When Consensus Meets Self-stabilization*
Self-stabilizing Failure-Detector, Consensus and Replicated State-Machine
(Extended Abstract)

Shlomi Dolev¹, Ronen I. Kat¹, and Elad M. Schiller²

¹ Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il, kat@cs.bgu.ac.il
² Division of Computer Science and Engineering, Chalmers University of Technology.
Göteborg, Sweden
elad@chalmers.se.

Abstract. This paper presents a self-stabilizing failure detector, asynchronous consensus and replicated state-machine algorithm suite, the components of which can be started in an arbitrary state and converge to act as a virtual state-machine.

Self-stabilizing algorithms can cope with transient faults. Transient faults can alter the system state to an arbitrary state and hence, cause a temporary violation of the safety property of the consensus. New requirements for consensus that fit the on-going nature of self-stabilizing algorithms are presented. The wait-free consensus (and the replicated state-machine) algorithm presented is a classic combination of a failure detector and a (memory bounded) rotating coordinator consensus that satisfy both eventual safety and eventual liveness.

Several new techniques and paradigms are introduced. The bounded memory failure detector abstracts away synchronization assumptions using bounded heartbeat counters combined with a balance-unbalance mechanism. The practically infinite paradigm is introduced in the scope of self-stabilization, where an execution of, say, $2^{64}$ sequential steps is regarded as (practically) infinite. Finally, we present the first self-stabilizing wait-free reset mechanism that ensures eventual safety and can be used in other scopes.

Keywords: Failure Detector, Consensus, State-Machine, Wait-Free, Distributed Reset, Self-Stabilization.

1 Introduction

Self-stabilization. Self-stabilization [13,14] is a fundamental property of a system that ensures automatic recovery of the system following the occurrence of

* Partially supported by the Israeli Ministry of Science, the Lynn and William Frankel Center for Computer Sciences, and the Rita Altura Trust Chair in Computer Sciences.
faults. Self-stabilizing systems are designed to be started in an arbitrary state and to converge to exhibit a desired behavior of the system. Recovery oriented computing, autonomic computing and self-* computing, e.g., [24], are research and industrial terms used extensively nowadays. The research and industrial activities in these fields may greatly benefit from using the well-understood and rigorous fundamentals of self-stabilization.

Consensus and failure detectors. Consensus is a fundamental and, in a sense, a complete problem in distributed computing. A distributed task is reduced to a centralized task by agreeing on the current distributed inputs (and the system state) and by consequently computing the fitting outputs. Unfortunately, as proved in [21] (and in [10,32] for shared memory) there is no asynchronous consensus algorithm even in executions in which just one process may stop taking steps. Fortunately, there are consensus algorithms, e.g., [29], that preserve safety (i.e., processes never decide on different values). Liveness is achieved in well-behaved executions (excluding, for instance, executions chosen according to [21]). Failure detectors, e.g., [8], form a mechanism that captures the synchronization requirements to obtain consensus liveness.

The consensus task is defined as a one-shot task, where the distributed system is started with inputs for each process and every non-crashed process must decide\(^1\) on a common value\(^2\) that appeared in one of the inputs\(^3\). In the scope of self-stabilization it is possible that the processes are started in a state in which each of them has already decided on a different value and does not take any further steps. Hence, one-shot self-stabilizing consensus is impossible. The definition of the consensus task in the scope of self-stabilization should incorporate the need for repeated invocations of the consensus, for example as the means of implementing a replicated state-machine. In such a case, the requirements for the self-stabilizing consensus must ensure eventual termination of initialized or non-initialized execution, as well as, all the consensus requirements for the set of processes that initialized a new session of the consensus execution. For instance, when considering the elegant algorithm presented in [29], and allowing an arbitrary state (and counters values), it is unclear whether there is a set of executions starting in such a state that will ensure termination. This is simply because, wrap around of counters is not considered. One may argue that a counter of 64 bits is practically infinite. This argument does not hold in the scope of self-stabilization since a single transient-fault may cause the counter to reach its upper bound at once. However, as we discuss in the sequel, we do consider an execution of \(2^{64}\) sequential steps as practically infinite.

Replicated state-machine. A bold application for consensus is an implementation of a fault-tolerant replicated state-machine, e.g., [29]. The abstraction of a replicated state-machine has been proven important in several domains, e.g., [7,16,17].

---

\(^1\) This is a termination requirement for the largest possible set of executions.

\(^2\) The agreement requirement must hold in every execution.

\(^3\) The validity requirement must hold in every execution.
Related work. The literature on failure detectors is rich; see for example the recent surveys [22,35]. We focus on the eventual strong $\Diamond S$ failure detector that is known to be the weakest failure detector required to solve consensus [8,9] in message-passing systems, when the majority of the processes are non-crashed. The weakest failure detector for the case when more than half of the processes may fail is considered in [12]. We note that the restriction on the number of failed processes does not apply to the shared-memory settings. This is due to the fact that a message can be delayed and the sending process can take additional steps, but a write to a register cannot be hidden from the other processes when the writing process takes additional steps. The $\Diamond S$ failure detector is an unreliable mechanism for detecting crashes. The $\Diamond S$ failure detector guarantees that (1) eventually every process that has crashed is permanently suspected by every non-crashed process (strong completeness property), and (2) there is a time after which some non-crashed processes are never suspected by any of the non-crashed processes (eventual weak accuracy property). The few self-stabilizing failure detectors mentioned in the literature uses message-passing systems. In [6] a self-stabilizing failure detector is presented in partial synchronous settings. In addition to the fact that the failure detector of [6] is not designed for shared memory systems, [6] does not handle the case of $n - 1$ crashed processes. The algorithm in [27] uses randomization to construct a self-stabilizing perfect failure detector. Recent research on failure detectors is focused in identifying the weakest synchrony requirements necessary for solving consensus. Aguilera et al [2] presented a leader election algorithm (i.e., the $\Omega$ failure detector) that requires $n - 1$ eventual timely outgoing links for at least one non-crashed process. In [3] the requirements were weakened to $f$ eventual timely outgoing links, where $f$ is the number of crashed processes. In [34] it is assumed that the outgoing links may be moving (i.e., the destinations are not fixed). Note that [2,3,34] require unbounded counters while a self-stabilizing failure detector may use only bounded counters.

Consensus is also an extensively studied topic in distributed computing, see, for example, [5,33]. In the case of eventual (unreliable and non-randomized) failure detectors there is a known bounded memory implementation for specific settings. For example, in [23], the case of three processes, where only one may fail, is considered. However, for the general case, to the best of our knowledge, there is no memory bounded consensus algorithm for asynchronous systems. The known algorithms require unbounded counters to preserve safety. We note that a bounded consensus algorithm is known for the case in which failures are instantly detected. Such failure detectors are inherently not self-stabilizing. A self-stabilizing consensus algorithm eventually ensures safety (in the presence of crash failures) and liveness (under reasonable synchronization assumptions for the failure detector).

Using asynchronous reset with the purpose of circumventing unbounded values is discussed in [28]. This asynchronous reset is not wait-free and requires a complete knowledge regarding failures of links and processes. That is, the algorithm is notified when edges become either active or inactive. A resetable vector
clocks that provide non-blocking resets in the absence of faults is presented in [4]. In case faults occur, the algorithm requires one blocking reset (i.e., a global reset) before the system stabilizes. Another solution for circumventing unbounded values is self-stabilizing timestamps [1]. $O(n)$ invocations of weak timestamps procedures (each invocation requires $\bar{O}(n)$ operations) are used in [1] in order to achieve bounded timestamps. We propose a wait-free reset that can be used for implementing self-stabilizing bounded timestamps which requires only $O(n)$ operations per invocation (following the convergence of the wait-free reset).

**Our contribution.** We present a new self-stabilizing failure detector, consensus and replicated state-machine algorithm suite in shared-memory settings. All components can be started in an *arbitrary state* and converge to act as a virtual state-machine. Thus, we gain a self-stabilizing infrastructure for the execution of self-stabilizing applications. In addition, we present the first wait-free reset technique.

- **Failure detector.** Our self-stabilizing failure detector does not use randomization and is designed for partial synchronous settings. In such settings, the interleaving order of (non-crashed processes) steps is eventually somewhat restricted. Roughly speaking, each process has a bounded heartbeat counter. The relative advances of the counter are compared to other processes. To avoid confusion that could arise due to the fact that the counters are bounded, we use wrap around flags that indicate when a process has wrapped its counter. We show that bounded flags are sufficient for computing of the relative speed of steps. This simple mechanism identifies crashed processes and allows the active set of processes to continue in their consensus task. Our failure detector uses synchronization assumptions that are analogous to the settings assumed in [2]. Note that our algorithm uses shared memory and not message-passing as in [2]. Moreover, our algorithm is memory bounded while [2] requires unbounded counters.

- **Consensus.** Our self-stabilizing algorithm is able to achieve eventual safety (and eventual liveness) using bounded memory. The consensus algorithm assumes the existence of the obtained self-stabilizing $\Diamond S$ failure detector. Generally speaking, the algorithm is a rotating coordinator algorithm that is based on the (memory unbounded) algorithm in [31]. The processes are sequentially assigned to be the consensus coordinator. A coordinating process that takes steps and is not suspected as crashed, will successfully bring the system to a *univalent configuration* (a configuration after which all decisions have the same value). A univalent configuration is reached, once the coordinating process succeeds in writing the proposed consensus value (which is one of the inputs) and at the moment that every process reads this proposed value prior to becoming current coordinator. The use of the eventual strong failure detector ensures that eventually one of the processes (i.e., $p$) is never suspected. It is possible that a univalent configuration is reached before the stage of the execution in which $p$ is never suspected as crashed. Otherwise, when $p$ becomes the current coordinator, no other process will become a new coordinator, and a univalent configuration is, therefore, imposed by $p$.

- **Replicated state-machine.** We use epochs of consensus invocations for deciding on the transition of the (distributed) state-machine. Note that due to the
When Consensus Meets Self-stabilization

Our contribution is also in the modification made to the traditional replicated state-machine, where the machine has to decide on a common state as well. Thus, we tolerate an arbitrary initial configuration in which processes have different notions regarding the current state of the state-machine. We suggest the use of hash functions to avoid copying the entire state when possible.

- Practically infinite executions and self-stabilizing wait-free reset. We define a set of executions that we call \textit{practically infinite executions}. Practically infinite executions are executions in which the time complexity measured by the longest \textit{happened before chain} \cite{30} is longer than, lets say $2^{64}$. Note that even if each step takes a single nanosecond, no computer system will last such a long period of time, and no client will wait for the last step in this sequence. In case the suite of asynchronous consensus algorithm and failure detector does not reach a decision within a period of time that corresponds to $2^{64}$ sequential steps, the decision value will clearly be obsolete.

The above argument is used in the scope of self-stabilization for the first time. Rather than assuming that $2^{64}$ counter is infinite, we argue regarding the amount of time required for a chain of $2^{64}$ sequential steps of counter increments. If due to a transient fault the counter reaches its upper bound at once, a wait-free reset takes place ensuring a subsequent practically infinite resetless execution.

\textbf{Paper organization.} The rest of the paper is organized as follows: The system settings appear in Section 2. The self-stabilizing failure detector is described in Section 3. The self-stabilizing consensus and state-machine algorithm appear in Section 4. Concluding remarks appear in Section 5. Details and proofs are omitted from this extended abstract and can be found in \cite{18}.

2 System Settings

We consider a shared memory system with a set of $\Pi$ communicating entities that we call \textit{processes}. There are $n$ processes in the system, with each process having a distinct identifier in the range of $0, \ldots, n-1$. Each process $p$ is associated with a set of atomic multi-reader/single-writer registers. A process $p \in \Pi$ can write only in its associated register and can read any register. Process $p$ writes the value $v$ to register $R$, by using the command \texttt{write}(R,v). Process $p_i$ reads the value of register $R$ by using the command \texttt{read}(R). We use capital letters for register names, e.g., $R_p$. We use lower case letters, e.g., $r_p$, for denoting the local variables of processes that contain the values read by the process from the register, i.e., $r_p$ contains the last value read from $R_p$.

Each process $p$ is modeled by a state-machine. We use a program in pseudo code to describe the state space and the transition function of $p$. In every given instance, the state of a process $p$ includes the process program counter, the values of $p$’s local variables, and the values of the registers associated with $p$. A state transition of a process $p$ is defined by an \textit{(atomic) step}. A step consists of a sequence of internal (program) computations that ends in a single \texttt{read} or
write operation to a register. A system configuration consists of the states of all the processes. An execution is a sequence of configurations and steps \( E = (c_1, a_1, c_2, a_2, \ldots) \), where configuration \( c_{i+1} \) is reached by executing a step \( a_i \) by one process. A task is defined by a set of executions called legal executions (\( LE \)). A configuration \( c \) is a safe configuration for an algorithm and task \( LE \) provided that any execution that starts in \( c \) is a legal execution (belongs to \( LE \)). An algorithm is self-stabilizing with relation to task \( LE \) if every infinite execution of the algorithm reaches a safe configuration with relation to the algorithm and the task. A process may fail by permanently stopping to execute steps. We say that such a process crashed. Hence, it does not execute any step in a suffix of the execution. Note that the output of a read command to a shared register of crashed process \( p \) is constant, as only \( p \) can write to its registers. A process that executes a step infinitely often is said to be non-crashed.

Our self-stabilizing consensus and wait-free reset algorithms are designed for asynchronous systems (with a failure detector) and our self-stabilizing failure detector algorithm assumes the existence of a (unknown to the processes) bound on the (relative) execution speed of the processes.

### 3 Self-stabilizing Failure Detector

Fischer, Lynch and Paterson [21], (and then [10,32]) have shown that consensus cannot be reached in message-passing (and then in shared memory) asynchronous environments. A failure detector [8] is an oracle that identifies crashed processes and helps to separate safety and liveness concerns in a way that may lead to a feasible safe solution for consensus in which liveness depends only on (synchronization or) scheduling of actions while safety always holds. We present a self-stabilizing ♦S failure detector that satisfies the following properties:

**Property 1 (Strong completeness).** Every execution has a suffix in which (eventually) every process that has crashed is permanently suspected by every non-crashed process.

**Property 2 (Eventual weak accuracy).** Every execution has a suffix in which some non-crashed processes are never suspected by any of the non-crashed processes.

**Partial Synchronous Settings and Requirements.** We assume the existence of a global clock \( t \) that is unknown to the processes. We use real time to argue concerning progress during executions. We make no assumptions regarding local clocks or clock synchronization.

We say that a source process is a process that executes any two successive steps exactly \( \delta \) time units (of the global clock) apart. Such a process is said to be executing steps on time. The definition is similar to the notion of eventual timely output links, in a message-passing model, of a process as in [2]. We also assume that no process executes two successive steps faster than a source process \( p \). Namely, for every non-crashed process \( q \) and any two successive steps \( a_i \) and \( a_{i+1} \) of a process \( q \), there are at least \( \delta \) time units between the time in which the communication actions of \( a_i \) and \( a_{i+1} \) have taken place.
We say that an infinite execution is admissible if it has at least one source process \( p \). Since no process is faster than \( p \), process \( p \) executes at least one step between any two successive steps of any process in an admissible execution. The failure detector task is defined by a set of executions in which the strong completeness (Property \([1]\)) and eventual weak accuracy (Property \([2]\)) hold.

**The Failure Detector.** The heart-beat mechanism is used for detecting crashed processes. Processes that are not crashed signal the rest of the processes by repeatedly changing a value in a register, read by the other processes. The failure detector algorithm maintains a set of suspicious processes in the local variable \( fd \) (failed detection). If process \( p \) suspects process \( q \) to have crashed, then \( q \in fd \).

A non-crashed process \( p \) continuously advances a “heartbeat” counter \((HB_p)\) in order to avoid being mistakenly suspected. Process \( p \) periodically compares the other processes’ counters progress over time.

Since a crashed process \( q \) does not advance its heartbeat counter \( HB_q \), every non-crashed process suspects \( q \) within a finite period of time. Process \( p \) uses a cyclic history array \( h_q \) for recording indications on the last \( k \) heartbeats (counter) of \( q \). If \( q \) increased its counter, \( p \) records 1 in an entry of \( h_q \). Otherwise, \( p \) records 0. Process \( q \) is suspected as crashed by \( p \) only if during the last \( k \) records of \( p \), \( q \) did not advance its heartbeat. Note that the “fastest process” (e.g., the source process) is never suspected since every process marks the continuous progress of the source process. A process that executes steps slowly may be falsely suspected as crashed when the history size, \( k \), is too small. A longer history provides a more robust and accurate indication of crashed processes. However, the need for accumulating a long history before making the decision delays failure indications.

Since a self-stabilizing failure detector has to use a bounded heartbeat counter, each counter is incremented modulo \( m \). In some cases, a slow process may (incorrectly) consider a process that has wrapped its counter, as crashed. Therefore, in case the counter of process \( p \) wraps, \( p \) uses a balance/unbalance \([15,20]\) protocol to ensure that \( p \) is not considered as crashed. That is, \( p \) keeps a flag \( WR_{p,q} \) for every process \( q \). Process \( q \) keeps a copy of \( p \)’s flag in \( LWR_{q,p} \) and makes sure that the copy always equals \( p \)’s value. When \( p \) wraps its counter, \( p \) assigns \( q \)’s flag \((WR_{p,q})\) a new value that is different from the value of \( LWR_{q,p} \) (\( q \)’s copy of \( WR_{p,q} \)). Therefore, when \( q \) reads \( p \)’s flag \((WR_{p,q})\), \( q \) notices the fact that \( p \)’s counter has wrapped. The counter size \((m)\) is chosen in a way that optimizes the number of wraps around (to zero) with relation to balance/unbalance usage. The registers \( WR_{p,q} \) and \( LWR_{q,p} \) contain a value of \( \{0, 1, 2\} \) in order to ensure that \( p \) can introduce a new value and make sure that \( q \) notices the change. The register has three values to ensure correct behavior given that processes \( p \) and \( q \) keep a local variable for both \( WR_{p,q} \) and \( LWR_{q,p} \). The value of the local variable may differ from the value of the register. We say that \( p \) is unbalanced towards \( q \) in case \( WR_{p,q} \neq LWR_{q,p} \). Otherwise, \( p \) is balanced towards \( q \). We use the predicate \( unbalance(q) \) to describe \( p \)’s view (indicated by the values of \( wr_{p,q} \) and \( lwr_{q,p} \)) regarding its state (balanced/unbalanced) towards \( q \). When \( p \)’s counter is wrapped, \( p \) writes a value to \( WR_{p,q} \) (for every \( q \)) in such a way that the predicate \( unbalance(q) \) is true. In case \( wr_{p,q} = lwr_{q,p} \), \( p \) increases \( lwr_{q,p} \).
by one modulo $3$. A process $q$ that notices the wrap indication (i.e., the predicate $\text{unbalance}(q)$ is true) copies $wr_{p,q}$’s value to $\text{LWR}_{q,p}$.

In Figure 1 we present the algorithm for the self-stabilizing failure detector. In every iteration of the program, process $p$ increases the heartbeat counter $HB_p$ (lines 1 and 2) and signals, by executing the procedure $\text{unbalance}_{\text{all}}$ (lines 3 and 4), to all other processes when the counter wraps around to zero. In lines f1 through f5 process $p$ flags an unbalance indication towards every other process $q$. In lines f2 and f3 process $p$ reads $\text{LWR}_{q,p}$ and $\text{WR}_{p,q}$. If $p$ is balanced towards $q$, then process $p$ unbalances and writes to register $\text{WR}_{p,q}$ (lines f4 and f5). A process $p$ becomes balanced towards $q$ by reading register $\text{WR}_{q,p}$ (line 9) and writing the value in register $\text{LWR}_{p,q}$ (line 18).

Process $p$ records the history of process $q$ in the cyclic history array $h_q$. Process $p$ keeps the history of length $k$ for each process. In line 5 process $p$ moves to the next history entry. In line 6 process $p$ initializes the list of new suspects in $\text{newfd}$. In lines 8 through 14 process $p$ reads the heartbeat counter and the balance/unbalance indications of every process $q$ as well as records $q$’s progress. If $q$ increases its counter (i.e., $q$ executed enough steps since $p$ last read $q$’s counter), then $p$ records 1; otherwise, $p$ records 0 in the history entry.

If every entry of $q$’s history is 0 (i.e., $q$ did not execute enough steps during the last $k$ iterations), then $q$ is suspected to have crashed (lines 15 and 16). Otherwise, $p$ does not suspect $q$. In lines 17 and 18, $p$ keeps the last value of $q$’s counter and writes the value of $\text{WR}_{q,p}$ to $\text{LWR}_{p,q}$ (i.e., balances towards $q$). In line 19, process $p$ updates its suspects list $f_d$ with the new computed suspicions.

Note that the requirement for the existence of a source process can be relaxed to, say, the existence of a set of processes that are “fast enough”. A process $p$ is considered “fast enough” when no process suspects $p$ as being crashed. Thus, the proofs hold for a much larger set of executions, for example, executions where a process $p$ exists for which the history associated with $p$ by every other

---

**Fig. 1.** Failure Detector for process $p$
process has at least one non-zero entry. That is, every other process increases its heartbeat at most $k$ times between any two successive heartbeat increases of $p$. The following Theorem states the correctness of the algorithm. Details are omitted from this extended abstract.

**Theorem 1.** *Every admissible execution of the failure detector algorithm (Figure [7]) has a suffix that satisfies the failure detector task. The suffix starts after every non-crashed process executes $(k + 3)\Delta$ steps.***

### 4 Self-stabilizing Consensus and Replicated State Machine

We will now describe the self-stabilizing consensus and the replicated state-machine algorithm using the eventual strong failure detector of the previous section.

A self-stabilizing replicated state-machine is a collection of processes, each of which independently implements a state-machine. Every non-crashed process executes the same sequence of transitions and reaches the same state. In order to guarantee that each process, eventually, executes the same transitions, we employ a sequence of consensus instances, one for each transition. Each instance has an *epoch* number, in which the processes decide on a single value (i.e., the transition) from the possible transitions (i.e., inputs) suggested by each process. We assume that the inputs are provided to the algorithm. The origin of the inputs is outside the scope of our work. The self-stabilizing consensus satisfies the following properties in the presence of the self-stabilizing $\Diamond S$ failure detector:

*Property 3 (Eventual termination).* Every execution has a suffix in which every non-crashed process decides on a value in every epoch.

*Property 4 (Eventual Validity).* Every execution has a suffix in which every non-crashed process decides on the initial value of some non-crashed process in every epoch.

*Property 5 (Eventual Agreement).* Every execution has a suffix in which no two non-crashed processes decide on different values in every epoch.

We use the self-stabilizing consensus to implement a replicated state-machine. The self-stabilizing replicated state-machine guarantees the following properties:

*Property 6 (Eventual Coordination).* Every execution has a suffix starting with some epoch $e$ such that for every epoch $e' > e$, no two processes execute a different transition.

*Property 7 (Eventual Consistency).* Every execution has a suffix starting with some epoch $e$ in which no two processes differ in their machine state in every epoch.
Asynchronous Settings and Requirements. The system is assumed to be completely asynchronous, i.e., there are no timing assumptions. Moreover, processes may be as slow (or fast) as one may choose them to be or they may stop operating altogether. Processes cannot distinguish a slow process from a crashed process. Each process is associated with a single multi-reader/single-writer register $R_p$ that holds $O(n \log n)$ bits.

The task of the self-stabilizing consensus is defined by a set of executions in which eventual termination (Property 3), validity (Property 4) and agreement (Property 5) hold. The task of the self-stabilizing replicated state-machine is defined by a set of executions in which eventual coordination (Property 6) and eventual consistency (Property 7) hold.

The Consensus and the State-Machine. Next we describe the consensus, the replicated state-machine and the wait-free reset that along with the failure detector form a bounded replicated state-machine that is self-stabilizing.

• Consensus with a rotating coordinator. The consensus ensures that no two processes will decide on different values in an epoch. Every process (from the set $\Pi$ of processes) either follows the previous decision of others, or strives to make a decision that the other processes would follow. A process, which does not copy a decision value from a process that has already decided, must execute the following sequence of scan (read from the registers of all processes) and write operations: announce, scan, propose, scan, and decide. Each of such sequences is identified by a unique sequence number that is called a round number.

The algorithm is based on the observation that there is no possible interleaving of such atomic operations that allows different (transition) values for the decide write operation. Consider a process $p$ with the smallest round number $r$ that proposes and subsequently decides with round number $r$. The algorithm states that any process with a round number $r'$ smaller than $r$ will not decide with $r'$. According to the algorithm, every process $q$ with a higher round number that proposes adopts the value proposed (and decided) by $p$. A process with a higher round number that does not adopt $p$’s proposed value must scan prior to $p$’s proposal. This implies that $q$ announced before $p$ proposed. In such a case, $p$ cannot decide since $p$ finds $q$’s higher round number. The above observation (i.e., that implies safety) is based on the assumption that the round numbers are ever increasing. We assume that the round number counters do not wrap in an execution that starts with counters initialized to zero. We show how to achieve such an execution in the sequel.

The system achieves consensus when a single process $p$ executes the above sequence of steps without crashing. A process that tries to execute such a sequence of steps is said to be the coordinator. The failure detector assists in moving the responsibility to the next process when a coordinator is suspected to have crashed. The rotating coordinator paradigm [8] states that in every round $r$ there is a single coordinator $p$ (i.e., $p = r \mod n$) that carries out the above sequence of steps. Every other process $q$ (i.e., $q$ is not a coordinator in $r$) waits for $p$ to decide within round $r$. Process $q$ repeatedly performs scans until some process decides, or until it is obvious (for $q$) that the coordinator will not decide
within $r$. For example, the coordinator (of round $r$) is in a higher round number than $r$ or $q$’s failure detector suspects the coordinator of round $r$. In such cases, $q$ moves to a new round number, by increasing $r$ by one.

**Self-stabilizing replicated state-machine.** The replicated state-machine ensures that every non-crashed process executes the same sequence of transitions and reaches the same state. Each transition is associated with an instance (i.e., epoch number) of the consensus. Every process computes the transition using the decision value (in an epoch) and a fixed, hardwired, transition function (of commands) in order to reach a new state.

Starting from an initial state, each process of the replicated state-machine executes the same transitions due to the fact that the consensus decision values in every epoch are the same. Periodically, process $p$ compares its epoch number and state with other processes. When $p$ finds another process with a higher epoch, $p$ increases its epoch number to the highest epoch ($p$ observed) and copies the state ($p$ read) from the process with the smallest identifier among the processes in the highest epoch.

The state-machine for process $p$ is represented by the tuple $\langle e_p, state_p \rangle$, where $e_p$ is the current epoch number and $state_p$ is the state reached in the previous epoch. When a decision is reached in epoch $e_p$, $p$ executes the $\text{decide}()$ procedure, computes the new state and increases its epoch number to $e_p + 1$. We assume that the epoch and the round numbers are practically unbounded (i.e., the real-time needed to reach the highest value is practically infinite) when the system is started from an initial state. In the case that the system starts from an arbitrary state, we show that there is an execution suffix in which the epoch and the round numbers are practically unbounded. That is, we allow the epoch and the round numbers to be reinitialized to zero.

**Self-stabilizing wait-free reset.** Next, we show how to resolve the contradiction between safety, which requires that epoch and round numbers are ever increasing, and the fact that the counters are bounded and may wrap around (to zero). The fact that a counter of 64-bits (or more) is practically infinite does not hold in the scope of self-stabilization. A single transient-fault (or incorrect initialization) may cause the counter to reach such a large value at once, not allowing the consensus algorithm to have enough rounds to reach a decision in an epoch. A similar argument applies to epoch numbers and the state-machine transitions. The goal of the reset mechanism is to set all epoch and round number counters (for every non-crashed process) to zero. Thus, following a reset, the consensus and the replicated state-machine algorithm will have practically unbounded number of rounds to reach a decision and an unbounded number of epochs to perform transitions.

We denote by $inf$ the maximal value of the epoch and the round number counters ($inf$ is related to the bounded size of the counter). In addition, we assume that, when a process increases its counter beyond $inf$, the counter value is not changed. We assume that the value $inf$ is very big and is reached only when a counter is initialized (or changed due to a transient fault) to some non-zero (large) value.
Whenever the epoch or the round number of process $p$ reaches $\text{inf}$, $p$ attempts to set all counters to zero by performing a reset and by assigning zero to the epoch and the round numbers. We associate every counter wrap with a reset sequence number. When $p$ performs a reset, $p$ increases by one its reset sequence number $rseq$. We will show that the reset sequence number should be in the range of 0 to $2n$. Each process $p$ keeps track of all the other processes reset sequence numbers in the array $rseq$, and $p$’s reset sequence number is kept in $rseq[p]$. In addition to the reset sequence number, $p$ uses a balance/unbalance protocol instance with every process $q$. The balance/unbalance protocol is used to identify slow or crashed processes. When $p$ performs a reset, $p$ ensures that for every process $q$, it holds that: $wr_{p,q} \neq lwr_{q,p}$ (unbalanced), by assigning a value to $wr_{p,q}$ if needed. That is, if $wr_{p,q} = lwr_{q,p}$, then $p$ increases $lwr_{q,p}$ by 1 modulo 3. Process $p$ evaluates the predicate $\text{isreset}(q)$ to be true if in a configuration $c$ the local variables of process $p$ indicate that the reset sequence number ($rseq_{p}[p]$ and $rseq_{q}[p]$) and/or the balance/unbalance flags ($wr_{p,q}$ and $lwr_{q,p}$) of $p$ and $q$ differ.

We say that $q$ is reset or $q$ is flagged as reset in case the predicate $\text{isreset}(q)$ is true for process $q$. The flag indicates that $q$ needs to set its counters to zero. The goal of a process that performs a reset is to invalidate the values of the epoch and the round counters of other process and signal them to assign zero to these counters thereafter. A process $q$ that is reset, does not strive to propose and decide in the consensus algorithm (and is ignored by other processes). When $q$ notes that process $p$ has flagged $q$ as reset, process $q$ acknowledges the reset by copying the values of $p$’s flags and the reset sequence number and by resetting the epoch/round number to 0. The acknowledgment indicates that $q$ has set its counters to zero according to the reset request.

Another possible function of the reset mechanism is to help maintaining the common state of the replicated state-machine. We suggest using the output of a hash function on the state instead of using the full state. Whenever a process finds a conflict of hash values the process invokes a reset that will set the machine state of each process to an initial, predefined state. In addition, we can enhance the probabilistic nature of hash functions collisions by communicating the full state (instead of the hash value) in every fixed number of epochs.

We now describe the consensus and the replicated state-machine algorithm. Figure 2 describes the registers, variables and macros. Figure 3 contains the procedures and Figure 4 describes the algorithm. The algorithm combines a replicated state-machine that executes transitions and a consensus that decides on the values of the transitions. A process $p$ writes only to register $R_p$. The register consists of a tuple $\langle v, g, e, r, state, wr, lwr, rseq \rangle$. $v$ is $p$’s estimate of the decision value in epoch $e$. The value of $g$ is the consensus phase tag, which can be either announce, propose or decide. $e$ and $r$ are the current epoch and round numbers. $state$ represents the last state of the state-machine. $wr$ and $lwr$ are arrays (of size $n$) of the balance/unbalance values, where the value of $wr[q]$ is used for the unbalancing action by $p$ and $lwr[q]$ is used for the balancing action by $p$ towards $q$. Finally, $rseq$ is an array of the reset sequence numbers (one reset sequence number for each process). In the sequel, we compare instances of $\langle e, r \rangle$. 
We say that \( \langle e_1, r_1 \rangle > \langle e_2, r_2 \rangle \) if \( e_1 > e_2 \), or if \( e_1 = e_2 \land r_1 > r_2 \). We say that a process has decided in epoch \( e \) if a non-reset process \( q \) with a higher epoch number exists, or if a register (of a non-reset process) contains the tag \textit{decide} for epoch \( e \). In such cases, the predicate \textit{decisionExists}(\( e \)) is true.

The procedure \textit{scan()} (Figure 3) reads the registers of all the processes. The values that are associated with the consensus are stored in the set \( m_p \) and the values for determining if process \( q \) is reset are stored in \( wr_q, lwr_q \) and \( rseq_q \). The values in \( m_p \) are tuples of \( \langle q, v_q, e_q, r_q, state_q \rangle \), where \( q \) represents the process identifier, \( v_q \) is the current estimate of process \( q \), \( e_q \) and \( r_q \) are the current epoch and round numbers of \( q \), and \( state_q \) is the state of the state-machine of process \( p \).

![Fig. 2. Definitions for process \( p \)](image)

The procedure \textit{next()} in Figure 3 advances \( p \) to a new round \( r + 1 \). First, \( p \) scans the registers (line g1). In case \( p \) is flagged as reset, \( p \) repeats the scan (line g3), reading the registers values after the reset. Then, \( p \) resets its epoch and round number to zero (line g4), obtains the estimate for the new epoch (line g5) and balances any balance/unbalanced flags and reset sequence numbers (lines g6 through g8). That is, \( p \) copies \( q \)'s unbalanced flag \( (wr_{q,p}) \) to \( lwr_{p,q} \) and copies \( q \)'s reset sequence number \( (rseq_{q}[q]) \) to \( rseq_{p}[q] \). In case no process flagged \( p \) as reset, \( p \) checks if a process has reached an epoch/round number higher than \( p \)'s epoch/round number and verifies the state consistency of the replicated state-machine (lines g9 through g14). In lines g9 and g10 process \( p \) finds, using the function \textit{index} − \textit{max}, a non-reset process \( q \) with the smallest identifier that has the maximal epoch/round number \( \langle e_q, r_q \rangle \). If \( \langle e_q, r_q \rangle > \langle e_p, r_p \rangle \), then \( p \) copies \( \langle e_q, r_q \rangle \) (lines g11 and g12) and the state of \( q \)'s state-machine (line g13). In case \( p \) does not copy an epoch or a round number from another process, \( p \) checks if the epoch/round number reached \textit{inf}. If \( e_p \) or \( r_p \) reached \textit{inf}, then \( p \) performs a reset (lines g15 through g19), by unbalancing (if the balance/unbalance instance from \( p \) to \( q \) is not already unbalanced) all other processes and by increasing its reset sequence number \( rseq_{p[p]} \) as well. Otherwise, in lines g20 and g21, process \( p \) increases its round number. The procedure \textit{decide()} (Figure 3) computes the
new state, advances to the next epoch and obtains the input for the new epoch. Figure 3 describes the consensus and the replicated state-machine code. In lines 1 and 2, process \( p \) initializes the state-machine. In lines 3 through 5, process \( p \) advances to the next round and announces its estimate. In case \( p \) is the coordinator, process \( p \) executes lines 7 through 20. In lines 7 through 10 process \( p \) checks if process \( q \) has already decided. If so, \( p \) decides after coping \( q \)’s decision value and the state of \( q \)’s state-machine. If no process reached a higher round than \( p \), then in lines 13 through 16, \( p \) adopts the latest estimate that a process has proposed. If no process has proposed, then \( p \) proposes its own estimate. In lines 17 through 20, \( p \) verifies once more that no other process has reached a higher round, and if so, \( p \) decides. In case \( p \) is not a coordinator (lines 22 through 29), \( p \) continuously reads the registers (i.e., the epoch/round numbers and the reset indications) and waits for the coordinator to reach \( p \)’s round number. Due to the fact that it is possible for the coordinator not to reach \( p \)’s round, \( p \) stops waiting if (1) \( p \) waits for itself, or (2) \( p \) is reset, (3) a process has decided within the epoch, or (4) the coordinator is suspected as a crashed process. In case \( p \) exits the loop after a process has decided with \( p \)’s epoch number, then \( p \) copies the decision value (lines 27 through 29).

We use the following in our proofs: a process \( p \), which executes steps in epoch \( e \) (i.e., \( e_p = e \)) and writes a tuple with the tag \( \text{decide} \) to register \( R_p \), is said to be decided in epoch \( e \). Note that a process is decided after executing the step in lines 9, 19 or 28. A restless execution of the state-machine is a practically infinite execution, starting in an arbitrary configuration, in which no process flags another as reset.

**Proof overview.** First, we show that there are infinitely many steps in which the round number is increased in a restless execution. In the sequel, we show that in every execution a restless execution exists. We prove that if there is a decision in a certain epoch \( e \), then there is a subsequent configuration in which a process has

```
procedure scan() :
f1. \( m_p \leftarrow \emptyset \)
f2. \( \forall q \in \Pi \) do
f3. \( \langle v_q, q, r_q, \text{state}_q, \text{wr}_q, \text{wr}_q, \text{rseq}_q \rangle \leftarrow \text{read}(R_q) \)
f4. \( m_p \leftarrow m_p \cup \{ (q, v_q, q, r_q, \text{state}_q) \} \)

procedure next() :
g1. scan()
g2. if (isreset(\( p \)))
g3. scan()
g4. \( \langle e_p, r_p \rangle \leftarrow (0, 0) \)
g5. \( v_p \leftarrow \text{new input}(e_p) \)
g6. \( \forall q \in \mathcal{P} \) do
   g7. \( \text{wr}_q, q \leftarrow \text{wr}_q, q \)
g8. \( \text{rseq}_q[q] \leftarrow \text{rseq}_q[q] \)
g9. \( \text{else if} \ (\exists \langle q, * , * , e_q, r_q, \text{state}_q \rangle \in m_p : (\langle e_q, r_q \rangle \geq \langle e_p, r_p \rangle) \land \neg \text{isreset}(q)) \)
g10. \( q \leftarrow \text{index-max}\{ \langle e_q, r_q \rangle | \langle q, * , * , e_q, r_q, \text{state}_q \rangle \in m_p : \neg \text{isreset}(q) \} \)
g11. if (\( (e_q, r_q) > (e_p, r_p) \))
g12. \( \langle e_p, r_p \rangle \leftarrow \langle q, r_q \rangle \)
g13. \( \text{state}_p \leftarrow \text{state}_q \)
g14. \( v_p \leftarrow \text{new input}(e_p) \)
g15. if (\( r_p = \text{inf} \lor e_p = \text{inf} \))
g16. \( \forall q \in \mathcal{P} : \text{wr}_q, q \leftarrow \text{wr}_q, q + 1 \ \text{mod} \ 3 \)
g17. \( \text{wr}_q, q \leftarrow \text{wr}_q, q + 1 \ \text{mod} \ 2n \)
g18. \( \text{rseq}_q[p] \leftarrow \text{rseq}_q[p] + 1 \ \text{mod} \ 2n \)
g19. \( \langle e_p, r_p \rangle \leftarrow (0, 0) \)
g20. \( \text{else} \)
g21. \( \langle e_p, r_p \rangle \leftarrow (e_p, r_p + 1) \)

procedure decide() :
h1. \( \text{state}_p \leftarrow \text{new state}(\text{state}_p, v_p) \)
h2. \( \langle e_p, r_p \rangle \leftarrow (e_p + 1, -1) \)
h3. \( v_p \leftarrow \text{new input}(e_p) \)
```

Fig. 3. Procedures for process \( p \).
an epoch number that is at least $e + 1$ (assuming a resetless execution). Next, we prove that eventually a process decides, using the eventual completeness and eventual accuracy of the failure detector indications (that are guaranteed in admissible executions). Note that this concludes the liveness arguments. Next, we turn to prove the eventual safety property. We show that any resetless execution has a suffix in which no two processes decide on different values with the same epoch number.

The rest of the proof focuses on proving that practically resetless executions must exist. We prove that eventually when $p$ invokes a reset, the values of the registers that are used for implementing the reset are not equal, and eventually, when $q$ notices the reset and acknowledges (the reset), these values are equal. We use the behavior of the balance/unbalance protocol and the reset sequence numbers to show that eventually when a process resets the system, the reset is successful. A successful reset is the one after which every process uses 0 as its epoch/round number or uses a counter value that has been incremented from 0 due to steps, for which this reset has a happen-before relation. In fact, we show that a successful reset takes place, when a process $p$ manages (when writing a reset indication) to introduce a new reset sequence number to every neighboring process $q$ and the first step of $q$, in which $q$ balances, is the one that also assigns 0 to the epoch and the round numbers. We identify crashed processes by the balance/unbalance protocol.

An important property of a successful reset is that following such a reset, in every practically infinite execution, the epoch and the round numbers of every process are always (much) smaller than $\inf$. The proof is based both on the

```
1. $(e_p, r_p) \leftarrow (0, -1)$
2. $v_p \leftarrow \text{new input}(e_p)$
3. do forever
4. $c \leftarrow r_p \mod n$
5. write($R_p, (v_p, \text{announce}, e_p, r_p, \text{state}_p, w rp, lwr_p, rseq)$)
6. if ($p = c$)
7. scan()
8. if $(\exists q, v_q, \text{decide}, e_p, r_q, \text{state}_q) \in m_p : \neg \text{isreset}(q)$
9. write($R_p, (v_q, \text{decide}, e_p, r_p, \text{state}_q, w rp, lwr_p, rseq)$)
10. decide()
11. else
12. if $(\forall q, v_q : e_q, r_q, *) \in m_p : (e_p, r_p) > (e_q, r_q) \vee \text{isreset}(q)$
13. if $(\exists q, v_q, \text{propose}, e_p, r_q, *) \in m_p : \neg \text{isreset}(q)$
14. $r_{\text{max}} \leftarrow \max\{r_q \mid (q, v_q, \text{propose}, e_p, r_q, \text{state}_q) \in m_p : \neg \text{isreset}(q)\}$
15. $(\text{state}_p, v_p) \leftarrow (\text{state}_q, v_q) : (q, v_q, \text{propose}, e_p, r_{\text{max}}, \text{state}_q) \in m_p : \neg \text{isreset}(q)$
16. write($R_p, (v_q, \text{propose}, e_p, r_p, \text{state}_p, w rp, lwr_p, rseq)$)
17. scan()
18. if $(\forall q, v_q : e_q, r_q, *) \in m_p : (e_p, r_p) > (e_q, r_q) \vee \text{isreset}(q)$
19. write($R_p, (v_q, \text{decide}, e_p, r_p, \text{state}_p, w rp, lwr_p, rseq)$)
20. decide()
21. else
22. do
23. scan()
24. $(e_p, r_p) \leftarrow (e, r) : (p, *, *, e, r, *) \in m_p$
25. $c \leftarrow r_p \mod n$
26. until $(\exists c : (e = p \vee \text{isreset}(c) \vee \text{decisionExist}(e_p) \vee c \in fd_p \vee (e, r_c) \in m_p : (e, r_c) > (e_p, r_p) \wedge \neg \text{isreset}(c))$
27. if $(\exists q, v_q, \text{decide}, e_p, r_q, \text{state}_q) \in m_p : \neg \text{isreset}(q)$
28. write($R_p, (v_q, \text{decide}, e_p, r_p, \text{state}_q, w rp, lwr_p, rseq)$)
29. decide()
```

Fig. 4. Replicated state-machine and consensus for process $p$
origin of any such epoch or round number, which is 0, and on the sequential increment operations. The time needed to execute these sequential operations is proportional to the value of the epoch/round numbers. Hence, reaching a value of $inf$ takes practically infinite time. Next, we prove that following a successful reset of $p$, any process $q$ may invoke reset at most once before acknowledging $p$’s reset. We use the above small epoch and round numbers property to conclude that no process $q$ invokes reset after $q$ balances, following a successful reset by a process $p$. Then, we show that following a successful reset there are at most $n - 1$ resets.

The final part of the proof shows that a successful reset takes place. The proof uses the eventual behavior of the balance/unbalance and the reset sequence numbers. The proof assumes to the contrary that there are (slightly less than) $2n^2$ unsuccessful resets. Every such reset, executed by a process $p$, is unsuccessful due to the fact that there is at least one process $q$, with a large epoch or round number, for which the unbalance attempt of $p$ has not succeeded (i.e., $q$ did not reset its epoch and round numbers following the unbalance attempt of $p$) and the other processes copied this large value. Note that this happens when $p$ does not know the actual state of the registers (i.e., when the values of the registers and the local variables differ). When such a scenario occurs, we say that $q$ interferes with the reset of $p$. The use of reset numbers that are incremented modulo $2n$ implies that $q$ may interfere in the resets of $p$ at most twice in every $2n$ sequential resets of $p$. Thus, when $p$ executes $2n$ unsuccessful resets there is at least one neighbor $q$ that interferes three times. Since three interferences of a process cannot happen, it holds that within at most $2n^2$ sequential resets (by any process) at least one reset is successful.

Theorem 2 uses the above observations about the existence of a successful reset to show that there is a practically infinite resetless execution. Moreover, assuming failure detector indications in such a resetless execution, both the consensus properties (eventual termination, validity and agreement) and the replicated state-machine properties (eventual coordination and consistency) hold. The following Theorem states the correctness of the algorithm. Details are omitted from this extended abstract.

**Theorem 2.** Every execution $E$ of the consensus and the replicated state-machine algorithm, with an eventual strong failure detector, has a practically infinite suffix after at most $2n^2 - n$ resets that satisfies the consensus and replicated state-machine tasks.

5 Concluding Remarks

While the definition of the consensus task is a combination of the safety and the eventual liveness properties, a self-stabilizing consensus ensures eventual safety and eventual liveness. Moreover, the self-stabilizing consensus task is suitable for on-going long-lived systems, in which there are repeated invocations of consensus incarnations. The self-stabilizing consensus will ensure the safety and the eventual liveness requirement starting from a consensus incarnation (epoch). In
fact, when started in a predefined initial configuration (with epoch and round numbers zero, and no resets or unbalance actions) safety is ensured as long as no transient faults occur.

To the best of our knowledge our work is the first to introduce a complete solution for a self-stabilizing asynchronous bounded memory consensus. Our solution starts from the design of the self-stabilizing eventual strong failure detector. Then, we present the asynchronous bounded self-stabilizing consensus that assumes an eventual strong failure detector. Finally, we expose all the details required for using the self-stabilizing consensus algorithm for implementing the self-stabilizing replicated state-machine, including stabilization of the bounded consensus incarnation (epoch) numbers.

New consideration, namely unboundedness, is introduced and used in our algorithm. One application of this work is extending the results in [11,19] to ensure the eventual stability of the consensus output, and this time in asynchronous executions in the presence of transient faults and crashes.

Acknowledgments. We would like to thank Gregory Chockler and Seth Gilbert for their fruitful discussions during the first stage of the research.

References


