# Snap-Stabilization in Message-Passing Systems[⋆]

Sylvie Delaët[1], Stéphane Devismes[2], Mikhail Nesterenko[3], and Sébastien Tixeuil[4]

[1] LRI UMR 8623, Université de Paris-Sud
sylvie.delaët@lri.fr
[2] VERIMAG UMR 5104, Université Joseph Fourier
stephane.devismes@imag.fr
[3] Computer Science Department, Kent State University
mikhail@cs.kent.edu
[4] LIP6 UMR 7606, Université Pierre et Marie Curie
sebastien.tixeuil@lip6.fr

**Abstract.** We consider *snap-stabilization* in message-passing systems. Snap-stabilization permits to design protocols that withstand transient faults: Any computation that is started after faults cease *immediately* satisfies the expected specification. Our contribution is twofold, as we demonstrate that in message passing systems *(i)* snap-stabilization is impossible for nontrivial problems if we consider channels with finite yet unbounded capacity, and *(ii)* snap-stabilization becomes possible in the same setting with bounded-capacity channels. The latter contribution is constructive, as we propose two snap-stabilizing protocols.

## 1 Introduction

*Snap-stabilization* [2] offers an attractive approach to transient fault tolerance. As soon as such faults end a snap-stabilizing protocol *immediately* operates correctly. Of course, not all safety predicates can be guaranteed when the system is started from an arbitrary global state. Snap-stabilization's notion of safety is *user-centric*: When the user initiates a request, the received response is correct. However, between the request and the response, the system can behave arbitrarily (except from giving an erroneous response to the user).

A related well-studied concept is *self-stabilization* [3]. After the end of the transient faults, a self-stabilizing protocol *eventually* satisfies its specification. Thus, snap-stabilization offers stronger safety guarantee than self-stabilization: It may take an arbitrary long time for a self-stabilizing protocol to start behaving correctly after the faults.

However, nearly every snap-stabilizing protocol presented so far assumes a high level communication model in which any process is able to read the states of every communication neighbor and update its own state in a single atomic step (this model is often referred to as the *state model*). Designing protocols with

[⋆] Due to the lack of space, many technical results have been omitted, see [1] (`http://arxiv.org/abs/0802.1123`) for more details.

forward recovery properties (such as self-stabilizing and snap-stabilizing ones) in the low level message-passing model is rather challenging. In this model, a process may either send a message to a single neighbor or receive a message from a single neighbor (but not both) together with some local computations; also messages in transit could be lost or duplicated.

Our contribution is twofold:

(1) We show that contrary to the high level state model, snap-stabilization is strictly more difficult to guarantee than self-stabilization in the low level message passing model. In more details, for nontrivial specifications, there exists no snap-stabilizing (even with unbounded memory per process) solution in message-passing systems with unbounded yet finite capacity channels. This is in contrast to the self-stabilizing setting, where solutions with unbounded memory per process [4], unbounded random sequences [5], or operating on a restricted set of specifications [6] do exist.
(2) We prove that snap-stabilization in the low level message passing model is feasible when channels have bounded capacity. Our proof is constructive, as we present snap-stabilizing protocols for *propagation of information with feedback* (PIF) and *mutual exclusion*.

## 2 Impossibility Results

We introduced the notion of *safety-distributed* specifications and shown that no problem having such a specification admits a snap-stabilizing solution in message-passing systems with finite yet unbounded capacity channels. Intuitively, safety-distributed specification has a safety property that depends on the behavior of more than one process. That is, certain process behaviors may satisfy safety if done sequentially, while violate it if done concurrently. For example, in mutual exclusion, a requesting process eventually executes the critical section but several requesting processes must not execute the critical section concurrently. Since most of classical synchronization and resource allocation problems are safety-distributed, this result prohibits the existence of snap-stabilizing protocols in message-passing systems if no further assumption is made.

This result hinges on the fact that after some transient faults the configuration may contain an unbounded number of arbitrary messages. Note that a safety-distributed specification involves more than one process and thus requires the processes to communicate to ensure that safety is not violated. However, with unbounded channels, each process cannot determine if the incoming message is indeed sent by its neighbor or is the result of faults. Thus, the communication is thwarted and the processes cannot differentiate safe and unsafe behavior.

## 3 Possibility Results

We shown that snap-stabilization becomes feasible in message-passing systems if the channels are of bounded known capacity. We present solutions to *propagation* *of information with feedback* (PIF) and *mutual exclusion.* The protocols assume fully-connected networks and use finite local memory at each process. The channels are lossy, bounded and FIFO. The program execution is asynchronous. To ensure nontrivial liveness properties, we make the following fairness assumption: If a sender process $s$ transmits infinitely many messages to a receiver process $r$ then, $r$ receives infinitely many of them. The message that is not lost is received in finite (but unbounded) time. If the channel is full when the message is transmitted, this message is lost. For simplicity, we consider single-message capacity channels. The extension to an arbitrary but known bounded message capacity channels is straightforward (see [7]).

## 3.1 PIF

The PIF scheme can be described as follows: When requested, a process – called *initiator* – starts the first phase of the PIF-computation by broadcasting a specific message $m$ into the network (the *broadcast phase*). Then, each non-initiator acknowledges to the initiator the receipt of $m$ (the *feedback phase*). The PIF-computation terminates when the initiator receives acknowledgments from every other process and decides taking account of these acknowledgments. Any process may need to initiate a PIF-computation. Thus, any process can be the initiator of a PIF-computation and several PIF-computations may run concurrently. Hence, any PIF protocol has to cope with concurrent PIF-computations.

A basic PIF implementation requires the following input/output variables:

– $\texttt{Req}_p$. This variable is used to manage the requests for the process $p$. $\texttt{Req}_p$ is set to `Wait` when $p$ is requested to perform a PIF. $\texttt{Req}_p$ is switched from `Wait` to `In` at the beginning of each PIF-computation (*n.b.* $p$ starts a PIF-computation only upon a request). Finally, $\texttt{Req}_p$ is switched from `In` to `Done` at the termination of each PIF-computation (this latter switch also corresponds to the *decision event*). Since a PIF-computation is started by $p$, we assume that $\texttt{Req}_p$ cannot be set to `Wait` before the termination of the current PIF-computation, *i.e.*, before $\texttt{Req}_p = \texttt{Done}$.[1]
– $\texttt{BMes}_p$. This buffer contains the message to broadcast.
– $\texttt{FMes}_p[1 \ldots n-1]$. $\texttt{FMes}_p[q]$ contains the acknowledgment for the broadcast message that $q$ sends to $p$.

Using these variables, a process $p$ is requested to broadcast a message $m$ when ($\texttt{BMes}_p$,$\texttt{Req}_p$) is set to ($m$,`Wait`). Consequently to this request, a PIF-computation is started, *i.e.*, $\texttt{Req}_p$ is set to `In`. This computation terminates when $\texttt{Req}_p$ is set to `Done`. Between the start and the termination, the protocol has to generate two types of event at the application level. First, a "**B-receive**$\langle m \rangle$ **from** $p$" event at any other process $q$. When this event occurs, the application at $q$ is assumed to treat the broadcast message $m$ and then to put an acknowledgment $Ack_m$ into $\texttt{FMes}_q[p]$. The protocol then transmits $Ack_m$ to $p$: This generates a

[1] Even if the current computation is due to a fault.

"**F-receive**$\langle Ack_m \rangle$ **from** $q$" event at $p$ so that the application at $p$ can access to the acknowledgment.

Note that the protocol has to operate correctly despite arbitrary messages in the channels left after the faults. Note also that the messages can be lost. To counter the message loss the protocol repeatedly sends duplicate messages. To deal with the arbitrary initial messages and the duplicates, we mark each message with a flag which takes its value in {0,1,2,3,4}. Two arrays are used to manage the flag marking:

– In $\texttt{State}_p[q]$, process $p$ stores a flag value that it attaches to the messages it sends to its $q$'th neighbor.
– In $\texttt{NState}_p[q]$, $p$ stores last flag that it receives from its $q^{th}$ neighbor.

Using these two arrays, our protocol proceeds as follows. When $p$ starts a PIF-computation, it sets $\texttt{State}_p[q]$ to 0, for every process $q$. The computation terminates when $\texttt{State}_p[q] \geq 4$ for every index $q$.

During the computation, $p$ repeatedly sends $\langle \texttt{PIF},\texttt{BMes}_p,\texttt{FMes}_p[q],\texttt{State}_p[q], \texttt{NState}_p[q] \rangle$ to every process $q$ such that $\texttt{State}_p[q] < 4$. When some process $q$ receives $\langle \texttt{PIF},B,F,pState,qState \rangle$ from $p$, $q$ updates $\texttt{NState}_q[p]$ to $pState$. Then, if $pState < 4$, $q$ sends $\langle \texttt{PIF},\texttt{BMes}_q,\texttt{FMes}_q[p],\texttt{State}_q[p], \texttt{NState}_q[p] \rangle$ to $p$. Finally, $p$ increments $\texttt{State}_p[q]$ only when it receives a $\langle \texttt{PIF},B,F,qState,pState \rangle$ message from $q$ such that $pState = \texttt{State}_p[q]$ and $pState < 4$.

The trick behind the algorithm is the following. Assume that $p$ starts to broadcast the message $m$. Then, while $\texttt{State}_p[q] < 4$, $\texttt{State}_p[q]$ is incremented only when $p$ received a message $\langle \texttt{PIF},B,F,qState,pState \rangle$ from $q$ such that $pState = \texttt{State}_p[q]$. So, $\texttt{State}_p[q]$ will be equal to four only after $p$ successively receives $\langle \texttt{PIF},B,F,qState,pState \rangle$ messages from $q$ with the flag values 0,1,2, and 3. Now, initially there is at most one message in the channel from $p$ to $q$ and at most another one in the channel from $q$ to $p$. So these messages can only cause at most two incrementations of $\texttt{State}_p[q]$. Finally, the arbitrary initial value of $\texttt{NState}_q[p]$ can cause at most one incrementation of $\texttt{State}_p[q]$. Hence, since $State_p[q] = 3$, we have the guarantee that $p$ will increment $State_p[q]$ to 4 only after it receives a message sent by $q$ after $q$ receives a message sent by $p$. That is, this message is a correct acknowledgment of $m$ by $q$.

It remains to see when generating the **B-receive** and **F-receive** events:

– Any process $q$ receives at least four copies of the broadcast message from $p$. But, $q$ generates a **B-receive** event only once for each broadcast message from $p$: When $q$ switches $\texttt{NState}_q[p]$ to 3.
– After it starts, $p$ is sure to receive the correct feedback from $q$ since it receives from $q$ a $\langle \texttt{PIF},B,F,qState, pState \rangle$ message such that $pState = \texttt{State}_p[q] = 3$. As previously, to limit the number of events, $p$ generates a **F-receive** event only when it switches $\texttt{State}_p[q]$ from 3 to 4. The next copies are ignored.

## 3.2 Mutual Exclusion

A *mutual-exclusion* mechanism ensures that a special section of code, called *critical section* (CS), can be executed by at most one process at any time. We adopt the specification proposed in [8]: Any process that requests CS enters in CS in finite time (*liveness*), and if a requesting process enters in CS, it executes CS alone (*safety*). It is important to note that, starting from any configuration, a snap-stabilizing mutual exclusion protocol cannot prevent several (non-requesting) processes to execute the CS simultaneously. However, it guarantees that every requesting process executes the CS alone.

Our snap-stabilizing mutual exclusion protocol is called $\mathcal{ME}$. As previously, $\mathcal{ME}$ uses the variable Req. A process $p$ sets $\mathcal{ME}.\texttt{Req}_p$ to Wait when it requests the access to the CS. Process $p$ is then called a *requestor* and assumed to not set $\mathcal{ME}.\texttt{Req}_p$ to Wait until $\mathcal{ME}.\texttt{Req}_p = \texttt{Done}$, *i.e.*, until its current request is done.

The main idea behind the protocol is the following: We assume identities on processes and the process with the smallest identity – called the *leader* – decides using a variable called Val which process can execute the CS. Val takes its value in $\{0 \ldots n-1\}$ and we assume that any process numbers its incoming channel from 1 to $n-1$. A process $p$ is authorized to access the CS, if $p$ is the leader and $\texttt{Val}_p$ is equal to 0, or $p$ is not the leader and the Val-value of the leader designates the link incoming from $p$ to the leader.

When a process learns that it is authorized to access the CS: (1) It first ensures that no other process can execute the CS; (2) It then executes the CS if it wishes to; (3) Finally, it notifies to the leader that it has terminated Step (2) so that the leader (fairly) authorizes another process to access the CS.

To apply this scheme, $\mathcal{ME}$ is executed by phases from Phase 0 to 4 in such way that each process goes through Phase 0 infinitely often. After requesting the CS, a process $p$ can access the CS only after executing Phase 0: $p$ can access to the CS only if $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$ and $p$ switches $\mathcal{ME}.\texttt{Req}_p$ from Wait to In only in Phase 0. Hence, our protocol just ensures that after executing Phase 0, a process always executes the CS alone. We describe the five phases of our protocol below:

*Phase 0.* When a process $p$ is in Phase 0, it starts a PIF-computation to collect the identities of all processes and to evaluate which one is the leader. It also sets $\mathcal{ME}.\texttt{Req}_p$ to In if $\mathcal{ME}.\texttt{Req}_p = \texttt{Wait}$. Finally switches to Phase 1.

*Phase 1.* When a process $p$ is in Phase 1, $p$ waits the termination of the previous PIF. Then, $p$ starts a PIF of the message ASK to know if it is authorized to access the CS and switches to Phase 2. Upon receiving a message ASK from the channel $p$, any process $q$ answers YES if $\texttt{Val}_q = p$, NO otherwise. Of course, any process will only take account of the answer of the leader.

*Phase 2.* When a process $p$ is in Phase 2, it waits the termination of the PIF started in Phase 1. After the PIF terminates, $p$ knows if it is authorized to access the CS. If $p$ is authorized to access the CS, $p$ starts a PIF of the message EXIT. The goal of this message is to force all other processes to restart to Phase 0. This ensures no other process executes the CS until $p$ notifies to the leader that it releases the CS. Indeed, due to the arbitrary initial configuration, some process $q \neq p$ may believe that it is authorized to execute the CS: If $q$ never starts Phase 0. On the contrary, after restarting to 0, $q$ cannot receive any authorization from the leader until $p$ notifies to the leader that it releases the `CS`. Finally, $p$ terminates Phase 2 by switching to Phase 3.

*Phase 3.* When a process $p$ is in Phase 3, it waits the termination of the last PIF. After the PIF terminates, if $p$ is authorized to execute the `CS`, then: (1) $p$ executes the `CS` and switches $\mathcal{ME}.\texttt{Req}_p$ from `In` to `Done` if $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$, then either (2.a) $p$ is the leader and switches $\texttt{Val}_p$ from 0 to 1 or (2.b) $p$ is not the leader and starts a PIF of the message `EXITCS` to notify to the leader that it releases the `CS`. Upon receiving such a message, the leader increments its variable `Val` modulus $n + 1$ to authorize another process to access the `CS`. Finally, $p$ terminates Phase 3 by switching to Phase 4.

*Phase 4.* When a process $p$ is in Phase 4, it waits the termination of the last PIF and then switches to Phase 0.

## 4 Conclusion

In this paper, we shown that *snap-stabilization* is impossible for a wide class of specifications in message-passing systems where the channel capacity is finite yet unbounded. However, we also show that *snap-stabilization* is possible in message-passing systems if we assume a bound on the channel capacity. The proof is constructive, as we presented two snap-stabilizing protocols for message-passing systems with a bounded channel capacity.

It is worth investigating if these results could be extended to more general networks, *e.g.* with general topologies, and/or where nodes are subject to permanent *aka* crash failures. On the practical side, our results imply the possibility of implementing snap-stabilizing protocols on real networks, and actually implementing them is a future challenge.

## References

1. Delaët, S., Devismes, S., Nesterenko, M., Tixeuil, S.: Snap-stabilization in message-passing systems. Research Report 6446, INRIA (2008)
2. Bui, A., Datta, A.K., Petit, F., Villain, V.: Snap-stabilization and pif in tree networks. Distributed Computing 20(1), 3–19 (2007)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
4. Gouda, M.G., Multari, N.J.: Stabilizing communication protocols. IEEE Trans. Computers 40(4), 448–458 (1991)
5. Afek, Y., Brown, G.M.: Self-stabilization over unreliable communication media. Distributed Computing 7(1), 27–34 (1993)
6. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators revisited. Journal of Aerospace Computing, Information, and Communication (2006)
7. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction (extended abstract). In: FOCS, pp. 268–277. IEEE, Los Alamitos (1991)
8. Cournier, A., Datta, A.K., Petit, F., Villain, V.: Enabling snap-stabilization. In: ICDCS, pp. 12–19. IEEE Computer Society, Los Alamitos (2003)