Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer

Shir Cohen ✉
Technion, Haifa, Israel

Idit Keidar ✉
Technion, Haifa, Israel

Abstract

We formalize Byzantine linearizability, a correctness condition that specifies whether a concurrent object with a sequential specification is resilient against Byzantine failures. Using this definition, we systematically study Byzantine-tolerant emulations of various objects from registers. We focus on three useful objects– reliable broadcast, atomic snapshot, and asset transfer. We prove that there exist n-process f-resilient Byzantine linearizable implementations of such objects from registers if and only if $f < \frac{n}{2}$.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases Byzantine linearizability, concurrent algorithms, snapshot, asset transfer

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.18

Related Version *Full Version*: https://arxiv.org/abs/2102.10597 [14]

1 Introduction

Over the last decade, cryptocurrencies have taken the world by storm. The idea of a decentralized bank, independent of personal motives has gained momentum, and cryptocurrencies like Bitcoin [23], Ethereum [25], and Diem [8] now play a big part in the world's economy. At the core of most of these currencies lies the asset transfer problem. In this problem, there are multiple accounts, operated by processes that wish to transfer assets between accounts. This environment raises the need to tolerate the malicious behavior of processes that wish to sabotage the system.

In this work, we consider the shared memory model that was somewhat neglected in the Byzantine discussion. We believe that shared memory abstractions, implemented in distributed settings, allow for an intuitive formulation of the services offered by blockchains and similar decentralized tools. It is well-known that it is possible to implement reliable read-write shared memory registers via message passing even if a fraction of the servers are Byzantine [1, 21, 24, 19]. As a result, as long as the client processes using the service are not malicious, any fault-tolerant object that can be constructed using registers can also be implemented in the presence of Byzantine servers. However, it is not clear what can be done with such objects when they are used by Byzantine client processes. In this work, we study this question.

In Section 4 we define *Byzantine linearizability*, a correctness condition applicable to any shared memory object with a sequential specification. Byzantine linearizability addresses the usage of reliable shared memory abstractions by potentially Byzantine client processes. We then systematically study the feasibility of implementing various Byzantine linearizable shared memory objects from registers.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 18; pp. 18:1–18:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We observe that existing Byzantine fault-tolerant shared memory constructions [20, 22, 1] in fact implement Byzantine linearizable registers. Such registers are the starting point of our study. When trying to implement more complex objects (e.g., snapshots and asset transfer) using registers, constructions that work in the crash-failure model no longer work when Byzantine processes are involved, and new algorithms – or impossibility results – are needed.

As our first result, we prove in Section 5 that an asset transfer object used by Byzantine client processes does not have a wait-free implementation, even when its API is reduced to support only transfer operations (without reading processes' balances). Furthermore, it cannot be implemented without a majority of correct processes constantly taking steps. Asset transfer has wait-free implementations from both reliable broadcast [7] and snapshots [17] (which we adapt to a Byzantine version) and thus the same lower bound applies to reliable broadcast and snapshots as well.

In Section 6, we present a Byzantine linearizable reliable broadcast algorithm with resilience $f < \frac{n}{2}$, proving that, for this object, the resilience bound is tight. To do so, we define a sequential specification of a reliable broadcast object. Briefly, the object exposes broadcast and deliver operations and we require that deliver return messages previously broadcast. We show that a Byzantine linearizable implementation of such an object satisfies the classical (message-passing) definition [10]. Finally, in Section 7 we present a Byzantine linearizable snapshot with the same resilience. In contrast, previous constructions of Byzantine lattice agreement, which can be directly constructed from a snapshot [6], required $3f + 1$ processes to tolerate f failures.

All in all, we establish a tight bound on the resilience of emulations of three useful shared memory objects from registers. On the one hand, we show that it is impossible to obtain wait-free solutions as in the non-Byzantine model, and on the other hand, unlike previous snapshot and lattice agreement algorithms, our solutions do not require $n > 3f$. Taken jointly, our results yield the following theorem:

▶ **Theorem 1.** *In the Byzantine shared memory model, there exist n-process f-resilient Byzantine linearizable implementations of reliable broadcast, snapshot, and asset transfer objects from registers if and only if $f < \frac{n}{2}$.*

Although the construction of reliable registers in message passing systems requires $n > 3f$ servers, our improved resilience applies to client processes, which are normally less reliable than servers, particularly in the so-called *permissioned model* where servers are trusted and clients are ephemeral.

In summary, we make the following contributions:

- Formalizing Byzantine linearizability for any object with a sequential specification.
- Proving that some of the most useful building blocks in distributed computing, such as atomic snapshot and reliable broadcast, do not have f-resilient implementations from SWMR registers when $f \geq \frac{n}{2}$ processes are Byzantine.
- Presenting Byzantine linearizable implementations of a reliable broadcast object and a snapshot object with the optimal resilience.

2 Related Work

In [4] Aguilera et al. present a non-equivocating broadcast algorithm in shared memory. This broadcast primitive is weaker than reliable broadcast – it does not guarantee that all correct processes deliver the same messages, but rather that they do not deliver conflicting messages. A newer version of their work [5], developed concurrently and independently of our work[1], also implements reliable broadcast with $n \geq 2f+1$, which is very similar to our implementation. While the focus of their work is in the context of RDMA in the *M&M* (message–and–memory) model, our work focuses on the classical shared memory model, which can be emulated in classical message passing systems. While the algorithms are similar, we formulate reliable broadcast as a shared memory object, with designated API method signatures, which allows us to reason about the operation interval as needed for proving (Byzantine) linearizability and for using this object in constructions of other shared memory objects.

Given a reliable broadcast object, there are known implementations of lattice agreement [16, 26], which resembles a snapshot object. However, these constructions require $n = 3f+1$ processes. In our work, we present both Byzantine linearizable reliable broadcast and Byzantine snapshot, (from which Byzantine lattice agreement can be constructed [6]), with resilience $n = 2f+1$.

The asset transfer object we discuss in this paper was introduced by Guerraoui et al. [17, 15]. Their work provides a formalization of the cryptocurrency definition [23]. The highlight of their work is the observation that the asset transfer problem can be solved without consensus. It is enough to maintain a partial order of transactions in the systems, and in particular, every process can record its own transactions. They present a wait-free linearizable implementation of asset transfer in crash-failure shared memory, taking advantage of an atomic snapshot object. We show that we can use their solution, together with our Byzantine snapshot, to solve Byzantine linearizable asset transfer with $n = 2f+1$.

In addition, Guerraoui et al. present a Byzantine-tolerant solution in the message passing model. This algorithm utilizes reliable broadcast, where dependencies of transactions are explicitly broadcast along with the transactions. This solution does not translate to a Byzantine linearizable one, but rather to a sequentially consistent asset transfer object. In particular, reads can return old (superseded) values, and transfers may fail due to outdated balance reads.

Finally, recent work by Auvolat et al. [7] continues this line of work. They show that a FIFO order property between each pair of processes is sufficient in order to solve the asset transfer problem. This is because transfer operations can be executed once a process's balance becomes sufficient to perform a transaction and there is no need to wait for all causally preceding transactions. However, as a result, their algorithm is not sequentially consistent, or even causally consistent for that matter. For example, assume process i maintains an invariant that its balance is always at least 10, and performs a transfer with amount 5 after another process deposits 5 into its account, increasing its balance to 15. Using the protocol in [7], another process might observe i's balance as 5 if it sees i's outgoing transfer before the causally preceding deposit. Because our solution is Byzantine linearizable, such anomalies are prevented.

3 Model and Preliminaries

We study a distributed system in the shared memory model. Our system consists of a well-known static set $\Pi = \{1, \ldots, n\}$ of asynchronous client processes. These processes have access to some shared memory objects. In the shared memory model, all communication between processes is done through the API exposed by the objects in the system: processes invoke operations that in turn, return some response to the process. In this work, we assume

[1] Their work [5] was in fact published shortly after the initial publication of our results [14].

a reliable shared memory. (Previous works have presented constructions of such reliable shared memory in the message passing model [1, 21, 24, 3, 19]). We further assume an adversary that may adaptively corrupt up to f processes in the course of a run. When the adversary corrupts a process, it is defined as *Byzantine* and may deviate arbitrarily from the protocol. As long as a process is not corrupted by the adversary, it is *correct*, follows the protocol, and takes infinitely many steps. In particular, it continues to invoke the object's API infinitely often. Later in the paper, we show that the latter assumption is necessary.

We enrich the model with a *public key infrastructure* (PKI). That is, every process is equipped with a public-private key pair used to sign data and verify signatures of other processes. We denote a value v signed by process i as $\langle v \rangle_i$.

Executions and Histories. We discuss algorithms emulating some object O from lower level objects (e.g., registers). An algorithm is organized as methods of O. A method execution is a sequence of *steps*, beginning with the method's invocation (invoke step), proceeding through steps that access lower level objects (e.g., register read/write), and ending with a return step. The invocation and response delineate the method's execution interval. In an *execution* σ of a Byzantine shared memory algorithm, each correct process invokes methods sequentially, where steps of different processes are interleaved. Byzantine processes take arbitrary steps regardless of the protocol. The *history* H of an execution σ is the sequence of high-level invocation and response events of the emulated object O in σ.

A *sub-history* of a history H is a sub-sequence of the events of H. A history H is *sequential* if it begins with an invocation and each invocation, except possibly the last, is immediately followed by a matching response. Operation *op* is pending in a history H if *op* is invoked in H but does not have a matching response event.

A history defines a partial order on operations: operation op_1 precedes op_2 in history H, denoted $op_1 \prec_H op_2$, if the response event of op_1 precedes the invocation event of op_2 in H. Two operations are concurrent if neither precedes the other.

Linearizability. A popular correctness condition for concurrent objects in the crash-fault model is linearizability [18], which is defined with respect to an object's sequential specification. A *linearization* of a concurrent history H of object o is a sequential history H' such that (1) after removing some pending operations from H and completing others by adding matching responses, it contains the same invocations and responses as H', (2) H' preserves the partial order $\prec_H$, and (3) H' satisfies o's sequential specification.

f-resilient. An algorithm is *f-resilient* if as long as at most f processes fail, every correct process eventually returns from each operation it invokes. A *wait-free* algorithm is a special case where $f = n - 1$.

Single Writer Multiple Readers Register. The basic building block in shared memory is a single writer multiple readers (SWMR) register that exposes *read* and *write* operations. Such registers are used to construct more complicated objects. The sequential specification of a SWMR register states that every read operation from register R returns the value last written to R. Note that if the writer is Byzantine, it can cause a correct reader to read arbitrary values.

Asset Transfer Object. In [17, 15], the asset transfer problem is formulated as a sequential object type, called *Asset Transfer Object*. The asset transfer object maintains a mapping from processes in the system to their balances[2]. Initially, the mapping contains the initial balances of all processes. The object exposes a *transfer* operation, *transfer(src,dst,amount)*, which can be invoked by process *src* (only). It withdraws *amount* from process *src*'s account and deposits it at process *dst*'s account provided that *src*'s balance was at least *amount*. It returns a boolean that states whether the transfer was successful (i.e., *src* had *amount* to spend). In addition, the object exposes a *read(i)* operation that returns the current balance of i.

4 Byzantine Linearizability

In this section we define Byzantine linearizability. Intuitively, we would like to tame the Byzantine behavior in a way that provides consistency to correct processes. We linearize the correct processes' operations and offer a degree of freedom to embed additional operations by Byzantine processes.

We denote by $H|_{correct}$ the projection of a history H to all correct processes. We say that a history H is Byzantine linearizable if $H|_{correct}$ can be augmented with operations of Byzantine processes such that the completed history is linearizable. That is, there is another history, with the same operations by correct processes as in H, and additional operations by another at most f processes. In particular, if there are no Byzantine failures then Byzantine linearizability is simply linearizability. Formally:

▶ **Definition 2** (Byzantine Linearizability). *A history H is Byzantine linearizable if there exists a history H' so that $H'|_{correct} = H|_{correct}$ and H' is linearizable.*

Similarly to linearizability, we say that an object is Byzantine linearizable if all of its executions are Byzantine Linearizable.

Next, we characterize objects for which Byzantine linearizability is meaningful. The most fundamental component in shared memory is read-write registers. Not surprisingly, such registers, whether they are single-writer or multi-writers ones are de facto Byzantine linearizable without any changes. This is because before every read from a Byzantine register, invoked by a correct process, one can add a corresponding Byzantine write.

In practice, multiple writers multiple readers (MWMR) registers are useless in a Byzantine environment as an adversary that controls the scheduler can prevent any communication between correct processes. SWMR registers, however, are still useful for constructing more meaningful objects. Nevertheless, the constructions used in the crash-failure model for linearizable objects do not preserve this property. For instance, if we allow Byzantine processes to run a classic atomic snapshot algorithm [2] using Byzantine linearizable SWMR registers, it will not result in a Byzantine linearizable snapshot object. The reason is that the algorithm relies on correct processes being able to perform "double-collect" meaning that at some point a correct process manages to read all registers twice without witnessing any changes. While this is true in the crash-failure model, in the Byzantine model this is not the case as the adversary can change some registers just before any correct read.

[2] The definition in [17] allows processes to own multiple accounts. For simplicity, we assume a single account per-process, as in [15].

Relationship to Other Correctness Conditions

Byzantine linearizability provides a simple and intuitive way to capture Byzantine behavior in the shared memory model. We now examine the relationship of Byzantine linearizability with previously suggested correctness conditions involving Byzantine processes.

PBFT [12, 11] presented a formalization of linearizability in the presence of Byzantine-faulty clients in message passing systems. Their notion of linearizability is formulated in the form of I/O automata. Their specification is in the same spirit as ours, but our formulation is closer to the original notion of linearizability in shared memory.

Some works have defined linearization conditions for specific objects. This includes conditions for SWMR registers [22], a distributed ledger [13], and asset transfer [7]. Our condition coincides with these definitions for the specific objects and thus generalizes all of them. Liskov and Rodrigues [20] presented a correctness condition that has additional restrictions. Their correctness notion relies on the idea that Byzantine processes are eventually detected and removed from the system and focuses on converging to correct system behavior after their departure. While this model is a good fit when the threat model is software bugs or malicious intrusions, it is less appropriate for settings like cryptocurrencies, where Byzantine behavior cannot be expected to eventually stop.

5 Lower Bound on Resilience

In shared memory, one typically aims for wait-free objects, which tolerate any number of process failures. Indeed, many useful objects have wait-free implementations from SWMR registers in the non-Byzantine case. This includes reliable broadcast, snapshots, and as recently shown, also asset transfer. We now show that in the Byzantine case, wait-free implementations of these objects are impossible. Moreover, a majority of correct processes is required.

▶ **Theorem 3.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist a Byzantine linearizable implementation of asset transfer that supports only transfer operations in a system with $n \leq 2f$ processes, f of which can be Byzantine, using only SWMR registers.*

Note that to prove this impossibility, it does not suffice to introduce bogus actions by Byzantine processes, because the notion of Byzantine linearizability allows us to ignore these actions. Rather, to derive the contradiction, we create runs where the bogus behavior of the Byzantine processes leads to incorrect behavior of the correct processes.

Proof. Assume by contradiction that there is such an algorithm. Let us look at a system with $n = 2f$ correct processes. Partition Π as follows: $\Pi = A \cup B \cup \{p_1, p_2\}$, where $|A| = f - 1$, $|B| = f - 1$, $A \cap B = \emptyset$, and $p_1, p_2 \notin A \cup B$. By assumption, $|A| > 1$. Let z be a process in A. Also, by assumption $|B| \geq 2$. Let q_1, q_2 be processes in B. The initial balance of all processes but z is 0, and the initial balance of z is 1. We construct four executions as shown in Figure 1.

Let σ_1 be an execution where, only processes in $A \cup \{p_1\}$ take steps. First, z performs *transfer*(z, p_1,*1*). Since up to f processes may be faulty, the operation completes, and by the object's sequential specification, it is successful (returns true). Then, p_1 performs *transfer*(p_1, q_1,*1*). By f-resilience and linearizability, this operation also completes successfully. Note that in σ_1 no process is actually faulty, but because of f-resilience, progress is achieved when f processes are silent.

Similarly, let σ_2 be an execution where the processes in $A \cup \{p_2\}$ are correct, and z performs *transfer*(z, p_2,*1*), followed by p_2 performing *transfer*(p_2, q_2,*1*).

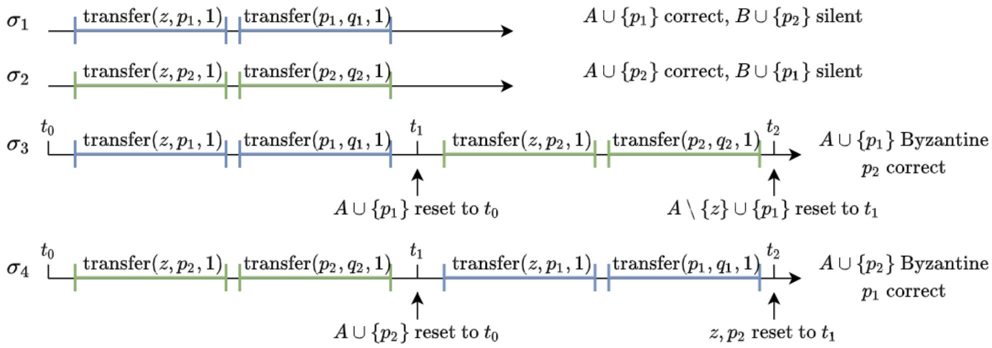

Figure 1 An asset transfer object does not have an f-resilient implementation for $n \leq 2f$.

We now construct σ_3, where all processes in $A \cup \{p_1\}$ are Byzantine. We first run σ_1. Call the time when it ends t_1. At this point, all processes in $A \cup \{p_1\}$ restore their registers to their initial states. Note that no other processes took steps during σ_1, hence the entire shared memory is now in its initial state. Then, we execute σ_2. Because we have reset the memory to its initial state, the operations execute the same way. When σ_2 completes, processes in $A \backslash \{z\} \cup \{p_1\}$ restore their registers to their state at time t_1. At this point, the state of z and p_2 is the same as it was at the end of σ_2, the state of processes in $A \setminus \{z\} \cup \{p_1\}$ is the same as it was at the end of σ_1, and processes in B are all in their initial states.

We construct σ_4 where all processes in $A \cup \{p_2\}$ are Byzantine by executing σ_2, having all processes in $A \cup \{p_2\}$ reset their memory, executing σ_1, and then having z and p_2 restore their registers to their state at the end of σ_2. At this point, the state of z and p_2 is the same as it was at the end of σ_2, the state of processes in $A \setminus \{z\} \cup \{p_1\}$ is the same as it was at the end of σ_1, and processes in B are all in their initial states.

We observe that for processes in B, the configurations at the end of σ_3 and σ_4 are indistinguishable as they did not take any steps and the global memory is the same. By f-resilience, in both cases q_1 and q_2, together with processes in B and one of $\{p_1, p_2\}$ should be able to make progress at the end of each of these runs. We extend the runs by having q_1 and q_2 invoke transfers of amount 1 to each other. In both runs processes in $B \cup \{p_1, p_2\}$ help them make progress. In σ_3, p_1 behaves as if it is a correct process and its local state is the same as it is at the end of σ_1, and in σ_4 p_2 behaves as if it is a correct process and its local state is the same as it is at the end of σ_2. Thus, σ_3 and σ_4 are indistinguishable to all correct processes, and as a result q_1 and q_2 act the same in both runs. However, from safety exactly one of their transfers should succeed. In σ_3, p_2 is correct and *transfer*$(p_2, q_2, 1)$ succeeds, allowing q_2 to transfer 1 and disallowing the transfer from q_1, whereas σ_4 the opposite is true. This is a contradiction. ◀

Guerraoui et al. [17] use an atomic snapshot to implement an asset transfer object in the crash-fault shared memory model. In addition, they handle Byzantine processes in the message passing model by taking advantage of reliable broadcast. In Appendix A we show that their atomic snapshot-based asset transfer can be easily adapted to the Byzantine settings by using a Byzantine linearizable snapshot, resulting in a Byzantine linearizable asset transfer. Their reliable broadcast-based algorithm, on the other hand, is not linearizable and therefore not Byzantine linearizable even when using Byzantine linearizable reliable broadcast. Nonetheless, Auvolat et al. [7] have used reliable broadcast to construct an asset transfer object where transfer operations are linearizable (although reads are not).

We note that our lower bound holds for an asset transfer object without read operations. In Algorithm 4 in Appendix A we construct an asset transfer object given a Byzantine linearizable snapshot (proofs appear in the full version [14]). The above discussion and the construction in Algorithm 4 lead us to the following corollary:

▶ **Corollary 4.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist an f-resilient Byzantine linearizable implementation of an atomic snapshot or reliable broadcast in a system with $f \geq \frac{n}{2}$ Byzantine processes using only SWMR registers.*

Furthermore, we prove in the following lemma that in order to provide f-resilience it is required that at least a majority of correct processes take steps infinitely often, justifying our model definition.

▶ **Lemma 5.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist an f-resilient Byzantine linearizable implementation of asset transfer in a system with $n \geq 2f+1$ processes, f of which can be Byzantine, using only SWMR registers if less than $f+1$ correct processes take steps infinitely often.*

Proof. Assume by way of contradiction that there exists an f-resilient Byzantine linearizable implementation of asset transfer in a system with $n \geq 2f+1$ processes where there are at most f correct processes that take steps infinitely often. Denote these f correct processes by the set A. Thus, there is a point t in any execution such that from time t, only processes in A and Byzantine processes take any steps. Starting t, the implementation is equivalent to one in a system with $n = 2f$, f of them may be Byzantine. This is a contradiction to Theorem 3. ◀

6 Byzantine Linearizable Reliable Broadcast

With the acknowledgment that not all is possible, we seek to find Byzantine linearizable objects that are useful even without a wait-free implementation. One of the practical objects is a reliable broadcast object. We already proved in the previous section that it does not have an f-resilient Byzantine linearizable implementation, for any $f \geq max\{3, \frac{n}{2}\}$. In this section we provide an implementation that tolerates $f < \frac{n}{2}$ faults.

6.1 Reliable Broadcast Object

The reliable broadcast primitive exposes two operations *broadcast(ts,m)* returning void and *deliver(j,ts)* returning m. When $deliver_j(i, ts)$ returns m we say that process j delivers m from process i in timestamp ts. The broadcast operation allows processes to spread a message m in the system, along with some timestamp ts. The use of timestamps allows processes to broadcast multiple messages.

Its classical definition, given for message passing systems [10], requires the following properties:

- Validity: If a correct process i broadcasts (ts, m) then all correct processes eventually deliver m from process i in timestamp ts.
- Agreement: If a correct process delivers m from process i in timestamp ts, then all correct processes eventually deliver m from process i in timestamp ts.
- Integrity: No process delivers two different messages for the same (ts, j) and if j is correct delivers only messages j previously broadcast.

In the shared memory model, the deliver operation for some process j and timestamp ts returns the message with timestamp ts previously broadcast by j, if exists. We define the sequential specification of reliable broadcast as follows:

▶ **Definition 6.** *A reliable broadcast object exposes two operations* broadcast(ts,m) *returning void and* deliver(j,ts) *returning m. A call to* deliver(j,ts) *returns the value m of the first* broadcast(ts,m) *invoked by process j before the deliver operation. If j did not invoke* broadcast *before the deliver, then it returns* $\perp$*.*

Note that as the definition above refers to sequential histories, the first broadcast operation (if such exists) is well-defined. Further, whereas in message passing systems reliable broadcast works in a push fashion, where the receipt of a message triggers action at its destination, in the shared memory model processes need to actively pull information from the registers. A process pulls from another process j using the *deliver(j,ts)* operation and returns with a value $m \neq \perp$. If all messages are eventually pulled, the reliable broadcast properties are achieved, as proven in the following lemma.

▶ **Lemma 7.** *A Byzantine linearization of a reliable broadcast object satisfies the three properties of reliable broadcast.*

Proof. If a correct process broadcasts m, and all messages are subsequently pulled then according to Definition 6 all correct processes deliver m, providing validity. For agreement, if a correct process invokes *deliver(j,ts)* that returns m and all messages are later pulled by all correct processes, it follows that all correct processes also invoke *deliver(j,ts)* and eventually return $m' \neq \perp$. Since *deliver(j,ts)* returns the value v of the first *broadcast(ts,v)* invoked by process j before it is called, and there is only one first broadcast, and we get that $m = m'$. Lastly, if *deliver(j,ts)* returns m, by the specification, j previously invoked *broadcast(ts,m)*. ◀

6.2 Reliable Broadcast Algorithm

In our implementation (given in Algorithm 1), each process has 4 SWMR registers: send, echo, ready, and deliver, to which we refer as *stages* of the broadcast. We follow concepts from Bracha's implementation in the message passing model [9] but leverage the shared memory to improve its resilience from $3f + 1$ to $2f + 1$. The basic idea is that a process that wishes to broadcast value v writes it in its send register (line 4) and returns only when it reaches the deliver stage. I.e., v appears in the deliver register of at least one correct process. Throughout the run, processes infinitely often call a *refresh* function whose role is to help the progress of the system. When refreshing, processes read all registers and help promote broadcast values through the 4 stages. For a value to be delivered, it has to have been read and signed by $f + 1$ processes at the ready stage. Because each broadcast message is copied to 4 registers of each process, the space complexity is $4n$ per message. Whether this complexity can be improved remains as an open question.

In the refresh function, executed for all processes, at first a process reads the last value written to a send register (line 16). If the value is a signed pair of a message and a timestamp, refresh then copies it to the process's echo register in line 18. In the echo register, the value remains as evidence, preventing conflicting values (sent by Byzantine processes) from being delivered. That is, before promoting a value to the ready or deliver stage, a correct process i performs a "double-collect" of the echo registers (in lines 19,21). Namely, after collecting $f + 1$ signatures on a value in ready registers, meaning that it was previously written in the echo of at least one correct process, i re-reads all echo registers to verify that there does not exist a conflicting value (with the same timestamp and sender). Using this method, concurrent deliver operations "see" each other, and delivery of conflicting values broadcast by a Byzantine process is prevented. Before delivering a value, a process writes it to its deliver register with $f+1$ signatures (line 22). Once one correct process delivers a value, the following deliver calls can witness the $f+1$ signatures and copy this value directly from its deliver register (line 11).

Algorithm 1 Shared Memory Bracha: code for process i.

shared SWMR registers: $send_i, echo_i, ready_i, deliver_i$

1: **procedure** CONFLICTING-ECHO($\langle ts, v\rangle_j$)
2: return $\exists w \neq v, k \in \Pi$ such that $\langle ts, w\rangle_j \in echo_k$

3: **procedure** BROADCAST(ts,val)
4: $send_i \leftarrow \langle ts, val\rangle_i$
5: **repeat**
6: $m \leftarrow$ deliver(i,ts)
7: **until** $m \neq \bot$ ▷ message is deliverable

8: **procedure** DELIVER(j,ts)
9: refresh()
10: **if** $\exists k \in \Pi$ and v s.t. $\langle\langle ts, v\rangle_j, \sigma\rangle \in deliver_k$ where σ is a set of $f+1$ signatures on $\langle ready, \langle ts, v\rangle_j\rangle$ **then**
11: $deliver_i \leftarrow deliver_i \cup \{\langle\langle ts, v\rangle_j, \sigma\rangle\}$
12: return v
13: return $\bot$

14: **procedure** REFRESH
15: **for** $j \in [n]$ **do**
16: $m \leftarrow send_j$
17: **if** $\nexists ts$, val s.t. $m = \langle ts, val\rangle_j$ **then** continue ▷ m is not a signed pair
18: $echo_i \leftarrow echo_i \cup \{m\}$
19: **if** $\neg$conflicting-echo(m) **then**
20: $ready_i \leftarrow ready_i \cup \{\langle ready, m\rangle_i\}$
21: **if** $\exists S \subseteq \Pi$ s.t. $|S| \geq f+1, \forall j \in S, \langle ready, m\rangle_j \in ready_j$ and $\neg$conflicting-echo(m) **then**
22: $deliver_i \leftarrow deliver_i \cup \{\langle m, \sigma = \{\langle ready, m\rangle_j | j \in S\}\rangle\}$ ▷ σ is the set of $f+1$ signatures

We make two assumptions on the correct usage of our algorithm. The first is inherently required as shown in Lemma 5:

▶ **Assumption 1.** *All correct processes infinitely often invoke methods of the reliable broadcast API.*

The second is a straight forward validity assumption:

▶ **Assumption 2.** *Correct processes do not invoke* broadcast(ts,val) *twice with the same ts.*

In the full version [14] we prove the correctness of the reliable broadcast algorithm and conclude the following theorem:

▶ **Theorem 8.** *Algorithm 1 implements an f-resilient Byzantine linearizable reliable broadcast object for any $f < \frac{n}{2}$.*

7 Byzantine Linearizable Snapshot

In this section, we utilize a reliable broadcast primitive to construct a Byzantine snapshot object with resilience $n > 2f$.

7.1 Snapshot Object

A snapshot [2] is represented as an array of n shared single-writer variables that can be accessed with two operations: *update(v)*, called by process i, updates the i^{th} entry in the array and *snapshot* returns an array. The sequential specification of an atomic snapshot is as follows: the i^{th} entry of the array returned by a *snapshot* invocation contains the value v last updated by an *update(v)* invoked by process i, or its variable's initial value if no update was invoked.

Following Lemma 5, we again must require that correct processes perform operations infinitely often. For simplicity, we require that they invoke infinitely many snapshot operations; if processes invoke either snapshots or updates, we can have each update perform a snapshot and ignore its result.

▶ **Assumption 3.** *All correct processes invoke snapshot operations infinitely often.*

7.2 Snapshot Algorithm

Our pseudo-code is presented in Algorithms 2 and 3. During the algorithm, we compare snapshots using the (partial) coordinate-wise order. That is, let s_1 and s_2 be two n-arrays. We say that $s_2 > s_1$ if $\forall i \in [n]$, $s_2[i].ts > s_1[i].ts$.

Recall that all processes invoke snapshot operations infinitely often. In each snapshot instance, correct processes start by collecting values from all registers and broadcasting their collected arrays in "start" messages (message with timestamp 0). Then, they repeatedly send the identities of processes from which they delivered start messages until there exists a round such that the same set of senders is received from $f+1$ processes in that round. Once this occurs, it means that the $f+1$ processes see the exact same start messages and the snapshot is formed as the supremum of the collects in their start messages.

We achieve optimal resilience by waiting for only $f+1$ processes to send the same set. Although there is not necessarily a correct process in the intersection of two sets of size $f+1$, we leverage the fact that reliable broadcast prevents equivocation to ensure that nevertheless, there is a common *message* in the intersection, so two snapshots obtained in the same round are necessarily identical. Moreover, once one process obtains a snapshot s, any snapshot seen in a later round exceeds s.

Each process i collects values from all processes' registers in a shared variable $collect_i$. When starting a snapshot operation, each process runs update-collect, where it updates its collect array (line 8) and saves it in a local variable c (line 9). When it does so, it updates the i^{th} entry to be the highest-timestamped value it observes in the i^{th} entries of all processes' collect arrays (lines 16 – 18). Then, it initiates the snapshot-aux procedure with a new

Algorithm 2 Byzantine Snapshot: code for process i.

shared SWMR registers: $\forall j \in [n]$ $collected_i[j] \in \{\perp\} \cup \{\mathbb{N} \times Vals\}$ with selectors ts and val, initially $\perp$
$\forall k \in \mathbb{N}$, $savesnap_i[k] \in \{\perp\} \cup \{\text{array of } n\ Vals \times \text{set of messages}\}$ with selectors $snap$ and proof, initially $\perp$
local variables: $ts_i \in \mathbb{N}$, initially 0
$\forall j \in [n]$, $rts_i[j] \in \mathbb{N}$, initially 0
$r, auxnum \in \mathbb{N}$, initially 0
$p \in [n]$, initially 1
$\forall j \in [n], k \in \mathbb{N}$, $seen_i[j][k]$,$senders_i \in \mathcal{P}(\Pi)$, initially $\emptyset$
$\sigma \leftarrow \emptyset$ set of messages

1: **procedure** UPDATE(v)
2:     **for** $j \in [n]$ **do** ▷ collect current memory state
3:         update-collect($collected_j$)
4:     $ts_i \leftarrow ts_i + 1$
5:     $collected_i[i] \leftarrow \langle ts_i, v\rangle_i$ ▷ update local component of collected

6: **procedure** SNAPSHOT
7:     **for** $j \in [n]$ **do** ▷ collect current memory state
8:         update-collect($collected_j$)
9:     $c \leftarrow collected_i$
10:     **repeat**
11:         $auxnum \leftarrow auxnum + 1$
12:         $snap \leftarrow$ snapshot-aux($auxnum$)
13:     **until** $snap \geq c$ ▷ snapshot is newer than the collected state
14:     return $snap$

15: **procedure** UPDATE-COLLECT(c)
16:     **for** $k \in [n]$ **do**
17:         **if** $c[k].ts > collected_i[k].ts$ and $c[k]$ is signed by k **then**
18:             $collected_i[k] \leftarrow c[k]$

auxnum tag. Snapshot-aux returns a snapshot, but not necessarily a "fresh" one that reflects all updates that occurred before *snapshot* was invoked. Therefore, snapshot-aux is repeatedly called until it collects a snapshot s such that $s \geq c$, according to the snapshots partial order (lines 10 – 13).

By Assumption 3 and since the *auxnum* variable at each correct process is increased by 1 every time snapshot-aux is called, all correct processes participate in all instances of snapshot-aux. When a correct process invokes a snapshot-aux procedure with auxnum, it first initiates a new reliable broadcast instance at line 28, dedicated to this instance of snapshot-aux. Note that although processes invoke one snapshot-aux at a time, they may engage in multiple reliable broadcast instances simultaneously. That is, they continue to partake in previous reliable broadcast instances after starting a new one. As another preliminary step of snapshot-aux, each correct process once again updates its collect array using the update-collect procedure (lines 30– 31) and broadcasts it to all processes at line 33.

Algorithm 3 Byzantine Snapshot auxiliary procedures: code for process i.

```
procedure MINIMUM-SAVED(auxnum)
    S ← {s | ∃j ∈ [n], s = savesnap_j[auxnum].snap and savesnap_j[auxnum].proof is a valid proof of s}
    if S = ∅ then
        return ⊥
    res ← infimum(S)                                   ▷ returns the minimum value in each index
    savesnap_i[auxnum] ← ⟨res, ⋃_{j∈[n]} savesnap_j[auxnum].proof⟩
    update-collect(res)
    return res

procedure SNAPSHOT-AUX(auxnum)
    initiate new reliable broadcast instance
    σ ← ∅
    for j ∈ [n] do                                     ▷ collect current memory state
        update-collect(collected_j)
    senders_i ← {i}                                    ▷ start message contains collect
    broadcast(0,⟨collect_i⟩_i)
    while true do
        cached ← minimum-saved(auxnum)                 ▷ check if there is a saved snapshot
        if cached ≠ ⊥ then return cached
        p ← (p + 1) mod n + 1                          ▷ deliver messages in round robin
        m ← deliver(p, rts_i[p])                       ▷ deliver next message from p
        if m = ⊥ then continue
        if rts_i[p]= 0 and m contains a signed collect array c then
                                                       ▷ start message (round 0)
            σ ← σ ∪ {m}
            update-collect(c)
            senders_i ← senders_i ∪ {j}
        else if m contains a signed set of processes, jsenders then
                                                       ▷ round r message for r > 0
            if jsenders ⊈ senders_i then
                continue                               ▷ cannot process message, its dependencies are missing
            σ ← σ ∪ {m}
            seen_i[j][rts_i[p]] ← jsenders ∪ seen_i[j][rts_i[p] − 1]
        rts_i[p] ← rts_i[p] + 1

        if received f + 1 round-r messages for the first time then
            r ← r + 1
            broadcast(r, ⟨senders_i⟩_i)

        if ∃s s.t. |{j| seen_i[j][s] = senders_i}| = f + 1 then    ▷ stability condition
            r ← 0
            senders_i ← ∅
            ∀j ∈ [n], k ∈ ℕ, seen_i[j][k] ← ∅
            cached ← minimum-saved(auxnum)             ▷ re-check for saved snapshot
            if cached ≠ ⊥ then return cached
            savesnap_i[auxnum] ← ⟨collect_i, σ⟩
                                  ▷ σ contains all received messages in this snapshot-aux instance
            return collect_i
```

During the execution, a correct processes delivers messages from all other processes in a round robin fashion. The local variable p represents the process from which it currently delivers. In addition, $rts[p]$ maintains the next timestamp to be delivered from p (lines 38, 49, 37). Note that if the delivered message at some point is $\perp$, $rts[p]$ is not increased, so all of p's messages are delivered in order (line 39).

Snapshot-aux proceeds in rounds, which are reflected in the timestamps of the messages broadcast during its execution. Each correct process starts snapshot-aux at round 0, where it broadcasts its collected array; we refer to this as its start message. It then continues to round $r+1$ once it has delivered $f+1$ round r messages (line 51). Each process maintains a local set $senders$ that contains the processes from which it received start messages (line 43). In every round (from 1 onward) processes send the set of processes from which they received start messages (line 52).

Process i maintains a local map $seen[j][r]$ that maps a process j and a round r to the set of processes that j reported to have received start messages from in rounds 1–r (line 48), but only if i has received start messages from all the reported processes (line 45). By doing so, we ensure that if for some correct process i and a round r $seen_i[j][r]$ contains a process l, l is also in $senders_i$. If this condition is not satisfied, the delivered counter for j ($rts[j]$) is not increased and this message will be repeatedly delivered until the condition is satisfied.

Once there is a process i such that there exists a round s and there is a set S of $f+1$ processes j for which $seen_i[j][s]$ is equal to $senders_i$, we say that the *stability condition* at line 53 is satisfied for S. At that time, i and f more processes agree on the collected arrays sent at round 0 by processes in $senders_i$, and $collect_i$ holds the supremum of those collected arrays. This is because whenever it received a start message, it updated its collect so that currently $collect_i$ reflects all collects sent by processes in $senders_i$. Thus, i can return its current collect as the snapshot-aux result. Since reliable broadcast prevents Byzantine processes from equivocating, there are f more processes that broadcast the same $senders$ set at that round, and any future round will "see" this set. As we later show, after at most $n+1$ rounds, the stability condition holds and hence the size of $seen$ is $O(n^3)$. Together with the collected arrays, the total space complexity is cubic in n.

To ensure liveness in case some correct processes complete a snapshot-aux instance before all do, we add a helping mechanism. Whenever a correct process successfully completes snapshot-aux, it stores its result in a savesnap map, with the auxnum as the key (either at line 24 or at line 59). This way, once one correct process returns from snapshot-aux, others can read its result at line 35 and return as well. To prevent Byzantine processes from storing invalid snapshots, each entry in the savesnap map is a tuple of the returned array and a proof of the array's validity. The proof is the set of messages received by the process that stores its array in the current instance of snapshot-aux. Using these messages, correct processes can verify the legitimacy of the stored array. If a correct process reads from savesnap a tuple with an invalid proof, it simply ignores it.

7.3 Correctness

We outline the key correctness arguments highlighting the main lemmas. Formal proofs of these lemmas appear in the full version [14]. To prove our algorithm is Byzantine linearizable, we first show that all returned snapshots are totally ordered (by coordinate-wise order):

▶ **Lemma 9.** *If two snapshot operations invoked by correct processes return s_i and s_j, then $s_j \geq s_i$ or $s_j < s_i$.*

Based on this order, we define a linearization. Then, we show that our linearization preserves real-time order, and it respects the sequential specification. We construct the linearization E as follows: First, we linearize all snapshot operations of correct processes in the order of their return values. Then, we linearize every update operation by a correct process immediately before the first snapshot operation that "sees" it. We say that a snapshot returning s *sees* an update by process j that has timestamp ts if $s[j].ts \geq ts$. If multiple updates are linearized to the same point (before the same snapshot), we order them by their start times. Finally, we add updates by Byzantine processes as follows: We add *update(v)* by a Byzantine process j if there is a linearized snapshot that returns s and $s[j].val = v$. We add the update immediately before any snapshot that sees it.

We next prove that the linearization respects the sequential specification.

▶ **Lemma 10.** *The i^{th} entry of the array returned by a* snapshot *invocation contains the value v last updated by an* update(v) *invoked by process i in E, or its variable's initial value if no update was invoked.*

Because an update is linearized immediately before some snapshot sees it and snapshots are monotonically increasing, all following snapshots see the update as well. Next, we prove in the two following lemmas that E preserves the real-time order.

▶ **Lemma 11.** *If a snapshot operation invoked by a correct process i with return value s_i precedes a snapshot operation invoked by a correct process j with return value s_j, then $s_i \leq s_j$.*

▶ **Lemma 12.** *Let s be the return value of a snapshot operation $snap_i$ invoked by a correct process i. Let $update_j(v)$ be an update operation invoked by a correct process j that writes $\langle ts, v\rangle$ and completes before $snap_i$ starts. Then, $s[j].ts \geq ts$.*

It follows from Lemma 12 and the definition of E, that if an update precedes a snapshot it is linearized before it, and from Lemma 11 that if a snapshot precedes a snapshot it is also linearized before it. The following lemma ensures that if an update precedes another update it is linearized before it. That is, if a snapshot operation sees the second update, it sees the first one.

▶ **Lemma 13.** *If update1 by process i precedes update2 by process j and a snapshot operation snap by a correct process sees update2, then snap sees update1 as well.*

Finally, the next lemma proves the liveness of our algorithm.

▶ **Lemma 14.** *(Liveness) Every correct process that invokes some operation eventually returns.*

We conclude the following theorem:

▶ **Theorem 15.** *Algorithm 2 implements an f-resilient Byzantine linearizable snapshot object for any $f < \frac{n}{2}$.*

Proof. Lemma 9 shows that there is a total order on snapshot operations. Using this order, we have defined a linearization E that satisfies the sequential specification (Lemma 10). We then proved that E also preserves real-time order (Lemmas 11 – 13). Thus, Algorithm 2 is Byzantine linearizable. In addition, Lemma 14 proves that Algorithm 2 is f-resilient. ◀

8 Conclusions

We have studied shared memory constructions in the presence of Byzantine processes. To this end, we have defined Byzantine linearizability, a correctness condition suitable for shared memory algorithms that can tolerate Byzantine behavior. We then used this notion to present both upper and lower bounds on some of the most fundamental components in distributed computing.

We proved that atomic snapshot, reliable broadcast, and asset transfer are all problems that do not have f-resilient emulations from registers when $n \leq 2f$. On the other hand, we have presented an algorithm for Byzantine linearizable reliable broadcast with resilience $n > 2f$. We then used it to implement a Byzantine snapshot with the same resilience. Among other applications, this Byzantine snapshot can be utilized to provide a Byzantine linearizable asset transfer. Thus, we proved a tight bound on the resilience of emulations of asset transfer, snapshot, and reliable broadcast.

Our paper deals with feasibility results and does not focus on complexity measures. In particular, we assume unbounded storage in our constructions. We leave the subject of efficiency as an open question for future work.

References

1 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

3 Yehuda Afek, David S Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM (JACM)*, 42(6):1231–1274, 1995.

4 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 409–418, 2019.

5 Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement, 2021. `arXiv:1905.12143`.

6 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement. In *International Workshop on Distributed Algorithms*, pages 35–53. Springer, 1992.

7 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bulletin of EATCS*, 3(132), 2020.

8 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

9 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

10 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

11 Miguel Castro, Barbara Liskov, et al. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

12 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

13 Vicent Cholvi, Antonio Fernandez Anta, Chryssis Georgiou, Nicolas Nicolaou, and Michel Raynal. Atomic appends in asynchronous byzantine distributed ledgers. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 77–84. IEEE, 2020.

14 Shir Cohen and Idit Keidar. Tame the wild with byzantine linearizability: Reliable broadcast, snapshots, and asset transfer. *arXiv preprint*, 2021. `arXiv:2102.10597`.

15 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

16 Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 674–683. IEEE, 2020.

17 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

18 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.

19 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3):451–500, 1998.

20 Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *International Symposium on Distributed Computing*, pages 487–489. Springer, 2005.

21 Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pages 311–325. Springer, 2002.

22 Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60(4):677–694, 2017.

23 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2009.

24 Rodrigo Rodrigues and Barbara Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, 2003.

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

26 Xiong Zheng and Vijay K. Garg. Byzantine lattice agreement in asynchronous systems. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14–16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.OPODIS.2020.4`.

A Byzantine Asset Transfer

In this section we adapt the asset transfer implementation from snapshots given in [17] to a Byzantine asset transfer. The algorithm is very simple. It is based on a shared snapshot array S, with a cell for each client process i, representing i's outgoing transactions. An additional immutable array holds all processes' initial balances. A process i's balance is computed by taking a snapshot of S and applying all of i's valid incoming and outgoing transfers to i's initial balance. A transfer invoked by process i checks if i's balance is sufficient, and if so, appends the transfer details (source, destination, and amount) to i's cell. Similarly to the use of dependencies in the (message-passing broadcast-based) asset transfer algorithm of [17], we also track the history of every transaction. To this end, we append to the process's cell also the snapshot taken to compute the balance for each transaction.

Algorithm 4 Byzantine Asset Transfer: code for process i.

```
shared Byzantine snapshot: S
initial– immutable array of initial balances
local variables: txns_i – sets of outgoing transaction, initially {}
ts_i ∈ ℕ, initially 0
snap – array of sets of transactions, initially array of empty sets      ▷ the last snapshot taken

struct txn contains:
    timestamp ts,
    source src,
    destination dst,
    amount amount

procedure BALANCE(j,snap)
    incoming ← 0
    outgoing ← 0
    for l ∈ [n] do
        for k ∈ snap[l] do
            if snap[l][k].dst = j and valid(snap[l][k]) then
                incoming ← incoming + snap[l][k].amount
    for k ∈ snap[j] do
        if valid(snap[j][k]) then
            outgoing ← outgoing + snap[j][k].amount
    return initial(j) + incoming − outgoing

procedure TRANSFER(src,dst,amount)
    ts_i ← ts_i + 1
    snap ← S.snapshot()
    if balance(src, snap) < amount then
        return false
    txns_i ← txns_i.append(⟨ts_i, src, dst, amount, snap⟩_i)
    S.update(txns_i)
    return true

procedure READ(j)
    snap ← S.snapshot()
    return balance(j, snap)
```