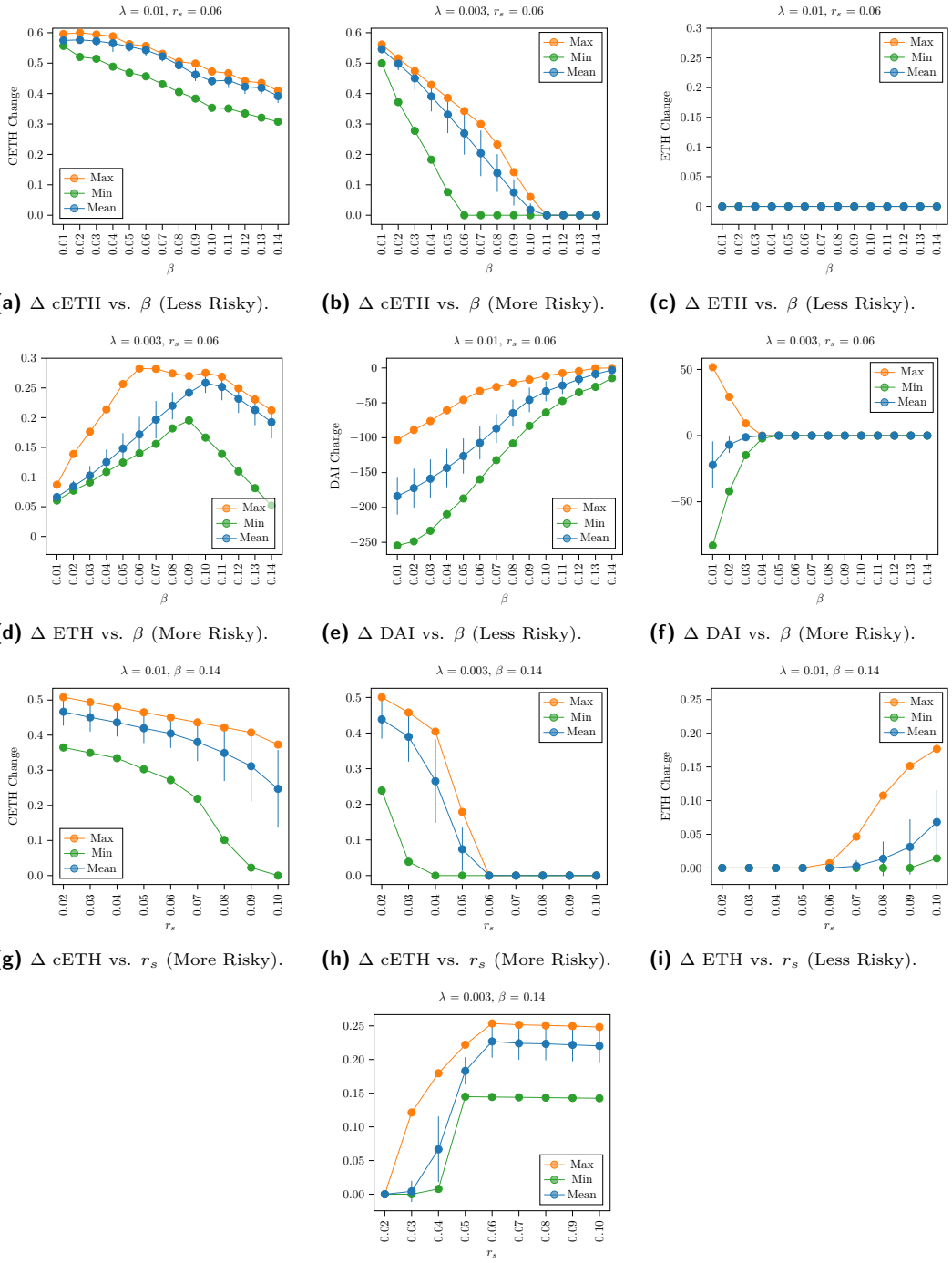
$$\mathbf{\Sigma} = \begin{bmatrix} 0.04 & 0 & 0 & 0 \\ 0 & 0.64 & 0.048 & 0.36 \\ 0 & 0.048 & 0.09 & 0.015 \\ 0 & 0.36 & 0.015 & 0.25 \end{bmatrix} \tag{7}$$

These parameters have been chosen arbitrarily for illustration, but intuitively encode the following assumptions. The returns and variance in increasing order are USD < cETH < DAI < ETH. Returns on USD is assumed to be uncorrelated with other assets while DAI is weakly correlated with ETH and cETH. ETH and cETH are relatively highly correlated with each other.

5.1 Baseline parameters and portfolio

Considering our baseline model with parameters $n = 10$, $\pi_{USD} = \$1000$, $\pi_{DAI} = \$1000$, $\pi_{eth} = \$0$, $\pi_{ceth} = \$0$, $r_s = 0.06$, $\beta = 0.01$, we analyze how does the transaction fee β, Stability Rate r_s and risk preference λ affect the P_{DAI} and the optimal portfolio of an investor. We fix $\lambda = 0.01$ for a risk-averse investor and $\lambda = 0.003$ for a risk-tolerant investor.

In a population of n investors with 2 possible risk values i.e., $\lambda \in \{0.003, 0.01\}$, we have 2^n different risk profiles for n investors. When $n = 10$, we have $2^{10} = 1024$ different risk profiles for an investor population. Also, when we fix λ for a single investor, we have 512 different risk profiles for the remaining 9 investors. In each of these 512 risk profiles, we find the assets bought or sold by the investor and we call the mean of this value as Mean Asset Change. For example, Mean cETH Change (Δ cETH) refers to the mean cETH bought or sold by an investor when we change λ for the other 9 investors. Mean DAI Settling Price is also similarly defined.



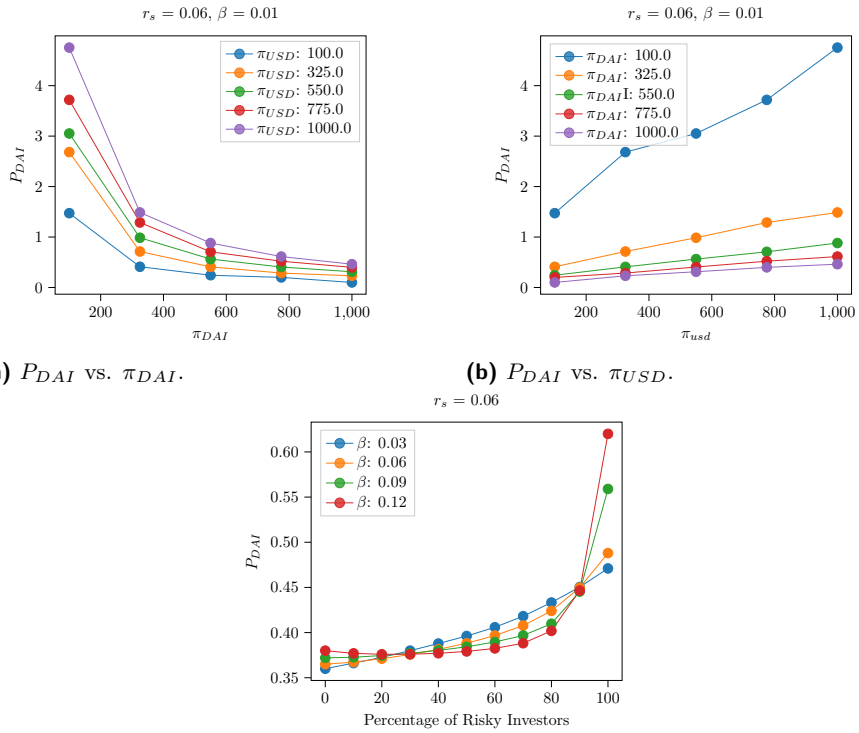
(a) Δ cETH vs. β (Less Risky). **(b)** Δ cETH vs. β (More Risky). **(c)** Δ ETH vs. β (Less Risky).

(d) Δ ETH vs. β (More Risky). **(e)** Δ DAI vs. β (Less Risky). **(f)** Δ DAI vs. β (More Risky).

(g) Δ cETH vs. r_s (More Risky). **(h)** Δ cETH vs. r_s (More Risky). **(i)** Δ ETH vs. r_s (Less Risky).

(j) Δ ETH vs. r_s (More Risky).

Figure 4 Investor Asset Trends.

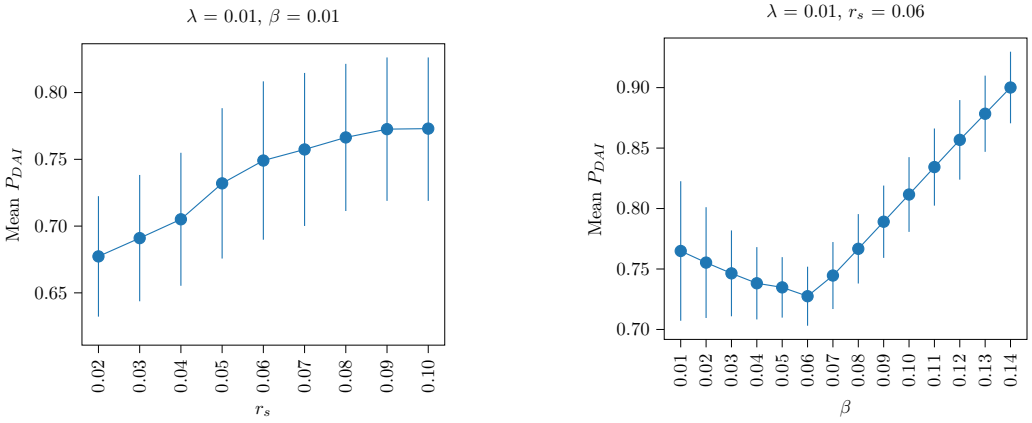


(a) P_{DAI} vs. π_{DAI}.

(b) P_{DAI} vs. π_{USD}.

(c) P_{DAI} vs. Percentage of risky investors.

Figure 5 P_{DAI} v/s Investor population aggregates.



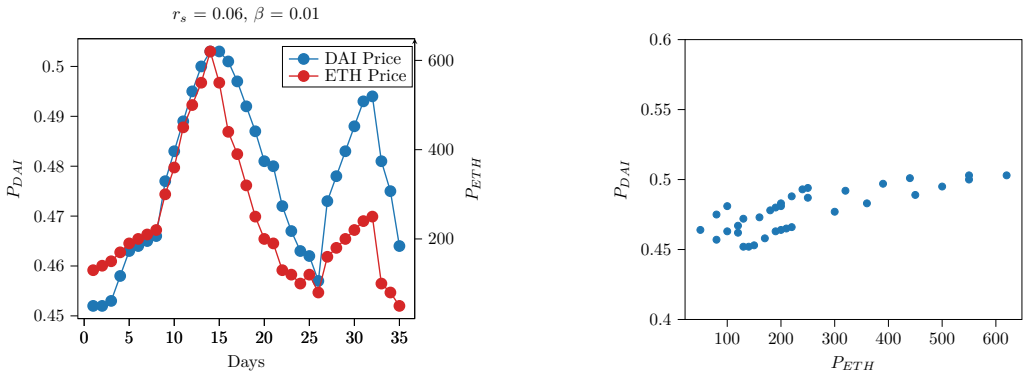
(a) Mean P_{DAI} vs. r_s.

(b) Mean P_{DAI} vs. β.

Figure 6 Mean P_{DAI} vs. r_s & β.

5.2 Investor Assets

In this section, we describe how does risk preference λ, transaction fee β & stability rate r_s impact an investor's asset allocation. In Figure 3, we analyze the impact of change in r_s on the distribution of an investor's assets. It is observed that once the value of r_s increases from 0.02 to 0.1, the distribution of different assets (shown in different colors) changes. An increase in r_s makes it costlier to open a CDP and therefore disincentivizes an investor from



(a) P_{DAI} & P_{ETH} over 35 days.

(b) Correlation of P_{DAI} & P_{ETH}.

Figure 7 Relationship b/w P_{DAI} & P_{ETH}.

holding cETH. At the same time, as β remains constant, the cost of holding ETH remains the same. It is evident from Figure 3 that an investor reduces its cETH holdings and increases its ETH holdings as r_s increases. This is because holding ETH becomes cheaper and more profitable given its high return rate.

5.2.1 Impact of Risk Preference

A risk-tolerant investor is more likely to invest in ETH as compared to a risk-averse investor given that ETH is the riskiest asset. In Figures 4a through 4j, we make the following observations,

- In Figures 4a, 4b the cETH holdings of a risk-tolerant investor quickly reach 0. It appears that a risk-tolerant investor is very sensitive to β.
- In Figures 4c, 4d, we see that a risk-tolerant investor invests in the riskiest asset i.e., ETH, while a risk-averse investor doesn't invest in ETH at all. For the risk-tolerant investor as β increases, we observe that ETH holdings first increase and then decrease. We believe that an investor prefers to convert its cETH to ETH as it offers a better return rate but once its cETH holdings reach 0, the taxation from β comes into the picture which causes a decline in ETH Holdings.
- In Figures 4e, 4f, we see that a risk-tolerant investor prefers to hold DAI vs. a risk-averse investor that wants to minimize total DAI held. At lower β's an investor is very sensitive to the risk parameters of other $n-1$ investors.
- In Figures 4g, 4h, we see that the cETH holdings of a risk-tolerant investor are also very sensitive to r_s. And from Figures 4i, 4j, we see that a risk-tolerant investor holds more ETH than a risk-averse investor.

5.2.2 Impact of Transaction Fee

In Figures 4a, 4b we see that as β goes up, the mean cETH change decreases. In Figure 4e, 4f we see that as we increase β, the absolute mean DAI Change also reduces to 0. Similarly, an investor is also less likely to buy/sell ETH. These trends are easy to explain because a transaction fee on buying/selling of any asset is similar to a tax. A higher β disincentivizes an investor from buying and selling assets.

In Figure 6b, we see that as β increases from 0.01 to 0.14 with π_{USD} = \$325 & π_{DAI} = \$300, mean P_{DAI} first decreases and then increases. An increase in β has two side effects i.e., reduction in DAI demand and reduction in DAI supply. We believe that, when DAI demand reduces more than DAI supply we see a decrease in P_{DAI} and when DAI supply reduces more than DAI demand we see an increase in P_{DAI}.

5.2.3 Impact of Stability Rate

In Figures 4g, 4h we see that as r_s increases from 0.02 to 0.1, with all parameters matching the baseline, the mean cETH change for an investor decreases because it becomes costlier to open a CDP for minting DAI. In Figures 4i, 4j it is also seen that the mean ETH change for an investor increases initially and then flattens out. The Stability Rate r_s doesn't affect the mean ETH change directly but as r_s increases, it becomes prohibitively expensive to hold cETH, and as a result, the investor converts its cETH to ETH which causes an increase in mean ETH change. As all of the cETH is converted to ETH, a further increase in r_s does not affect the mean ETH change. Also, a change in r_s does not directly affect an investor's willingness to buy/sell DAI.

5.3 DAI Settling Price

In this section we analyze the impact of stability rate r_s, transaction fee β, mean USD holdings for the investor population π_{USD}, mean DAI holdings for the investor population π_{DAI}, price of ETH P_{ETH} and investor risk preference on the DAI Settling Price P_{DAI}.

5.3.1 Impact of mean DAI and USD holdings

In Figure 5a, we see that as π_{DAI} in the market increases, P_{DAI} decreases because an increase in DAI supply while keeping the demand constant drives down P_{DAI}. Similarly in Figure 5b, we see that as π_{USD} in the market increases P_{DAI} increases because as investors have more money to spend, they want to invest more in stable assets such as DAI. An increase in demand for DAI while keeping supply constant drives up the P_{DAI}.

5.3.2 Impact of Investor Risk Preference

In Figure 5c, as we increase the percentage of risk-tolerant investors in the market, P_{DAI} increases. As a risk-tolerant investor prefers to hold more DAI than a risk-averse investor, we can say that a risk-tolerant investor has a tendency to buy DAI and a risk-averse investor has a tendency to sell DAI. As we increase the number of risk-tolerant investors in the market, two things occur. Firstly, with more risk-tolerant investors we have more investors with a higher DAI demand which increases the total DAI demand in the market. Secondly, with less risk-averse investors we have fewer investors willing to sell DAI which reduces the total DAI supply in the market. These two factors are sufficient to drive up P_{DAI}.

5.3.3 Impact of stability rate

In Figure 6a, we see that as r_s increases from 0.02 to 0.1, with π_{USD} = \$325 & π_{DAI} = \$300, mean P_{DAI} increases. This is because with an increase in r_s, people are less likely to open a CDP to mint DAI which reduces the DAI supply keeping DAI demand constant. This directly leads to an increase in P_{DAI}.

5.3.4 Impact of ETH price

In Figure 7a, we observe how P_{DAI} changes as we perform the market simulation over multiple days with an external ETH price feed. We observe that the P_{DAI} closely mirrors the changes in P_{ETH}. From Figure 7b we see that the P_{DAI} is highly correlated to P_{ETH}. We also observe that P_{DAI} varies slightly with large P_{ETH} changes showing that the P_{DAI} is resistant to rapid P_{ETH} changes.

6 Conclusions

We have presented DAISIM, the first open-source computational simulation of the single-collateral DAI stablecoin from MakerDAO. The simulation models investors as rational portfolio optimizers and simulates DAI trading on a market to determine the DAI price as a function of various relevant parameters. In future work this simulation could be used to develop automated mechanisms to steer or control the price of DAI by modifying relevant control parameters. We also plan to extend DAISIM to handle the newer multi-collateral version of DAI that has been introduced more recently.

References

1 Jeremy Clark, Didem Demirag, and Seyedehmahsa Moosavi. Demystifying stablecoins. *Queue*, 18(1):39–60, 2020.
2 Wanyun Gu and Tanay Kothari. Simulating stablecoin systems with latent market confidence index. *Available at SSRN*, 2018.
3 Lewis Gudgeon, Daniel Perez, Dominik Harz, Arthur Gervais, and Benjamin Livshits. The decentralized financial crisis: Attacking defi. *arXiv preprint arXiv:2002.08099*, 2020.
4 Robert A Haugen and Robert A Haugen. *Modern investment theory*, volume 5. Prentice Hall Upper Saddle River, NJ, 2001.
5 R K Lyons and G V Natraj. What keeps stablecoins stable? Online at `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3508006`. Accessed: 2020-05-03.
6 MakerDAO Project. Makerdao whitepaper: Overview of the dai stablecoin system. Online at `https://makerdao.com/en/whitepaper/#an-overview-of-the-maker-protocol-and-its-features`. Accessed: 2020-12-18.
7 A Mundt, D Harz, L Gudgeon, J Y Liu, and A Minca. Stablecoins 2.0: Economic foundations and risk-based models. `arXiv:2006.12388v3[econ.GN],21Oct2020`.
8 A Mundt and A Minca. (in)stability for the blockchain: Deleveraging spirals and stablecoin attacks. `arXiv:1906.02152v2[q-fin.TR],4Jun2020`.
9 A K Mundt and A Minca. While stability lasts: A stochastic model of stablecoins. `arXiv:2004.01304v1[q-fin.TR],2Apr2020`.
10 Defi pulse: Maker. Online at `https://defipulse.com/maker`. Accessed: 2019-12-18.
11 The Blockchain Team. The state of stablecoins. report online at `https://www.blockchain.com/ru/static/pdf/StablecoinsReportFinal.pdf`. Accessed: 2019-12-18.
12 The Reserve Project. The state of stablecoins 2019: hype vs reality in the race for stable, global, digital money. report online at `https://reserve.org/stablecoin-report`. Accessed: 2019-12-18.

TimeFabric: Trusted Time for Permissioned Blockchains

Aritra Mitra
University of Waterloo, Canada

Christian Gorenflo
University of Waterloo, Canada

Lukasz Golab
University of Waterloo, Canada

S. Keshav
University of Cambridge, UK

Abstract

As the popularity of blockchains continues to rise, blockchain platforms must be enhanced to support new application needs. In this paper, we propose one such enhancement that is essential for financial applications and online marketplaces – support for time-based logic such as verifying deadlines or expiry dates and examining a time window of recent account activity. We present a lightweight solution to reach consensus on the current time without relying on external time oracles. Our solution assigns timestamps to blocks at transaction validation time and maintains a cache reflecting the effects of recent transactions. We implement a proof-of-concept prototype, called TimeFabric, in Hyperledger Fabric, a popular permissioned blockchain platform, and experimentally demonstrate high throughput and minimal overhead (approximately 3%) of maintaining trusted time. We also demonstrate a 2x performance improvement due to the cache, compared to reconstructing account histories from the ledger.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Permissioned Blockchain, Timestamp, Clock, Sliding Window, Hyerpleger Fabric

Digital Object Identifier 10.4230/OASIcs.FAB.2021.4

Supplementary Material *Software (Source Code)*:
`https://github.com/aritramitra14/fabric/tree/timefabric`

1 Introduction

Blockchain systems have received substantial interest due to their ability to maintain a trusted transaction log in a decentralized environment. The earliest platform, Bitcoin [12], allowed the exchange of digital currency among peers in a distributed network. Ethereum [20] then introduced smart contracts, which are Turing-complete stored procedures that expanded the applicability of blockchains beyond cryptocurrencies into finance [11] [6], supply chain management [14] and healthcare [1]. Recently, *permissioned* systems such as Hyperledger Fabric [2] have been proposed for enterprise settings in which only authenticated entities participate in the network.

When processing transactions, blockchain systems must accomplish two goals: consensus on the order of transactions and consensus on the validity of transactions. In early blockchains such as Bitcoin and Ethereum, the miner selected to create a block provides consensus on order, and validity is independently verified by each peer in the network. However, in

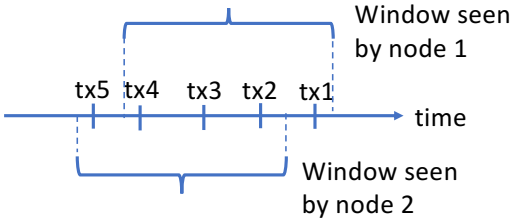


Figure 1 Different sliding windows seen by nodes with different clocks.

permissioned blockchains such as Fabric, consensus on order is obtained by using an ordering service and consensus on validity by using a subset of peers in the network (details in Section 2).

In simple cases, transaction validity can be determined based on account balance alone. Many blockchain systems thus use an account-based data model, in which each peer maintains the current state of each account in a so-called *state database*. This allows peers to validate transactions without having to reconstruct account balances from the entire history stored in the ledger. However, as permissioned blockchains gain traction in enterprise settings, blockchain systems must be enhanced to support new application needs. In this paper, we target applications such as financial services and online auctions and marketplaces, in which determining transaction validity is more complex and depends on *time*.

For example, assume a decentralized retail setting with a blockchain platform operated by manufacturers, sellers and regulators. The platform must not allow the participating entities to manipulate timestamps in an attempt to sell expired products. Furthermore, in a financial setting, a bank may allow an overdraft (i.e., allow a withdrawal despite insufficient funds) if an account is in good standing based on recent transactions. Thus, access to a time *window* of recent account activity is required when executing these transactions.

To validate transactions whose correctness depends on time, a common solution is to obtain the current time from an external trusted oracle, along with the oracle's certificate of the current timestamp. This allows each peer to establish validity, and for all peers to come to the same deterministic conclusion. However, this approach breaks down when validity is *independently* determined by multiple peers, as is the case in permissioned blockchains such as Fabric or Corda [5]. In these situations, we need to obtain consensus on the current time among the endorser peers as a precondition to obtaining consensus on validity. Otherwise, it may be impossible to agree on the transaction outcome. For example, when processing an overdraft transaction, peer nodes with different current times may consider a different window of recent activity. We show an example in Figure 1, with two nodes and five recent transactions. Node 1 considers a window with transactions tx1 through tx4. Node 2 uses a different current time and considers a different window, with transactions tx2 through tx5.

To address the above issues in support of smart contracts with time-based logic, we make the following contributions.

1. *Trusted time for time-based transactions:* Instead of relying on external time oracles, we propose a light-weight consensus mechanism for time that assigns a trusted timestamp to each block. Block timestamps can then be used by the network to deterministically execute time-based smart contracts.
2. *Data layer support for time-based transactions:* We extend the account-based data model to store a sliding window of recent states, effectively maintaining a cache reflecting the effects of recent transactions. If a peer node needs to examine the recent history of an account, it can access the cache instead of reconstructing the account history from the ledger.

3. *Implementation and experimental evaluation:* We implement our solution, called TimeFabric, in Fabric 1.4, and experimentally verify that the overhead of maintaining trusted time is low (under 3%) and that the cache reduces the time to retrieve a window of recent history by a factor of two. Notably, we make minimal changes to Fabric's transaction processing methodology and we preserve Fabric's modular design, which allows different consensus algorithms to be plugged in without affecting transaction execution. The TimeFabric source code is publicly available at `https://github.com/aritramitra14/fabric/tree/timefabric`.

While we use Fabric in our proof-of-concept implementation, our solution generally applies to any blockchain in which multiple entities independently judge the validity of transactions. Thus, we allow a migration path for these types of blockchains if they do not wish to trust an external oracle to validate time-based transactions.

The remainder of this paper is organized as follows. Section 2 provides background information, including an overview of Hyperledger Fabric, Section 3 presents our solution, Section 4 discusses the experimental results, Section 5 reviews previous work, and Section 6 concludes the paper with directions for future work.

2 Background

Blockchain platforms can be categorized as public, or permissionless, and private, or permissioned; the former allows anyone to join the network whereas a private blockchain, commonly used in enterprise collaborations, includes a *membership service* that only allows authenticated entities to participate in the network. However, the authenticated entities do not have to fully trust each other.

Public blockchains such as Bitcoin and Ethereum follow an *Order-Execute* (OE) transaction model. Transactions are first ordered using a protocol such as Proof of Work, and then are executed sequentially by each node. In contrast, Fabric follows an *Execute-Order-Validate* (EOV) model, alternatively referred to a *Simulate-Order-Validate-Commit* model [17], in which transactions are executed in parallel in a sandboxed environment, ordered, and validated before being committed to the ledger. We explain the details below, and we summarize the transaction processing workflow in Figure 2 (ignore the steps marked in red, which correspond to our modifications in TimeFabric and will be discussed later).

2.1 Hyperledger Fabric Overview

Entities participating in a Fabric network are called nodes and can be categorized as *peers* and *orderers*. Peers execute smart contracts, called chaincode in Fabric. Orderers, collectively referred to as the *ordering service*, are responsible for transaction ordering and creation of blocks. Each peer maintains a local copy of the ledger as well as a *state database* (LevelDB by default), which is a key-value representation of the current state of the ledger. A record in the state database contains three pieces of information: a key (e.g., account ID), a value (e.g., the current account balance), and a version number. The state database is used during transaction processing; for example, it can be used to determine whether a given account has a sufficient balance to make a purchase without having to retrieve all the transactions for this account from the ledger. Whenever a transaction (i.e., the execution of a smart contract) is committed to the ledger, the effects of the transaction are persisted in the state database. That is, the new values are written to the database and the corresponding version numbers are incremented. Old versions are eventually discarded from the state database by a background garbage-collection process.

Fabric's Execute-Order-Validate transaction processing protocol proceeds as follows.

2.1.1 The Execute Step

Client applications submit *transaction proposals* to the Fabric network (step 1 in Figure 2). A subsets of peers, called *endorsers*, concurrently simulate the execution of the corresponding smart contracts in a sandboxed environment, i.e., without persisting the effects in the state database. Two such endorsers are shown in Figure 2. Each endorser then sends a response to the client application if the corresponding smart contract was successfully simulated. The response contains the endorser's signature as well as a *read set* and *write set*, which consist of the keys and their version numbers that were read from the state database, and keys (plus their new values) that were updated, respectively, during the simulated execution of the transaction proposal. The write sets thus capture the effects of transactions that must eventually be reflected in the state database.

2.1.2 The Order Step

An endorsement policy, set by the network, specifies the number of endorsements a transaction needs. After a client application receives the required number of endorser responses (step 3 in Figure 2), it sends the transaction proposal, with endorsements attached, to the orderers (step 4 in Figure 2). The orderer nodes run a consensus protocol to determine the order of transactions received from various client applications. Fabric allows various consensus protocols to be plugged into the ordering stage (e.g., Kafka or Raft), with crash-fault (rather than Byzantine fault) tolerant protocols used in practice since the participants in a permissioned blockchain system are known and incentivized to behave honestly. Transactions, with endorsements attached, are segmented into blocks; a block is created if the maximum number of transactions per block (set by the application) arrive or if a block timeout period is exceeded (the default block timeout in Fabric is two seconds). Blocks are then disseminated to the peers (step 5 in Figure 2). Note that orderers are only responsible for ordering the transactions and batching them into blocks; they do not examine transaction contents for correctness or validity.

2.1.3 The Validate Step

Finally, peers serially *validate* (endorser signatures and read-write sets of) transactions in a block, and, upon successful validation, persist the effects of transactions in the local state database and append the block to the local copy of the ledger (step 6 in Figure 2; committer peers are the non-endorsing peers). Transaction validation succeeds if the version numbers of the keys in the transaction read sets are the same as the current version numbers in the state database.

Validation is required because transaction proposals are executed in parallel during the initial Execute stage, and thus transaction conflicts may arise. For example, suppose two client transactions wish to withdraw money from the same account, with key 123, whose current version number in the world state is 100. Suppose no other transactions in this block touch this key. The read sets of both of these transactions include key 123 with version number 100. During validation, the first of these transactions will be committed because key 123 still has version number 100 in the state database (it has not been modified by any other transaction from this block). After the first transaction is committed, the new version of key 123 will be 101. Now, the second transaction fails because the version number of key 123 in its read set is 100, but it is 101 in the state database. Failed (or aborted) transactions are marked as such and remain in the block.

Transaction validation prevents read-write and write-write conflicts. In a given block, at most one transaction can write to a key, and if another transaction only reads this key without writing to it, this transaction must be ordered before the one that writes to this key (otherwise, the version numbers will not match). This prevents double-spending, but may also prevent legitimate transactions from being committed. In the above example, even if there is sufficient balance in account 123 for both withdrawals, only the first transaction will succeed. The second transaction will need to be re-submitted by the client application for re-endorsement, and will be put in a new block for validation.

Note that once the transactions in a block have been ordered, they are sequentially validated by each peer in the same order. As a result, each peer makes the same transaction commit (or not) decisions, and thus each peer stores the same version of the ledger and the state database. Also note that smart contracts are not re-executed during validation; only their effects are persisted in the state database.

2.2 Timestamps and Account Histories in Hyperledger Fabric

We now outline existing Fabric functionality related to transaction timestamps and transaction histories. Clients can set transaction timestamps when creating transaction proposals, which are recorded in the transaction header and ultimately appear in the blockchain. Fabric exposes a method *GetTxTimestamp(transaction_id)* for chaincode to access transaction timestamps. However, transaction timestamps are *not* endorsed during the execute step or verified during the validate step.

Furthermore, chaincode can call *GetHistoryForKey(key)* to obtain a history of *all* values for a given key, along with the transaction timestamps corresponding to each update (querying a specific time window is not supported). This is done by consulting an index that points to (the blocks containing) transactions that have modified a given key. These transactions are then retrieved from the blockchain to compute the history, which is expensive. This index is stored in the state database, in addition to the keys and their latest values.

3 Our Solution

In this section, we present our solution to support smart contracts with time-based logic. Our design goals are:

1. to provide a trusted and consistent time reference for peers that validate transactions, without the need to consult external oracles,
2. to process transactions that reference this trusted time efficiently, with minimal overhead,
3. and to preserve the underlying blockchain system architecture as much as possible while making minimal modifications.

We address goal #1 in Section 3.1 and goal #2 in Section 3.2. We then describe implementation details of our proof-of-concept, TimeFabric, which is based on Hyperledger Fabric 1.4 (Section 3.3), followed by a discussion of TimeFabric's failure model compared to the underlying Fabric (Section 3.4).

3.1 Trusted Time

Our approach to maintain trusted time consists of the following steps.

1. **Validation of transaction timestamps**. Peers that validate transactions are given an additional responsibility: to ensure that the transaction timestamp is within δ time units of the current trusted time (this will be defined shortly). Thus, transactions with

timestamps too far into the past or the future will be rejected. The value of δ can be set in the corresponding smart contract code, and we will discuss how to choose an appropriate value for δ in Section 3.3.

2. **Assigning trusted block timestamps**. Additionally, peers that validate transactions need to assign *block timestamps*. In particular, they set the block timestamp to be the most recent or the median transaction timestamp within the block, among transactions that have been validated and have not been rejected[1]. However, if this timestamp is older than the timestamp of the previous block, then the timestamp of the new block equals the timestamp of the previous block plus a small constant ϵ (in our implementation, $\epsilon = 1$ millisecond). To do this, when validating transactions within a block, each peer must keep track of the latest transaction timestamp seen, and finally append it to the block. Block timestamps become part of the blockchain and are included in the block hash for immutability.
3. **A heartbeat mechanism**. Suppose no transactions arrive for some time, say, one minute. Then, when a transaction finally arrives, its timestamp would be a minute into the future relative to the timestamp of the latest committed block. To ensure that trusted time moves forward, we require a "dummy" client that sends mock transactions even during periods of inactivity. A mock transaction updates a reserved "dummy" account with a random value, and its transaction timestamp equals the local time of the client. We will discuss how often these mock transactions need to be sent in Section 3.3.

Time thus advances one block at a time, based on *validated* transaction timestamps, giving every peer a common time reference. At any point, the current trusted time, or *block time*, as required for transaction validation, is the time of the latest block that has been committed to the ledger. The block time is used for any reference to time in a smart contract.

Our solution uses one timestamp per block rather than one timestamp per transaction for several reasons. The first is efficiency: in general, obtaining consensus on a value in a decentralized setting is expensive. The second is to ensure a monotonically increasing time reference: transactions within a block may not be ordered by their timestamps.

3.2 Data Layer Support

Given our notion of trusted time, we create three methods that are accessible to smart contracts.

1. *GetTimenow()* returns the current block time.
2. *GetHistoryRangeForKey(key, start, end)* returns a history of values for a given key with block timestamps in the interval $[start, end]$.
3. *GetStateWindow(key,window_length)* is a wrapper over *GetTimenow()* and *GetHistoryRangeForKey()*. It obtains a history of values for a given key with block timestamps in the interval $[current\_time - window\_length, current\_time]$.

GetTimenow() can be used when validating transaction timestamps, which can then be used during smart contract execution, e.g., to verify if deadlines are met. A simple implementation of this method is to extract the timestamp from the latest block in the ledger. Another option is to cache the block time at the validating peers.

GetHistoryRangeForKey() is meant to be used during smart contract execution to retrieve recent histories. A naive implementation is to reconstruct the account history from the ledger, which is expensive. Our solution is to maintain a cache capturing the effects of recent

[1] We will discuss the choice between maximum and median transaction timestamps in Section 3.3.

transactions. First, we assume that validating peers use the account-based data model and already maintain a key-value state database with the current account states. Additionally, we require each validating peer to maintain a *cache database*. Each record in the cache database is a key-value pair. The key is a concatenation of the corresponding key in the state database and the block timestamp of the transaction that updated the key. The value is the corresponding updated value. For example, suppose that key 123 is updated to have value 50 by a transaction belonging to a block with Unix timestamp 1607994614. The corresponding key-value pair in the state database is $(123, 50)$, plus the version number. The key-value pair in the cache database is $(123 : 1607994614, 50)$.

To populate the cache database, we make another modification to the validating peers. In addition to writing key-value pairs to the state database, we require the validating peers to write key-value pairs (with timestamps concatenated to the key) to the cache database. *GetHistoryRangeForKey()* can then be answered via a range query on the key against the cache database. For example, a query for the history of key 123 between Unix timestamps 1600000000 and 1607994614 becomes a range query against the cache database for keys in the range from $123 : 1600000000$ to $123 : 1607994614$.

There is one important distinction between the state database and the cache database. In the former, values of existing keys are updated since only the most recent value needs to be stored. In contrast, the cache database is append only: an update of the state database results in a new key added to the cache database since keys in the cache database include block timestamps. Thus, if not *pruned*, the cache database will grow indefinitely.

To avoid this problem, we borrow a common solution, similar to the calendar queue, used by data stream management systems to maintain sliding windows [7]. The idea is to partition, or shard, the cache database by time, and, instead of deleting individual records over time, periodically drop the oldest part. For example, suppose that an application requires a 7-day history. Peers may partition the cache database by day. Every day, a new part is added to store new records generated that day, and the oldest day is dropped. The window length and the number of shards are parameters that may be decided by the network along with other blockchain configuration parameters.

Technically, there is no limit on how much history can be stored in the cache database. However, to ensure high transaction throughput, it should be ensured that the cache database (and, of course, the state database) fits in memory.

3.3 TimeFabric Implementation

We now discuss the implementation details of TimeFabric, which is based on Hyperledger Fabric 1.4. Our modifications to Fabric's transaction validation pipeline are shown in red in Figure 2 and are explained below.

In the validation step, peers have two additional tasks:

1. In Fabric, transactions are validated by committer peers once a block is received from the ordering service. Each transaction in the block is unpacked and validated by the committer peer in parallel using multiple Go routines. At this stage, we additionally identify the maximum or median timestamp across the valid transactions, and we insert this timestamp into the block metadata.
2. We add a cache database that must by maintained by the peers over time (i.e., periodically create new shards and drop old shards). We modify the block commitment stage to add this new database (which is a hashmap in our implementation). Each transaction in a block is unpacked to extract the write-sets. We then compute new keys to be written to the cache database by concatenating the timestamp to the original key, and we insert this key-value pair to the cache database.

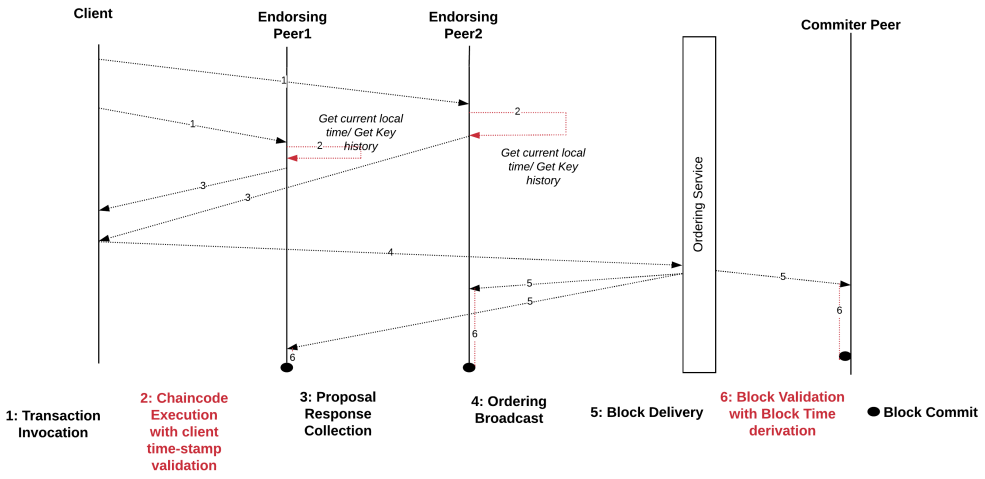


Figure 2 Transaction flow in Hyperledger Fabric. In TimeFabric, we make changes in steps 2 and 6, shown in red.

In the execute step, endorsing peers have one additional task: validate transaction timestamps by comparing them to the current block time (via the new method *GetTimenow()*). We implemented this method in the Fabric RPC server by querying the ledger to retrieve the latest block, and extract the block timestamp from the block metadata. We considered caching the block time at the endorsers, but the performance gains were minimal since the latest block is already cached in memory by Fabric.

Additionally, smart contracts have access to recent histories via *GetStateWindow()*, which queries the cache database (and the state database for the latest value). In our implementation, the partitioned cache database consists of separate instances of hashmaps, and *GetHistoryRangeForKey()* is handled by issuing a range query against each instance.

We note a subtle but important issue related to read set validation in TimeFabric. Assume a transaction that fetches a window of recent account history, including the current balance, for account 123, and updates the account balance if the account history satisfies some condition. This transaction can use *GetHistoryRangeForKey()*, which fetches a window of recent history of key 123 from the cache database. However, we wish to re-use Fabric's transaction conflict logic during transaction validation. For example, this transaction should not be committed if another transaction from the same block had updated account 123. To identify these types of conflicts, we modify *GetHistoryRangeForKey()* to also fetch the latest key-value pair from the state database (in addition to fetching the history of this key from the cache database). Next, only the records read from the state database are validated to make sure the version numbers match; records in the cache database are never updated (only new keys are added), so their version numbers are always '1' and do not need to be validated. In our example, only the latest version of key 123 obtained from the state database is validated, and the window of recent history of key 123 obtained from the cache database is not. However, the transaction's read set contains all keys read from the state database and the cache database for auditability (recall that the read and write sets becomes part of the blockchain).

Finally, there are no modifications to Fabric's ordering step. This satisfies design goal #3: Fabric's modular design suggests that orderers should only be responsible for ordering transactions. To maintain compatibility with various plug-and-play consensus algorithms for the ordering step, our modification are restricted to the endorsers and the validators.

In addition to the above changes to Fabric, our implementation of TimeFabric requires the addition of a dummy client for heartbeats (recall Section 3.1). To decide when to send a heartbeat, we observe that Fabric orderers disseminate a new block when it is full (contains the maximum number of transactions) or if it contains at least one transaction and no other transaction has arrived for two seconds (the default timeout period). Thus, we configure the dummy client to send a mock transaction every two seconds.

Now, recall the δ parameter for transaction timestamp validation. In our implementation, block time is permitted to be two seconds in the past in the worst case, if no new transactions have arrived and a heartbeat transaction was just generated. To account for this delay and network delays between clients and the Fabric/TimeFabric network, we set δ to two seconds plus the expected network delay. Large values of δ should be avoided to prevent malicious clients from submitting transactions with future timestamps and therefore advancing the block time too quickly. On the other hand, delays must be taken into account to ensure that legitimate transactions are not rejected, although a client who experiences an unusually long delay can always resubmit its transaction. Note that in a permissioned blockchain, even malicious clients need permission to access the blockchain by requesting access credentials from a membership service. Hence, clients exhibiting malicious behaviour can be ejected from the system. Nevertheless, if one wishes to err on the side of caution, malicious behavior can be reduced by setting the block timestamp to the median value of the timestamps in a block, rather than the maximum, since the median is harder to manipulate.

Finally, recall that our solution assigns one timestamp per block rather than one timestamp per transaction. However, observe that block timestamps alone produce totally ordered key histories in TimeFabric because Fabric's validation step ensures that a key can be updated at most once per block (recall Section 2).

3.4 TimeFabric Failure Model

In this section, we discuss the impact of our modifications on the failure model of the system. In Fabric, the membership service that authenticates the participating entities must be fault-tolerant, and this does not change in TimeFabric. Similarly, we do not change Fabric's ability to plug in various ordering algorithms, which can be crash-fault or Byzantine-fault tolerant, as desired by the application.

We also retain Fabric's endorsement policies, specifying the number of endorser responses required by a client transaction. Having to collect multiple endorser responses prevents collusion between client applications and an endorser, and this extends to TimeFabric's endorsement of transaction timestamps.

Furthermore, the ledger is replicated among the peers, each block contains a hash pointer to the previous block to ensure immutability, and every peer independently validates transactions and appends new blocks to the chain, as in Fabric. TimeFabric adds block timestamping to each peer's responsibilities, resulting in the same failure model: any inconsistencies at one peer can be easily detected by comparing other peers' ledgers. In contrast to Fabric, TimeFabric peers also maintain a cache database. In case of a crash fault, a peer can rebuild its cache database by unpacking transactions from recent blocks. (Similarly, a peer (in Fabric and TimeFabric) recovering from a failure can rebuild its state database from the ledger).

Finally, as for the mock client that implements the heartbeat mechanism, we install one such client at each endorser for crash-fault tolerance.

Table 1 End to End Throughput.

Fabric 1.4	TimeFabric
2927 ± 136	2831 ± 196

4 Experiments

In this section, we experimentally evaluate TimeFabric, which we implemented in Fabric version 1.4 (our modifications remain compatible with the recent release of version 2 since we do not change Fabric's modular design). We use six local servers connected through a 1Gbit/s switch. In practice, Fabric deployments may be geo-distributed, but our experiments focus on commit overhead and database access times at individual peers, which are independent of how the peers nodes are distributed. Each server is equipped with two Intel Xeon CPU E5-2620 v2 processors at 2.10 GHz, and 64 GB of RAM. Our experiments are conducted using Fabric binaries and we only use docker containers for the chaincode runtime environment. All tests are conducted with non-conflicting and valid transactions to ensure that all transactions go through the entire lifecycle (endorsement, ordering, validation and commit) without being aborted. This helps us to evaluate the worst-case performance of the system in terms of transaction throughput.

Our experiments have two goals: *1)* evaluating our implementation of trusted block time and *2)* evaluating the performance of the new API to obtain the current block time and a recent history for a given key. To evaluate the implementation of block time, we measure the overhead introduced by our changes to the Fabric transaction processing lifecycle, specifically, the overhead incurred by committer peers. To isolate this overhead, we send pre-endorsed transactions to the orderer and measure the transaction throughput at committer peers. We also measure the latency of the block time, i.e., how far back it is compared to the wall clock, for various block sizes. To evaluate the performance of the new API, we measure the runtime overhead of our new method *GetTimenow()*, and we compare our method *GetStateWindow()* to Fabric's *GetHistoryForKey()*.

4.1 Block Time Implementation

Committer Overhead. In this experiment, we compare the transaction throughput at the committer peer for Fabric 1.4 and TimeFabric. We use a single endorser and a single committer peer, a solo orderer, and four client machines that generate transaction proposals. We first send 25,000 transaction proposals from each client to the endorser and obtain the proposal responses. We then set up 25 threads in each client (totaling 100 threads) to send a total of 100,000 transactions to the orderer. Subsequently, we measure the total time by the committer peer to commit all the blocks to the ledger and then derive the throughput. Following prior work on improving the throughput of Fabric [8], we set the block size to 100. We conduct 30 runs and report the mean throughput and the standard deviation in Table 1. This experiment shows that our changes only add about 3% overhead to the block validation and commit process.

Block Time Latency. In this experiment, we record the time difference between an endorser's local clock and the block time, i.e., the time assigned to the latest committed block. We expect lower latencies for smaller block sizes, with size corresponding to the number of transactions per block. Since we want to measure the latency from the point of view of a single endorsing peer, we use a single peer with a solo orderer and one client node. We

Table 2 Block time latency for various block sizes.

Block Time Latency					
Block Size	**50**	**75**	**100**	**125**	**150**
Mean (ms)	97	186	192	244	480
Median (ms)	90	131	175	223	285
Range (ms)	51-2343	85-2445	103-2591	104-2603	164- 2670

execute a smart contract that calls our method, *GetTimenow()*, to obtain the current block time. The smart contract then calculates the difference between its local clock and the block time, and writes this difference to a new key in the state database. That is, the sole purpose of this smart contract is to record block time latencies. We execute 25000 such transactions for varying block sizes, and we compute the mean and median latencies as well as the latency range, as seen by these transactions.

We show the results in Table 2. We observe that mean latency increases with the block size. However, as we noted earlier, prior work observed the highest throughput at a block size of 100. Given this block size, the mean block time latency is under 200 milliseconds. Note that these result correspond to a scenario in which transactions arrive continuously and blocks fill up naturally, without the need for heartbeat transactions to create new blocks. As we discussed earlier, if transactions stop arriving, then the block time latency increases to just over two seconds, which is the timeout period plus the time to commit the block with the heartbeat transaction.

4.2 Time Query Performance

Endorser Overhead of GetTimenow(). In this experiment, we measure the performance of *GetTimenow()* by monitoring the endorsement time for transactions on a single peer. For this, we implement a smart contract that corresponds to a retail purchase transaction for a perishable product. The transaction is endorsed if its timestamp is earlier than product expiry date; if so, the chaincode additionally decrements the available quantity of the product, which involves one key read and one key write. In TimeFabric, the chaincode calls *GetTimenow()* to obtain the time. In Fabric, the chaincode simply obtains the local time at the endorser. We send a series of transactions to the endorsing peer from a single client and calculate the total time for obtaining all the responses. We repeat this experiment by varying the number of transactions and recording the endorsement time.

We show the results in Figure 3, which reveals that the performance overhead of *GetTimenow()* is statistically insignificant.

Endorser Overhead of GetStateWindow(). We compare the performance of *GetStateWindow()* in our implementation against *GetHistoryForKey()* in Fabric 1.4. Since Fabric fetches key histories directly from blocks, we expect a performance improvement in our implementation that uses the cache database for recent history. We start by loading the state database with 500 keys, and then each key is updated between 10 an 200 times, depending on the experiment. The chaincode for this experiment corresponds to a financial overdraft transaction: it reads the full history of the key (between 10 and 200 values, depending on the experiment, to simulate different window lengths) and writes a new value for this key if the history shows that this account has maintained some minimum balance. We use a single client to execute the transactions for all 500 keys and we record the total time for collecting all proposal responses from a single endorser.

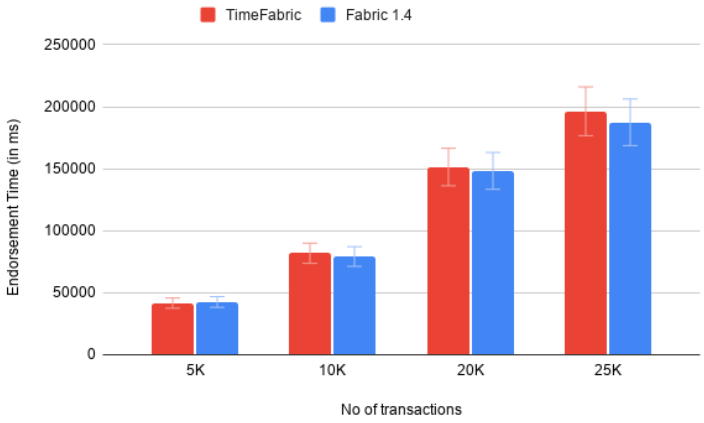


Figure 3 Endorsement time comparison.

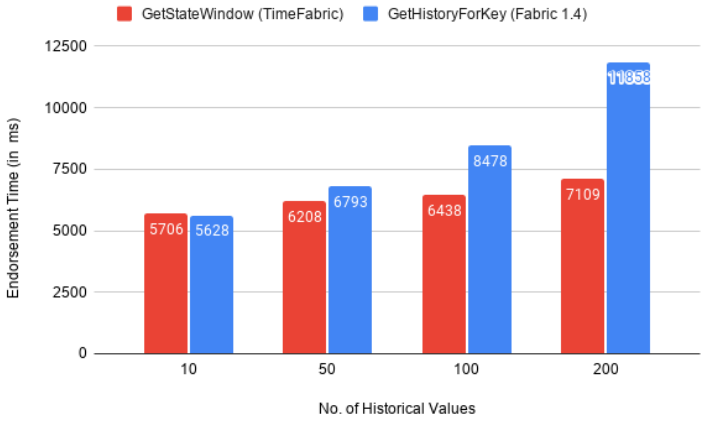


Figure 4 Endorsement time for window queries.

We show the results in Figure 4. The performance of Fabric's *GetHistoryForKey()* degrades as the window length increases since there is more history to retrieve. On the other hand, the running time of our implementation of *GetStateWindow()* increases only slightly as the window length increases. For a window of 200 historical values, TimeFabric is nearly twice as fast as Fabric 1.4.

5 Related Work

Hyperledger Fabric is actively being developed and various performance optimizations have recently been proposed, including adding parallelism and caching to the transaction processing pipeline[8, 17]. Our solution is compatible with these optimizations since our modifications leave Fabric's modular structure intact.

Perhaps the closest work to ours is that of Zan and Xu [22], which proposes to add a separate global clock node to Fabric, whose purpose is to periodically synchronize the local clocks of endorsers, orderers and committers during the transaction lifecycle. Although this approach can improve the accuracy of local clocks, it cannot fully synchronize them, as we do using block time. Additionally, our solution goes one step further to ensure that time-related operations such as sliding windows can be done efficiently.

FabricSharp [16] is a proposal to add timestamp-based optimistic concurrency control to Fabric. However, instead of using physical time, FabricSharp uses block sequence numbers and it does not solve our problem of maintaining trusted time for use by smart contracts. This precludes, for example, applications that depend on a time window.

LineageChain [15] extends Fabric by exposing *provenance* information, i.e., key histories, to smart contracts. For efficiency, LineageChain maintains an index over the provenance tree. This is conceptually similar to our use of the cache database to speed up sliding window queries. However, LineageChain does not offer a notion of time and its provenance queries do not support sliding windows.

An index to speed up temporal queries in Fabric was proposed in [9]. Account histories are stored in blocks on the file system and the index consists of pointers stored in the state database. The pointers identify blocks that contain transactions for a given account whose timestamps are within a given interval. The index is meant for off-line analytics over account histories. In contrast, our solution maintains an in-memory time window of the effects of recent transactions for use by smart contracts. Furthermore, our solution includes a notion of trusted time, whereas the index proposed in [9] was based on unverified transaction timestamps.

Next, we review time-related concepts in permissionless blockchains such as Bitcoin and Ethereum. In systems that use Proof of Work for consensus, block timestamps are usually set by the miners when forming new blocks. Ethereum enforces a protocol to not accept a new block if the timestamp provided by the miner is earlier than timestamp of the previous block. Additionally, if a block timestamp is set in future, other mining nodes may not want to build on that block, resulting in forks. Bitcoin's protocol is to not propagate a block whose miner-assigned timestamp is earlier than the median of the previous 11 blocks or more than two hours into the future. We incorporate similar constraints in our solution: block timestamps must be monotonically increasing, and they are based on verified transaction timestamps that cannot be too far in the past or the future.

While protocols exist in permissionless systems to reject blocks with suspicious timestamps, there has also been work describing attacks related to time manipulation [19],[4],[21],[3]. These works highlight vulnerabilities but do not propose solutions, except [18] – in that work, focusing on Bitcoin, a verifier node requests a timestamping authority (TSA) to validate block timestamps. The verifier node unpacks the block header, has the TSA timestamp the block, and includes the hash of the data in a subsequent transaction that is included in the next block. The next block header is again unpacked, timestamped by TSA and returned to the verifier. As a result, any discrepancy in block time can be found by comparing the block time (set by the miner) against the two timestamps obtained from the TSA. Our solution avoids a timestamping authority and instead leverages the additional trust inherent in permissioned blockchains by using client transaction timestamps (properly verified) as a basis of trusted block timestamping.

Finally, other studies such as [13] and [10] argue that block sequence numbers are intrinsic to blockchains and best represent the temporal progression of a blockchain. Reference [13] specifically states that any reference to an external time oracle violates the decentralized property of a blockchain network. Our solution avoids the use of external time oracles, and, again, leverages the additional trust inherent in permissioned systems to assign block timestamps.

6 Conclusions

In this paper, we presented a method to support smart contracts with time-based logic referencing current time or a time window of recent history. We showed that existing solutions such as querying an external time oracle, break down for systems in which multiple peers independently validate transactions. Instead, our solution assigns trusted block timestamps at transaction validation time, which can then be used by all peers to reach consensus on time-based transaction validity. To ensure that time-based smart contracts can be executed efficiently, our solution also adds a cache database storing a window of recent transactions. We implemented a proof-of-concept prototype, TimeFabric, on top of Hyperledger Fabric. Experimental results show that our modifications add little overhead to the transaction processing pipeline in Fabric and that time-based smart contracts can be executed efficiently by fetching account histories from the cache.

In future work, we plan to investigate new applications that can leverage trusted time and access to sliding windows of account histories enabled by TimeFabric, in areas such as finance, retail, supply chains and online auctions.

References

1 Rishav Raj Agarwal, Dhruv Kumar, Lukasz Golab, and Srinivasan Keshav. Consentio: Managing consent to data access using permissioned blockchains. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC*, pages 1–9, 2020.

2 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

3 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint archive*, 2016:1007, 2016.

4 Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 453–467. Springer, 2017.

5 Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

6 Luisanna Cocco, Andrea Pinna, and Michele Marchesi. Banking on blockchain: Costs savings thanks to the blockchain technology. *Future internet*, 9(3):25, 2017.

7 Lukasz Golab and M. Tamer Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

8 Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 455–463. IEEE, 2019.

9 Himanshu Gupta, Sandeep Hans, Kushagra Aggarwal, Sameep Mehta, Bapi Chatterjee, and Praveen Jayachandran. Efficiently processing temporal queries on hyperledger fabric. In *34th IEEE International Conference on Data Engineering, ICDE*, pages 1489–1494, 2018.

10 Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

11 Larissa Lee. New kids on the blockchain: How bitcoin's technology could reinvent the stock market. *Hastings Bus. LJ*, 12:81, 2015.

12 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

13 Ricardo Pérez-Marco. Blockchain time and heisenberg uncertainty principle. In *Science and Information Conference*, pages 849–854. Springer, 2018.

14 Hubert Pun, Jayashankar M Swaminathan, and Pengwen Hou. Blockchain adoption for combating deceptive counterfeits. *Production and Operations Management*, 2021.

15 Pingcheng Ruan, Gang Chen, Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance for blockchain. *Proc. VLDB Endow.*, 12(9):975–988, 2019.

16 Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference*, pages 543–557, 2020.

17 Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference*, pages 105–122, 2019.

18 Pawel Szalachowski. (short paper) towards more reliable bitcoin timestamps. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 101–104. IEEE, 2018.

19 Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.

20 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

21 Karl Wüst and Arthur Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.

22 Chao Zan and Hai-Chuan Xu. A global clock model for the consortium blockchains. In *International Conference on Blockchain and Trustworthy Systems*, pages 71–80. Springer, 2019.

Dynamic Curves for Decentralized Autonomous Cryptocurrency Exchanges

Bhaskar Krishnamachari
Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA

Qi Feng
Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA

Eugenio Grippo
Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA

Abstract

One of the exciting recent developments in decentralized finance (DeFi) has been the development of decentralized cryptocurrency exchanges that can autonomously handle conversion between different cryptocurrencies. Decentralized exchange protocols such as Uniswap, Curve and other types of Automated Market Makers (AMMs) maintain a liquidity pool (LP) of two or more assets constrained to maintain at all times a mathematical relation to each other, defined by a given function or curve. Examples of such functions are the constant-sum and constant-product AMMs. Existing systems however suffer from several challenges. They require external arbitrageurs to restore the price of tokens in the pool to match the market price. Such activities can potentially drain resources from the liquidity pool. In particular dramatic market price changes can result in low liquidity with respect to one or more of the assets and reduce the total value of the LP. We propose in this work a new approach to constructing the AMM by proposing the idea of *dynamic curves*. It utilizes input from a market price oracle to modify the mathematical relationship between the assets so that the pool price continuously and automatically adjusts to be identical to the market price. This approach eliminates arbitrage opportunities and, as we show through simulations, maintains liquidity in the LP for all assets and the total value of the LP over a wide range of market prices.

2012 ACM Subject Classification Applied computing → Online banking

Keywords and phrases Decentralized Exchange, Automated Market Maker, Decentralized Finance, Dynamic Curves

Digital Object Identifier 10.4230/OASIcs.FAB.2021.5

Related Version *Full Version*: https://arxiv.org/abs/2101.02778

1 Introduction

Since the introduction of Bitcoin as the first peer-to-peer digital cash [15], the birth of different cryptocurrencies has revolutionized the world of finance [21]. As of the time of writing this article, it is estimated that the total cryptocurrency market capitalization is more than $600 Billion, involving thousands of different coins [10].

Traditionally, cryptocurrency exchanges, which use an order book mechanism, are centralized. They suffer from concerns about the concentration of financial power [14] and being prone to a single point of failure, resulting in a potentially significant loss of funds when attacked [3]. Additionally, they also pose a liquidity problem for tokens with a smaller market capitalization resulting in barriers to entry to the financial market [14].

On the other hand, it is difficult to implement the order book model in a decentralized manner in the form of a blockchain smart contract [20] [3]. First, market makers will face high gas costs to execute transactions, regardless of their sizes [20]. Second, it will require a complex matching algorithm to support a variety of order types [3].

Automated Market Makers such as Hanson's logarithmic market scoring rules (LMSRs) are widely used in traditional prediction markets to address the problem of low liquidity and trading volume [13, 19]. An LMSR-based AMM is also used in decentralized prediction markets such as Gnosis [12] and Augur [16]. Given certain similar market characteristics, curve-based Automated Market Makers (AMM) were recently introduced to address the challenges in a currency exchange context. They are currently one of the areas of decentralized finance receiving the most attention. Instead of relying on the traditional market makers to provide liquidity, decentralized exchanges utilizing curve-based AMMs, such as Bancor [14], Uniswap [1], StableSwap/Curve [11] and many others implement a liquidity pool (LP) using smart contracts on a blockchain. In this model, liquidity providers supply single or multiple types of tokens to the designated liquidity pools, and traders exchange against the pools of tokens instead of relying on order matching. The liquidity pool of these AMMs track a pre-defined mathematical function (curve), thus determining how many tokens of one type to provide to a trader in exchange for a certain amount of another. Curve-based AMMs provide a continuous supply of liquidity compared to the order book model. Additionally, depending on the mathematical function (curve) utilized, they can potentially allow for a wide range of exchange prices. However, the token price within a liquidity pool for a given AMM (which we refer to as the pool price) might be different from the market price.

When such a gap occurs on a decentralized AMM-based exchange, arbitrageurs may have the opportunity to buy or sell tokens to set the pool price equal to the market price, restoring equilibrium. However, in some cases, particularly when the market price changes dramatically, the AMM-based LP could lose liquidity with respect to one or more of the assets. We propose in this work a new approach to constructing the AMM by proposing the idea of *dynamic curves*. It utilizes input from a market price oracle to modify the mathematical relationship between the assets so that the pool price continuously and automatically adjusts to be identical to the market price. This eliminates arbitrage opportunities and, as we show through simulations, helps the AMM-based LP maintain liquidity and total value over a wide range of market prices.

The following are the key contributions of this work:

- We present a simple and unified mathematical and conceptual framework (in section 3) describing existing curve-based AMMs and key metrics such as pool price, slippage, divergence loss. It unifies much of what is known about them today. We believe this section will be of independent interest to researchers starting out in this area.
- We focus on the liquidity problem posed by arbitrageurs on existing AMMs, especially when the market price for one of the assets becomes too high, causing asset depletion and value reduction of the liquidity pool.
- We present a new dynamic curve mechanism, which is general enough to be adapted to any monotonic function/curve used on an AMM. This mechanism relies on an external market price oracle and eliminates arbitrageurs. We illustrate the mechanism concretely through generalizations of both constant-sum and constant-product models.
- We present numerical simulations showing the clear advantages of our proposed dynamic curve mechanism in a) retaining greater liquidity in the pool to benefit small traders, b) retaining greater total value in the liquidity pool, and c) functioning effectively over a much larger range of market prices.

The rest of this paper is organized as follows: we present and discuss relevant prior work in section 2. In section 3 we give a unified treatment and definition of key concepts and metrics relevant to decentralized AMMs. In section 4 we propose and describe our new dynamic AMM models. In section 5 we present agent-based simulations and compare the

performance of four different AMMs, including two static AMMs (Constant- Sum AMM, Constant-Product AMM) and their two dynamic generalizations that we introduce in this work. We present concluding comments in section 6.

2 Related Work

Bancor was the first DEX to implement the type of AMM called Bonding Curve, which provides continuous liquidity [14]. In this type of AMM, there is a single token (Bancor Network Token - BNT) used as an intermediate currency. There are separate pools for each non-native currency to be traded against BNT. This model is a little different from the arbitrary two-asset curve based AMM's that we focus on in this paper (though there are significant connections as well). In curve-based AMMs, any two currencies could be traded directly against each other.

Borrowing solutions from the prediction market, Buterin [6] first proposed such a curve-based AMM for a decentralized exchange. Specifically, he proposed the Constant Product Curve. It is a convex curve that takes the form of $x \cdot y = k$, where x and y are the total supply of two tokens in a liquidity pool and k is the product constant. It was subsequently implemented by Adams *et al.* [1] to create Uniswap.

With the shape of a downward-sloping straight line, the Constant Sum Curve [5][7] takes the form of $x + y = k$. x and y are the total supply of two tokens in a liquidity pool, and k is the sum constant. StableSwap/Curve [11] implemented an AMM curve that is a blend of Constant Sum and Constant Product to provide continuous liquidity, price stability and a built-in pool balancing mechanism.

Wang [19] proposed the Constant Ellipse Curve AMM with the general form of $(x-a)^2 + (y-a)^2 + b \cdot xy = C$, in which a and b are constant. One can choose between the concave and the convex curve in the first quadrant [19]. Wang also presents the curve corresponding to the LMSR rule.

Angeris and Chitra analyze such curve-based AMMs, which they refer to as constant function market makers in the general case, i.e., with arbitrarily many tokens [2]. They analyze various mathematical properties of such AMMs, including formulating the optimal arbitrage by traders as a convex optimization problem.

2.1 Performance metrics for AMMs

Slippage and divergence loss are the two main factors contributing to the proposal and adoption of different AMMs. The former is directly tied to the loss of traders, while the latter is directly connected to the liquidity providers' returns.

Slippage is the difference between the expected and actual trade execution price [18], and in the AMM context, it is defined as the gap between the pool price before a trade and the effective price obtained for the trade (see section 3.3). As long as token price changes during trade, slippage incurs. In addition, when large trades happen compared to pool size, slippage increases dramatically, resulting in lower trading profits [9].

Divergence loss, sometimes called impermanent loss, incurs when liquidity providers withdraw liquidity with the presence of a difference in token price before and after trades [17]. If funds are pulled out during a large price swing, liquidity providers will suffer a loss of total asset value, compared to simply holding the assets [8]. Given trades might affect token pool prices and hence divergence loss, it is important to distinguish between regular and arbitrage trading. Divergence loss due to arbitrage trading in closing pool and market price gap can be mitigated by incorporating reliable oracles to protect liquidity providers [4].

2.2 Slippage and Divergence Loss in AMMs

In Bancor, given the dynamic pool price by design, trades experience slippage. Divergence loss incurs as the pool price is intrinsic and relies on arbitrageurs to close price gaps [14]. To protect liquidity providers, Bancor v2 integrates with Chainlink price oracle to reduce divergence loss from arbitraging [4].

In Uniswap, similar to Bancor, the pool price is inherently unstable and the size of trades in relation to pool size affects pool price to different extents. The larger the trades are, the higher slippage and divergence loss can occur.

The constant-sum curve has zero slippage [5, 7] and no divergence loss (as we show in section 3). However, because it has a fixed price and finite liquidity, it is only suitable for stablecoins and could easily be depleted of one of its pool assets; for this reason, it is primarily of theoretical interest [5, 7]. We use it as a baseline in our work.

StableSwap/Curve introduces an invariant that allows trading on a Constant Sum shaped curve when the portfolio is relatively balanced and switch trading to a Constant Product shaped curve when imbalanced [11]. Such a design allows much lower slippage and divergence loss but is only applicable to stablecoins as the price of the desired trading range is always close to 1.

The constant ellipse curve introduced by Wang [19] has a fixed price range compared to that of a Constant Product Curve and thus a fixed range of slippage and divergence loss. Wang also concludes that the LMSR curve would not be suitable for exchanges if the numbers of the two tokens are not balanced in the liquidity pool.

The proposal in this paper presents an approach for AMM-based decentralized exchanges using dynamic curves that eliminates the possibility of arbitrage and thus any divergence loss. Instead, as we show, depending on the chosen family of curves, any slippage loss incurred by traders is converted to an equivalent gain for the liquidity providers. A dynamic version of the constant sum curve is a special case of our proposed solution, and in that case, there is no slippage loss at all.

2.3 Simulation

There are two simulations conducted respectively on StableSwap/Curve and Uniswap v1 to evaluate the DEXes performance. Egorov [11] suggests that StableSwap/Curve generates 312% APR and 0.06% fee per trade for liquidity providers with total liquidity of $30000 in DAI, USDC and USTD over 6 months. Angeris *et al.* [3] conduct an agent-based simulation to test the hypothesis that Uniswap has a robust market mechanism with little arbitrage opportunities under various market conditions. Three types of agents, including profit-maximizing arbitrageurs, traders with exogenous motives and liquidity providers (both active/Markowitz portfolio optimizing and passive), interact in the Uniswap and a stochastic reference markets. The results show that Uniswap tracks market prices closely in different market environments, and Constant Product Curves have the potential to be price oracles [3].

3 Background - AMM Curves and Key Metrics

Consider a liquidity pool with two coins, whose amounts are denoted by x and y. For convenience, we will refer to these two tokens as X and Y. The AMM will allow the exchange of one token for another following a given function f as follows:

$$y = f(x) \tag{1}$$

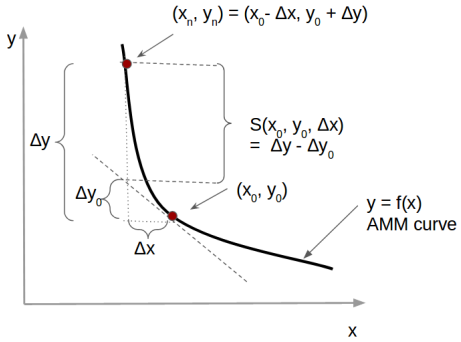


Figure 1 Illustration of Price Slippage on a Trade.

We refer to a plot of this function showing all allowed combinations of y and x as the *AMM curve*. For example, there can be a constant product curve, which is $y = \frac{k}{x}$ or a constant sum curve, which would be denoted as $y = c - x$. It is generally considered reasonable for the AMM curve to be convex and monotonically decreasing because this ensures (as we shall see in the next section) that the price for the token X is monotonically decreasing as a function of its availability in the pool, as should be expected of a typical supply curve.

3.1 Price Curve

Given an AMM curve, we can derive the price of the X token as follows:

$$p_X(x, y) = -\frac{dy}{dx} \tag{2}$$

For example, for the constant product curve, we would get $p_X(x, y) = \frac{k}{x^2}$ and likewise for the constant sum curve, we would get that $p_X(x, y) = 1$.

A plot of $p_X(x, y)$ versus x shows how the price of token X varies with its supply in the liquidity pool. Such a curve is referred to as a *price curve*. Note that if $f(x)$ is monotonically decreasing, then the price will always be positive, and if $f(x)$ is convex, then the price curve will be monotonically decreasing (as it should, being a type of supply curve).

3.2 Value of the pool

Given the definition of price, we can also assess the value of a given liquidity pool (measured in terms of Y) as follows:

$$V_p(x, y) = p_X \cdot x + y \tag{3}$$

3.3 Slippage

For curve-based AMM, slippage is defined as the loss incurred by a trader due to the price mismatch between the pool price at which the trade is initiated and the effective price obtained during the trade. Let us consider a trader seeking to buy Δx units of token X when the LP is at a state (x_0, y_0). Say that on the curve, the new point after the trade will be (x_n, y_n), where $x_n = x_0 - \Delta x$. The amount that the trader would then need to put into the LP will be $\Delta y = y_n - y_0$. If the pool price at the original point was p_0, then the buyer would have to pay $\Delta y_0 = p_0 \Delta x$. The difference between Δy and Δy_0 is defined as the slippage loss $S(x_0, y_0, \Delta x)$. This is illustrated in Figure 1.

Similarly, when the trader wishes to sell Δx units of token X, the gap between the $\Delta y_0 = p_0 \Delta x$ that the trader would like to receive and the $\Delta y = y_0 - y_n$ that he will actually receive would be the slippage loss on the sale, which could be expressed as $S(x_0, y_0, -\Delta x)$.

It is easy to see that on a constant sum AMM, the slippage loss is always 0 (because the price is constant at all points on the curve, or, equivalently, the tangent line at any point and the AMM curve always coincide). On any strictly convex curve, because the tangent line is always below the curve, the slippage loss will always be a positive quantity (i.e., the trader always incurs a penalty). The total slippage will be higher for a larger trade, and therefore acts as a disincentive for a trader to make large trades with the LP.

3.4 Divergence Loss

In general, when a trade is made, the price may change, as the original pair of values (x_o, y_o) moves to a new pair (x_n, y_n) following the curve, resulting in a new price p_n. Accordingly, the value of the liquidity pool could potentially decrease after a trade. This decrease as a relative or percentage decrease is referred to as divergence loss δ, and can be formally defined as follows:

$$\delta = \frac{V_{p_n}(x_n, y_n) - V_{p_n}(x_0, y_0)}{V_{p_n}(x_0, y_0)} \tag{4}$$

We work out below the divergence loss for the two example curves.

3.4.1 Divergence loss for constant-product curve

For the constant product curve, recall that the following hold:

$$p_n(x_n, y_n) = \frac{k}{x_n^2} \implies x_n = \sqrt{\frac{k}{p_n}} \tag{5}$$

$$y_n = \frac{k}{x_n} \implies y_n = \sqrt{k \cdot p_n} \tag{6}$$

Similarly, we also have that $x_o = \sqrt{\frac{k}{p_o}}$ and $y_o = \sqrt{k \cdot p_o}$.
Then we can define $V_{p_n}(x_n, y_n)$ as follows:

$$\begin{aligned} V_{p_n}(x_n, y_n) &= p_n \cdot x_n + y_n \\ &= p_n \sqrt{\frac{k}{p_n}} + \sqrt{k \cdot p_n} \\ &= 2\sqrt{k \cdot p_n} \end{aligned} \tag{7}$$

Likewise, we can define $V_{p_n}(x_o, y_o)$ as follows:

$$\begin{aligned} V_{p_n}(x_o, y_o) &= p_n \cdot x_o + y_o \\ &= p_n \sqrt{\frac{k}{p_o}} + \sqrt{k \cdot p_o} \end{aligned} \tag{8}$$

Based on the above two equations, we can calculate the divergence loss as follows:

$$\delta = \frac{2\sqrt{k \cdot p_n} - p_n \sqrt{\frac{k}{p_o}} + \sqrt{k \cdot p_o}}{p_n \sqrt{\frac{k}{p_o}} + \sqrt{k \cdot p_o}} \tag{9}$$

Denoting by ρ the ratio of the two prices $\frac{p_n}{p_o}$, the divergence loss for the constant product curve can be simplified to:

$$\delta = \frac{2\sqrt{\rho} - 1 - \rho}{1 + \rho} \tag{10}$$

This result is given in [17].

3.4.2 Divergence loss for constant-sum curve

Here the price is always 1. The two values can be written as follows:

$$\begin{aligned} V_{p_n}(x_o, y_o) &= x_o + y_o = c \\ V_{p_n}(x_n, y_n) &= x_n + y_n = c \end{aligned} \tag{11}$$

Since both are the same, the liquidity pool does not show any change in value, and thus the divergence loss in this case will be 0.

4 Dynamic curves

In the prior work on AMMs, the curve has a fixed form and the exact shape is determined by the initial total liquidity. E.g., in the constant product curve, the parameter $k = x_i \cdot y_i$ where x_i, y_i are the initial amounts of the two tokens. In other words, the curve can only change if the liquidity providers add/remove tokens from the pool, but not from trading activity.

Consider a trade that happens while the market price of token X remains unchanged at some price p_{mkt}. If the pool changes from the state (x_o, y_o) to a new state (x_n, y_n), then the pool price would potentially change from $p(x_o, y_o)$ to $p(x_n, y_n)$ (assuming the curve is not the constant-sum curve in which case there is no change in the pool price). This can result in at least a temporary difference between the pool price and the market price. As we do in the rest of the paper, we are assuming here that the pool's capitalization is a relatively small fraction of the total market capitalization of the underlying assets so that the market price is not determined or affected by the pool price.

Another reason for a temporary difference between the pool price and the market price could be that the market price changes due to some external market conditions. In either case, traditionally, it is expected that these temporary differences will be erased by the action of arbitrageurs, restoring the pool price back to the market price.

We propose a new mechanism that instead changes the curve every time the market price changes in such a way as to ensure that the current pool price will always equal the market price, *without requiring action by external arbitrageurs*. We illustrate below how this new mechanism would generalize the constant-product and constant-sum curves – the same approach can be used to generalize other smooth, decreasing, convex curves to the dynamic setting as well.

4.1 Dynamic curve adjustment to generalize constant-sum

In this case, we can describe the market-price-tracking dynamic curve as follows:

$$p_{mkt}(t) \cdot (x(t) - a(t)) + y(t) = c \tag{12}$$

Here, the parameter $a(t)$ will also be adjusted dynamically when the market price changes, to ensure that the new linear curve passes through the current pair of $(x(t), y(t))$ values. For simplicity, say the market is initialized at some pair $(x(0), y(0)$ at a market price of 1. Then c could be set to be $x(0) + y(0)$, with the original $a(0) = 0$.

If the market shifts to a price of $p_{mkt}(t)$ at some time t and the liquidity pool at this arbitrary time is $(x(t), y(t))$, then the value of $a(t)$ will also be adjusted as follows to match the above dynamic curve:

$$a(t) = x(t) - \frac{c - y(t)}{p_{mkt}(t)} \tag{13}$$

Intuitively, this dynamic curve is always a line that has the slope corresponding to the current market price and always passing through the current liqudity pair $(x(t), y(t))$.

Any trade that happens uses the current (instantaneous) curve. This allows the constant sum AMM to flexibly support a wider range of market prices while still providing 0 slippage compared to the original design (which allows only a fixed pool price and thus will not work when the market price is dramatically different).

4.2 Dynamic curve adjustment to generalize constant-product

In this case, we can describe the market-price-tracking dynamic curve as follows:

$$w(t) \cdot (x(t) - a(t)) \cdot y(t) = k \tag{14}$$

Or alternatively, as:

$$y(t) = \frac{\frac{k}{w(t)}}{x(t) - a(t)} \tag{15}$$

Note that in the above expressions, $x(t)$ and $y(t)$ must always be strictly positive; $a(t)$ must be constrained to be always strictly less than $x(t)$; and $w(t)$ should always be strictly positive. The instantaneous price corresponding to the dynamic version of the constant product curve can be defined as follows:

$$p_X(t) = \frac{k}{w(t)} \cdot \frac{1}{(x - a(t))^2} \tag{16}$$

When the market price changes, then both $w(t)$ and $a(t)$ will have to be changed in order to (a) make sure that the new market price $p_{mkt}(t)$ matches $p_X(t)$ in equation (16) and (b) x(t),y(t) match the curve described in equation (14). Thus we have to solve two equations and two unknowns. The solution turns out to be the following :

$$\begin{aligned} a(t) &= x(t) - \frac{y(t)}{p_{mkt}(t)} \\ w(t) &= \frac{k \cdot p_{mkt}(t)}{y(t)^2} \end{aligned} \tag{17}$$

We remark: the first expression above ensures the requirement mentioned above that $a(t)$ will remain strictly less than $x(t)$ and the second expression ensures that $w(t)$ is strictly positive, so long as k, $p_{mkt}(t)$, $x(t)$ and $y(t)$ are all kept strictly positive at all.

4.3 From divergence loss to slippage gain

As with the static setting, there is no slippage loss for traders or divergence loss in the case of dynamic constant sum AMM. This is because in the absence of any change in market price, the pool price does not change during a trade.

In the case of the dynamic constant product AMM, the traders do experience a slippage loss just like in the static constant product AMM case. However, corresponding gain in value is accrued entirely to the liquidity pool and could be referred to as a *slippage gain* for the LP. Further, in the absence of change in the market price, because the pool price does not change in the dynamic constant product curve, there is no divergence loss. Rather, the LP benefits from each trade by the same slippage gain. Thus, the dynamic constant product AMM provides a strict improvement from the LP's perspective. This result, in fact, generalizes to the dynamic version of any strictly convex curve, as we show below.

Proposition 1. *In a dynamic AMM based on a family of monotonically decreasing $y = f(x)$ curves that are strictly convex, when the market price remains fixed, the LP will gain value after each trade by an amount equivalent to the slippage loss of the trader.*

Proof. Assuming the market price does not change during a trade, for any strictly convex curve, the trader suffers a slippage loss at each trade. This is because the tangent to the curve (whose slope is equal to the pool price and therefore the market price) lies below the curve if it is strictly convex. If the trader buys X tokens, it will therefore have to give the pool an amount of Y tokens that exceeds what it should have given at the current market price. Likewise, if the trader sells X tokens, it will receive an amount of Y tokens less than what it should have received at the current market price. The gap, in either case, corresponds to the slippage loss. An equivalent amount is gained by the LP (when the trader buys X tokens, the excess Y tokens are sent to the LP; when the trader sells X tokens, the gap corresponds to Y tokens are withheld by the LP). There is no other source of divergence loss for the LP because the pool price is readjusted to the market price immediately after execution of the trade - its value increases precisely by the amount of slippage loss experienced by the trader. ◀

5 Simulation Results

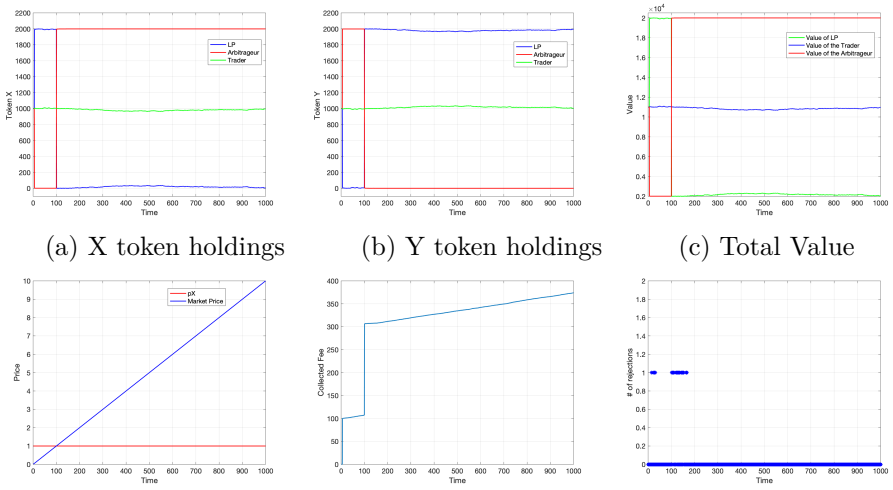
We conduct a number of simulations to compare the performance of four different AMM's, two static AMM's (Constant-Sum AMM, Constant-Product AMM) and their two dynamic generalizations that we have introduced in section 4. Although it is not practically implemented due to its shortcomings, we nevertheless use the constant-sum AMM as a baseline to highlight how the dynamic version of this scheme has advantages and may be more interesting from a practical perspective.

5.1 Simulation Setup

In all our simulations, we consider the three parties involved:

- The liquidity pool (LP).
- A trader (which could be viewed as a collection or series of independent traders who are merely interested in exchanging a relatively small amount of one of the tokens in the LP for another).
- An arbitrageur (a profit-maximizing agent responding specifically to any gaps in price between the pool and the broader market; the arbitrageur in our simulation could also be viewed as a collection or series of independent arbitragers).

Initially, all three parties have 1000 X tokens and 1000 Y tokens. At each time, a trader makes a random trade (buy or sell X token) drawn from a standard normal distribution (i.i.d. Gaussian process with zero mean and unit variance). The relatively small standard



(a) X token holdings (b) Y token holdings (c) Total Value

(d) Market vs Pool Price (e) Collected Fees (f) Declined Trades

Figure 2 Simulation of a Static Constant Sum AMM.

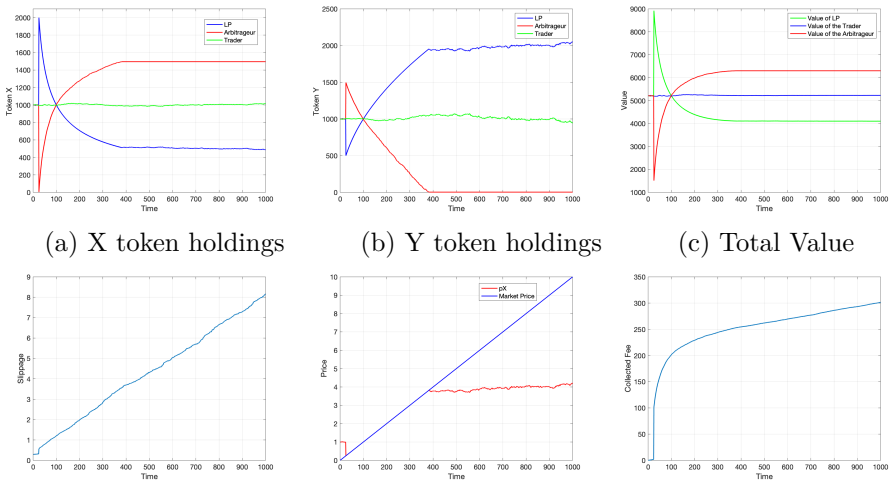
deviation would result in relatively light trading or profit from transaction fees collected from trades (that are not for the purpose of arbitrage). We use the exact same sequence of random trades in evaluating all four AMM's, to enable a fair comparison. We model the market price as undergoing a linear increase from 0 to 10 over the course of 1000 time steps.

5.2 Static Constant Sum AMM

Figure 2 shows the results for the baseline static constant AMM. As shown by Figure 2(d), the market price is initially below the pool price of 1, and after 100 time steps it switches to being above the pool price. As can be seen from Figure 2 (a), (b), the arbitrageur initially sells all its X tokens to buy up all of the Y tokens from the LP when the market price is low, and then it buys up all the X tokens from the LP when the market price becomes high. By draining the pool of its liquidity in one of its assets, to the pool's disadvantage, the arbitrageur effectively extracts most of the total value from the LP. Figure 2(c) shows the total value of the LP, the arbitrageur and the trader measured in terms of the market price at the end of the simulation (when X tokens are worth 10 Y tokens); it shows that the LP ends up with only about a tenth of the value it had at the beginning of the simulation. The LP does collect some fees from the arbitrageur (per Figure2(e)), but they are relatively modest compared to the loss in value. Although the trader doesn't suffer slippage loss in this case, it can be disappointed and face a rejection of its requested trade whenever the pool lacks sufficient tokens of the type sought by the trader (as shown in figure 2(f); this happens particularly early on when the arbitrageur is able to cause the LP to be nearly empty of one of its assets, then another, disrupting the LP's ability to serve the regular trader).

5.3 Static Constant Product AMM

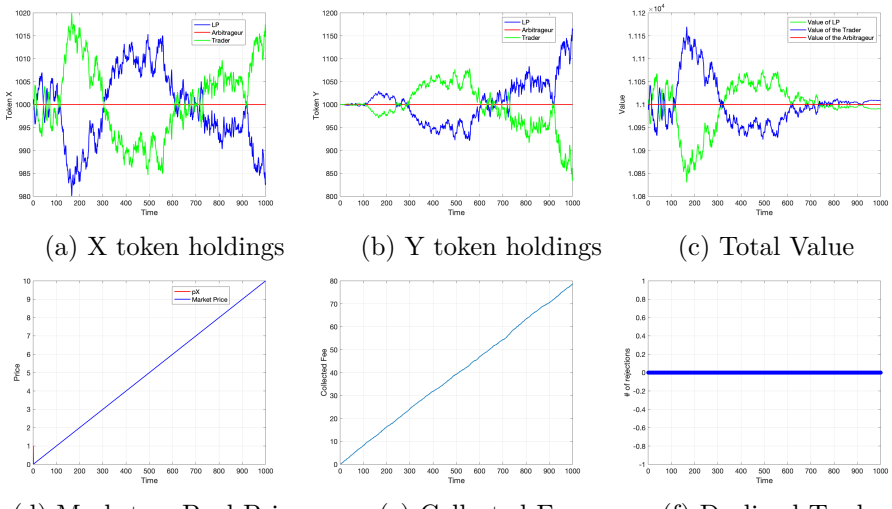
Figure 3 shows the results for the static constant product AMM. Here too, we can see from Figures 3(a) and (b), the arbitrageur has a big impact on the token holdings of the LP. Over time, as the market price goes up, the arbitrageur keeps buying X tokens from the pool, equalizing the pool price and the market price. Eventually the arbitrageur has no more Y tokens available (after which point the pool prize stays below the market price, see



(a) X token holdings (b) Y token holdings (c) Total Value

(d) Cumulative Slippage (e) Market vs Pool Price (f) Collected Fees

Figure 3 Simulation of a Static Constant Product AMM.

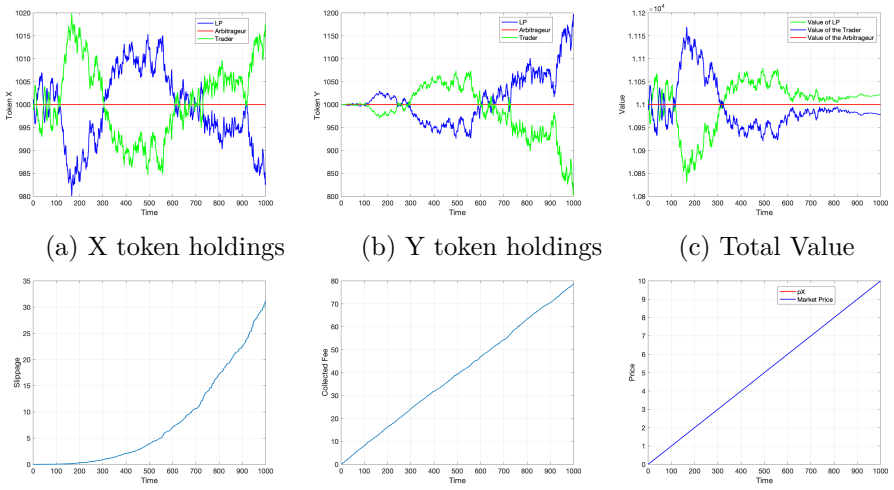


(a) X token holdings (b) Y token holdings (c) Total Value

(d) Market vs Pool Price (e) Collected Fees (f) Declined Trades

Figure 4 Simulation of a Dynamic Constant Sum AMM.

Figure 3(e)). As shown in Figure 3(c), the total value of the LP reduces while that of the arbitrageur grows until the latter runs out of Y tokens. In reality, however, a well-funded arbitrageur would keep going until all X tokens are depleted from the LP, causing it to lose even more value. The trader suffers some constant slippage on average in every trade, as illustrated by the cumulative slippage showing a linear increasing trend in Figure 3(d). There are again some collected fees from the arbitrageur and the regular trader, but here too they pale in comparison to the loss in value caused by the arbitrageur's actions. It is clear that just like with the constant sum setting, in the long term, the LP is at the mercy of the arbitrageur in the presence of significant market price volatility.



(a) X token holdings (b) Y token holdings (c) Total Value

(d) Cumulative Slippage (e) Collected Fees (f) Market vs Pool Price

Figure 5 Simulation of a Dynamic Constant Product AMM.

5.4 Dynamic Constant Sum AMM

Figure 4 shows the results for the dynamic constant sum AMM. Here the arbitrageur is eliminated as the pool price is automatically adjusted to the market price after each trade. The total tokens of either types X or Y remain the same and the holdings of the LP and the trader are mirror images of each other, as shown in Figures 4(a) and 4(b). The total value of the LP doesn't show a dramatic change other than random fluctuation caused by stochastic trading on the pool. In the absence of aggressive arbitrage, the collected fees are somewhat modest. As shown by Figure 4(e), the LP collects the expected amount of fees (about 80 over 1000 trades, corresponding to the expectation of the absolute value of a normal random variable). A noticeable fact about the dynamic constant sum AMM is that *the LP maintains more than 85% of the original amount of tokens of both types at all times* during the simulation. This is in sharp contrast to the liquidity problems observed in the static settings (Figures 2 and 3). As a result, there are no declined trades, as evidenced by Figure 4(f). Another related comment to be made here is that there is little incentive for large trades in this dynamic AMM because the price is always pegged to the market price and the transaction fees (being a constant percentage) impose a higher cost in absolute terms on larger trades.

5.5 Dynamic Constant Product AMM

Finally, Figure 5 shows the results for the dynamic constant product AMM. Here too, the arbitrageur is eliminated as the pool price is automatically adjusted to the market price after each trade (see Figure 5(f)), and the X and Y tokens held by the trader and LP mirror each other as shown in Figure 5(a) and Figure 5(b). The total value of the LP in Figure 5(c) doesn't show a dramatic change other than the fluctuation caused by stochastic trading on the pool and a slight increase in value for the LP due to the slippage gain discussed earlier in this paper. The corresponding cumulative slippage loss for the trader, equivalent to the cumulative slippage gain for the LP, is shown in Figure 5(d), and it is this gain that results in a slight positive drift in the LP value over time. As with the dynamic constant sum AMM, in the absence of aggressive arbitrage, the LP collects the expected amount of fees from the

regular trader (≈ 80 over 1000 trades, corresponding to the expectation of the absolute value of a normal random variable). Compared to the dynamic constant sum setting, here, there is even less incentive for any trader to execute a large trade as they would suffer a large slippage loss in addition to paying transaction fees. Figure 5(a) and Figure 5(b) show that the LP maintains high liquidity in both assets again – more than 90% of the original amount of tokens of both types at all times during the simulation.

6 Conclusions

We have given a detailed introduction to curve-based AMMs for decentralized cryptocurrency exchanges in this work. We introduced a new approach to operating such curve-based AMM decentralized exchanges that utilizes an oracle with a real-time market price feed to continuously and automatically adjust the pool price to the market price by dynamically adjusting the curves over time. We showed that in such a dynamic AMM, there is no room for arbitrage. The slippage loss for traders is converted to an equivalent gain for the liquidity pool. The LP maintains a high level of liquidity in all assets and its total value stays fairly stable over time. Such a dynamic AMM results in a slightly lower collection of transaction fees due to the elimination of arbitrageurs, but this loss is more than offset by the ability to maintain the total value of the pool.

From a practical perspective, implementing such a dynamic AMM requires the use of a low-latency and accurate market price oracle. It would be of great interest to develop and evaluate a real-world decentralized exchange based on this approach.

References

1 Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core, 2020. URL: https://uniswap.org/whitepaper.pdf.

2 Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. *arXiv preprint*, 2020. arXiv:2003.10001.

3 Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets. *Cryptoeconomic Systems Journal*, 2019.

4 Bancor. Announcing Bancor V2. URL: https://blog.bancor.network/announcing-bancor-v2-2f56b515e9d8, November 2020. Accessed: 2020-12-18.

5 Dmitriy Berenzon. Constant function market makers: Defi's "zero to one" innovation. URL: https://medium.com/bollinger-investment-group/constant-function-market-makers-defis-zero-to-one-innovation-968f77022159, May 2020. Accessed: 2020-12-18.

6 Vitalik Buterin. Let's run on-chain decentralized exchanges the way we run prediction markets. URL: https://www.reddit.com/r/ethereum/comments/55m04x/lets_run_onchain_decentralized_exchanges_the_way/, October 2016. Accessed: 2020-12-18.

7 Chainlink. How to bring more capital and less risk to automated market maker dexs. URL: https://blog.chain.link/challenges-in-defi-how-to-bring-more-capital-and-less-risk-to-automated-market-maker-dexs/, June 2020. Accessed: 2020-12-18.

8 Chainsecurity. Upcoming exchange concepts – a farewell to the order book? URL: https://chainsecurity.com/disruption-or-disillusion/. Accessed: 2020-12-18.

9 Richard Chen. A comparison of decentralized exchange designs. URL: https://thecontrol.co/a-comparison-of-decentralized-exchange-designs-1deef249f56a, April 2019. Accessed: 2020-12-18.

10 CoinMarketCap. Cryptocurrency prices, charts and market capitalizations. URL: https://coinmarketcap.com/. Accessed: 2020-12-18.

11 Michael Egorov. Stableswap – efficient mechanism for stablecoin liquidity. URL: https://curve.fi/files/stableswap-paper.pdf, 2019.

12 Gnosis. White paper. URL: https://github.com/gnosis/research/blob/master/gnosis-whitepaper.pdf, 2017.
13 Robin Hanson. Logarithmic markets scoring rules for modular combinatorial information aggregation. *The Journal of Prediction Markets*, 1(1):3–15, 2007.
14 Eyal Hertzog, Guy Benartzi, and Galia Benartzi. Bancor protocol: Continuous liquidity for cryptographic tokens through their smart contracts. *Bancor White Paper*, 2018.
15 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Online article*, 2019.
16 Jack Peterson, Joseph Krug, Micah Zoltu, Austin K Williams, and Stephanie Alexander. Augur: a decentralized oracle and prediction market platform. *arXiv preprint*, 2015. arXiv:1501.01042.
17 Pintail. Uniswap: A good deal for liquidity providers? URL: https://pintail.medium.com/uniswap-a-good-deal-for-liquidity-providers-104c0b6816f2, August 2020. Accessed: 2020-12-18.
18 Totle. Do dexs have a slippage problem? URL: https://medium.com/totle/do-dexs-have-a-slippage-problem-68c3a4fe5161, September 2019. Accessed: 2020-12-18.
19 Yongge Wang. Automated market makers for decentralized finance (defi). *arXiv preprint*, 2020. arXiv:2009.01676.
20 Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain, 2017. URL: https://github.com/0xProject/whitepaper.
21 Lawrence H White. The market for cryptocurrencies. *Cato J.*, 35:383, 2015.