When Consensus Meets Self-Stabilization
Self-Stabilizing Failure-Detector, Consensus and Replicated State-Machine
(Extended Abstract)

Shlomi Dolev* Ronen I. Kat† Elad M. Schiller‡

Abstract

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

Self-stabilizing algorithm 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 and new techniques 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.

*Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel. Email: dolev@cs.bgu.ac.il Partially supported by the Israeli Ministry of Trade and Industry, the Lynn and William Frankel Center for Computer Sciences, and the Rita Altura Trust Chair in Computer Sciences.

†Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel. Email: kat@cs.bgu.ac.il Partially supported by the Lynn and William Frankel Center for Computer Sciences.

‡Division of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden. Email: elad@chalmers.se.
1 Introduction

Self-stabilization. Self-stabilization [10, 11] is a fundamental property of a system, that ensures automatic recovery of the system following the occurrence of 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., [20] 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 [17] there is no asynchronous consensus algorithm even in executions in which at most one processor may stop taking steps. Fortunately, there are consensus algorithms e.g., [25] 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 [17]). Failure detectors e.g., [7] form a mechanism that captures 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 will be started in a state in which each (of the processes) had 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, say, as means of implementing a replicated state-machine. In such a case, the requirements for the self-stabilizing consensus must ensure eventual termination in initialized or non-initialized execution and ensure all the consensus requirements for the set of processes that initialized a new session of the consensus execution.

For example, when considering the elegant algorithm presented in [25], 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. That is simply because, wrap around of counters is not considered. One may argue that a counter of 64 bits is practically infinite. This argument will 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., [25]. The abstraction of a replicated state-machine has been proven important in several domains e.g., [6, 13, 14].

Our contribution. We present the first self-stabilizing failure detector, consensus and replicated state-machine algorithm suit. 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. The self-stabilizing failure detector is designed for semi-synchronous settings. In such settings, the interleaving order of (a non-crashed processes) steps is eventually somewhat restricted. Roughly speaking, each processor has a bounded heartbeat counter. The relative advances of the counter are compared to other processes. To

---

\(^1\)This is a termination requirement for the largest possible set of asynchronous/semi-synchronous executions.

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

\(^3\)Validity requirement must hold in every execution.
avoid confusion that could arise due to the fact that the counters are bounded, we use wrap around flags that indicate when a process had wrapped its counter. The flags are used to facilitate the correct computation 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. In other words, (1) every correct process eventually suspects all the processes that had crashed (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). Such a failure detector is known as an eventual strong failure detector $\diamondsuit$. In [8, 7] it was shown that the eventual strong failure detector is the weakest failure detector to solve consensus.

**Consensus.** Our self-stabilizing consensus algorithm assumes the existence of the obtained self-stabilizing $\diamondsuit$ failure detector. Roughly speaking, the algorithm is a rotating coordinator algorithm that is based on the (memory unbounded) algorithm of [27]. 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 are with 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 once every process reads this proposed value prior to becoming current coordinator. The use of eventual strong failure detector ensures that eventually one of the processes $p$ is never suspected. It is possible that a univalent configuration will be reached before the stage of the execution from 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 will therefore be imposed by $p$.

**Replicated state-machine.** We use epochs of consensus invocations for deciding on the transition of the (distributed) state-machine. 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 practically infinite executions. Practically infinite executions are executions in which the time complexity measured by the longest happened before chain [26] is longer than, 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 may wait for the last step in this sequence. In case the suit of asynchronous consensus algorithm and failure detector does not reach a decision within a period that corresponds to $2^{64}$ sequential steps, the decision value will clearly be obsolete.

We use the above argument for the first time in the scope of self-stabilization. Rather than assuming that $2^{64}$ counter is infinite, we argue concerning the time that a chain of sequential steps of $2^{64}$ counter increments requires. 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.

**Related work.** The literature on failure detectors is rich; see for example the recent surveys [18, 29]. However, the few self-stabilizing failure detectors either assume synchronous execution [5] or use randomization to construct self-stabilizing perfect failure detectors [23]. Our self-stabilizing failure detector does not require randomization.

Consensus is also an extensively studied topic in distributed computing, see for example [4, 28]. A bounded memory version exists for specific settings. For example, in [19] the case of three processes where only one may fail is considered. However, for the general case, there is no self-stabilizing consensus for asynchronous systems. An asynchronous consensus algorithm requires either (1) a failure detector whose properties always hold, such a failure detector is inherently not self-stabilizing or (2) an eventual failure detector, such consensus algorithm requires unbounded counters to preserve safety.

Self-stabilizing consensus eventually ensures safety in the presence of crash failures and liveness (under reasonable synchronization assumptions for the failure detector). Our algorithm is the first to achieve the above.
Using asynchronous reset with the purpose of circumventing unbounded values is discussed in [24]. 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 active or inactive. A resetable vector clocks that provide non-blocking resets in the absence of faults is presented in [3]. In case of faults, 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 \( O(n) \) operations) are used in [1] in order to achieve bounded time-stamp. We propose wait-free reset that can be used for implementing self-stabilizing bounded time-stamp that after its convergence requires only \( O(n) \) operations per invocation.

**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 appear in the appendix.

### 2 System Settings

We consider a shared memory system with a set of \( \Pi \) communicating entities that we call *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 \( R \). A process \( p \in \Pi \) can write only in its associated register and can read any register. Process \( p \) writes to register \( R \) the value \( v \), 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, e.g., \( r_p \) 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 *atomic step*. An atomic step consists of a sequence of internal (program) computations that ends in a single *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 atomic steps \( E = (c_1, a_1, c_2, a_2, \ldots) \), such that configuration \( c_{i+1} \) is reached by executing an atomic 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 a task \( LE \) if any execution that starts in \( c \) is a legal execution (belongs to \( LE \)). An algorithm is *self-stabilizing* with relation to a 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 atomic steps. We say that such a process is *crashed*. Hence, it does not execute any atomic 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 an atomic 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 some (unknown to the processes) bound on the (relative) execution speed of the processes.

### 3 Self-Stabilizing Failure Detector

Fischer, Lynch and Paterson [17] have shown that a consensus cannot be reached in an asynchronous environment. A failure detector [7] 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 \( \diamond S \) failure detector that satisfies the following properties:
Property 1 (Strong completeness:) Every execution has a suffix in which every non-crashed process suspects all the processes that had crashed.

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.

3.1 Semi-Synchronous Settings and Requirements

We assume the existence of a global clock that is unknown to the processes. We use real time to argue concerning progress during executions. We make no assumptions regarding local clocks or clocks synchronization.

We say that a strictly correct process is a process that executes any two successive atomic steps exactly \( \delta \) time units (of the global clock) apart. Such a process is said to be executing atomic steps on time. The definition is similar to the notion of eventual timely output links, in a message-passing model, of some process as in [2]. We also assume that no process executes two successive steps faster than a strictly correct process. 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 \( a_{i+1} \) took place.

We say that an infinite execution is admissible if it has at least one strictly correct process. Since no process is faster than \( p \), process \( p \) executes at least one atomic step in between any two successive atomic 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.

3.2 The Failure Detector

The failure detector algorithm maintains a set of suspicious processes in the local variable \( \text{fd} \) (failed detection). If process \( p \) suspects process \( q \) to be crashed, then \( q \in \text{fd} \). 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 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 strictly correct process) is never suspected since every process notes the continuous progress of the strictly correct 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 had wrapped its counter, as crashed. Therefore, in case the counter of a process \( p \) wraps, \( p \) uses a balance/unbalance [12, 16] protocol to ensure that \( p \) is not considered as crashed. That is, \( p \) keeps a flag \( WR_{p,q} \) for every other process \( q \). Process \( q \) keeps a copy of \( p \)’s flag in \( LW_{R_{q,p}} \) and makes sure that the copy always equals to \( p \)’s value. When \( p \) wraps its counter, \( p \) assigns \( q \)'s flag a value that is different from the value of its copy of \( q \)'s flag. Therefore, when \( q \) reads \( p \)'s flag, \( q \) notices the fact that \( p \)'s counter had 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 \( LW_{R_{q,p}} \) contain a value in \( \{0, 1, 2\} \) 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 \( LW_{R_{q,p}} \). The value of the local variable may differ from the value
of the register. We say that $p$ is unbalanced towards $q$ in case $W_{R_{p,q}} \neq LWR_{q,p}$. Otherwise, $p$ is balanced towards $q$. We use the predicate $unbalance(q)$ to describe $p$’s view (indicated by the values of $w_{r_{p,q}}$ and $lwr_{q,p}$) regarding its state (balanced/unbalanced) towards $q$. When $p$’s counter is wrapped, $p$ writes to $W_{R_{p,q}}$ (for every $q$) a value, such that the predicate $unbalance(q)$ is $true$. In case $w_{r_{p,q}} = lwr_{q,p}$, $p$ increases $lwr_{q,p}$ by one modulo 3. A process $q$ that notices the wrap indication (i.e., the predicate $unbalance(q)$ is $true$) copies $w_{r_{p,q}}$’s value to $lwr_{q,p}$.

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 $unbalance_all()$ (lines 3 and 4), to the 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 $LWR_{q,p}$ and $W_{R_{p,q}}$. If $p$ is balanced towards $q$ then process $p$ unbalances and writes to register $W_{R_{p,q}}$ (lines $f4$ and $f5$). A process $p$ becomes balanced towards $q$ by reading register $W_{R_{q,p}}$ (line 9) and writing the values in register $LWR_{q,p}$ (line 18).

Process $p$ records the history of process $q$ in the cyclic history array $h_q$. Process $p$ keeps the history of length $k$ per process. In line 5 process $p$ moves to the next history entry. In line 6 process $p$ initializes the list of new suspects in $newfd$. In lines 8 through 14 process $p$ reads the heartbeat counter and balance/unbalance indications of every process $q$ and 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 as 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 $W_{R_{q,p}}$ to $LWR_{p,q}$ (i.e., balances towards $q$). In line 19, process $p$ updates its suspects list $fd$ with the new computed suspicions.

**Theorem 3.1** Every admissible execution of the failure detector algorithm (Figure 1) has a suffix that satisfies the failure detector task. The suffix starts after every non-crashed process executes $(k + 3)\Delta$ atomic 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 that 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 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 in which no two processes execute a different transition.*

**Property 7** Eventual Consistency: *Every execution has a suffix in which no two processes differ in their machine state in every epoch.*

### 4.1 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.

### 4.2 The Consensus and the State-Machine

**Consensus with a rotating coordinator.** The consensus ensures that no two processes will decide on different values in an epoch. Every process either follows the previous decision of others, or strives to make a decision that the other processes would follow. A process that does not copy a decision value from a process that had 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 smaller round number $r'$ (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 atomic 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 be crashed. The **rotating coordinator** paradigm \([7]\) 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 atomic steps. Every other process \( q \) (i.e., \( q \) is not a coordinator in \( r \)) waits for \( p \) to decide in round \( r \). Process \( q \) repeatedly performs *scans* until some process decides, or until it is obvious (for \( q \)) that the coordinator will not decide in \( r \). E.g., 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, 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, a process 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 *decide() procedure*, computes the new state and increases its epoch number to \( e_p + 1 \).

**Self-stabilizing wait-free reset.** Next, we show how to resolve the contradiction between the 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 some epoch. A similar argument applies to epoch numbers and the state-machine 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 some counter is initialized (or changed due to a transient fault) to some non-zero (large) value.

Whenever the epoch or round number of process \( p \) reaches \( inf \), \( p \) performs a *reset* and assigns zero to both counters. 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: \( wr_{p,q} \neq lw_{q,p} \) (unbalanced), by assigning a value to \( wr_{p,q} \) if needed. That is, if \( wr_{p,q} = lw_{q,p} \) then \( p \) increase \( lw_{q,p} \) by 1 modulo 3. Process \( p \) evaluates the predicate \( reseted(q) \) 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 \( lw_{q,p} \) ) of \( p \) and \( q \) differs.
We say that \( q \) is reseted or \( q \) is flagged as reset in case the predicate \( \text{reseted}(q) \) is true for process \( q \). A process \( q \) that is reseted, does not strive to propose and decide in the consensus algorithm (and is ignored by other processors). When \( q \) note that some process \( p \) had flagged \( q \) as reset, process \( q \) acknowledges the reset by copying the values of \( p \)'s flags and reset sequence number and by resetting the epoch/round number to 0.

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 the full state. Whenever a process had reached an epoch/round number higher than \( q \)'s reset sequence number (lines 22 through 29), process \( q \) resets its epoch and round number to zero (line 26). That is, reseted process \( q \) does not strive to propose and decide in the consensus algorithm (and is ignored by other processors).

We now describe the consensus and the replicated state-machine algorithm in Figure 2. 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 \( \{ v, g, e, r, state, wr, lwr, rseq \} \). \( v \) 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 \( \langle e_1, r_1 \rangle \geq \langle e_2, r_2 \rangle \) if \( e_1 > e_2 \) or if \( e_1 = e_2 \) and \( r_1 > r_2 \). We say that some process had decided in epoch \( e \) if there exists a non-reseted process \( q \) with a higher epoch number or if some register (of a non-reseted process) contains the tag decide for epoch \( e \). In such cases, the predicate \( \text{decisionExists}(e) \) is true.

The procedure \( \text{next}() \) 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/unbalance flags and reset sequence numbers (lines g6 through g8). That is, \( p \) copies \( q \)'s unbalanced flag \( (wr_{p,q}) \) to \( lwr_{p,q} \) and copies \( q \)'s reset sequence number \( (rseq_{q_p}[q]) \) to \( rseq_{q_p}[q] \). In case no process flagged \( p \) as reset, \( p \) checks if some process had 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 \( \text{index} - \text{max} \), a non-reseted process \( q \) with the smallest identifier that has the maximal epoch/round number \( \langle e_q, r_q \rangle \). If \( \langle e_q, r_q \rangle \geq \langle e_p, r_p \rangle \), 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 some other process, \( p \) checks if the epoch/round number reached \( \text{inf} \). If \( e_p \) or \( r_p \) reached \( \text{inf} \) 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 \( rseq_{p}[p] \). Otherwise, in lines g20 and g21, process \( p \) increases its round number.

The procedure \( \text{scan}() \) 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 reseted are stored in \( wr_{q_p}, lwr_{q_p} \) and \( rseq_{q_p} \). 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 \( q \).

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 some process \( q \) had already decided. If so, \( p \) decides after copying \( q \)'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 some process had proposed. If no process proposed, then \( p \) proposes its own estimate. In lines 17 through 20, \( p \) verifies once more that no other process had 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
Figure 2: Replicated state-machine and consensus for process p.

coordinator not to reach p's round, p stops waiting if (1) p wait for itself, (2) p is reseted, or (3) some process had decided in the epoch. In case p exits the loop after some process had 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 that executes atomic steps in epoch e (i.e., ep = e) and writes to register Rp a tuple with the action decide is said to be decided in epoch e. Note that a process is decided after executing the atomic 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 (Lemma 4.1) in a restless execution. In the sequel, we show that in every execution there exists a restless execution. Lemma
Lemma 4.2 proves that if there is a decision in a certain epoch \(e\), then there is a subsequent configuration in which some process has an epoch number that is at least \(e + 1\) (assuming a resetless execution). Next, in Lemma 4.3 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 Lemma 4.3 concludes the liveness arguments. Next, we turn to prove eventual safety property. Lemma 4.4 shows 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, 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 one after which every process uses 0 as its epoch/round number or a counter value that was incremented from 0 due to steps, for which there exists a happen-before relation to this reset operation. 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 atomic step of \(q\), in which \(q\) balances, is 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 on the origin of any such epoch or round number, which is 0 and on the sequential increment operations. Moreover, 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 \(p\) 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 some process \(p\), is unsuccessful due to the fact that there is at least one process \(q\) for which the unbalance attempt of \(p\) did not succeed (i.e., \(q\) did not reset its epoch and round numbers following the unbalance attempt of \(p\)). Note that this happens when \(p\) does not know the actual state of the registers. We say that, \(q\) interferes in the reset of \(p\), when such a scenario occurs. 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. However, that is not possible. Therefore, within at most \(2n^2\) sequential resets (by any process) at least one reset is successful.

Theorem 4.5 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.

**Lemma 4.1** In every execution \(E\), every process executes the procedure next() of the consensus algorithm infinitely often.

**Lemma 4.2** In every resetless execution \(E\) of the consensus algorithm, if a non-reseted process decides in epoch \(e\), then all non-crashed processes advance to an epoch equal or greater than \(e + 1\).

**Lemma 4.3** In every resetless execution \(E\) of the consensus algorithm with a failure detector, at least one non-crashed process will decide in epoch \(e\) (and advances to epoch \(e + 1\)).

**Lemma 4.4** Every resetless execution \(E\) of the consensus algorithm has a suffix in which no two processes decide on different values in some epoch \(e\).

**Theorem 4.5** Every execution \(E\) of the algorithm in Figure 2, 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, 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 some consensus incarnation (epoch). In fact, when started in a predefined initial configuration (with epoch and round values 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 in the design of a self-stabilizing eventually strong failure detector. Then, we present an asynchronous bounded self-stabilizing consensus that assumes an eventually strong failure detector. At last, we expose all the details required for using a self-stabilizing consensus algorithm for implementing a 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 is to extend the results in [9, 15] to ensure the eventual stability of the consensus output, 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
Additional Details for The Failure Detector

The code in Figure 1 is designed in a way that ensures that $p$ increments its heartbeat by executing line 1 and 2 every $\Delta$ atomic steps of $p$. One may assume that $p$ counts the number of its atomic steps starting with the atomic step in
line 1 of Figure 1. In case this number \( x \) is smaller than \( \Delta \) when the program counter of \( p \) reaches line 1 again, \( p \) executes \( \Delta - x - 1 \) empty steps before executing line 1 again. One may also assume that an empty step is a step in which \( p \) reads one of its associated registers while not changing the state of any process or register.

The requirement for the existence of a strictly correct process can be relaxed; e.g., to the existence of a set of processes that are “fast enough”. A process \( p \) is considered “fast enough” when no process suspects \( p \) as crashed. Thus, the proofs hold for a much larger set of executions, for example, executions in which there exists a process \( p \) for which the history associated with \( p \) by every other 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 increase steps of \( p \).

The following definitions are needed for Lemma 3.1, Lemma 3.2 and Lemma 3.3. Let \( a_{q,l} \) denote the execution of line \( l \) by process \( p \) in an admissible execution \( E \) of the code presented in Figure 1, where \( 8 \leq l \leq 18 \) and where \( p \) carries out the iteration that accesses a (register or local) variable that is associated with the process \( q \). We denote by \( c_{q,l} \) the configuration that immediately follows \( a_{q,l} \). Let \( c_{q,1}^{1}, c_{q,1}^{2}, \ldots, c_{q,1}^{x} \), be the first \( x \) consecutive configurations in \( E \) that immediately follow \( a_{q,l} \). Note that \( c_{q,1}^{x} \in E \) exists because line 1 of the code is repeatedly executed every \( \Delta \) atomic steps of \( p \).

Lemma 3.1 demonstrates that Property 1 holds for admissible executions of the algorithm presented in Figure 1, and that the response time is a function of \( k \) (the length of the history \( h_q \)).

**Lemma 3.1** In every admissible execution \( E \) of the failure detector algorithm a non-crashed process \( p \) suspects every crashed process \( q \) within at most \( (k + 3)\Delta \) atomic steps.

**Proof:** Since process \( q \) is crashed, it holds that the values of registers \( HB_q, LW R_{p,q} \) and \( WR_{q,p} \), do not change in \( E \). After \( c_{q,10}^{1} \), process \( p \) executes lines 1 through 19 of the algorithm, and the values of \( new h b_q, wr_{p,q} \) and \( lw r_{q,p} \) are, therefore, fixed in every configuration in the execution that starts in \( c_{q,10}^{2} \).

From \( c_{q,10}^{2} \) it holds that: \( LW R_{p,q} = WR_{q,p} \), the predicate \( unbalance(q) \) is false and \( l h b_q = HB_q \). Since \( q \)’s heartbeat did not change, \( new h b_q = l h b_q \) in \( c_{q,10}^{2} \). Process \( p \) evaluates the if-statement condition of line 11 to be false, and assigns 0 to \( h_q[i] \). The same arguments hold for all \( x \geq 2 \) \( c_{q,10}^{x} \) and we have \( h_q[i + x \mod k] = 0 \). Therefore, in \( c_{q,10}^{k+2} \) it holds that \( \forall i h_q[i] = 0 \). The if-statement’s condition of line 15 is true and \( p \) adds \( q \) to \( new f d \) (line 16) and to \( f d \) (line 19). Therefore, after executing \( (k + 3)\Delta \) atomic steps process \( p \) suspects process \( q \). \( \square \)

Next, we show that there is a suffix of \( E \) in which every non-crashed process \( p \) detects every wrap of \( q \)’s counter (i.e., when \( q \) writes 0 to \( HB_q \)). We note that the suffix starts within no more than \( 2\Delta \) steps (of \( p \)). Process \( p \) detects a wrap of \( q \)’s counter when the predicate \( unbalance(q) \) is true. Let \( E \) be an admissible execution that starts in an atomic write step to \( HB_q \) by a strictly correct process \( q \).

**Lemma 3.2** Every admissible execution \( E \) has a suffix in which a non-crashed process \( p \) detects a counter wrap of a strictly correct process \( q \).

**Proof:** Let \( c \) be a configuration that follows \( q \)’s counter wrap (line 2) after \( c_{q,18}^{2} \). We denote by \( a_1, a_2 \) and \( a_3 \) the first three atomic steps (in lines \( f2, f3 \) and \( f5 \)) after \( c \) in which \( q \) reads \( LW R_{p,q}, WR_{q,p} \) and, if \( WR_{q,p} = LW R_{p,q} \), writes to \( WR_{q,p} \). Let \( c' \) be the first configuration that follows \( a_3 \), or if line \( f5 \) is not executed, then follows \( a_2 \). We denote by \( c_{q,18}^{i} (i \geq 2) \) the first configuration (that follows \( a_{q,18} \)) immediately before \( a_1 \) and by \( v_0 \) the value written to \( LW R_{p,q} \). Since \( a_1 \) follows \( c_{q,18}^{i} \), the value read in \( a_1 \) is \( v_0 \). Let the value read in \( a_2 \) (from \( WR_{q,p} \)) be \( v_1 \).

Consider the case that \( v_0 = v_1 \). Then \( q \) executes \( a_3 \) and writes a value \( v \neq v_0, v_1 \) in \( WR_{q,p} \). Hence, in either \( c_{q,10}^{i+1} \) or \( c_{q,10}^{i+2} \) it holds that the predicate \( unbalance(q) \) is true.
Consider the case in which $v_0 \neq v_1$. Then, $q$ does not change the value of $WR_{q,p}$ due to the fact that $WR_{q,p}$ is not change until $p$ writes $v_1$ to $LWR_{p,q}$. Therefore, in $c_{q,10}^{i+1}$ it holds that the predicate $unbalance(q)$ is $true$.

Hence, a process $p$ that executes $2\Delta$ steps can detect the fact that a strictly correct process $q$ wrapped its counter.

Lemma 3.3 shows that Property 2 holds for admissible executions that satisfy Lemma 3.2 of the algorithm presented in Figure 1.

**Lemma 3.3** In every admissible execution $E$ of the failure detector algorithm, with $k > 2$, every non-crashed process $p$ does not suspect any strictly correct process $q$ within at most $4\Delta$ atomic steps.

**Proof:** Let $E$ be an admissible execution that starts when $q$ writes to register $HB_q$ (line 2). The value of $newhb_q$ in $c_{q,8}^2$ is the result of a read atomic step from register $HB_q$.

Due to the fact that $q$ is strictly correct and executes an atomic step in between every two atomic steps of $p$, process $q$ changes its heartbeat counter (at least once) between any two executions of $a_{q,8}$.

$h(q,l) (l > 1)$ denotes the number of times that process $q$ changed its heartbeat counter between $c_{q,8}^{l-1}$ and $c_{q,8}^l$. For $l > 2$, if $h(q,l)$ is not a multiple of $m$, then in $c_{q,8}^l$ it holds that $newhb_q - lhb_q \mod m > 0$ and that process $p$ evaluates the condition of the if-statement of line 11 to be $true$ and assigns $1$ to $h_q[z]$ (line 12).

Consider the case that $h(q,l)$ is a multiple of $m$. Then $q$ wraps its counter between $c_{q,8}^{l-1}$ and $c_{q,8}^l$. By Lemma 3.2 there exists, following the wrap, a configuration $c_{q,10}^i$ in which the predicate $unbalance(q)$ is $true$.

We now show that $l + 1 \geq i \geq l - 1$. Let $c'$ be the first configuration after the wrap, in which $WR_{q,p} \neq LWR_{p,q}$ (Lemma 3.2). By the fact that the $q$ wraps its counter after $c_{q,8}^{l-1}$ then $i \geq l - 1$. Since $q$ wraps its counter prior to $c_{q,8}^l$ and $q$ is strictly correct, $c'$ occurs prior to $c_{q,8}^{l+1}$. Hence, in either $c_{q,10}^l$ or $c_{q,10}^{l+1}$ it holds that $unbalance(q)$ is $true$ and process $p$ evaluates the condition of the if-statement of line 11 to be $true$ and assigns $1$ to $h_q[z]$ (line 12).

Thus, if $k > 2$, then at least one entry in the history is always $1$ and $q$ is never suspected after $4\Delta$ steps (and obviously process $q$ never suspects itself).

Theorem 3.1 follows from Lemma 3.1 and Lemma 3.3.

**Theorem 3.4** Every admissible execution of the failure detector algorithm (Figure 1) has a suffix that satisfies the failure detector task. The suffix starts after every non-crashed process executes $(k + 3)\Delta$ atomic steps.

### 3.1 Self-Stabilization Consensus Proofs

**Lemma 4.1** In every execution $E$, every process executes the procedure next() of the consensus algorithm infinitely often.

**Proof:** Every line of the algorithm in Figure 2 contains a finite (length) computation step, or possibly a bounded sequence of local computations. Every loop statement in the algorithm, except in lines 22 through 26, can be exchanged with a sequential (bounded) code. Therefore, in case the procedure next() is not executed infinitely often, lines 22 through 26 are executed forever.
Assume in contradiction that $E'$ is a suffix of $E$ in which some process executes lines 22 through 26 forever. Let $p$ be a process that has the smallest epoch/round number $\langle e_p, r_p \rangle$ that executes lines 22 through 26 forever and let $q$ be the coordinator in round $r_p$.

Since process $p$ waits forever, by line 26 of the algorithm, $p$ is not the coordinator of round $r_p$ (i.e., $p \neq q$) and $q$ is not crashed. Therefore, $q$ is either reseted or has an epoch/round number smaller than or equal to $\langle e_p, r_p \rangle$. Consider the case in which $q$ is flagged as reset. By the above properties, $q$ is not crashed and, by the fact that $q$ is reseted (i.e., $\text{reseted}(q) = \text{true}$), $q$ does not execute lines 22 through 26 forever. Therefore, $q$ exits the loop and (eventually) executes the procedure $\text{next}()$. Thus, $q$ acknowledges the reset. Note that once $q$ acknowledges the reset, $q$ cannot be flagged as reset again, unless $p$ is also flagged as reset. In such a case, process $p$ exits the loop in lines 22 through 26. Hence, there exists a suffix of $E'$ in which $q$ is not flagged as being in reset.

Consider the case that $q$ is not flagged as being in reset and is in epoch/round $\langle e_q, r_q \rangle$. In case $e_p > e_q$, the predicate $\text{decisionExist}(e_q)$ is $\text{true}$ and $q$ always exits the loop in lines 22 through 26. Hence, $q$ executes the procedure $\text{next}()$ and increments its epoch number to at least $e_p$. Note that if $q$ advances to a higher epoch number than $e_p$, then $p$ exits the loop. In case that $e_q = e_p$ and $r_q \leq r_p$, then $q$ cannot execute lines 22 through 26 forever, since according to our assumption, $p$ is in the smallest epoch/round number. Then, $q$ always increases its round number (or matches the highest round number in $e_q$). Hence, eventually $e_q > e_p$ and $p$ exits the loop in contradiction to our assumption.

Thus, no process can execute lines 22 through 26 forever. Hence, $p$ executes the procedure $\text{next}()$ infinitely often.

**Lemma 4.2** In every resetless execution $E$ of the consensus algorithm, if a non-reseted process decides in epoch $e$, then all non-crashed processes advance to an epoch equal or greater than $e + 1$.

**Proof:** Suppose to the contrary that a non-crashed process $p$ never advances to a higher epoch than $e$ (or decides), even when some (other) non-reseted process $q$ has decided in epoch $e$ in round $r$.

Consider the case that process $q$ has decided and advanced to the next epoch $e + 1$. By Lemma 4.1, $p$ executes the procedure $\text{next}()$. Process $q$ is never flagged as reset in $E$ and is in epoch $e + 1$. Then, process $p$ advances to epoch $e + 1$ or higher (line $g11$).

Consider the case that process $q$ has decided in epoch $e$ and round $r$, but did not advance to the next epoch $e + 1$. In other words, every process that reads the value of register $R_q$ finds that $q$ had decided and is in epoch/round $\langle e, r \rangle$. Let $c_1$ be the first configuration in which process $q$ has decided. Then, in any configuration following $c_1$ the value of $R_q$ is $\langle v, \text{decide}, e, r, \text{state}, \text{reset} \rangle$. By the fact that $p$ never decides and is not crashed, we have that $p$ executes the procedure $\text{next}()$ infinitely often. Consider the first execution of the procedure $\text{next}()$ after $c_1$. Following line 3 it holds that $p$ is not flagged as being in reset (lines $g2$ through $g5$) and that $p$ is in epoch $e$ at the highest round number in $e$ (lines $g10$ through $g14$).

Now consider the case in which $p$ executes epoch $e$ in some round $r'$. Consider the case that $p$ is the coordinator in round $r'$. Then, the predicate in the if-statement in line 8 is evaluated to $\text{true}$ and $p$ decides and continues to epoch $e + 1$. This is contrary to the assumption that $p$ never continues to an epoch higher than $e$. Hence, $p$ is never a coordinator following the first execution of the procedure $\text{next}()$ after $c_1$.

Consider the case in which $p$ is not the coordinator in round $r'$. Let $c_2$ be the first configuration after $c_1$, following the execution of the last atomic step of line 23 (in the $\text{scan}()$ procedure). By the fact that $q$ decided, the predicate $\text{decisionExist}(e)$ is $\text{true}$ in $c_2$. Hence, process $p$ executes lines 27 through 29. The condition of the if-statement in line 27 is evaluated to $\text{true}$ ($q$ has decided) and, therefore, process $p$ decides and continues to epoch $e + 1$ (the procedure $\text{decide}()$). This is contrary to our assumption.

Hence, every non-crashed process continues to epoch $e + 1$ or higher.
**Lemma 4.3** In every resetless execution $E$ of the consensus algorithm with a failure detector, at least one non-crashed process will decide in epoch $e$ (and advances to epoch $e + 1$).

**Proof:** Suppose to the contrary that no process ever decides in any epoch number. By the failure detector strong completeness property (1), there is a configuration $c_1$ after which every process that crashes is suspected by every non-crashed process. By the failure detector eventual accuracy property (2), there is a configuration $c_2$ after which some non-crashed process $p$ is never suspected by any other process. This implies that $p$ executes steps in $E$. Therefore $p$ is not flagged as reset after $c_2$.

Let $c_3$ be the configuration by which all faulty processes have already crashed and let $c_4$ be the configuration by which all non-faulty processes have executed line 3 of the algorithm. Let $c_{\text{max}}$ be the latest configuration among $c_1$, $c_2$, $c_3$, and $c_4$. Let $\langle e, r \rangle$ be the highest epoch/round number that a non-reseted process reached [[(i.e., in a register or a local variable)]] in configuration $c_{\text{max}}$. We note that only non-crashed processes take steps in the execution that starts in configuration $c_{\text{max}}$.

Let $\langle e, r_p \rangle$ be the earliest epoch/round, where $r_p > r$ (in epoch $e$), in which $p$ is the coordinator and $p$ is not suspected by any non-crashed process. Due to the fact that $E$ is practically infinite, such a round $r_p < \inf$ exists. By the assumption that no process ever decides, [[the fact]] that process $p$ is not crashed (and not suspected as crashed) and that at least one non-reseted process is in epoch $e$, it holds that process $p$ reaches epoch/round $\langle e, r_p \rangle$. Note that no process can advance beyond round $r_p$ in epoch $e$. Moreover, process $p$ cannot decide before or in round $\langle e, r_p \rangle$. Process $p$, acting as a coordinator, executes lines 7 through 20 in round $\langle e, r_p \rangle$. Since process $p$ does not decide in epoch/round $\langle e, r_p \rangle$ it holds that $p$ found some process $q$ that advanced beyond round $e \cdot r_p$ when executing line 12 or line 18 in epoch/round $\langle e, r_p \rangle$. Process $q$ reached round $\langle e, r_p \rangle$ after configuration $c_{\text{max}}$ (as $r_p > r$). Then, by line 26 of the algorithm, process $p$ can continue to round $r_p + 1$ only if: (1) $q$ is the coordinator of $r_p$ (false), (2) $q$ is flagged as being in reset ($false$), (3) $p$ is suspected to be crashed ($false$), (4) $p$ finished executing round $r_p$ ($false$), or (5) some process decides ($false$, by assumption). Hence, no such process $q$ exists and $p$ does not find such a process when executing lines 12 or 18. Therefore, process $p$ decides in epoch $e$. 

A configuration $c$ that immediately follows an atomic step of process $p$ in line 15 is named proposing configuration (for process $p$). Note that, according to the algorithm, a process that decides in epoch/round $\langle e, r \rangle$ by executing line 19, must reach a proposing configuration in round $\langle e, r \rangle$, prior to deciding.

**Lemma 4.4** Every resetless execution $E$ of the consensus algorithm has a suffix in which no two processes decide on different values in some epoch $e$.

**Proof:** Let $E = E_1 \circ E_2$, where $E_1$ is the smallest prefix of $E$, where each process executes at least once (or never in $E$) the procedure $\text{next}()$. Let $e'$ be the highest epoch number (either in the registers or in local variables) in the first configuration of $E_2$. We now show that no two processes decide differently in any epoch $e > e'$.

Let $p$ be a coordinator process that decides in round $\langle e, r \rangle$ in $E_2$ by executing line 19, such that, the proposing configuration $c$ of process $p$ and round $r$ is the first in $E_2$. We denote by $v$ the value proposed by $p$ in round $r$. There exists such a process by the fact that $E_2$ is a resetless execution and there exist at least one process that decides without copying some other process decision (i.e., executing lines 10 or 29). Note that a process that decides by executing lines 10 or 29 only copies an existing decision value.

Suppose to the contrary that process $q$ is the first process that proposes, by executing the atomic step in line 16, a value $v' \neq v$ in epoch $e$ and in round $r' > r$. According to the algorithm, process $q$ read the value of $R_p$ before proposing. Consider the case that process $q$ read the value of $R_p$ after process $p$'s proposing configuration. By the fact that $q$ is the first process that proposes a value $v' \neq v$ in $r' > r$ it holds that the value of the proposal with the
highest round number is $v$. Hence, process $q$ assigns $v_p \leftarrow v$, in contrary to the assumption that $q$ proposed $v' \neq v$. Consider the case that process $q$ reads the value of $R_p$ before process $p$'s proposing configuration. In such a case, process $q$ executed the atomic step in line 5 before $p$'s proposing configuration. Note that process $p$ executes line 17 after its proposing configuration. Then, in line 18, process $p$ finds that process $q$ is in a higher round than $p$. Hence, $p$ does not decide, contrary to the fact that $p$ decided in $\langle e, r \rangle$.

4.2 Additional Details for the Self-stabilizing Wait-Free Reset Mechanism

Next we prove the existence of a resetless execution. We use the balance/unbalance protocol to detect crashed processes in our wait-free reset mechanism.

A process $p$ that executes lines g16 through g20 is said to be performing a reset (with a sequence number) $reset_p$. The configuration $c_{reset}$ that immediately follows an atomic step in which $p$ writes to $R_p$ following a reset is named $p$ reset configuration. The execution that starts in $c_{reset}$ and continues until $p$ performs another reset (if ever) is denoted by $E_{reset}$. A process $q$ is said to acknowledge $p$'s reset if $E_{reset}$ is composed of a non empty prefix $E_{q, noack}$ and a suffix $E_{q, ack}$, such that, in $E_{q, noack}$ it holds that $unbalanced(p, q)$ is true and in $E_{q, ack}$ it holds that $unbalanced(p, q)$ is false and that $\langle e_q, r_q \rangle = \langle 0, 0 \rangle$ in the first configuration of $E_{q, ack}$. We note that $p$ itself always acknowledges its reset. Hence, for $p$ it holds that $E_{reset} = E_{p, ack}$. A reset performed by process $p$ is said to be a successful reset if every process $q$ acknowledges $p$'s reset. A process that does not acknowledge $p$ reset is said to interfere with the reset. Note that a crashed process $q$ can interfere with $p$'s reset at most once. After two resets performed by $p$, $q$ is always unbalanced towards $p$.

Lemma 4.5 Let $E_{reset}$ be an execution that starts after a successful reset performed by $p$. Every process $q$ that already acknowledged $p$'s reset, copies the epoch or the round numbers (lines g8 through g12) only from processes that had acknowledged $p$'s reset.

Proof: Assume in contradiction that $q$ copies an epoch or a round number from a process $u$ that did not acknowledged $p$’s reset. Let $a$ and $a'$ be the two atomic steps in which $q$ reads $u$’s epoch or round number (from $R_u$) and writes the copied value (to $R_q$).

By the fact that $E_{q, ack}$ starts with $q$ writing $\langle e_q, r_q \rangle = \langle 0, 0 \rangle$ in $R_q$ it holds that $a, a' \in E_{q, ack}$. Since $q$ copied $u$’s epoch or round number before $u$ acknowledged $p$’s reset then $a \in E_{u, noack}$. Moreover, since $q$ read $R_p$ after $p$’s reset then the predicate reseted($u$) is true. Hence, $q$ does not copy $u$’s epoch or round number contrary to the assumption that $q$ copies an epoch or round number from a process $u$ that did not acknowledge $p$’s reset.

Lemma 4.6 In every execution suffix $E$ that starts with a successful reset, performed by $p$, every process may execute at most one reset before acknowledging $p$’s reset.

Proof: Let $E_{reset}$ be an execution that starts with a successful reset, performed by process $p$. Assume in contradiction that process $q$ performs two resets, in $c_1$ and $c_2$, before $q$ acknowledges $p$’s reset (i.e., $c_1, c_2 \in E_{q, noack}$).

According to the algorithm, between any two resets, $q$ must execute lines g1 through g22 of procedure $next()$. Hence, $q$ reads all the registers in line g1 following $c_1$ and before $c_2$. Therefore, $q$ evaluates the if-statement condition of line g2 to be true and in the next atomic write step $q$ acknowledges $p$’s reset. Thus, $c_2 \in E_{q, ack}$ contrary to the assumption that $q$ performs another reset before acknowledging $p$’s reset.

Lemma 4.7 Every execution suffix $E$ that starts with a successful reset has a practically infinite resetless execution.
Proof: Let $c_1$ be the first configuration in $E$, in which process $p$ performed a successful reset. We denote by $c_{rq}$ the first configuration following a reset performed by process $q$ before acknowledging $p$'s reset. In case process $p$ does not perform a reset (before acknowledging $p$'s reset) then $c_{rq} = c_{rp}$. By Lemma 4.6, a process that acknowledges $p$'s reset, executes at most one reset.

We denote by $c_{i,j}$ the earliest configuration in which process $q$ that acknowledged $p$'s reset, writes $\langle e_p, r_q \rangle = \langle i, j \rangle$. Such configurations exists in $E$ by the fact that a process always increases its round (epoch) number by 1 in lines $g21$ ($h2$, respectively) or copied a higher round (epoch) number in line $g12$. Let $q$ be the first process that performs a reset after acknowledging $p$'s reset. According to the algorithm (lines $g16$ through $g20$), $q$ performs a reset when either $e_p = \inf$ or $r_p = \inf$. Hence, in $E$ there exists a practically infinite (i.e., $0 < i, j < \inf$) number of configurations $c_{i,j}$.

In $E$ there exist at most $n$ resets until $e_p = \inf$ or $r_p = \inf$. Thus, by the pigeon hole principle, it holds that there are two consecutive resets for which there exist at least $\inf \frac{n}{n+1}$ atomic steps in between. Note that following a reset configuration $c_{ru}$, a process may temporary assign zero to its epoch or round number. Hence, the number of atomic steps may be greater than $\inf \frac{n}{n+1}$.

Thus, after a successful reset there exists a practically infinite resetless execution.

Lemma 4.8 In every execution $E$ in which there are $2n^2 - 2n + 1$ resets, there exists a successful reset.

Proof: Assume in contradiction that there are more than $2n^2 - 2n + 1$ unsuccessful resets in $E$. The assumption yields the existence of a process $p$ that performs more than $2n - 1$ unsuccessful resets. Hence, some process interferes with $p$'s reset more than twice. That is, for such a process, there exist a happen-before chain of read and write operations of epoch or round numbers to an epoch or round number before $p$'s reset. Let $q$ be the first process that interferes three times with $p$'s reset.

Let $c_{i_1}$, $c_{i_2}$ and $c_{i_3}$ be the first three consecutive reset configurations in which $q$ interferes with $p$'s reset. Note that $c_{i_3}$ is a $p$ reset configuration that occurs after $p$ performs at least 3 resets and no more than $2n - 1$ resets (every process interferes twice and $q$ interferes three times). Consider the case that $q$ is crashed in $E$. Then, following $c_{i_1}$, process $p$ reads $R_q$. In $c_{i_2}$ it holds that $q$ is always unbalanced towards $p$ contrary to the assumption that $q$ interferes with the reset in $c_{i_2}$. Hence, $q$ is not crashed.

Consider the case that $q$ is not crashed. Since $q$ interferes in $c_{i_1}$, $c_{i_2}$ and since $p$'s reset sequence number changes in $c_{i_2}$, $q$ must write to $R_q$ a new reset sequence number. Let $a_1$ be the atomic write step in which $q$ writes $p$'s reset sequence number that appears in configuration $c_{i_2}$. By the fact that every write is followed by a read, $q$ executes an atomic step $a_2$ that reads $p$'s reset sequence number after $a_1$. By similar arguments, $q$ must write again in order to interfere with the reset in $c_{i_3}$. We denote by $a_3$ the atomic step in which $q$ writes a value that interferes with $p$'s reset in $c_{i_3}$ and we denote by $a' q$'s atomic step that reads $R_p$ immediately before $a_3$. Note that $a'$ and $a_3$ are executed in the procedure next().

By the fact that $p$ executes less than $2n$ resets after $c_{i_1}$ and until $c_{i_3}$, process $p$ writes a new sequence number in $c_{i_3}$. Since $q$ reads $R_p$ following $c_{i_1}$, it holds that $q$ cannot write the reset sequence number used in $c_{i_3}$ prior to reading the value from $R_p$ (i.e., following $c_{i_3}$). Thus, $a'$ and $a_3$ are executed after $c_{i_3}$. Hence, when $q$ evaluates the if-statement in line $g2$, the predicate $\text{unbalanced}(p, q)$ is true and $q$ acknowledges $p$'s reset, which is contrary to the fact that $q$ interferes with $p$'s reset following $c_{i_3}$.

Therefore, every process may interfere only twice and $p$ performs a successful reset after at most $2n - 2$ unsuccessful resets. Thus, some process performs a successful reset in any sequence of at least $(2n - 2)n + 1$ resets in $E$. ■
4.3 Self-Stabilization Consensus and Replicated state-machine Proofs

**Theorem 4.9** Every execution $E$ of the algorithm in Figure 2, 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.

**Proof:** First, we show that the consensus task is satisfied. Then, we show how the replicated state-machine task is achieved using the consensus task. In case there are $2n^2 - 2n + 1$ resets in $E$, then by Lemma 4.8, there exists a successful reset. By Lemma 4.6, after a successful reset every process executes at most one reset. Thus, after at most $2n^2 - n$ resets there exists a practically infinite resetless execution (Lemma 4.7). In case there are less than $2n^2 - 2n + 1$ resets in $E$, then by the pigeon hole principle there is a practically infinite execution interval in which no process performs a reset.

Therefore, by Lemma 4.3 and Lemma 4.4, the eventual termination, validity and agreement properties hold in every epoch starting with $e$. Hence the consensus task is satisfied.

By the fact that a state-machine transition is derived directly from the decision of the consensus algorithm in some epoch, the state-machine eventual coordination property holds in every epoch equal or higher than $e$. Next, we show that no two processes differ in the state of the replicated state-machine in every epoch $e' > e$. Note that the eventual consistency is trivial when only one process reaches epoch $e' > e$.

Whenever a process adopts a decision estimate or a decision value from another process (lines $g9$ through $g15$), the state of the replicated state-machine is copied. Therefore, no two processes that decides in epoch $e' > e$ differ in their machine state in epoch $e'$. Thus, we only have to consider the case in which a process reaches epoch $e'$ by copying the epoch number of another process (i.e., executing lines $g9$ through $g15$). By the fact that such a process must copy its state from another process, it cannot introduce a new (different) state in $e'$.

Hence, no two processes that reaches epoch $e' > e$ differ in the state of the replicated state-machine. Therefore, the eventual consistency property holds in every epoch $e' > e$. Thus, the replicated state-machine task is satisfied.]]