Self-Stabilizing Replicated State Machine Coping with Byzantine and Recurring Transient Faults

Shlomi Dolev* Amit Hendin† Maurice Herlihy‡ Maria Potop Butucaru§ Elad Michael Schiller¶

Abstract

The ability to perform repeated Byzantine agreement lies at the heart of important applications such as blockchain price oracles or replicated state machines. Any such protocol requires the following properties: (1) *Byzantine fault-tolerance*, because not all participants can be assumed to be honest, (2) r*ecurrent transient fault-tolerance*, because even honest participants may be subject to transient "glitches", (3) *accuracy*, because the results of quantitative queries (such as price quotes) must lie within the interval of honest participants' inputs, and (4) *self-stabilization*, because it is infeasible to reboot a distributed system following a fault.

This paper presents the first protocol for repeated Byzantine agreement that satisfies the properties listed above. Specifically, starting in an arbitrary system configuration, our protocol establishes consistency. It preserves consistency in the face of up to $\lceil n/3 \rceil - 1$ Byzantine participants *and* constant recurring ("noise") transient faults, of up to $\lceil n/6 \rceil - 1$ additional malicious transient faults, or even more than $\lceil n/6 \rceil - 1$ (uniformly distributed) random transient faults, in each repeated Byzantine agreement.

1 Introduction

In finance, a *price oracle* (or just *oracle*) is a service that connects users (often smart contracts on blockchains) with timely real-world data. Examples might include the spot price of Bitcoin, the current dollar/euro exchange rate, or nightly temperatures in Florida citrus groves. Today, oracle services are a growing business sector, led by companies such as Chainlink [7], Pyth Network [2], and others [30].

Much hinges on price oracle accuracy. Oracle price data drives automated trading algorithms, and unexpected fluctuations may trigger ruinous margin calls or liquidations. To avoid costly errors or fraud, commercial oracle services typically employ a variety of ad-hoc techniques, typically aggregating and smoothing data from multiple sources [30].

This paper introduces a class of Byzantine agreement algorithms well-suited to provide a systematic, rigorous foundation for *decentralized* price oracles immune to tampering from a small coalition of participants. A decentralized price oracle consists of a set of n processes, each connected to a distinct data source, such as a stock market feed or a sensor. A user query causes the processes to reach consensus on an aggregate price to be delivered to that user. Here is a list of (informally stated) properties a consensus protocol for a price oracle should satisfy.

- **Byzantine Fault-Tolerance**: Because the stakes are high, one cannot assume all participants are honest. Our protocol tolerates $\lceil n/3 \rceil - 1$ Byzantine (permanently dishonest) processes.
- **Recurrent Transient Fault-Tolerance**: Even honest participants may be subject to recurring *transient* errors, where an otherwise honest participant periodically undergoes an incorrect state change, but then resumes honest behavior. Our protocol tolerates $\lceil n/6 \rceil - 1$ recurring intermittent transient errors.
- **Price Accuracy**: As conditions change, honest participants may receive different prices from their sources, while dishonest participants may claim arbitrary prices. Even in the face of differing inputs and disruptive participants, it is not acceptable to deliver a null $\perp$ price to a user. Instead, our protocol guarantees that the price delivered to the user lies within the *range* of prices proposed by honest participants. (Under some circumstances, the protocol can be adapted to return specific values, such as the internal mode or median.)

*Ben-Gurion University of the Negev, Israel. Email: dolev@bgu.ac.il
†Ben-Gurion University of the Negev, Israel. Email: hendina@post.bgu.ac.il
‡Brown University, USA. Email: mph@cs.brown.edu
§Sorbonne Université, LIP6, Paris, France. Email: maria.potop-butucaru@lip6.fr
¶Chalmers University of Technology, Sweden. Email: elad.schiller@chalmers.se

- **Self-Stabilization**: When something goes wrong, it is not feasible to restart a decentralized service. An oracle state is *legitimate* if it could have been produced by a failure-free execution. Our protocol is *self-stabilizing*: starting from any (possibly illegitimate) state, the system will eventually enter a legitimate state, and remain in legitimate states until the next transient failure, see [12].

These properties are useful for applications beyond price oracles. For example, a *replicated state machine* is a fundamental task in distributed computing in which multiple processes coordinate to act as a single process (state machine). Devising replicated state machine protocols that work in the face of imperfect communication and Byzantine participants is a classic problem of distributed computing [24, 29].

Related work. Prior work on Byzantine agreement has addressed tolerating Byzantine participants and faults, *e.g.,* [17, 8]. By contrast, our work here investigates the effects of *repeated* transient faults in the scope of *repeated* consensus to realize a replicated state machine. Our protocol has more flexibility in choosing decision values than protocols restricted to approximate agreement [11], median [35], or interval agreement [28].

Stolz and Wattenhofer [35] propose a Byzantine agreement protocol that satisfies median validity, meaning that the non-faulty processes decide on a value that is at most t places away from the median value of the non-faulty nodes, where t is the upper limit of the number of Byzantine processes. Melnyk and Wattenhofer [28] propose a Byzantine agreement protocol that satisfies interval validity, which requires the non-faulty processes to decide values close to the k^{th} smallest initial value of the non-faulty processes. While both protocols take the median of a (possibly not identical) vector of the inputs structured by each process, here we ensure that each non-Byzantine process structures an *identical* vector of the inputs. While these two works aim to approximate almost equal (but possibly different) values decided by non-Byzantine participants, our approach obtains consistency on an entirely agreed-upon vector. All non-Byzantine participants agree upon each vector entry before taking the median. In order to obtain such an identical vector, another Byzantine agreement algorithm, that may return $\perp$, is employed; we chose a classical one presented in [36].

Binum et. al. [6] presented a self-stabilizing Byzantine replicated machine; however, they do not address the tolerance of recurring transient faults nor ensure strong validity, namely, the usage of the preceding state replica under one-third Byzantine and one-sixth recurring transient faults. Several approaches were suggested to seamlessly cope with transient faults, including superstabilization for tolerating the dynamicity of the communication graph [15], fault containment, and local stabilization for coping with a close-by corruption of the state of the participants, [22, 1]. Here, we introduce the possibility of tolerating Byzantine and recurrent (arbitrary and randomized) transient faults while ensuring a replicated state machine with strong validity. We ensure that the replicated state agreed upon by the non-Byzantine participants is used when computing the next replicated state, despite recurring transient faults that corrupt the replicas maintained by a bounded number of them.

Several prior works address the combination of Byzantine fault tolerance and self-stabilization in a replicated state machine protocol. Dolev *et al.* [13] propose a self-stabilizing Byzantine-tolerant replicated state machine architecture, relying on failure detectors to manage faulty behavior. While their design also targets recovery and Byzantine resilience, it assumes transient faults are rare and isolated. In contrast, our solution explicitly tolerates recurring transient faults that may continuously affect system execution. Georgiou et al. [20] propose a loosely self-stabilizing binary Byzantine consensus protocol for asynchronous message-passing systems without relying on digital signatures. Their work focuses on binary input domains, whereas our approach targets multivalued consensus with stronger interval validity guarantees under recurring transient faults. Browenstein *et al.* [8] presented a self-stabilizing Byzantine replicated state machine that preserves the global state privacy, based on secure multi-party computation.

In the scope of Blockchain, Dolev et al. [16] present techniques for using wallet information to reconstruct corrupted (or even erased) Blockchain records. Georgiou *et al.* [21] present a self-stabilizing, Byzantine-tolerant framework for recycling single-shot consensus objects in long-lived computations, allowing the reuse of object identifiers and other finite resources. Recently, Duvignau *et al.* [18, 19] present self-stabilizing versions of reliable broadcast and multivalued consensus protocols designed for Byzantine environments. Like our work, their approach addresses fault tolerance from arbitrary initial states and supports recovery in the presence of transient faults. However, it does not account for the recurring nature of such faults, which is a central focus of our system settings.

While previous works focus on resource reuse and managing global variables under adversarial conditions, our work targets agreement and state machine replication with recurring transient fault containment and strong validity on the replicated state guarantees.

Paper roadmap. System settings appear in the next section, Section 2, the requirements of the Byzantine agreement and the replicated state machine appear in Section 3. Madian-based multi-valued Byzantine agreement is presented in Section 4, then our replicated state machine appears in Section 5. Lastly, conclusions appear in Section 6. Some details appear in the Appendix, and some are omitted from this extended abstract.

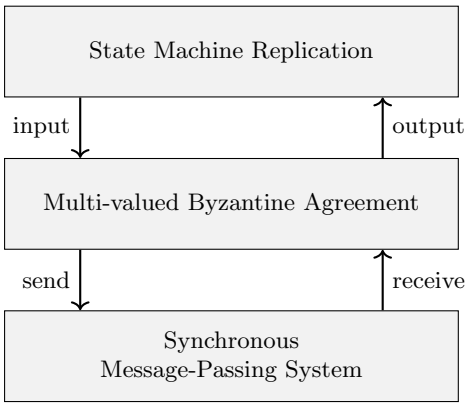

Figure 1: Layered architecture showing interactions between the system components.

2 System Settings

The Byzantine Agreement problem was first defined in [31]. The agreement task involves a set of processes that attempt to agree on a value by executing a common protocol during which they communicate. In Byzantine agreement, some processes act maliciously, sending the worst possible sequence of (inconsistent) messages. The Byzantine participants can be two-faced, sending or omitting conflicting messages to different participants or fainting crashes.

We assume a fully synchronous system model, dividing time into discrete, globally synchronized rounds. A global pulse, implemented via an external timing mechanism, defines the start of each round. All non-faulty processes send their messages simultaneously at the beginning of the round, and all messages are delivered before the next round begins.

The *processes* $p_1, \ldots, p_n$ are a set of processes that execute the same algorithm. Each process can send messages to any other process. In each round, a process may send different messages to different recipients. Processes proceed according to global rounds. The communication graph is a clique: each process has a unique identifier and can communicate directly with all other processes. The sender's identity is known upon receipt, either encoded in the message or inferred from the communication channel.

The system state (also called a *configuration*) at any time consists of the vector of the local states of all processes. Due to synchronous delivery, message buffers are assumed empty at round boundaries. We define a system execution as a sequence of global configurations resulting from atomic computation steps executed in rounds. Each round triggered by a global pulse consists of four phases: (i) an *input phase*, in which every process receives an external input value simultaneously; (ii) a *send phase*, where each process broadcasts messages; (iii) a *receive phase*, where all messages are delivered; and (iv) a *computation phase*, where each process updates its local state based on received messages and input.

We consider *Byzantine faults* [31], where faulty processes may arbitrarily deviate from the algorithm and act maliciously. To enable system recovery and continuous functionality of the system [31, 5], the number of Byzantine processes is bounded throughout execution by $t < \lfloor n/3 \rfloor$.

We also account for *transient faults*, as introduced in Dijkstra's seminal work on self-stabilization [10]. These faults occur only before the start of the execution, possibly due to temporary violations of fault thresholds, and may arbitrarily corrupt the initial system configuration. From that point on, the fault thresholds, *e.g.,* number of Byzantine processes, are assumed to remain within tolerated bounds, and no transient fault can arbitrarily change the system state.

In contrast, we define *recurring transient faults* (also called *online transient faults*) as arbitrary local state corruptions that may occur throughout execution. These affect at most one process per corruption event and are assumed to occur with bounded intensity. Specifically, the number of such corruptions in any given round is strictly fewer than $r < \lceil n/6 \rceil - 1$.

The bound on r ensures that the median used in our agreement algorithms remains within the range of correct values. Since up to $\lfloor n/3 \rfloor$ processes may be Byzantine and some correct values may be lost or unavailable, allowing more than $\lfloor n/6 \rfloor - 1$ recurring corruptions could cause the median to reflect faulty or adversarial values. This threshold ensures that a sufficient number of uncorrupted, correct inputs influence the outcome.

Due to the presence of transient faults, the system may begin in an arbitrary configuration, meaning that transient faults may corrupt each process's state. Our algorithms are designed to be self-stabilizing, *i.e.,* they converge to correct behavior even in the presence of a bounded (*e.g.,* one third of the processes) number of Byzantine processes together with a bounded (*e.g.,* one sixth of the processes) recurring state corruptions.

Legitimate States and Self-Stabilization. A system state is said to be *legitimate* if it satisfies all

requirements specified by the problem definition (Section 3). An algorithm is *self-stabilizing* if, from any arbitrary state, the system is guaranteed to reach a legitimate configuration in a finite number of rounds and remains in legitimate configurations as long as the number of faults per round remains within tolerated thresholds. Our algorithms stabilize even when starting from an arbitrarily corrupted state due to transient faults, and continue to function correctly under a bounded number of Byzantine processes and recurring state corruptions.

We define the *stabilization time* as the worst-case number of synchronous rounds required for the system to reach a legitimate configuration from an arbitrary state, assuming up to t Byzantine faults and up to r recurring state corruptions per round.

We summarize the system description and main terms used in the sequel.

Processes $p_1, \ldots, p_n$ are a set of processes that execute the same protocol. Each process can send a message to any other process. Processes execute the protocol according to sequential global pulses. Messages are sent when a pulse occurs and are received before the next pulse appears. The communication graph is a clique. Each message is identified by the communication link through which it arrives. Therefore, the identity of the process that sent a message is known to the receiver.

Pulse number A global pulse is associated with a global counter modulo a given integer b. The processes are exposed to the global pulse and the global pulse number [1]. We denote a global pulse that triggers the transition of the replicated state machine as $Pulse$, each such $Pulse$ is invoked when the global counter of pulses is zero. Then we have intermediate pulses, for which the pulse counter value is $1, 2, \ldots, b-1$, denoted $IPulse$ to finalize the needed computation for the next replicated state. In the sequel, we may omit mentioning the $IPlus$ that controls the advance in the intermediate round stages whenever there is no possible confusion.

Faulty (Byzantine) Process A faulty process is a process that deviates from the protocol. This includes processes that crash during execution or have unlimited computation power to send the most malicious sequence of messages to prevent the other processes from reaching an agreement. There are at most f faulty processes where $f < \lceil n/3 \rceil - 1$.

Transient Faults A (spontaneous) change of a state of a process to an arbitrary state in the domain of its states. There are at most r transient faults where $r < \lceil n/6 \rceil - 1$ in situations with no prior knowledge of the distribution of transient faults.

3 Problem Statement and Solution Framework

We study a class of agreement problems for systems that tolerate both recurring transient faults and Byzantine faults, provided that the number of Byzantine processes is less than $\lfloor n/3 \rfloor$ and the number of recurring transient faults is less than $\lceil n/6 \rceil - 1$. We focus on three variants of Multi-valued Byzantine Agreement (MVBA). Our main contribution is a protocol that achieves *interval validity* and consistency, building upon an algorithm that guarantees only *weak validity*. For comparison, we also recall the definition of MVBA with *strong validity*, and briefly mention the related concept of *approximate agreement*.

Figure 1 illustrates the layered architecture of our solution framework. At the bottom lies the synchronous message-passing system (Section 2), which provides reliable, round-based communication among processes, activated by global pulses. On top of this network layer, the MVBA layer (Definition 2, Algorithm 1) implements agreement primitives that tolerate Byzantine faults and recurring transient faults. This layer combines a multi-valued agreement structure with most-common and median-based selection mechanisms to ensure consistency and interval validity. It interacts with the communication layer via the `send` and `receive` interfaces, and with higher layers via synchronous `input` and `output` operations. At the top of the stack is the *State Machine Replication* layer (Definition 4, Algorithm 2), which relies on the agreement abstraction to execute deterministic transitions consistently across all correct replicas.

We clarify that, for the sake of a simple presentation, the studied problems are specified only by the requirements critical to understanding our contribution. For a more comprehensive treatment of consensus problems in distributed systems, we refer the reader to e.g, [32, 23, 27, 3]. It is also known that in synchronous message-passing systems, any consensus algorithm that terminates within a bounded number of rounds can be transformed into a self-stabilizing algorithm that recovers within the same bound after the occurrence of the last transient fault. See, for example, Theorem 2 of Georgiou *et al.* [21]. Therefore, from this point onward, our work focuses exclusively on protocols that tolerate both recurring transient faults and Byzantine faults.

[1] The global pulse and global pulse number can themselves be an output of semi-synchronous algorithms, *e.g.,* [17, 26].

3.1 Multi-valued Byzantine Agreement (MVBA)

Our solution framework uses the problem specified in Definition 2 to construct a replicated state machine (Definition 4) that tolerates both Byzantine faults and recurring transient faults.

Definition 1 (Reference Problem: MVBA with Strong Validity)**.** *A set of n processes executes a Multi-valued Byzantine Agreement algorithm. For all $1 \leq i \leq n$, each process p_i starts with an input value $v_i \in V$ from some domain of values V and returns a decision value $d_i \in V$ within a known constant number of synchronous rounds. The algorithm is correct if it satisfies the following properties:*

- Consistency: *All non-faulty processes decide on the same value.*
- Strong Validity: *All non-faulty processes decide on a value proposed by a non-faulty process.*

A variation on Definition 1 substitutes the strong validity requirement with *discrete interval validity* (Definition 2), which captures the intuition that the decided value must lie within the interval of values proposed by non-faulty processes, even if a Byzantine process proposes it.

Definition 2 (Target Problem: MVBA with Discrete Interval Validity)**.** *A Multi-valued Byzantine Agreement algorithm satisfies* discrete interval validity *if, in addition to satisfying consistency (Definition 1), all non-faulty processes decide on the same value $d \in V$, and this integer value lies within the range of the inputs proposed by non-faulty processes.*

Formally, if $T = \{v_i \mid p_i \text{ is non-faulty}\}$ is the multiset of inputs from non-faulty processes, then $d \in \{\min T, \ldots, \max T\}$.

An MVBA algorithm with discrete interval validity accepts values that Byzantine processes may have proposed, as long as they fall within the range of correct inputs and do not violate agreement. The value chosen does not result from the (not necessarily integer) mean computation used to approximate the input value of the non-faulty processes, see *e.g.,* [11]. Unlike weak validity (defined next), discrete interval validity does not introduce an artificial default value $\perp$.

In the sequel, we use interval validity instead of discrete interval validity whenever no confusion can arise.

3.2 Our Building Block: MVBA with Weak Validity

Our solution framework for MVBA with interval validity builds upon a Byzantine Agreement algorithm that guarantees only *weak validity*, such as the one proposed by Turpin and Coan [36]. Note that weak validity permits a decision on the default value $\perp$, which may not be acceptable for application-level correctness or progress.

Definition 3 (Building Block: MVBA with Weak Validity)**.** *A Multi-valued Byzantine Agreement algorithm satisfies* weak validity *if all non-faulty processes decide on the same value (*i.e., *satisfy consistency from Definition 1), and the decided value is either:*

- *a value proposed by a non-faulty process, if all such processes proposed the same input; or*
- *a value from the decision domain V, which may include a default value $\perp$ otherwise.*

A related but distinct problem is *approximate agreement* [11], in which the decision values of non-faulty processes must be within a small ϵ range of each other and lie within the bounds of the correct inputs.

3.3 Our Application: State Machine Replication

The state machine is a foundational model of computation with many variations. In general, a state machine M has a set of states $q_1, q_2, \ldots$ and a transition function $\delta(q, in)$ that takes a state q and an input in and returns the resulting state. A state machine is deterministic if δ always returns the same result for the same input and state.

The State Machine Replication problem was introduced by Lamport in [25] and later investigated under different settings. In [33], the authors specify the state machine replication problem in distributed systems prone to failures. A distributed protocol implementing state machine replication should verify the following two properties:

Agreement Every non-faulty copy of the state machine receives every command;

Order Commands are processed in the same order by every non-faulty copy of the state machine.

A *replicated state machine* consists of multiple identical replicas of M that maintain synchronized states and execute transitions based on agreed inputs. The following is a formal definition of a replicated state machine based on the definitions in [14, 34].

Definition 4. State Machine: *Let M be a deterministic state machine. Let $S = \{s_1, s_2, \dots\}$ be its set of states, $\Sigma = \{in_1, in_2, \dots\}$ be its input alphabet, and $\delta : S \times \Sigma \to S$ be the transition function.*

Replicas: *The processes $p_1, \dots, p_n$ that maintain local copies of M and apply transitions using δ.*

Program: *A program is a sequence of atomic steps, each consisting of a local computation followed by a communication step.*

Configuration and Global Input: *A* configuration *is defined by a function over the replica states (*e.g., *the most common state), and the* global input *is defined by aggregating local inputs (*e.g., *the median of all inputs).*

Execution: *A replicated execution is an alternating sequence $R = c_1, in_1, c_2, in_2, \dots$ such that for all $i > 0$, $\delta(c_i, in_i) = c_{i+1}$.*

See *e.g.,* [4] for the importance of fault-tolerant replicated state machines in the scope of Blockchain technologies.

4 Median-based Multi-valued Byzantine Agreement

We design a one-shot Multi-valued Byzantine Agreement algorithm that returns the most frequently occurring value among the agreed-upon inputs, provided that value's observed frequency surpasses the threshold required to guarantee validity. Otherwise, the algorithm returns the median value. In cases where applications tolerate a default value $\perp$, it is always trivially possible to return that value. However, we are interested in applications where default values would not be satisfactory. Our algorithm works by first reaching agreement on a vector of inputs using an agreement algorithm that satisfies a weaker validity condition. After this step, each non-faulty process obtains an identical vector of values. The process then removes all default entries ($\perp$), sorts the resulting list—with repetitions preserved—and selects the median. If the list contains an even number ℓ of non-default elements, the algorithm returns the entry at position $\ell/2$ in the sorted list. We introduce a tunable threshold parameter α, which controls when a value is considered sufficiently frequent to be selected directly. In situations where assumptions about the distribution of values of transient faults can be made, reducing the parameter α allows for greater transient fault tolerance, up to a third of all processes in the best case. When no assumptions can be made about the distribution of values of transient faults, setting α to a sixth of all processes yields the greatest transient fault tolerance.

In the context of replicated state machine applications, we aim to prefer the most common value in the decision vector, provided it occurs frequently enough to ensure strong validity. Specifically, if a value appears in more than one-third of the entries plus an additional margin α, it is selected. Otherwise, the median value is used. The parameter α accounts for the possible influence of recurring transient faults, allowing the algorithm to distinguish between values that reflect genuine consensus and those that may result from adversarial skew.

4.1 Algorithm description

We now present Algorithm 1, which implements the median-based multi-valued Byzantine agreement. The algorithm begins by allocating an array A to hold values received from other processes (line 13). Each process then broadcasts its own input value to all others (lines 14-15) and records received values in A (lines 16-17). Since messages include sender identities, each process can associate received values with sender indices, ensuring that $A[i]$ stores the value claimed by process p_i.

Each process then invokes a weak-validity multi-valued Byzantine agreement algorithm on every entry in A, running these instances in parallel (lines 18-19). The algorithm `WeakMVBA` is used here, which can be instantiated with the algorithm from Turpin and Coan [36], listed in the Appendix for completeness. After this step, each non-faulty process holds an array A in which each entry $A[i]$ reflects the agreed-upon value for process p_i's input. Since the same agreement algorithm is applied to each index, and all non-faulty processes use the same inputs and follow the same logic, their resulting arrays are identical. The process then calls `select_value`(A) (line 20) to determine the final decision value, to be returned.

The function `select_value` processes the agreed-upon vector A to determine the final output. It first removes all occurrences of the default value $\perp$ to produce a cleaned array $A_{\not\perp}$ (line 2), then counts the frequency of each remaining value using a dictionary C (lines 4–6). The most common value m is identified via an arg max operation (line 7). If m appears in at least $\lfloor k/3 \rfloor + 1 + \alpha$ entries (line 8), it is returned as the final decision.

Otherwise, the array $A_{\not\perp}$ is sorted (line 10), and the median element—specifically, the $\lfloor k/2 \rfloor$-th entry—is returned (line 11). This rule ensures consistency across all non-faulty processes and protects against biased skewing from faulty or corrupted inputs.

The parameter α serves as a tunable resilience margin that reflects assumptions about the distribution and coordination of recurring transient faults. In adversarial scenarios, where faults may intentionally reinforce each other to skew the output, a conservatively set $\alpha = \lceil n/6 \rceil - 1$ ensures correctness despite worst-case behaviour. In more benign environments, such as when transient faults follow a uniform or independent distribution, it is often safe to choose a smaller α, since the likelihood of multiple faults supporting the same incorrect value is significantly reduced.

As noted earlier, we instantiate `WeakMVBA` to be the algorithm of Turpin and Coan [36], which returns a default value $\perp$ when agreement cannot be reached. However, the specific choice of agreement algorithm is not essential—any one-shot MVBA algorithm satisfying weak validity (*i.e.,* agreeing on a correct input when all non-faulty processes begin with the same value, and otherwise deciding on a value from the decision domain including $\perp$) is sufficient. This modularity enables reuse and flexibility in system design.

Our decision rule then selects the most frequently occurring agreed-upon value—excluding $\perp$—if it appears at least $\lfloor n/3 \rfloor + 1 + \alpha$ times. Otherwise, the rule returns the median value from the sorted array $A_{\not\perp}$. In case of ties (*e.g.,* even-length arrays), we break them deterministically by selecting the lower of the two middle values.

The returned value by the Algorithm 1 is always consistent across the processes and satisfies the weak validity; proof provided next.

4.2 Correctness Proof

We begin by proving that Algorithm 1 has consistency and interval validity despite less than $\lfloor n/3 \rfloor$ Byzantine faults. This is shown in Lemmas 1 and 2. Then we further expand this result by showing that Algorithm 1 remains correct despite $\lceil n/6 \rceil - 1$ transient faults in addition to $\lfloor n/3 \rfloor$ Byzantine faults, as shown in Theorem 5.

Lemma 1. *Algorithm 1 satisfies the consistency property of Definition 1 despite less than $\lfloor n/3 \rfloor$ Byzantine faults.*

Proof. Let K be the set of $\lfloor \frac{2n}{3} \rfloor + 1$ indexes such that p_j is non-faulty for all $j \in K$ and let A_i denote the local array A of process p_i. When p_i reaches line 18, it holds that $A_i[j] = v_j$ for all $j \in K$.
Therefore, for any $i \in K$, due to the weak validity of `WeakMVBA`, it holds that in line 19, `WeakMVBA`$(A_i[j])$ returns v_j since all correct participants executed the protocol with the same input value, thus no action taken by the faulty processes could change the result due to the f-fault tolerance of the protocol `WeakMVBA`.

Moreover, based on the consistency of `WeakMVBA`, when all non-faulty processes reach line 20, we have that $A_j = A_i$ for all $j, i \in K$, that is, A is consistent, for entries $\ell \notin K$ too.

Every key inserted into the dictionary C is a value from A (line 6). This means that for any m we choose (line 7), there exists an i such that $A[i] = m$ as m is a value of C. Therefore, since A is identical in all correct processes, returning m (line 9) returns the same output to all correct processes. □

Lemma 2. *Algorithm 1 satisfies the interval validity property (Definition 2) despite less than $\lfloor n/3 \rfloor$ Byzantine faults.*

Proof. We prove interval validity by showing that any value returned by the algorithm is within the interval of values of non-faulty processes. We do so by looking at the possible cases the Byzantine processes can create and showing that none violate interval validity. Since any value is either returned in line 9 or 11, we address each case separately. Recall the set K from the proof of Lemma 1. Let $T = \{v_j | j \in K\}$ be a set of values of non-faulty processes and let $I = \{x | \min T \leq x \leq \max T\}$ be the interval of the values of non-faulty processes. By Lemma 1, when all processes reach line 9, A is consistent across the non-faulty processes. Consider the possible cases for the distribution of values in A:

1. All of the faulty processes sent values inside of I
2. All of the faulty processes sent values outside of I
3. Some of the faulty processes sent values outside of I

- In case 1, all values in A are within I; thus, any decision value satisfies the interval validity property.
- *Notice that we return values only on lines 9 and 11 (which get returned again in line 20).*
- In cases 2 & 3, consider situations where m is returned (line 9). This only happens if at least $\lfloor \frac{n}{3} \rfloor + 1$ processes send value m due to the condition on line 8. There are at most $f < \lfloor \frac{n}{3} \rfloor + 1$ faulty processes. Therefore they can only set $\lfloor \frac{n}{3} \rfloor$ entries of A. Thus, the faulty processes cannot force m to be outside of I, therefore $m \in I$.

```
   /* accepts a vector of inputs and returns one item from the vector, never returns ⊥          */
 1 function select_value(A)
       Data: 0 ≤ α < ⌈n/6⌉ − 1 a parameter reflecting recurring transient fault distribution
       /* Create new array A_⊥ as a copy of A with ⊥ values removed                             */
 2     A_⊥ ← remove_bottom(A);
       /* After removing bottom values, we get a new array length k ≤ n                        */
 3     k ← |A_⊥|;
       /* Take C to be a dictionary where keys are from V and values are their frequency in A_⊥ */
 4     C ← an empty dictionary;
 5     foreach i ∈ [1...k] do
 6         C[A_⊥[i]] ← C[A_⊥[i]] + 1;
       /* Take m to be the most common value, that is, the key with the highest value in C      */
 7     m ← arg max_{v∈A_⊥} C[v];
       /* If the most common value appears at least ⌊k/3⌋ + 1 + α times, return it              */
 8     if C[m] ≥ ⌊k/3⌋ + 1 + α then
 9         return m;
       /* Else, return the median of the proposed values                                        */
10     A' ← sort(A_⊥);
11     return A'[⌊k/2⌋];
12 function median_byzantine_agreement(v ∈ V)
       Data: P[1...n] array of addresses of all the parties
       /* Allocate an empty array to hold the initial values of all parties                     */
13     A[1...n] ← ⊥;
       /* Broadcast my initial value to all parties (including self)                            */
14     foreach i ∈ [1...n] do
15         send v to P[i];
       /* Save the initial values received from other parties (including self)                  */
16     foreach value u_i received from party p_i do
17         A[i] ← u_i;
       /* Execute the Multi-Valued Byzantine Agreement protocol with Weak Validity in parallel to
          reach agreement on each value in A                                                    */
18     foreach i ∈ [1...n] do
19         A[i] ← WeakMVBA(A[i]);
20     x ← select_value(A);
21     return x;
```

Algorithm 1: Median-based Byzantine Agreement protocol

- *Notice that in line 10, A' is created by sorting $A_\perp$, which in turn is a copy of A.*
- In cases 2 & 3 again, consider now situations where $A'[\lfloor \frac{n}{2} \rfloor]$ (the median of A) is returned (line 11). At most, $\lfloor n/3 \rfloor$ entries of A have a value outside of I. Therefore, there are at least $\lfloor \frac{2n}{3} \rfloor + 1$ entries of A with a value inside of I. The same holds for A', as a copy of the values of A. This implies that values of A that are outside of I are either in the top third or the bottom third, but never in the middle of A'. Therefore, the median of A must be inside of I, that is, $A'[\lfloor \frac{n}{2} \rfloor] \in I$.

□

Theorem 3. *Algorithm 1 implements the Multi-valued Byzantine Agreement specification (Definition 2) with interval validity in systems with less than $\lfloor n/3 \rfloor$ Byzantine processes.*

Proof. Follows from Lemmas 1 and 2. □

Lemma 4. *Any Byzantine agreement protocol that satisfies weak validity remains consistent despite transient faults.*

Proof. Unlike a Byzantine process, a process that experiences a transient fault broadcasts the same value to all the other processes. It can differ from the value the process computed in the previous pulse, or even be the default value $\perp$. Therefore, the value's impact on the decision computation is consistent across the non-faulty processes. □

Theorem 5. *Algorithm 1 implements the Multi-valued Byzantine Agreement specification (Definition 2) with interval validity in systems with less than $\lfloor n/3 \rfloor$ Byzantine processes and less than $\alpha < \lceil n/6 \rceil - 1$ malicious transient faults.*

Proof. To prove consistency and interval validity still hold despite additional malicious transient faults, we consider the case in which a system of n processes executing Algorithm 1 experiences α malicious transient faults in addition to having at most $\lfloor n/3 \rfloor$ Byzantine processes. First, we show that consistency holds, then we show validity holds up to the set parameter α of malicious transient faults. To do so, we reexamine the cases from the proof of Lemma 2, this time considering malicious transient faults.

Consistency follows from Lemma 1 and Lemma 4. By Lemma 4, every entry $A[i]$ where i is the index of a process with a transient fault is consistent across the non-faulty processes. Consider again the three possible cases for the values of A from the proof of Lemma 2, this time, accounting for the additional transient faults.

- In case 1, as in Lemma 2, interval validity property is trivially satisfied
- In case 2 & 3, consider situations where m is returned (line 9). Let $x \notin I$ be the most common value proposed by a process with a transient fault. Suppose then that x is the most common value in A, this implies that after line 7, $m = x$. Byzantine processes do not control transient processes. Therefore, in order to maximize the values in A to be outside of I, all Byzatine processes must propose the value x. Thus, the frequency of x in A is at most $\alpha + \lfloor n/3 \rfloor$, due to the assumption of at most α transient faults. But due to the condition on line 8, m will only be returned if the frequency of m is greater than $\lfloor n/3 \rfloor + 1 + \alpha$. Therefore, faulty processes (Byzantine + transient) will always fall short of the threshold. Therefore, if m is returned, then $m \in I$.
- In case 2 & 3 again, consider now situations where $A'[\lfloor \frac{n}{2} \rfloor]$ is returned (line 11). There are at most $\lfloor n/2 \rfloor - 1$ values outside of I in A since $\alpha < \lceil n/6 \rceil - 1$. Therefore there are at least $\lfloor \frac{n}{2} \rfloor + 1$ values inside of I in A. *Notice that in line 10, A' is created by sorting $A_\perp$, which in turn is a copy of A.* The same holds for A'. The values outside of I are placed in either the bottom half or top half of A', but in either case, there are at most $\lfloor n/2 \rfloor - 1$ of them Thus $A'[\lfloor \frac{n}{2} \rfloor] \in I$.

□

Corollary 6. *If the values yielded by recurrent transient faults are maliciously chosen, the bound on the tolerated recurrent transient faults is $\alpha = \lceil n/6 \rceil - 1$. If the values of the recurrent transient faults are (*e.g., *uniformly) distributed, then α reflects the expected number (with safety factor) of recurrent transient faults that may happen to choose a malicious value.*

We assume adversarial behavior if there is no prior knowledge regarding the distribution of transient faults. Therefore, we can tolerate at most $\lceil n/6 \rceil - 1$ transient faults since the non-faulty processes must have a majority to guarantee interval validity. However, transient faults may be regarded as non-malicious; viewed as a change of a value to hold a value chosen according to a uniform distribution, their number only slightly contributes to the value chosen by the Byzantine processes.

5 Byzantine and Transient Faults Resilient State Machine Replication

The following algorithm solves the state machine replication task in a system of n processes, where up to $\lfloor n/3 \rfloor$ may be Byzantine and up to $\lceil n/6 \rceil - 1$ may experience recurring transient faults per synchronous round. The algorithm achieves this by reaching agreement on both the current state and the next input using repeated invocations of Algorithm 1, the multi-valued Byzantine agreement with interval validity. Execution proceeds in iterations called Pulses, each consisting of several synchronous communication rounds. As mentioned in Section 2, global pulses are provided by an external timing mechanism that defines the start of each round. We clarify that within each Pulse, several communication rounds are executed, and the external mechanism is responsible for coordinating the algorithm's steps. Each *Pulse* is triggered externally, and the algorithm waits for the next *Pulse* to trigger before executing the next iteration of the main loop.

The initialization steps preceding the `Pulse` loop in Algorithm 2 are included for completeness, but they are not essential for ensuring self-stabilization. Rather, they help define the starting state when reasoning about eventual safety, in line with the discussion in [9], which emphasizes the role of initial values in comparing the guarantees of self-stabilizing versus non-self-stabilizing algorithms.

```
1 function state_machine_replication(input, s)
      Global Automaton: M a state machine with states Q = {q_0 ... q_m}, alphabet Σ, and transition
                        function δ : Q × Σ → Q
      /* Initialization for completeness, see [9]                                          */
2     input_value ← input;
3     current_state ← s;
      /* Main loop, each iteration is externally triggered by a Pulse, the execution
         waits until the next Pulse                                                        */
4     Upon Pulse
          /* Agree on next input                                                           */
5         input_value ← median_byzantine_agreement(input_value);
          /* Agree on current state                                                        */
6         current_state ← median_byzantine_agreement(current_state);
          /* Apply state transition from state current_state with input input_value on
             machine M                                                                     */
7         Apply state δ(current_state, input_value) to M;
```

Algorithm 2: State Machine Replication protocol

The following lemma establishes the correctness of Algorithm 2. For simplicity, we assume that recurring transient faults may only occur at the beginning of each `Pulse`. That is, when a new instance of the Byzantine agreement protocol is invoked, each non-Byzantine process broadcasts either a correct value from its local state or a corrupted value due to a transient fault. Any other types of state deviation or faulty behavior are conservatively attributed to Byzantine processes.

Lemma 7. *Consider the execution of Algorithm 2 between two successive pulses, $Pulse_i$ and $Pulse_{i+1}$, in the presence of less than $\lfloor n/3 \rfloor$ Byzantine processes and less than $\lceil n/6 \rceil - 1$ processes with corrupted state (due to recurrent transient fault), then all non-Byzantine processes reach the same state just before $Pulse_{i+1}$.*

Proof. Each participant p_i starts the execution with some input $input_i$ and state s_i, which is its replica of the replicated state machine. In line 5, the participants execute Algorithm 1 with $input_i$ as input; thus, by Theorem 5, every non-faulty process has the same value in *input_value* after line 5. In line 6, every non-faulty process proposes a state; thus, by Theorem 5, after line 6, every non-faulty participant has the same value in *current_state*. Due to the determinism of δ, since both *input_value* and *current_state* are consistent across the non-faulty parties, $\delta(current\_state, input\_value)$ is the same for all non-faulty parties. □

Lemma 8. *Just before every pulse, $Pulse_i$, $i > 1$, every non-faulty process executing Algorithm 2 is in a state that is derived from a state of a non-faulty process just before $Pulse_{i-1}$ and the commonly decided input $input_{i-1}$ despite less than $\lfloor n/3 \rfloor$ Byzantine faults and less than $\lceil n/6 \rceil - 1$ transient faults.*

Proof. By Lemma 7, after execution of $Pulse_i$, every non-faulty process is in the same state, denoted as s. Due to recurring transient faults, some of the non-faulty processes may change to a state different from s. Let r be the number of non-faulty processes that experience a transient fault after the execution of $Pulse_i$. In line 6, all non-faulty processes propose state s while the r transient faulty and Byzantine processes propose some other state. There are at most $\lceil n/6 \rceil - 1$, therefore $r < \lceil n/6 \rceil - 1$. At least $\lfloor n/2 \rfloor + 1$ non-faulty processes, thus at least $\lfloor n/2 \rfloor + 1$ processes propose the value s for the agreement step in line 6. Therefore, in line 7 of Algorithm 1, m is chosen to be s in all non-faulty processes. Due to the majority condition in line 8, m is returned in line 9 (both

lines of Algorithm 1). Thus, after line 6, it holds that $current\_state = s$ for every non-faulty process. Theorem 5 shows that $input\_value$ is consistent across the non-faulty process. Therefore, every non-faulty process will apply the state $\delta(s, input\_value)$ at the end of the execution of $Pulse_{i+1}$ (line 7). □

We conclude the series of proofs with the following theorem that declares a constant stabilization time, too.

Theorem 9. *In the presence of* $\lceil n/3 \rceil - 1$ *Byzantine participants and* $\lceil n/6 \rceil - 1$ *recurrent transient faults, Algorithm 2 satisfies the agreement and order requirements, satisfying strong validity on the state and interval validity on the inputs, from the second replicated state and onwards.*

Benign transient faults. We now discuss benign, recurring, transient faults, where the outcome of the recurring transient fault is a value chosen randomly and uniformly across all possible values. The discussion regards the Byzantine agreement on the state, as the Byzantine agreement on the inputs may not be subject to recurring transient faults. Recall that the Byzantine agreement on the inputs yields interval validity using only the median criteria.

Let x be the total number of recurring transient faults in a pulse $Pulse$. Let y be the most popular value among the z possible values that can be obtained due to recurrent transient faults. Let w be the number of processes with value y. Since the recurrent transient faults are randomly chosen, w is a function of x and z. Since the $\lceil n/3 \rceil - 1$ Byzantine participants can benefit by joining a value y to compete with the $\lfloor 2n/3 \rfloor + 1 - x$ non-Byzantine that do not experience recurrent transient faults in $Pulse$, the strong validity holds when $\lceil n/3 \rceil - 1 + w < \lfloor n/3 \rfloor + 1 + \alpha < \lfloor 2n/3 \rfloor + 1 - x$.

In other words, in a given $Pulse$, the $\lceil n/3 \rceil - 1$ Byzantine processes may choose to join the most popular value resulting from the randomly and uniformly chosen values of recurrent transient faults. When the number of recurrent transient faults is slightly less than $\lceil n/3 \rceil$, (leading to a total of almost two-thirds of the processes being faulty) and the possible number of possible values of the recurrent transient faults outcomes, z, is large (say, exponentially larger than the number of processes) then the value chosen by the non-faulty processes has a high probability of staying the most popular value. Schemes mentioned in the related work cannot support strong validity under such severe settings. In particular, any scheme based on the mean may decide on a value (at least slightly) different from the noncorrupted state of a non-Byzantine participant.

Thus, we obtain the following theorem:

Theorem 10. *When the recurring transient faults are uniformly distributed, there are* α *values that preserve strong validity on the state with high probability even when the number of recurring transient faults exceeds* $\lceil n/6 \rceil$.

6 Conclusions and Discussions

We investigated self-stabilizing Byzantine-tolerant algorithms that cope with arbitrary initial configurations (due to the occurrence of any number of transient faults) and Byzantine behaviours, converging to a desired behaviour despite Byzantine and recurrent transient faults. A replicated state machine that withstands such a combination of faults and maintains a strong validity property for the state component under such severe conditions is presented. Interestingly, in order to verify the strong properties of self-stabilizing Byzantine-tolerant state machine replication, our implementation is constructed on top of a self-stabilizing Byzantine-tolerant agreement protocol that only needs to guarantee discrete interval validity (a weak form of validity). Our replicated state machine transition benefits from strong validity for the state component and nondefault interval validity for the outside inputs. Still, when the inputs of many non-Byzantine participants are identical, a reasonable assumption in many scenarios, strong validity is also guaranteed for the outside inputs.

Practical and timely applications include Blockchain Oracle's services that report stock prices or currency exchange rates. Current practice is ad hoc, mostly using a combination of meetings and voting. Our infrastructure enables new such services with provable systematic guarantees.

Further note that the assumption of a global clock pulse may be relaxed using versions of self-stabilizing Byzantine clock synchronization algorithms, *e.g.,* [11, 26] that withstand recurring transient faults. As the protocol we employ [36] uses two rounds to complete the consensus, we may start the activity of [36] in every odd clock value. For other synchronous consensus, a similar modulo approach can be used.

Acknowledgments

This work was supported by the Google Research Grant, the Rita Altura Trust Chair in Computer Science, the BGU Data Center, the Frankel Center for Computer Science, and the Israeli Science Foundation (Grant No. 465/22). This work was also supported by Vinnova (Strategic Vehicle Research and Innovation, FFI) under project 2024-03687, 'MAGIC: Meeting Automotive Liability Challenges through Forensics Soundness'.

References

[1] Yehuda Afek and Shlomi Dolev. Local stabilizer. *J. Parallel Distributed Comput.*, 62(5):745–765, 2002.

[2] Pith Data Association. Pyth network: A first-party financial oracle. `chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.pyth.network/whitepaper_v2.pdf?ref=pyth-network.ghost.io`, September 2023. As of 21 May 2023.

[3] Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

[4] Rida A. Bazzi and Maurice Herlihy. Clairvoyant state machine replication. *Inf. Comput.*, 285(Part):104701, 2022.

[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[6] Alexander Binun, Thierry Coupaye, Shlomi Dolev, Mohammed Kassi-Lahlou, Marc Lacoste, Alex Palesandro, Reuven Yagel, and Leonid Yankulin. Self-stabilizing Byzantine-tolerant distributed replicated state machine. In *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, 2016, November 7-10, 2016, Proceedings*, volume 10083 of *LNCS*, pages 36–53, 2016.

[7] Lorenz Breidenbach, Alex Coventry, Christian Cachin, Steve Ellis, Andrew Miller, Ari Juels, Brendan Magauran, Sergey Nazarov, Alexandru Topliceanu, and Fan Zhang. Chainlink 2.0 and the future of decentralized oracle networks. `https://chain.link/whitepaper`, April 2021. As of 21 May 2025.

[8] Dan Brownstein, Shlomi Dolev, and Muni Venkateswarlu Kumaramangalam. Self-stabilizing secure computation. *IEEE Trans. Dependable Secur. Comput.*, 19(1):33–38, 2022.

[9] Sylvie Delaët, Shlomi Dolev, and Olivier Peres. Safe and eventually safe: Comparing self-stabilizing and non-stabilizing algorithms on a common ground. In Tarek F. Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems, 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings*, volume 5923 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2009.

[10] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[11] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986.

[12] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[13] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller. Self-stabilizing Byzantine tolerant replicated state machine based on failure detectors. In *Stabilization, Safety, and Security of Distributed Systems (SSS 2018)*, volume 11201 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2018.

[14] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Self-stabilizing Byzantine tolerant replicated state machine based on failure detectors. In *Cyber Security Cryptography and Machine Learning - Second International Symposium, CSCML 2018, Beer Sheva, Israel, June 21-22, 2018, Proceedings*, volume 10879 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2018.

[15] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chic. J. Theor. Comput. Sci.*, 1997, 1997.

[16] Shlomi Dolev and Matan Liber. Towards self-stabilizing blockchain, reconstructing totally erased blockchain. *Inf. Comput.*, 285(Part):104881, 2022.

[17] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *J. ACM*, 51(5):780–799, 2004.

[18] Romaric Duvignau, Michel Raynal, and Elad Michael Schiller. Self-stabilizing Byzantine fault-tolerant repeated reliable broadcast. *Theoretical Computer Science*, 967:113–130, 2023.

[19] Romaric Duvignau, Michel Raynal, and Elad Michael Schiller. Self-stabilizing multivalued consensus in the presence of Byzantine faults and asynchrony. *Theor. Comput. Sci.*, 1039:115184, 2025.

[20] Chryssis Georgiou, Ioannis Marcoullis, Michel Raynal, and Elad Michael Schiller. Loosely-self-stabilizing Byzantine-tolerant binary consensus for signature-free message-passing systems. In Karima Echihabi and Roland Meyer, editors, *Networked Systems - 9th International Conference, NETYS 2021, Virtual Event, May 19-21, 2021, Proceedings*, volume 12754 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2021.

[21] Chryssis Georgiou, Michel Raynal, and Elad Michael Schiller. Self-stabilizing Byzantine-tolerant recycling. In Shlomi Dolev and Baruch Schieber, editors, *Stabilization, Safety, and Security of Distributed Systems - 25th International Symposium, SSS 2023, Jersey City, NJ, USA, October 2-4, 2023, Proceedings*, volume 14310 of *Lecture Notes in Computer Science*, pages 518–535. Springer, 2023.

[22] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Comput.*, 20(1):53–73, 2007.

[23] Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

[24] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Mae Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2):4:1–4:49, 2018.

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[26] Christoph Lenzen and Joel Rybicki. Self-stabilising byzantine clock synchronisation is almost as easy as consensus. *J. ACM*, 66(5):32:1–32:56, 2019.

[27] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[28] Darya Melnyk and Roger Wattenhofer. Byzantine agreement with interval validity. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018*, pages 251–260. IEEE Computer Society, 2018.

[29] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. White paper, 2008.

[30] Amirmohammad Pasdar, Young Choon Lee, and Zhongli Dong. Connect api with blockchain: A survey on blockchain oracle implementation. *ACM Comput. Surv.*, 55(10), February 2023.

[31] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[32] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.

[33] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[34] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[35] David Stolz and Roger Wattenhofer. Byzantine agreement with median validity. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[36] Russell Turpin and Brian A. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, 1984.

A Appendix

A.1 Byzantine Agreement protocol of Turpin and Coan [36]

Below is the exact verbatim definition of the protocol in [36].

> Each process sends its initial value to every other process in the first round. A process is said to be perplexed if, in the first round, it receives at least as many $(P - T)/2$ initial values different from its own. Processes that are not perplexed are said to be content. Each perplexed process sends a message to every other process in the second round. The semantics of this message are just "I am perplexed".
>
> Each process maintains three local variables: two arrays indexed by the process number and a boolean. These variables are assigned values during the first two rounds. For process j, and $i \neq j$, these variables are defined as follows:
>
> $v(j)$ The processe's initial value.
>
> $v(i)$ The initial value received from process i
>
> $p(j)$ A boolean that is set true if and only if process j is perplexed, that is, $v(j) \neq v(i)$ for at least as many as $(P - T)/2$ distinct values of i
>
> $p(i)$ A boolean that is set true if and only if process i sent a message claiming it is perplexed
>
> alert A boolean that is set true if and only if at least as many as $P - 2T$ elements of p are true
>
> The binary computation is used to reach an agreement on alert. If the binary computation agrees alert = true, there are correct processes with different initial values from V. In this case, all correct processes use a predefined default value from V as the result of the extended computation. If agreement is alert = false, then all correct content processes have the same initial value from V. This value is the result of the extended computation. Perplexed processes deduce this result by using the initial value that is common to a majority of the content processes. Each perplexed process tabulates as votes the values $v(j)$ for which $p(j)$ is false. The majority vote is for the value favored by the correct content processes.

Next, we present a pseudocode format for the algorithm of [36].

```
1  function turpin_coan_ext(j, v ∈ V)
       Data: P[1...n] array of addresses of all the parties
       /* Allocate an empty array to hold the initial values of all parties */
2      A[1...n] ← ⊥;
       /* Counter to check perplexity */
3      c ← 0;
       /* Counter to check alertness */
4      d ← 0;
       /* Broadcast my initial value to all parties */
5      foreach i ∈ [1...n] do
6          send v to P[i];
       /* Save the initial values received from other parties */
7      foreach value u_i received from party p_i do
8          A[i] ← u_i;
       /* Check if current party is perplexed */
9      foreach i ∈ [1...n] do
10         if A[i] ≠ v ∧ i ≠ j then
11             c ← c + 1;
               /* If more than (n−f)/2 sent values are different from the current party, then
                  the current party is perplexed */
12             if c ≥ (n−f)/2 then
                   /* Broadcast the fact that the current party is perplexed to all other
                      parties */
13                 foreach i ∈ [1...n] do
14                     send message "p_j is perplexed" to P[i];
                   /* Stop checking for perplexity */
15                 break;
       /* Save the perplexity flags received from other parties */
16     foreach message "p_i is perplexed" received from party p_i do
17         d ← d + 1;
           /* If more than n − 2t parties are perplexed, then turn on the alert flag[36],
              that is, return ⊥ */
18         if d > n − 2f then
19             return ⊥;
20     alert ← binary_agreement(alert);
       /* If the system is not alert, return the majority value from the received values
          */
21     return the most common value in A;
```

Algorithm 3: Byzantine Agreement protocol of Turpin and Coan [36]