arXiv:1011.3632v1 [cs.DC] 16 Nov 2010

Stabilizing data-link over non-FIFO channels with optimal fault-resilience

Shlomi Dolev[‡] Swan Dubois[§] Maria Potop-Butucaru[§] Sébastien Tixeuil[§]

Abstract

Self-stabilizing systems have the ability to converge to a correct behavior when started in any configuration. Most of the work done so far in the self-stabilization area assumed either communication via shared memory or via FIFO channels. This paper is the first to lay the bases for the design of self-stabilizing message passing algorithms over unreliable non-FIFO channels. We propose a fault-send-deliver optimal stabilizing data-link layer that emulates a reliable FIFO communication channel over unreliable capacity bounded non-FIFO channels.

1 Introduction

Self-stabilization [5, 6, 12] is one of the most versatile techniques to sustain availability, reliability, and serviceability in modern distributed systems. After the occurrence of a catastrophic failure that placed the system components in some arbitrary global state, self-stabilization guarantees recovery to a correct behavior in finite time without external (*i.e.* human) intervention.

As self-stabilization is usually considered a hard property to satisfy, most related works used a simple communication model where processes can determine the current state of every neighbors (and update their own state accordingly) in an atomic manner (this model is referred to in the literature as the *state model* or systems with *central/distributed demon*). Asynchronous message passing is a more realistic way, compared to the state model, for the communication of processes in distributed systems. In such settings processes communicate by exchanging messages, where sending and receiving message are two separate atomic actions. Transformers for shared memory protocols to act in message passing systems, assuming the existence of FIFO channels, have been suggested, see *e.g.* [7, 6]. At the core of those transformers are the *data-link* protocols, that permit to reliably exchange information between neighboring processes in the message passing model. In addition, several self-stabilizing protocols (*i.e.* [9, 9, 2]) that are directly written in the message-passing model use an underlying data-link protocol as a building block.

Related Works. The most-studied data-link protocol, namely the alternating bit protocol (ABP), was proved to satisfy some stabilization properties [1, 8, 3]: in any execution of ABP, there exists a suffix that satisfies the specification (*i.e.* the ABP is *pseudo-stabilizing*). However, the impossibility

[‡]Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel. Email: `dolev@cs.bgu.ac.il`. The work started while this author was a visiting professor at LIP6. Research supported in part by the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Deutsche Telekom, US Air-Force and Rita Altura Trust Chair in Computer Sciences.

[§]LIP6, UMR 7606, Université Pierre et Marie Curie & INRIA, France.

to bound the amount of time before this suffix is reached makes the ABP unsuitable for most tasks. In [10, 7], Gouda and Multari and Dolev, Israeli, and Moran independently prove that for a wide class of problems (including data-link construction) guaranteeing self-stabilization when channels have unbounded initial capacity requires some kind of unboundedness in the protocol (either unbounded memory in [10], the existence of some aperiodic function [1], or access to a probabilistic variable [1]). Those approaches require to implement infinite objects with finite memory, and are thus unlikely to be actually used in real systems. Also, the expected time before reaching a stable global state depends on the initial contents of communication channels, and is thus unbounded.

Most recent works took the more realistic approach of assuming channels with bounded initial capacity. The token passing protocol in [8] can be used as a self-stabilizing ABP on bounded channels and only uses bounded memory. Howell *et al.* [11] provide another data-link protocol over bounded channels with the additional desirable property the that underlying communication channels are unreliable (*i.e.* they may loose or duplicate messages). Later, Varghese [13] presented self-stabilizing solutions for a wide class of problems (including data-link) in the same setting using only bounded memory. The FIFO ordering is crucial for the stabilization since solution relies on the fact that a sequence number that is unique in the system is eventually generated and flushes every stale message in transit. A common drawback of all aforementioned self-stabilizing data-link solutions is that they assume a FIFO order on messages in the underlying communication channels.

Another drawback of previously mentioned self-stabilizing data-link solutions is that they do not consider the quantitative impact of faults from the perspective of the upper layer protocol (*i.e.* the layer that actually uses the data-link). Indeed, starting from an arbitrary global state where channels may initially contain messages of arbitrary content, being able to bound the number of messages sent that are lost or duplicated, or the number of fake messages that are actually delivered to the destination is a very important matter. The bound on the number of faulty messages delivered by a data-link protocol is an important criteria for the data-link usability in larger application, in order to ensure the fault-resiliency of the global protocol stack. To our knowledge, only [9, 4] addresses, to some extent, this concern. A snap-stabilizing data-link (and global reset) for bounded capacity FIFO channels appears in [9]. In [4] a snap-stabilizing solution to the propagation of information with feedback (PIF) problem is presented. The solution can be seen as a data-link protocol when reduced to a 2-processes system. Snap-stabilization implies that any message that is actually sent by the sender process is eventually received by the receiver process, so the number of lost messages is 0. However, nothing can be deduced relatively to the duplicated or fake messages. Concerning the self-stabilizing protocols, only an order of magnitude on those numbers can be inferred from the stabilization time (if m messages sent or received are required to enter a legitimate global state from any arbitrary initialization, then at most m messages could be lost, duplicated, or wrongly delivered). To our knowledge, the question of fault-resilience optimality for data-link protocols has never been raised before, although it has important practical consequences.

Our contribution. Our contribution in this paper is twofold:

1. We define complexity metrics that are related to the fault-resilience of data-link protocols, and present impossibility results in the context of self-stabilization (*i.e.* the ability to recover from any arbitrary initial global state). In particular, we prove that no data-link protocol can prevent one duplication of the first message sent, and the delivery of a single fake message.

2. We present a data-link protocol that is optimal with respect to all presented fault-resilience metrics. Moreover, unlike previous self-stabilizing solutions that operate assuming the underlying communication channels preserve FIFO ordering, the channels we consider may indeed reorder messages, having some of them remain in the channel for an arbitrary long time. The strong fault-resilience property exhibited by our protocol make it particularly suitable for inclusion as a building block in more complex applications.

Paper organization. The paper is organized as follows. Section 2 proposes the network model and hypothesis and then, the data-link problem specification. Section 3 introduces two lower bounds results that justify our optimality claim. In Section 4, we propose our optimal stabilizing data-link protocol altogether with its correctness proof.

2 Model

2.1 System model

A *message-passing distributed system* consists of n *processes*, $p_0, p_1, p_2, \ldots, p_{n-1}$, connected by *communication links* through which messages are sent and received. Two processes connected through a communication link are referred in the following as neighboring processes.

As emphasized in [1] the purpose of a data-link protocol is to reliably transmit messages from one end of a communication medium (link) to the other end. Ideally, messages have to arrive without duplication or loss and in the order they have been sent. Therefore, we focus in the following the communication between two neighboring processes p_i and p_j where p_i is the sender and p_j is the receiver. The communication link between the two processes p_i and p_j is denoted in the following (p_i, p_j) and is composed of two virtual directed channels (i, j) and (j, i). The channel (i, j) is used to send messages from p_i to p_j while the channel (j, i) is used to send acknowledgments from p_j to p_i. In systems where p_j is also message sender, two additional virtual channels are used to carry the messages from p_j to p_i and acknowledgments from p_i to p_j.

We assume in the following that the capacity of each directed channel is c packets (*i.e.* low level messages). Note that in the scope of self-stabilization, where the system copes with an arbitrary starting configuration, there is no deterministic data-link simulation that uses bounded memory when the capacity of channels is unbounded [10, 8].

The channels are non-FIFO and not necessarily reliable (*i.e.* packets may not follow the FIFO order and may be lost). Additionally, their delivery time is unbounded. That is, any non lost packet is received in a finite but unbounded time. Each channel (i, j) is *weakly fair* in the sense that if the sender sends infinitely often a packet on the channel, then the receiver receives this packet an infinite number of time. Sending a packet to a channel whose capacity is exhausted (*i.e.* the channel already contains c packets) results in loosing a packet (either a packet already in the channel or the packet being sent).

As we deal with arbitrary initial corruption, a channel may initially contain up to c ghost packets (*i.e.* packets that have never been sent and contain arbitrary content).

A processor is modeled by a state machine that executes *steps*. Channels are modeled as sets (rather than queues to reflect the non-FIFO order). For example, the c-bounded channel (i, j) (used to send messages from p_i to p_j) is modeled by a c-sized set denoted by s_{ij}.

In each step, a processor changes its local state (*i.e.* the state of its local memory), and executes a single communication operation, which is either a *send* operation or a *receive* operation. The communication operation changes the state of an attached channel. In case the communication operation is a send operation from p_i to p_j then s_{ij} is a union of s_{ij} in the previous state with the sent packet. If the obtained union does not respect the bound $|s_{ij}| = c$ then an arbitrary message in the obtained union is deleted. In case the communication operation is a receive operation of a (non null) packet m (m must exist in s_{ji} of the previous state), then m is removed from s_{ji}. A receive operation by p_i from p_j may result in a null packet even when the s_{ji} is not empty, thus allowing unbounded delay for any particular packet. Packet losses are modeled by allowing spontaneous packet removals from the set.

A *configuration* of the system is the product of the local states of processes in the system and of their incident channels.

An *execution* is a sequence of configurations, $E = (C_1, C_2, \ldots)$ such that C_i, $i > 1$, is obtained from C_{i-1} when at least one process in the system executes a step.

2.2 Problem Specification

The specification of reliable communication between two neighboring processors starting from an arbitrary state shares similarities with the specification of reliable broadcast. The **(α, β, γ)-Stabilizing Data-Link communication** over c-bounded channels satisfies the following properties starting from an arbitrary configuration (with p_i and p_j being respectively the sender and the receiver):

- **α-Validity** Every message sent from p_i to p_j after the first α ones is eventually delivered to p_j.
- **β-Duplication** Every message sent from p_i to p_j after the first β ones is delivered at most once to p_j.
- **γ-Creation** At most γ ghost messages are delivered to p_j.
- **FIFO-Delivery** Messages sent from p_i to p_j are delivered in FIFO order.

3 Lower Bounds

In this section, we propose two impossibility results related to the possible values for the parameters β and γ. We prove that the lower bounds for β and γ parameters is 1. These results confirm the claim that the protocol we propose is optimal since it implements a $(0, 1, 1)$-Stabilizing data-link.

Theorem 1 *There exists no (α, β, γ)-Stabilizing Data-Link communication algorithm over c-bounded channels with $\gamma = 0$.*

Proof: Any (α, β, γ)-Stabilizing Data-Link communication algorithm over c-bounded channels must have an instruction which delivers messages to the receiver processor. As the program counter may be corrupted and channels may contains up to c ghost messages in the initial configuration, the receiver processor may execute this instruction during its first step and delivers at least one ghost message. □

Theorem 2 *There exists no* (α, β, γ)*-Stabilizing Data-Link communication algorithm over* c*-bounded channels with* $\beta = 0$*.*

Proof: Let $\mathcal{A}$ be any (α, β, γ)-Stabilizing Data-Link communication algorithm over c-bounded channels. Theorem 1 shows us that $\gamma > 0$. This implies that $\mathcal{A}$ delivers at least one ghost message m to p_j.

Assume now that the first (real) message sent by p_i to p_j and delivered to p_j by $\mathcal{A}$ is the same message m. This message has been sent by p_i only one time but has been delivered to p_j at least two different times. This message has been duplicated by $\mathcal{A}$. This implies that $\beta \geq 1$. □

In the next section, we present a protocol that is optimal with respect to α, β, and γ parameters.

4 Stabilizing Data-Link over non-FIFO Channels

The concept used in the design of the data-link protocol is to let the sender use a mechanism based on the capacity so that the sender can ensure the execution of an operation in the receiver side. More precisely, the receiver acts only upon receiving a packet from the sender. The sender may repeatedly send a particular packet, and in each time the receiver receives a packet it acknowledges the packet arrival. When the sender receives $3c + 2$ such acknowledgments, the sender is sure that the receiver received at least $2c + 2$ copies of the packet that are currently sent. Assume that whenever the receiver receives $2c+2$ copies of a packet the receiver delivers the content of the packet and resets its counting while remembering the last packet that caused the delivery. Once the sender gets the $3c + 2$ acknowledgments the sender is ready to send a different packet (an alternating bit value can ensure that identical consecutive messages are carried by different packets).

4.1 A $(0, 1, 1)$-Stabilizing Data-Link Protocol

The rationale of the protocol consists in adding safety extensions to the well known alternating bit protocol (*a.k.a.* ABP). First, the receiver can deliver a message only if $c + 1$ copies of this message have previously been received: this ensures that at least one of these is genuine (*i.e.* was actually sent by the sender), also a message is delivered only if the expected bit alternates with the one of the previously received message (similarly to the ABP). Second, the sender will expect for each message sent at least $3c+2$ acknowledgments with a matching alternating bit. As up to c acknowledgments could be ghost, this implies that $2c+2$ of these acknowledgments were actually sent by the receiver. One such acknowledgment could be sent by the received due to bad initialization, c of them could be due to c initial ghost messages in the reverse direction, and the remaining $c+1$ can only originate from genuine messages from the sender, that triggered a delivery at the receiver. In order to ensure 0-validity, the sender alternates between actual messages and synchronization messages (that is, two series of packets are sent for every user message). In the remaining, $< SYNCHRO >$ denotes a synchronization message.

Our $(0, 1, 1)$-stabilizing data-link protocol is presented as Figure 2. The protocol is composed from two functions: **Send** (which is executed on the sender side) and **Receive** (which is executed on the receiver side). When the application layer on the sender side wants to send a message m, it invokes **Send(m)** (if **Send(m)** is already in execution, the application layer waits after its termination) whereas the **Receive** function is always executed on the receiver side. When the **Receive** function has a message to deliver at the application layer on the receiver side, it executes

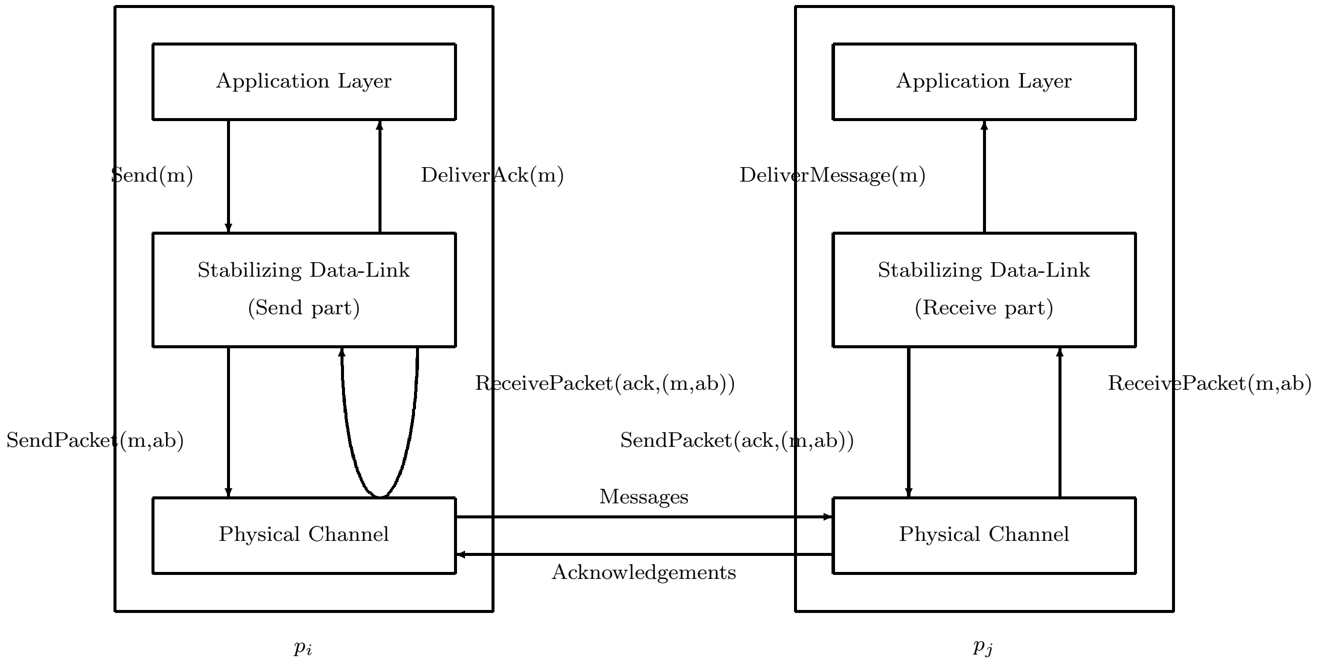

Figure 1: General organization of our system.

DeliverMessage(m**)** which transmits m at the application layer. Finally, each delivered message is acknowledged to the application layer on the sender side by **DeliverAck(**m**)**.

Functions **Send** and **Receive** must interact with the physical channel in order to exchange messages. For this, we assume that the channel provides two operations. First, it provides an operation to send a message or an acknowledgment, respectively **SendPacket(**m**,**ab**)** and **SendPacket(ack,(**m**,**ab**))** where m is the message and ab its alternating bit value. This operation puts m (or its acknowledgment) in the channel if it is possible (if this operation leads to more than c messages in the channel, one of them is arbitrarily deleted). Second, it provides an operation to receive a message or an acknowledgment, respectively **ReceivePacket(**m**,**ab**)** and **ReceivePacket(ack,(**m**,**ab**))** where m is the message and ab its alternating bit value. On the receiver side, **ReceivePacket(**m**,**ab**)** is executed when the channel has a message to deliver and when **Receive** is not in execution. It sets then m and ab to actual values of the delivered message. In other words, the reception (for the data-link protocol) on the receiver side is message-driven. On the sender side, **ReceivePacket(ack,(**m**,**ab**))** is executed by the data-link protocol. It checks whether the first waiting message in the channel (if any) matches with an acknowledgment of the parameter (m,ab). It returns **true** if this is the case, **false** otherwise. In any case, the first waiting message (if any) is deleted from the channel. The architecture of our system is summarized in Figure 1.

Queue operations:
The [] operator takes a message m and a boolean b as operands, and either enqueues $(m, ab, 0)$ (if $(m, ab, *)$ is not present in Q, then if the queue contained $c+1$ elements, the last element of the queue is dequeued) or returns a pointer to the *count* value associated to m and ab in Q. Any time a tuple value is changed in the queue, this tuple is promoted at the top of the queue, and the size of the queue does not change. The $\perp$ assignment to a queue Q denotes the fact that Q is emptied.

Send

input:
m: message to be sent
persistent variables:
ab: alternating bit value
ack: integer denoting the number of acknowledgments received for the current ab value

```
01: $ab := \neg ab$
02: $ack := 0$

03: $while\ ack < 3c+2$
04:   SendPacket $(<SYNCHRO>, ab)$;
06:   $if$ ReceivePacket $(ack, (<SYNCHRO>, ab))$
07:     $ack := ack + 1$;

08: $ab := \neg ab$
09: $ack := 0$

10: $while\ ack < 3c+2$
11:   SendPacket $(m, ab)$;
12:   $if$ ReceivePacket $(ack, (m, ab))$
13:     $ack := ack + 1$;

14: DeliverAck $(m)$
```

Receive

persistent variables:
last_delivered: boolean that states the alternating bit value of the last delivered message
Q: queue of size $c+1$ of 3-tuples $(m, ab, count)$, where m is a message, ab is an alternating bit value, and *count* is an integer denoting the number of packets (m, ab) received for the corresponding m and ab since the last **DeliverMessage** occurred.

```
01: $upon$ ReceivePacket $(m, ab)$
02:   $Q[m, ab] := min(Q[m, ab] + 1, c+1)$
03:   $if\ Q[m, ab] \geq c+1\ then$
04:     $if\ last\_delivered \neq ab\ then$
05:       $if\ m \neq <SYNCHRO>\ then$
06:         DeliverMessage $(m)$
07:       $last\_delivered := ab$
08:     $Q := \perp$
09:   SendPacket $(ack, (m, ab))$
```

Figure 2: $\mathcal{SDL}$: a $(0,1,1)$-Stabilizing Data-Link protocol

4.2 Correctness proof

In this section, let p_i and p_j be two neighboring nodes that execute $\mathcal{SDL}$, p_i being the sender and p_j the receiver. Let e be an execution starting from an arbitrary configuration.

Lemma 1 *When the system starts in a configuration where $ab \neq last\_delivered$, any message (either a $< SYNCHRO >$ message or a normal message) sent from p_i to p_j is delivered to p_j.*

Proof: Consider an initial configuration where $ab \neq last\_delivered$ and let m be a message sent by p_i with ab as alternating bit value. Assume m is never delivered. That is, p_j never executes lines 05-06 in the receiver's code. In turn, tests on lines 03 or 04 never evaluate to true simultaneously.

As $last\_delivered \neq ab$ in the initial configuration and $last\_delivered$ may change only when m is delivered (line 07), we know that the test on line 04 is always true (since m is never delivered by assumption).

This implies that $Q[m, b] \geq c + 1$ never evaluates to true (test on line 03). This implies that the sender stops sending (m, b) before the (m, b) counter reached $c + 1$ which is impossible. The reason is as follows. In order to stop sending the same message, p_i must get $3c+2$ acknowledgments with the expected content $(ack, (m, b))$. If such $3c + 2$ acknowledgments are indeed received, this implies that the receiver issued at least $2c + 2$ of those acknowledgments, and thus received $2c + 2$ packets (m, b). Consider the first such packet (m, b) received by the receiver. If there is no reset of the receiver's queue following this packet, the head of the queue now contains an entry (m, b, x) that can not be deleted until the receiver resets the entire queue. Indeed, at most c packets are initially present in the receiver's input channel, that can create at most c entries in the queue. Since the queue is of size $c + 1$, the $(m, b, *)$ tuple remains. Now, if the receiver sends $c + 1$ packets $(ack, (m, b))$, it implies that the receiver's queue for entry $(m, b, *)$ was incremented $c + 1$ times which invalidates the assumption. We can conclude that m is delivered in a finite time.

Note that after each delivery ab and $last\_delivered$ have the same value with the execution of the line 06 of the algorithm. Hence the next invocation of the send primitive by p_i will make the values ab and $last\_delivered$ different. In this way, we can repeat the reasoning and obtain the lemma. □

Lemma 2 *When the system starts in a configuration where $ab = last\_delivered$, only the first message (either a $< SYNCHRO >$ message or a normal message) sent from p_i to p_j is not delivered to p_j.*

Proof: Consider an initial configuration where $ab = last\_delivered$. Let m be a message sent by p_i with the value ab for the alternated bit champ.

Since the test in the line 04 of the receiver code evaluates to false, the deliver of m is not executed. However, since p_i keeps sending messages and p_j acknowledges these messages the send procedure returns and the next invocation of this procedure will execute line 01 of sender code.

It follows that the system reach in a finite time a configuration where $ab \neq last\_delivered$. Then, Lemma 1 implies that every message sent by p_i after the first message is delivered to p_j. □

Lemma 3 *$\mathcal{SDL}$ satisfies the 0-validity property.*

Proof: Assume that p_i starts $Send(m)$ in an arbitrary configuration. Then, p_i sends first a $< SYNCHRO >$ message (which may be lost if $ab = last\_delivered$ in the start configuration by Lemma 2).

Then, we know by Lemma 2 that we have $ab = last\_delivered$ when p_i starts to send m (since it has executed line 08 of the sender's code). By Lemma 1, we are ensured that m is eventually delivered to p_j and we obtain the result. □

Lemma 4 *$\mathcal{SDL}$ satisfies the 1-duplication property.*

Proof: Assume that at least one genuine message have been already delivered by $\mathcal{SDL}$ to p_j. Assume that there exists after that a message (m, b) sent by p_i to p_j which is delivered twice.

This implies that the line 06 in the receiver's code is executed twice which is false and the reason is the following. After the first delivery of m the receiver empties the queue and makes *last_delivered* $= ab$ (see proof of Lemma 2). Even if the p_i keeps invoking **SendPacket** (m, ab) until it receives the $3c+2$ acknowledgments, none of these messages will be delivered since for each of them the test in line 04 returns false.

This implies that only the first message may be duplicate. □

Lemma 5 *$\mathcal{SDL}$ satisfies the 1-creation property.*

Proof: Initially the channel (i, j) may contain at most c ghosts messages. In the worst case, the receiver's queue also contains an entry for each of the ghost with the counters initialized to c.

Let (g, b) be the first ghost message received by p_j with alternated bit champ set to b and assume $b \neq$ *last_delivered*. Then p_j delivers g and empties the queue. The main consequence is that none of the $c-1$ remaining ghost messages will be delivered. □

Lemma 6 *$\mathcal{SDL}$ satisfies the FIFO-delivery property.*

Proof: Consider the second message (m', b') sent by p_i to p_j. Following the first reset, each entry in the receiver's queue now never goes beyond c except for the entry related to (m', b') and possibly the entry related to (m, b) (the first message sent by p_i to p_j). The reason is that there are at most c dangling messages in the input channel of the receiver, and that the previous reset of the queue has nullified all values. Now, since the sender receives $3c + 2$ $(ack, (m', b'))$ packets from the receiver, in turn the receiver must receive $2c + 2$ (m', b') packets. In turn, this implies that the receiver executes at least two resets, exactly one of them being related to the delivery of m'. The first reset could be related to (m, b) (but then the *last_delivered* bit guarantees that the message m is not delivered), or to (m', b') (and then the message m' is delivered and no reset can then be related to (m, b)). □

Theorem 3 *$\mathcal{SDL}$ satisfies the $(0, 1, 1)$-Stabilizing Data-Link Communication specification.*

Proof: This is a direct consequence of Lemmas 3, 4, 5 and 6. □

References

[1] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[2] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Brief announcement: Sharing memory in a self-stabilizing manner. In *Proceedings of DISC 2010*, Lecture Notes in Computer Science, Boston, Massachusetts, USA, September 2010. Springer Berlin / Heidelberg.

[3] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[4] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. *Journal of Parallel and Distributed Computing (JPDC)*, 2010.

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

[6] S. Dolev. *Self-stabilization.* MIT Press, March 2000.

[7] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[8] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.

[9] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithms. In Alexander A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 2006.

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

[11] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. In Anish Arora, editor, *WSS*, pages 62–69. IEEE Computer Society, 1999.

[12] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.

[13] George Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.