Snap-Stabilization in Message-Passing Systems
Brief Announcement

Sylvie Delaët
sylvie.delaet@lri.fr
LRI-CNRS UMR 8623
Université de Paris XI

Stéphane Devismes
stephane.devismes@lri.fr
LRI-CNRS UMR 8623
Université de Paris XI

Mikhail Nesterenko
mikhail@cs.kent.edu
Computer Science Department
Kent State University

Sébastien Tixeuil
sebastien.tixeuil@lip6.fr
LIP6-CNRS UMR 7606
Université Paris VI

1 Introduction

In this announcement, we report recent results [4] (available at `http://arxiv.org/abs/0802.1123`) where we address the open problem of snap-stabilization in message-passing systems.

The concept of *snap-stabilization* [3] offers an attractive approach to transient fault tolerance. As soon as such fault ends 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, then 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* [5]. After the end of a transient fault, 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 fault.

However, all snap-stabilizing protocols presented thus far used a high-atomicity execution model: each process is able to read the states of its neighbors and update its own state in one atomic step. It was unclear if snap-stabilization is possible in more realistic finer atomicity execution models such as message-passing systems.

The contribution of this work is twofold.

(1) We prove that for non-trivial problem specifications, there exists no snap-stabilizing solution in message-passing systems with unbounded yet finite capacity channels. This negative result stands even if the processes have unbounded memory. In contrast, there is a number of self-stabilizing message-passing protocols [6, 1, 2].

(2) We show that snap-stabilization in the low level message passing model is possible if the channels have bounded capacity. We present snap-stabilizing solutions to several classic problems: propagation with feedback (PIF), identifier discovery and mutual exclusion.

2 Impossibility Results

We define *safety-distributed* problem specification. We then show that a safety-distributed specification cannot have a snap-stabilizing solution in message-passing systems even if each process is allowed to use an unbounded memory. Since most 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.

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 the mutual exclusion problem, each process has to eventually enter the critical section yet not two processes can be in the critical section concurrently.

Theorem 1 *There exists no* safety-distributed *specification that allows a snap-stabilizing solution in message-passing systems with unbounded capacity channels.*

The proof of Theorem 1 hinges on the fact that after the transient fault 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 show that snap-stabilization becomes feasible in message-passing systems if the channels are of bounded known message capacity. We present solutions to propagation of information with feedback (PIF), identifier-discovery, 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 non-trivial 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 [2]).

To illustrate the operation of our protocols, we present our message-passing snap-stabilizing PIF. The PIF specification is 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. Then, every non-initiator acknowledges to the initiator the receipt of m. The PIF-computation terminates when the initiator receives acknowledgments from every other process and decides taking these acknowledgments into account. 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.

Note that a snap-stabilizing PIF 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 use two variables:

- $\texttt{State}_p[1 \ldots n-1] \in \{0,1,2,3,4\}^{n-1}$. In $\texttt{State}_p[q]$, process p stores a flag value that it attaches to the messages it sends to its q'th neighbor.
- $\texttt{NState}_p[1 \ldots n-1] \in \{0,1,2,3,4\}^{n-1}$. In $\texttt{NState}_p[q]$, p stores last flag that it receives from its q^{th} neighbor.

In addition, we use two buffer variables:

- $\texttt{B-Mes}_p$. This buffer contains the message to broadcast.

- $\texttt{F-Mes}_p[1 \ldots n-1]$. $\texttt{F-Mes}_p[q]$ contains the acknowledgment for the broadcast message that q sends to p.

Using these variables, our protocol proceeds as follows. When p starts a PIF-computation, it puts the message m to broadcast in $\texttt{B-Mes}_p$ and sets $\texttt{State}_p[q]$ to 0, for all q such that $1 \leq q < n$. The computation terminates when $\texttt{State}_p[q] = 4$ for every index q.

During the computation, p repeatedly sends $\langle \texttt{B-Mes}_p, \texttt{F-Mes}_p[q], \texttt{State}_p[q], \texttt{NState}_p[q] \rangle$ to every q such that $\texttt{State}_p[q] \neq 4$. When some process q receives $\langle B, F, pState, qState \rangle$ from p, q updates $\texttt{NState}_q[p]$ to $pState$. Then, if $pState < 4$ (*i.e.*, if p is still waiting for an acknowledgment from q), q stores the acknowledgment for B in $\texttt{F-Mes}_q[p]$ and sends $\langle \texttt{B-Mes}_q, \texttt{F-Mes}_q[p], \texttt{State}_q[p], \texttt{NState}_q[p] \rangle$ to p. Finally, p increments $\texttt{State}_p[q]$ only when it receives a $\langle B, B, qState, pState \rangle$ message from q such that $\texttt{State}_p[q] = pState$ and $pState < 4$.

Hence, after p starts the computation, $\texttt{State}_p[q]$ equals to 4 only after p successively receives $\langle B, F, qState, pState \rangle$ messages from q with the flag values 0,1,2, and 3. Now, considering the arbitrary initial value of $\texttt{NState}_q[p]$ and the at most two arbitrary messages initially in the link $\{p,q\}$ (one in the channel from p to q and one in the channel from q to p), we are sure that after p starts, p receives a $\langle B, F, qState, pState \rangle$ from q with $pState = State_p = 3$ only if this message was sent by q after q receives a message sent by p. That is, this message is a correct acknowledgment of m by q and is the only acknowledgement from q that is delivered by p to the application that requested the PIF, hence the correct operation of our protocol.

4 Conclusion

We addressed the open problem of possibility of designing snap-stabilizing protocols for message-passing systems. We proved that it is impossible for a wide class of problems in case the channel capacity is either infinite or finite yet unbounded. On the positive side, we showed that snap-stabilization is possible if we assume a bound on the channel capacity. In this model, We presented snap stabilizing protocols for three well-known problems.

It is worth investigating if the results presented in this paper 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] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[2] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *FOCS*, pages 268–277. IEEE, 1991.

[3] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[4] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. Research Report 6446, INRIA, February 2008.

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

[6] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.