Self-Stabilization of Byzantine Protocols

Ariel Daliot and Danny Dolev

School of Engineering and Computer Science, The Hebrew University of Jerusalem, Israel. {adaliot,dolev}@cs.huji.ac.il

Abstract. Awareness of the need for robustness in distributed systems increases as distributed systems become integral parts of day-to-day systems. Self-stabilizing while tolerating ongoing Byzantine faults are wishful properties of a distributed system. Many distributed tasks (e.g. clock synchronization) possess efficient non-stabilizing solutions tolerating Byzantine faults or conversely non-Byzantine but self-stabilizing solutions. In contrast, designing algorithms that self-stabilize while at the same time tolerating an eventual fraction of permanent Byzantine failures present a special challenge due to the "ambition" of malicious nodes to hamper stabilization if the systems tries to recover from a corrupted state. This difficulty might be indicated by the remarkably few algorithms that are resilient to both fault models. We present the first scheme that takes a Byzantine distributed algorithm and produces its self-stabilizing Byzantine counterpart, while having a relatively low overhead of $O(f')$ communication rounds, where f' is the number of actual faults. Our protocol is based on a tight Byzantine self-stabilizing pulse synchronization procedure. The synchronized pulses are used as events for initializing Byzantine agreement on every node's local state. The set of local states is used for global predicate detection. Should the global state represent an illegal system state then the target algorithm is reset.

1 Introduction

On-going faults whose nature is not predictable or that express complex behavior are most suitably addressed in the Byzantine fault model. It is the preferred fault model in order to seal off unexpected behavior within limitations on the number of concurrent faults. Most distributed tasks require the number of concurrent Byzantine faults, f, to abide by the ratio of $3f < n$, where n is the network size. See [13] for impossibility results on several consensus related problems such as clock synchronization. Additionally, it makes sense to require systems to resume operation after a major failure without the need for an outside intervention and/or a restart of the system from scratch. E.g. systems may occasionally experience short periods in which more than a third of the nodes are faulty or messages sent by all nodes may be lost for some time due to a network failure.

Such transient violations of the basic fault assumptions may leave the system in an arbitrary state from which the protocol is required to resume in realizing its task. Typically, Byzantine algorithms do not ensure convergence in such cases, as strong assumptions are usually made on the initial state and thus merely focus on preventing Byzantine faults from notably shifting the system state away from the goal. A *self-stabilizing* algorithm bypasses this limitation by being designed to converge within finite time to a desired state from any initial state. Thus, even if the system loses its consistency due to a transient violation of the basic fault assumptions (e.g. more than a third of the nodes being faulty, network disconnected, etc.), then once the system becomes coherent again the

protocol will successfully realize the task, irrespective of the resumed state of the system. In trying to combine both fault models, Byzantine failures present a special challenge for designing stabilizing algorithms due to the "ambition" of malicious nodes to incessantly hamper stabilization, as might be indicated by the remarkably few algorithms resilient to both fault models.

We present an algorithm for transforming any Byzantine protocol to its self-stabilizing semi-synchronous counterpart, which is to the best of our knowledge, the first general scheme to do so for arbitrary protocols in the Byzantine fault model. Our result operates in the semi-synchronous network model typical of Byzantine protocols, though our scheme will also transform any asynchronous algorithm into its self-stabilizing semi-synchronous counterpart. Transient failures can practically be equivalent to the existence of an unbounded number of concurrent Byzantine failures. No distributed algorithm can reach its goal deterministically, in the face of permanent unbounded Byzantine failures, unless digital signatures are used. In a self-stabilizing paradigm, using digital signatures to counter Byzantine nodes exposes the protocols to "replay-attack" which might empty its usefulness.

Thus, deterministic protocols that tolerate permanent unbounded Byzantine failures by using digital signatures do not guarantee operation from arbitrary states and are thus not self-stabilizing. Hence, in order to self-stabilize and tolerate unbounded Byzantine failures it is essential to assume that eventually the bound on the permanent number of Byzantine failures is less than a third of the network. From this arbitrary state our protocol causes the user's target algorithm to converge efficiently. Therefore our result is stronger than just resilience to permanent unbounded Byzantine faults.

The algorithm assumes the existence of a module that delivers synchronized pulses to all the nodes. The function of the pulse synchronization is to align the activities of the participating nodes in a self-stabilizing and fault-tolerant manner. The use of an external pulse module subjects the protocol to a single point of failure. This necessitates an internal pulse mechanism in order to guarantee continuous function of the system at times that the external pulse is missing, which obliterates the benefit of circumventing any internal mechanisms with external ones. The only distributed internal protocols that delivers periodic synchronized pulses in a self-stabilizing manner tolerant to Byzantine faults are [7, 8].

The idea of the algorithm, in a bird's-eye view, is to run at each node, in the background, the self-stabilizing Byzantine protocol that periodically invokes tightly synchronized pulses. Subsequent to a pulse, the node initiates Byzantine agreement on its local application state. This ensures that following some bounded time there is consensus on the local state of every node (inclusive of faulty nodes). All correct nodes then evaluate whether this global application snapshot corresponds to a legal state of the basic program and, if required, collectively reset it at the next pulse.

The overhead of our protocol is $O(f')$ communication rounds, where f' is the actual number of permanent faults, in addition to the time complexity of the transformed non-stabilizing algorithm. We utilize a Byzantine Agreement protocol that works in a time-driven manner that we have presented in [9], which makes the agreement procedure progress as a function of the actual message transmission times and not the upper bound on the message transmission times. Consequently, the additional overhead can in effect be very low.

We postulate that the semi-synchronous network model is a very realistic and ubiquitous model that is essentially the underlying setting of overlay networks and even the internet. Our result implies that the semi-synchronous network model allows for a very extensive treatment of different models of fault tolerance.

2 Related Work

There are very few specific protocols that tolerate both transient failures as well as permanent Byzantine faults. In this section we survey most of them. Towards the end of the section we describe a few general schemes that aim at stabilizing arbitrary asynchronous non fault tolerant algorithms. To the best of our knowledge our result is the only general scheme that transforms an arbitrary Byzantine algorithm into a multitolerant program that is self-stabilizing in the presence of permanent Byzantine failures.

The concept of *multitolerance* is coined by Kulkarni and Arora [2, 17] to describe the property of a system to tolerate multiple fault-classes. They present a component based method for designing multitolerant programs. It is shown how to step-wise add tolerance to the different fault-classes separately. They design as an example a repetitive agreement protocol tolerant to Byzantine failures and to transient failures. Similarly, mutual exclusions for transient and permanent (non Byzantine) faults is designed. In [16] a multitolerant program for distributed reset is designed that tolerates transient and permanent crash failures. It is not shown how the method can be utilized for designing arbitrary algorithms, rather, particular problems are addressed and protocols are specifically designed for these problems using the method.

Nesterenko and Arora [20] define and formalize the notion of *local tolerance* in a multitolerant fault model of unbounded Byzantine faults that eventually comply with the $3f < n$ ratio. Local tolerance refers to the property of faults being contained within a certain distance of the faulty nodes so that nodes outside this containment radius are able to eventually attain correct behavior. They present two locally tolerant Byzantine self-stabilizing protocols for the particular problems of graph coloring and the dining philosophers problem.

Other examples are the two randomized self-stabilizing Byzantine clock synchronization algorithms presented by Dolev and Welch [12]. Both protocols have exponential convergence time. Our deterministic self-stabilizing Byzantine clock synchronization algorithm in [6] converges in linear time[1].

Many papers have been published that seek to find a universal technique to convert an arbitrary asynchronous protocol into a self-stabilizing equivalent. Thus these works have very limited handling of faults besides the transient faults. The concept of a *self-stabilizing extension* of a non-stabilizing protocol is brought by Katz and Perry [15]. They show how to compile an arbitrary asynchronous protocol into a self-stabilizing equivalent by centralized predicate evaluation. A self-stabilizing version of Chandy-Lamport snapshots that is recurrently executed is developed. The snapshot is evaluated for a global inconsistency and a distributed reset is done if necessary. This is improved by the local checking method of Awerbuch et al., [4]. Kutten and Patt-Shamir [18] present a time-adaptive transformer which stabilizes any non-stabilizing protocol in $O(f')$ time but on the expense of the space and communication complexities. A stabilizer that takes any off-line or on-line algorithm and "compiles" a self-stabilizing version of it is presented by Afek and Dolev [1]. The stabilizer has the advantage of being local, whereby local it is meant that as soon as the system enters a corrupt state, that fact is detected and second that the expected computation time lost in recovering from the corrupted state is proportional to the size of the corrupted part of the network. In a seminal paper by Arora and Gouda [3] a distributed reset protocol for shared memory is presented which tolerates fail-stop

[1] Note that the pulse synchronization procedure used in [6] has a flaw, as pointed out by Mahyar Malekpour from NASA LaRC and Radu Siminiceanu from NIA. A correct version can be found in [8].

failures. Note that the fail-stop failure assumption (as opposed to the sudden crash faults) makes the protocol non-masking and thus doesn't truly tolerate permanent faults. Moreover it has a relatively costly convergence time.

Gopal and Perry [14] present a framework for unifying process faults and systemic failures, i.e. ongoing faults and self-stabilization. Their scheme works in a fully synchronous system and is a "compiler" that creates a self-stabilizing version of any fault-tolerant fully synchronous algorithm. They assume the non-stabilizing algorithm works in synchronous rounds. Assuming a fully synchronous system is a strong assumption as it obliterates the need to consider the loss of synchronization of the rounds following a transient failure. Their scheme only assumes the loss of agreement on the round number itself. To overcome this following a systemic (transient) failure, at each round some sort of "agreement" is done on the round number. They assume the register holding the round number is unbounded, which is not a realistic assumption. In a self-stabilizing scheme a transient failure can cause the register to reach its upper limit. Thus they do not handle the overflow and wrap-around of the round number which is a major flaw. The permanent faults that the framework tolerates are any corruption of process code. This may seem very similar to Byzantine faults but the difference hinges on a subtle but significant dissimilarity. It is assumed that corruption of process code cannot result in malicious or two-faced behavior whereas Byzantine failures allow for any adversary behavior. This difference results in the FLM result [13] for Byzantine behavior, in which at least $3f + 1$ nodes are required to mask f failures. Conversely, corruption of process code imposes no such bound on the number of concurrent failures.

Note that being in an illegal global state is a stable predicate of the system state of a non-stabilizing program as otherwise it would either be self-stabilizing or not have the closure property that is required of any "rational" non-stabilizing algorithm (i.e. if in a legal state then stay in a legal state). A more general way of presenting our scheme is as a self-stabilizing Byzantine method for detection of stable predicates in semi-synchronous networks (see [21] for non fault-tolerant predicate detection in semi-synchronous networks). Distributed reset is just one particular action that can be done upon the detection of a certain predicate. Examples of other predicate detection uses are deadlock detection, threshold detection, progress detection, termination detection, state variance detection (e.g. clock synchronization), among others.

3 Model and Definitions

The environment is a semi-synchronous network model of n nodes that communicate by exchanging messages. We assume that the message passing allows for an authenticated identity of the senders. The communication network does not guarantee any order on messages among different nodes. Individual nodes have no access to a central clock and there is no external pulse system. The hardware clock rate (referred to as the *physical timers*) of correct nodes has a bounded drift, ρ, from real-time rate. When the system is not coherent then there can be an unbounded number of concurrent Byzantine faulty nodes, the turnover rate between faulty and non-faulty nodes can be arbitrarily large and the communication network may behave arbitrarily.

Definition 1. *A node is* **non-faulty** *at times that it complies with the following:*

1. *Obeys a global constant* $0 < \rho << 1$ *(typically* $\rho \approx 10^{-6}$*), such that for every real-time interval* $[u, v]$:

   $(1-\rho)(v-u) \leq$ *'physical timer'*$(v) -$ *'physical timer'*$(u) \leq (1+\rho)(v-u)$.

2. *Operates according to the instructed protocol.*
3. *Processes any message of the instructed protocol within π real-time units of arrival time.*

A node is considered **faulty** if it violates any of the above conditions. We allow for Byzantine behavior of the faulty nodes. A faulty node may recover from its faulty behavior once it resumes obeying the conditions of a non-faulty node. For consistency reasons, the "correction" is not immediate but rather takes a certain amount of time during which the non-faulty node is still not counted as a correct node, although it supposedly behaves "correctly"[2]. We later specify the time-length of continuous non-faulty behavior required of a recovering node to be considered **correct**.

Definition 2. *The communication network is* **non-faulty** *at periods that it complies with the following:*

1. *Any message sent by any non-faulty node arrives at every non-faulty node within δ real-time units;*
2. *All messages sent by a non-faulty node and received by a non-faulty node obey FOFI order.*

Basic notations:

- $d \equiv \delta + \pi$. Thus, when the communication network is non-faulty, d is the upper bound on the elapsed real-time from the sending of a message by a non-faulty node until it is received and processed by every correct node.
- A "*pulse*" is an internal event targeted to happen in tight synchrony at all correct nodes. A ***Cycle*** is the "ideal" time interval length between two successive pulses that a node invokes, as given by the user. The actual cycle length has upper and lower bounds and can be shortened to $cycle_{min}$ by faulty nodes. (see [8] for the details of the *pulse synchronization*).
- σ represents the upper bound on the real-time between the invocation of the pulses of different correct nodes (*tightness of pulse synchronization*)[3].
- $pulse\_conv$ represents the convergence time of the underlying pulse synchronization module.
- $agreement\_duration$ represents the maximum real-time required to complete the chosen Byzantine consensus/agreement procedure[4].

Note that n, f and $Cycle$ are fixed constants and thus non-faulty nodes do not initialize with arbitrary values of these constants. It is required that $Cycle$ is chosen s.t. $cycle_{min}$ is large enough to allow our protocol to terminate in between pulses.

A recovering node should be considered correct only once it has been continuously non-faulty for enough time to enable it to go through a complete "synchronization process". This is the time it takes, from any state, to complete two concomitant pulses that are in synchrony with all other correct nodes.

Definition 3. *A node is* **correct** *following $pulse\_conv + 2 \cdot Cycle + \sigma$ real-time of continuous non-faulty behavior.*

---

[2] For example, a node may recover with arbitrary variables, which may violate the validity condition if considered correct immediately.

[3] The specific pulse synchronization used ([8]) achieves $\sigma \leq 3d$.

[4] We differentiate between *consensus* on an initial value held by all nodes and *agreement* on an initial value sent by a specific possibly faulty node.

Definition 4. *The system is said to be* **coherent** *at times that it complies with the following:*

1. *At least $n - f$ of the nodes are correct, where $n \geq 3f + 1$;*
2. *The communication network has been continuously non-faulty for at least $pulse\_conv + 2 \cdot Cycle + \sigma$ real-time units.*

The reference to correct instead of non-faulty nodes circumvents the ability of the turnover rate between faulty and non-faulty behavior of nodes to hinder the system from ever converging to a legal state. Hence, if the system is not coherent then there can be an unbounded number of concurrent faulty nodes; the turnover rate between faulty and non-faulty nodes can be arbitrarily large and the communication network may behave arbitrarily. When the system is coherent, then the network and a large enough fraction of the nodes $(n-f)$ have been non-faulty for a sufficiently long time period for the pre-conditions for convergence of the protocol to hold. The assumption in this paper, as underlies any other self-stabilizing algorithm, is that eventually the system becomes coherent. Note that being coherent does not imply that the system is in a legal state.

The self-stabilization paradigm assumes that all variables and program registers are volatile and thus prone to corruption or can initialize with arbitrary assignments. Conversely, it assumes that the code (the instructed protocol) is not dynamic and can thus be stored on non-volatile or non-corruptible storage. Furthermore, it is assumed in the paradigm that any access to an external module utilized by the system is eventually restored. E.g., any dependency on continuous time correlated to real-time without access to an external time source, can not be handled in the context of self-stabilization as no algorithm can restore the reference to external time without access to the external time source.

A *local state* of a node is comprised of the program counter and an assignment of values to the local variables. A node switches from one local state to another through a computation step. A *global state* of a system of nodes is the set of local states of its constituents nodes and the contents of the FIFO communication channels. A *local application state* is a subset of the variables of the local state that are relevant for the application. Two local states are said to be *distinct* if they represent local states on different nodes. A *global application state* is a collection of all the distinct constituent local application states at a certain moment. A *global application snapshot* is any collection of distinct local application states. An *execution* of a program P is a possibly infinite sequence of global states in which each element follows from its predecessor by the execution of a single computation step of P. We define E to be the set of all possible execution sequences of a program P.

Definition 5. *An initial state is said to be* ***normal*** *if the program counter of each correct node is 0 and the communication channels are empty.*

Definition 6. *A* ***normal execution*** *is an execution whose initial state is normal and has entirely occurred while the system is coherent.*

Definition 7. *A global application state is said to be* ***legal*** *if it could occur in a normal execution.*

Definition 8. *A* ***legal execution*** *is an execution that is a non-empty suffix of a normal execution.*

We define NE, $(NE \subset E)$, to be the set of normal executions of P (also denoted $NE(P)$). Equivalently, we define LE, $(LE \subset E)$, to be the set of legal

executions of P (denoted $LE(P)$ respectively). The legal global states and the set of legal executions are determined by the particular task in the specific system and its respective normal executions. This cannot be characterized in general terms regardless of the actual problem definition that program P seeks to solve.

The self-stabilization of a system is informally defined by the requirement that every execution in E has a non-empty suffix in LE. We adopt the definitions of a self-stabilizing extension of a non-stabilizing program from [15]:

Definition 9. *A **projection** of a global state onto a subset of the variables and the messages on the channels is the value of the state for those variables and messages.*

Definition 10. *Program Q is an **extension** of program P if for each global state in $NE(Q)$ there is a projection onto all variables and messages of P such that the resulting set of sequences is identical to $NE(P)$, up to stuttering*[5]*.*

Note that when one considers only those portions of Q's global state that correspond to P's variables and messages and if repetitions of states are ignored, then the legal executions of P and Q are identical. Thus, a state of Q is a legal state of Q iff the projection onto P is a legal state of P. The program P to be extended is called **the basic program**.

Definition 11. *Program Q is a **self-stabilizing extension** of a program P if Q is an extension of P and any execution in E(Q) has a non-empty suffix whose projection onto P is in LE(P).*

Thus, informally, if Q is a self-stabilizing extension of P then the projection of Q onto P is self-stabilizing. Therefore we refer to Q as a **stabilizer** of P.

4 A Byzantine Stabilizer

Intuitively, the task of stabilizing a program should supposedly be rather straightforward: Every period of time, make all nodes report their internal states, then sift through the collected states and search for a possibly global inconsistency in the algorithm as emerges from the global snapshot. Upon such an inconsistency make all nodes reset to a consistent state. Below we display a conceptual view of the scheme:

At "*time − to − exchange − states***" do**
1. Send local state to all nodes and Byzantine Agree on every node's state;
2. All correct nodes now see the same global snapshot;
3. Check if global snapshot represents a legal state;
4. If not then reset the basic program;
5. If yes but your state is corrupt then repair state;

This greatly simplified scheme does not address the many subtle problems that surface when facing transient faults and permanent Byzantine faults: How do you synchronize the point in time for reporting the internal states? How do you ensure that the global snapshot is concurrent enough to be meaningful? How do you prevent Byzantine nodes from causing correct nodes to see differing global snapshots? How does the predicate detection mask Byzantine values?

We address the synchronization issue by employing an underlying Byzantine self-stabilizing pulse synchronization procedure. The pulse is essentially used as

[5] When comparing sequences, adjacent identical states are eliminated; this is called the elimination of stuttering in [15].

the event that helps to determine when to report the local state. The "meaningfulness" of the global snapshot is addressed by the observation that many algorithms have identifiable events in their executions. In a semi-synchronous protocol different nodes should execute the same events within a small bounded time of each other. If all correct nodes report their local states and clock time[6] at such an event (denoted *sampling point*) then the combination of clock time and the emergent global snapshot can be used for deducing whether the protocol is in a legal state. As an example, consider that the events are the beginning of a round, in case the basic program works in rounds. Thus all correct nodes should, whenever the system is in a legal state, reach the event of a specific round within bounded clock time of each other. By instructing the nodes to report their state (round number) and clock time at the specific round, it can be deduced whether this event indeed happened within the legal bounded time. If so, then that implies that the global snapshot taken carries meaningful information about the global state of the system. By evaluating this global predicate a decision can be made as of the legality of the global state and a reset can be done, if required. If the reported clock times are "too far" from each other then this is a sufficient indication that the system is not in a legal state and thus should be reset.

The issue of Byzantine nodes and values are tackled by initiating Byzantine agreement on the reported states. This ensures that all correct nodes have identical views of the global snapshot.

Our scheme stabilizes any Byzantine protocol that has such events (sampling points) during the execution, which can be identified by checking the program counter and local state. Otherwise, it is required that the basic program signals when to read and report the local state. We argue that this definition covers an extensive set of protocols. Programs that work in round structure is just a specific and easily identifiable example of such protocols. We assume for simplicity that the sampling points are taken at least 4σ apart on the same node in order to be able to differentiate between adjacent sampling points due to the synchronization uncertainties. It remains open whether this bound is really required. In Section 5 we give a detailed example of how to extend a specific clock synchronization algorithm that does not operate in a round structure.

Note that we do not aim at achieving a consistent global snapshot in the Chandy-Lamport sense (see [5]), which is not clearly defined in the Byzantine fault model. For our purposes a projection of the local state to the application state suffices in order to detect states that violate the assumptions of the basic program on its initial states, which rendered it non-stabilizing in the first place.

Generally, the extension of the basic program is established through a user-supplied wrapper function, so called because it "wraps" the basic program and functions as an interface between the basic program and the stabilizer. Note that the wrapper procedures must be supplied by the implementor. This is because it is a semantic matter to determine whether the global application state predicate indicates an illegal state that violates the assumptions of the basic program. For the sake of modularity and readability the wrapper is divided into two distinct modules according to its two main functions. The GetState_Wrapper() module interprets the local state of the basic program and returns the local state at the sampling points. The EvalState_Wrapper() module evaluates the agreed global application snapshot and determines whether it is legal with respect to the application. It also instructs a node how to repair its local application state as a function of the global application snapshot, should a node detect that its local application state is inconsistent with the legal global application snapshot.

[6] Note that the clock time can be the elapsed time on a node's timer since the pulse. The synchronization of the pulses implies synchronization of these clocks.

Restrictions on the basic program:

R1: The basic program at all correct nodes can be initialize within at least σ real-time units apart. The procedure INIT_BASIC_PROGRAM initializes it.
R2: The basic program can tolerate that up to f of the nodes can choose to keep values from previous incarnations of the basic program (e.g. for replay of digital signatures).
R3: Has repeated *sampling points* during execution that can be identified through the local state. The sampling points are such that if all correct nodes report their state at the same corresponding sampling point then the global application snapshot is "meaningful" with respect to the application.
R4: During a legal execution all the correct nodes' sampling points are within Δ real-time units of each other. The background pulse algorithm implies that $\Delta \geq \sigma$, because the pulse skew may cause the nodes to reach the sampling points up to σ real-time units of each other.
R5: There exists a value Σ, such that in every time-window that is at least some Σ real-time units long every correct node has at least one sampling point. This value also covers the initialization period of the basic program.
R6: The set of legal application states of the basic program can be determined by evaluating a predicate on the application state variables. An additional requirement is that if up to f non-faulty nodes detect that their own local state is inconsistent with a legal global application snapshot then it can be repaired without needing a global reset[7].
R7: The basic program has a closure property with regards to the legal global states. I.e. if the system is in a legal state and the system is coherent then it stays in a legal state as long as the system stays coherent.

To formalize the intuition we give a more refined presentation of the algorithm:

At "*pulse*" event Do */* received the internal pulse event */*
1. Revoke possible other instances of the algorithm and clear the data structures;
2. If (*reset*) then Do invoke INIT_BASIC_PROGRAM;*/* reset the Basic Program */*

/ Lines 3,4 are executed by the* GETSTATE_WRAPPER() *procedure */*
3. Upon a sampling point Do
4. Set $Timer$:= elapsed time since pulse;
5. Record *app_state* & invoke BYZ_AGREEMENT on (*app_state*, $Timer$);

/ Line 6 is executed about agreement_duration time after the* $f{+}1^{st}$ *agreement */*
6. Sift through agreed values for a cluster of $\geq n - f$ values whose $Timer$s within 2Δ of each other, thus comprising a meaningful global application snapshot;
7. If no such cluster exists then Do *reset* := *true*;

/ Lines 8,9,10 are executed by the* EVALSTATE_WRAPPER() *procedure */*
8. Else Do predicate evaluation on the global application snapshot;
9. If global application snapshot is not legal Do *reset*:=*true*;
10. Else If you are not part of the cluster Do Repair your application state;

---

[7] A basic program that lacks this property might not converge to a legal state.

The complete algorithm, denoted ByzStabilizer, is given below:

Algorithm 1 ByzStabilizer /* *executed at node q* */

At "*pulse*" event Do /* *received the internal pulse event* */
Begin
1. Revoke possible other instances of ByzStabilizer and clear the data structures;
2. $Timer := 0$; $T_{pivot} := 0$;
3. If (*reset*) then Do invoke Init_Basic_Program; /* *reset the Basic Program* */
4. Wait until $Timer = \sigma \cdot (1 + \rho)$ time units;

/* *read&agree state at sampl. point; collect f+1 agreed states in window* */
5. Do
6. Invoke in the background $RecState :=$ GetState_Wrapper();
7. If $RecState \neq \perp$ then Do invoke Byz_Agreement(q, $RecState$, $Timer$);
8. $AS := \{(p, S, T) \mid$ Byz_Agreement returned $S \neq \perp\}$; /* *add agreed state* */
9. $Agr\_nodes := \{p_i \mid (p_i, \_, T_i) \in AS,\ \sigma + \Delta \leq T_i \leq \Sigma + \Delta\}$; /* *minimal* T_i */
10. Until $(\| Agr\_nodes \| \geq f + 1$ or $Timer > \Sigma + \Delta + agreement\_duration)$;

/* *collect agreed states, until no more possible states from correct nodes* */
11. Do
12. $AS := \{(p, S, T) \mid$ Byz_Agreement returned $S \neq \perp\}$; /* *add agreed state* */
13. $Agr\_nodes := \{p_i \mid (p_i, \_, T_i) \in AS,\ \sigma + \Delta \leq T_i \leq \Sigma + \Delta\}$; /* *minimal* T_i */
14. Let *pivot* be the $f{+}1^{st}$ node in *Agr_nodes*, in ascending order by their min. T_i;
15. Until $Timer \geq T_{pivot} + (\sigma + \Delta + agreement\_duration) \cdot (1 + \rho)$ time units;

/* *seek cluster of* $\geq n - f$ *values whose Timers within* 2Δ *of each other* */
16. $AS' := \{(p, S, T) \in AS \mid \sigma + \Delta \leq T \leq T_{pivot} + \Delta \cdot (1 + \rho)\}$;
17. $Cluster\_rep := \{(p_c, S_c, T_c) \in AS' \mid$
$\| \{p' \mid (p', S', T') \in AS\ \&\ T_c \leq T' \leq T_c + 2\Delta\ \&\ S_c \sim S'\} \| \geq n - f\}$;

/* *if no cluster do reset, otherwise evaluate snapshot of earliest cluster* */
18. If $\| Cluster\_rep \| = 0$ then Do $reset :=$ true; /* *if no n-f sized cluster found* */
19. Else Do $(p_c, S_c, T_c) := \min_T\{(p, S, T) \in Cluster\_rep\}$;/* *else seek earliest cluster* */
20. $globAppSnapshot := \{(p', S', T') \in AS \mid T_c \leq T' \leq T_c + 2\Delta\ \&\ S_c \sim S'\}$;
21. $reset :=$ EvalState_Wrapper($globAppSnapshot$);/**reset,repair or nothing*/
End

The internal pulse event is delivered by the pulse synchronization procedure (presented in [8]). The synchronization of the pulses ensures that the ByzStabilizer procedure is invoked within σ real-time units of its invocation at all other correct nodes. Note that we do not assume any correlation between the pulse cycle and any internal cycles or rounds of the basic program. Hence at the time of the pulse, the basic program may be in any of its states. The Byzantine agreement procedure used, Byz_Agreement, is essentially the consensus procedure of [9]. We present its agreement equivalent in Section 7.

Line 1: Following the pulse any possible on-going invocation of ByzStabilizer (and thus any on-going Byz_Agreement or instance of the wrappers, but not the execution of the basic program) is revoked and all data structures that are not used by the basic program are cleared. The exception is the "reset" variable that is not cleared. Note that the application state, as it belongs to the basic program, remains intact.

Line 2-3: Each node p initializes a $Timer$ that holds the elapsed clock time since the last pulse invocation, before possibly doing a reset of the basic program.

Lines 4-7: When the GetState_Wrapper() wrapper procedure encounters a sampling point subsequent to the pulse, at elapsed time = $Timer$, then it records the local application state into the *RecState* variable. Agreement is then invoked on (p, *RecState*, $Timer$). The procedure GetState_Wrapper() sanity checks the state recorded at line 6, thus if it detects that the local application state is invalid or corrupt it will return $\perp$.

Lines 8-15: Target at identifying the $f+1^{st}$ (time-wise) distinct node whose value has been agreed upon, denoted the *pivot* node. Note that after a bounded time all correct nodes will identify the same pivot node. The time appearing in the agreed value of the pivot node is denoted T_{pivot}. The variable AS holds the set of agreed states. The variable *Agr_nodes* holds the set of nodes whose values have been agreed on.

Lines 16-17: A bounded period of time subsequent to T_{pivot}, all correct nodes must have terminated agreement on all nodes' values. It is then, that a cluster of at least $n-f$ agreed values is searched for, such that their $Timer$s are within 2Δ of each other.

Line 18: Such a cluster, if exists, comprises a meaningful global application snapshot. Otherwise, the global application state must be in an illegal state.

Lines 19-21: If a cluster is detected, then the EvalState_Wrapper procedure evaluates the global application snapshot. It determines whether the node must repair its local application state; whether a global reset should be scheduled at the next pulse invocation or whether the global application state is assumed to be legal and thus nothing is done. The $\sim$ notation denotes equality between cluster identifiers.

The following Lemma and Theorem apply as long as the system is coherent:

Lemma 1. *If the system is in an arbitrary global state then, within finite time, subsequent to line* 17 *of the* ByzStabilizer *algorithm there is agreement on the set Cluster_rep.*

Theorem 1. ByzStabilizer *is a self-stabilizing extension of any algorithm that complies with restrictions R1-R7.*

Proof. **Convergence:** Let the system be coherent but in an arbitrary global state, s, with the nodes holding arbitrary local application states. The pulse synchronization procedure is self-stabilizing, thus, independent of the system's initial state within a finite time the pulses are invoked regularly and synchronously with a tightness of σ real-time units. At the pulse invocation all remnants of previously invoked ByzStabilizer, inclusive of its sub-procedures such as the agreement and wrappers, are flushed by all the correct nodes. Following Lemma 1, subsequent to line 17 of ByzStabilizer there is consensus on the selected cluster (including of the empty cluster). At line 18 there may be one of two possibilities:

1. $\| \; Cluster\_rep \; \| = 0$: This necessarily implies the basic program is in an illegal state. In this case all correct nodes will do $reset := true$. At the next pulse all correct nodes will reset the basic program and thus converge to a legal state.
2. *A cluster was detected*: In this case subsequent to line 20 the variable globAppSnapshot, which holds the cluster whose states are the earliest agreed on since the pulse, will be generated at all correct nodes. Again, there are two cases to consider:
   (a) *The sampling points are within Δ real-time of each other*:
   Thus all correct nodes have initiated an agreement on their state within Δ real-time units of time T_{pivot} at the pivot node. Hence all correct nodes

are represented in the cluster. The reset variable will be set at line 21 by the EVALSTATE_WRAPPER predicate detection procedure. If the procedure returns that the globAppSnapshot is legal then all correct nodes do nothing. Otherwise all correct nodes will reset the basic program at the next pulse and thus the system converges to a legal global state.

(b) *The sampling points are not within Δ real-time of each other*: There are two cases to consider:
   i. *All correct nodes are represented in the cluster*:
   Thus the basic program is unsynchronized within the uncertainty window. If the EVALSTATE_WRAPPER procedure detects the illegality of the global state then all correct nodes will reset at next pulse, otherwise the illegality will not be detected and all correct nodes will not reset the basic program at the next pulse.
   ii. *At least one correct nodes is not represented in the cluster*: Again there are two cases:
      A. *The* EVALSTATE_WRAPPER *procedure evaluates in line* 21 *the application snapshot as illegal*: Then all correct nodes reset at the next pulse and the system attains a legal global state.
      B. *The* EVALSTATE_WRAPPER *procedure evaluates in line* 21 *the application snapshot as legal*: This is due to faulty nodes that "fill-in" for the lacking correct values, then these correct nodes that are not represented will detect so and must repair their local states. Thus no correct node does a reset at the next pulse. By restriction R6, a repair is done by the EVALSTATE_WRAPPER procedure as a function of the global application snapshot such that the new global state will be legal. □

Closure: Following Lemma 1 the closure proof reduces to case (2.a.) in the proof of convergence, for the case in which the global state is legal. Thus, following restriction R4 the EVALSTATE_WRAPPER procedure evaluates correctly that the global snapshot is legal and thus all correct nodes do *reset* :=*false*.

This concludes the proof of the theorem. □

5 Example of Stabilizing a Non-stabilizing Algorithm

To illustrate our method and to elucidate its generality we will provide a specific example of the conversion of a well known non-stabilizing algorithms to its stabilizing counterpart.

To stabilize the protocol using our scheme the following needs to be identified: the application state, the sampling points, the bound Δ on the real-time skew between correct nodes' sampling points in a legal state, the GETSTATE_WRAPPER procedure, the EVALSTATE_WRAPPER procedure and how it characterizes the legal states and how it does a repair, the initialization of the basic program following a global reset, the required minimal length of the cycle.

Consider the Byzantine clock synchronization algorithm in [10]. Informally that algorithm operates as follows: The processes resynchronize their clocks every *PER* time period. A process expects the time at the next resynchronization to equal *ET*. When a process's local time reaches *ET* it broadcasts a (signed) message stating "the time is *ET*". Alternatively, when a process receives such a message from $f + 1$ distinct nodes it knows that at least one correct node advanced its local time to *ET* and thus it resets its clock to *ET*. Note that this algorithm does not utilize a rounds structure.

It is interesting to note that the candidate protocol above uses signed messages in a way that does not comply with R2, because replay of signed messages

from previous incarnations of the protocol can destroy the synchronization of the clocks of the correct nodes. One can transform the protocol to conform with R2, by using Byzantine Agreement instead of sending signed messages. The difficulty above is inherent in stabilizing protocols that use digital signatures.

- The application state will be comprised of the ET variable only.
- Practically any point throughout the inter-PER period avoiding the vicinity of the resynchronization events is safe for sampling. For illustrative purposes we will define a sampling point at every time that equals $ET + PER/2$. It is clear that the ET variable is quiescent around this point when the algorithm is in a legal global state.
- The algorithm can be initialized with the required bound of σ real-time units between the different nodes. This will not affect the precision of the algorithm which will stay d. That will yield a real-time skew between correct nodes' sampling points in a legal state of $\Delta = d + PER \cdot (1 + \rho)$.
- The sampling point is identified by the GetState_Wrapper procedure through the local state event of $clocktime = ET + PER/2$, at which the ET value is read into the localAppState variable.
- The EvalState_Wrapper procedure identifies the legal application states as those in which there are at least $n - f$ identical ET values. A repair is done by a node by setting its ET value to equal the other $n - f$ or more ET values in the application snapshot if it was evaluated as legal.
- Following a reset a node should initialize the algorithm by setting its ET variable to some pre-defined value, e.g. $ET = 0$. As mentioned before, the initial skew of σ will affect the accuracy but not the precision, as early and fast nodes will reach their subsequent ET before the others, but the others late and slow nodes will set their clock accordingly upon receiving $f + 1$ messages which is uncorrelated to the initialization skew.
- The required minimal cycle length equals $PER/2$ in case the pulse correlates with the reading of the sampling point and some correct nodes will have to wait until the next sampling point. The protocol then needs to allow for a full Byzantine agreement to terminate, in addition to a few round-trip rounds. Thus the required minimal cycle length equals $PER/2 + (2f + 3)$ rounds.

6 Analysis

We require *Cycle* to be chosen s.t. $cycle_{min} > \sigma + \Sigma + agreement\_duration$.

From an arbitrary state in which the system is coherent it can take up to *pulse_conv* real-time until the pulses synchronize. Subsequent to the pulses it can take in the order of $\Sigma + agreement\_duratione$ real-time to reach a decision on a reset. The steady-state time complexity equals the time overhead from the pulse until the EvalState_Wrapper procedure terminates. Again this equals about $\Sigma + agreement\_duration$ time. With few faults and/or a fast network this becomes in the order of Σ, which is largely determined by the user and can be as low as $4d$ if the basic program allows for frequent sampling points. The message complexity is expressed in point-to-point messages. The message complexity of the steady state is roughly n^2 messages for the pulse synchronization procedure, and $f' \cdot n^2$ for the agreement algorithm.

Note that the agreement instances initiated by correct nodes will always terminate within 2 communication rounds, this is due to the early stopping property of the consensus algorithm which terminates within 2 rounds if all correct nodes hold the same initial agreement value. Thus the communication complexity is that of the actual number of faulty nodes.

The algorithm is *fault-containing*, in the sense that if faulty nodes behave "correctly" such that a correct node detects that it is not in synch with a legal global snapshot then the node can "repair" itself. Thus even though we present a reset-based protocol, repair is done up to a certain amount of concurrent faults. This is because our protocol is Byzantine resilient, thus a non-Byzantine fault or inconsistency will be masked by the protocol while the affected non-faulty node can perform a repair. Only if there should be more than f faults and inconsistencies would a system reset be performed.

The algorithm is also time-adaptive, the number of rounds executed in every cycle equals the number of actual faults, f'. This is due to the early-stopping feature of the agreement algorithm which terminates within $f' \leq f$ rounds.

Note that if solving a certain Byzantine problem can be reduced to consensus (or agreement) on the future value of the global state at the next pulse, (e.g. token circulation, see [9]), as opposed to reaching agreement on the current value of every node, then the agreement algorithm presented can be used to achieve 2-round early stopping subsequent to every pulse. Thus based on the global application snapshot at the last pulse, it can be calculated what the global state should be at this pulse. Thus if all correct nodes previously agreed on the state of every other node, which comprises the global snapshot, then they can enter agreement with consensus on the expected states for all nodes. The early stopping feature of the consensus algorithm in [9] ensures that if all correct nodes hold the same initial value to be agreed on then consensus is reached within two rounds. This makes the steady-state case extremely cost-efficient with a minimal overhead of 2 rounds. Only following a transient failure might full agreement be executed on the values of the faulty nodes, since different correct nodes may then hold different values for the same nodes.

Acknowledgements: We wish to thank Shlomi Dolev and Hanna Parnas for stimulating discussions with regards to the current result.

References

1. Y. Afek, S. Dolev, "*Local Stabilizer*", Proc. of the 5th Israeli Symposium on Theory of Computing Systems (ISTCS97), Bar-Ilan, Israel,74-84. June 1997.
2. A. Arora and S. Kulkarni, "*Component Based Design of Multitolerance*, IEEE Transactions on Software Engineering, Vol. 24, No.1, January 1998, pp. 63-78.
3. A. Arora and M. Gouda, "*Distributed Reset*, In Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science, number 472 in Lecture Notes in Computer Science, pages 316–333, 1990.
4. B. Awerbuch, B. Patt-Shamir and G. Varghese, "*Self-Stabilization by Local Checking and Correction*, In Proceedings of the 32nd IEEE Symp. on Foundation of Computer Science, 1991.
5. K. M. Chandy and L. Lamport, "*Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. on Computer Systems, Vol. 9(1):63–75, 1985.
6. A. Daliot, D. Dolev and H. Parnas, "*Linear Time Byzantine Self-Stabilizing Clock Synchronization*", In Proceedings of 7th International Conference on Principles of Distributed Systems (OPODIS-2003), La Martinique, France, December, 2003.
7. A. Daliot, D. Dolev and H. Parnas, "*Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks*", In Proceedings of the Sixth Symposium on Self-Stabilizing Systems, DSN SSS '03, San Francisco, June 2003. See also LNCS 2704.
8. A. Daliot, D. Dolev and H. Parnas, "*Self-Stabilizing Byzantine Pulse Synchronization*", Technical Report TR2005-84, Schools of Engineering and Computer Science, The Hebrew University of Jerusalem, Aug. 2005. Url: http://leibniz.cs.huji.ac.il/tr/841.pdf

9. A. Daliot, and D. Dolev, "*Self-Stabilizing Byzantine Token Circulation*", Technical Report TR2005-77, Schools of Engineering and Computer Science, The Hebrew University of Jerusalem, June 2005. Url: http://leibniz.cs.huji.ac.il/tr/834.pdf
10. D. Dolev, J. Y. Halpern, B. Simons, and R. Strong, "*Dynamic Fault-Tolerant Clock Synchronization*", Journal of the ACM, Vol. 42, No.1, pp. 143-185, 1995.
11. S. Dolev, "*Self-Stabilization*", The MIT Press, 2000.
12. S. Dolev, and J. L. Welch, "*Self-Stabilizing Clock Synchronization in the presence of Byzantine faults*", Journal of the ACM, Vol. 51, Issue 5, pp. 780 - 799, 2004.
13. M. J. Fischer, N. A. Lynch and M. Merritt, "*Easy impossibility proofs for distributed consensus problems*", Distributed Computing, Vol. 1, pp. 26-39, 1986.
14. A. S. Gopal and K. J. Perry, "*Unifying self-stabilization and fault-tolerance*", IEEE Proceedings of the 12th annual ACM symposium on Principles of distributed computing, Ithaca, New York, 1993.
15. S. Katz, K. J. Perry, "*Self-Stabilizing Extensions for Message-Passing Systems*", Distributed Computing 7(1): 17-26 (1993)
16. S. Kulkarni and A. Arora, "*Multitolerance in distributed reset*, Chicago Journal of Theoretical Computer Science, Special Issue on Self-Stabilization, 1998.
17. S. Kulkarni and A. Arora, "*Compositional Design of Multitolerant Repetitive Byzantine Agreement*, Proceedings of the 18th Int. Conference on the Foundations of Software Technology and Theoretical Computer Science, India, 1997.
18. S. Kutten and B. Patt-Shamir, "*Time-adaptive self stabilization*, In PODC97 Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, pages 149-158, 1997.
19. J. Lundelius, and N. Lynch, "*An Upper and Lower Bound for Clock Synchronization*," Information and Control, Vol. 62, pp. 190-205, Aug/Sep. 1984.
20. M. Nesterenko and A. Arora, "*Local Tolerance to Unbounded Byzantine Faults*", IEEE SRDS, pages 22-31, 2002.
21. S. D. Stoller, "*Detecting Global Predicates in Distributed Systems with Clocks*", Distributed Computing, 13(2):85-98, April 2000.
22. Sam Toueg, Kenneth J. Perry, T. K. Srikanth, "*Fast Distributed Agreement*", SIAM Journal on Computing, 16(3):445-457, June 1987.

7 Appendix - The Byz_Agreement Procedure

The Byzantine Agreement module extends the approach taken in [9] in using explicit time bounds in order to address the variety of potential problems that may arise when the system is stabilizing.

We assume that timers of correct nodes are always within $\bar{\sigma}$ of each other. More specifically, we assume that nodes have timers that reset periodically, say at intervals $\leq Cycle'$. Let $T_p(t)$ be the reading of the timer at node p at real-time t. We thus assume that there exists a bound such that for every real-time t, when the system is coherent,

$$\forall p, q \text{ if } \bar{\sigma} < T_p(t), T_q(t) < Cycle' - \bar{\sigma} \text{ then } |T_p(t) - T_q(t)| < \bar{\sigma} .$$

The bound $\bar{\sigma}$ includes all drift factors that may occur among the timers of correct nodes during that period. When the timers are reset to zero it might be, that for a short period of time, the timers may be further apart. The pulse synchronization algorithm [8] satisfies the above assumptions and implies that $\bar{\sigma} > d$.

We use the following notations in the description of the agreement procedure:

- Let $\bar{d}$ be the duration of time equal to $(\bar{\sigma} + d) \cdot (1 + \rho)$ time units on a correct node's timer. Intuitively, $\bar{d}$ can be assumed to be a duration of a "phase" on a correct node's timer.

– The *consensus-broadcast* and the *broadcast* primitives are defined in [9]. Note that an *accept* is issued within the broadcast primitive.

The BYZ_AGREEMENT algorithm is presented in a somewhat different style. Each step has a condition attached to it, if the condition holds and the timer value assumption holds, then the step is to be executed. Notice that only the step needs to take place at a specific timer value. It is assumed that the internal procedures invoked as a result of the BYZ_AGREEMENT procedure are implicitly associated with the agreement procedure.

```
Algorithm Byz_Agreement on (p, Val, T)          /* invoked at node q */

broadcasters := ∅; value :=⊥;
if p = q then send (initialize, q, Val, T + d̄, 1) to all;      /* the General */
by time (T + d̄) :
   if received (initialize, p, Val, T + d̄, 1) then
      consensus-broadcast(p, Val, T + d̄, 1);
by time (T + 3d̄) :
   if accepted (p, v, T + d̄, 1) then
         value := v;
by time (T + (2f + 3)d̄) :
   if value ≠⊥ then
         broadcast (p, value, T + d̄, ⌊(T_q − T − d̄)/(2d̄)⌋ + 1);
         stop and return value.
at time (T + (2r + 1)d̄) :
   if (|broadcasters| < r − 1) then
         stop and return value.
by time (T + (2r + 1)d̄) :
   if accepted (p, v', T + d̄, 1) and r − 1 distinct messages (p_i, v', T + d̄, i)
            where ∀i, j  2 ≤ i ≤ r, and p_i ≠ p_j ≠ p then
         value := v';
```

Fig. 1. The BYZ_AGREEMENT algorithm

The BYZ_AGREEMENT algorithm satisfies the following typical properties:

Termination: The protocol terminates in a finite time;
Agreement: The protocol returns the same value at all correct nodes;
Validity: If the initiator is correct, then the protocol returns the intiator's value;

Nodes stop participating in the BYZ_AGREEMENT protocol when they are instructed to do so. They stop participating in the broadcast primitive $2\bar{d}$ after they terminate BYZ_AGREEMENT.

Definition 12. *We say:*
A node ***returns*** *a value m if it has stopped and returned value $= m$.*
A node p ***decides*** *if it stops at that timer time and returns a value $\neq\perp$.*
A node p ***aborts*** *if it stops and returns $\perp$.*

Theorem 2. *The* BYZ_AGREEMENT *satisfies the Termination property. When $n > 3f$, it also satisfies the Agreement and Validity properties.*

Proof. The proof follows very closely to the proof of the Byz-Consensus algorithm in [9]. Notice, that there is a difference of one $\bar{d}$ resulting from the initiation of the protocol by a specific node, followed by a consensus. Another difference is that the General itself is one of the nodes, so if it is faulty there are only $f-1$ potential faults left.

Lemma 2. *If a correct node aborts at time $T+(2r+1)\bar{d}$ on its timer, then no correct node decides at a time $T+(2r+1)'\bar{d} \geq T+(2r+1)\bar{d}$ on its timer.*

Lemma 3. *If a correct node decides by time $T+(2r+1)\bar{d}$ on its timer, then every correct node decides by time $T+(2r+3)\bar{d}$ on its timer.*

Termination: Lemma 3 implies that if any correct node decides, all decide and stop. Assume that no correct node decides. In this case, no correct node ever invokes a broadcast $(p, v, T+\bar{d}, \_)$. By the consensus-broadcast properties in [9], no correct node will ever be considered as broadcaster. Therefore, by time $T+(2f+3)\bar{d}$ on their timers, all correct nodes will have at most f broadcasters and will abort and stop. □
Agreement: If no correct node decides, then all abort, and return to the same value. Otherwise, let q be the first correct node to decide. Therefore, no correct node aborts. The value returned by q is the value v of the accepted $(p, v, T+\bar{d}, 1)$ message. By the consensus-broadcast properties in [9], all correct nodes accept $(p, v, T+\bar{d}, 1)$ and no correct node accepts $(p, v', T+\bar{d}, 1)$ for $v \neq v'$. Thus all correct nodes return the same value. □
Validity: If the initiator q is correct, all the correct nodes invoke the consensus-broadcast with the same value v' and invoke the protocol with the same timer time $(T+\bar{d})$. By the consensus-broadcast properties in [9], all correct nodes will stop and return v'. □

Thus the proof of the theorem is concluded. □