Stabilizing Sharing Memory Robustly in Message Passing
(Extended Abstract)

Noga Alon∗ Hagit Attiya§ Shlomi Dolev† Swan Dubios‡
Maria Gradinariu‡ Sébastien Tixeuil‡

April 17, 2010

Abstract

An implementation of self-stabilizing multiple-reader single writer atomic register in message passing system is presented. The implementation is based on a new combinatorial structure for bounded time stamps for which there exists a time-stamp greater than any arbitrary set (of a particular size) of time-stamps.

The implementation employs a new self-stabilizing FIFO data link algorithm over bounded non-FIFO link.

1 Introduction

Distributed computing theory is proven to be extremely important in the daily practice of current systems [2, 6, 3, 11, 12]. The important terms in todays distributed systems include availability, reliability, serviceability, dynamic and fault-tolerance. Still, only a very few system are designed to withstand transient and permanent faults. The ability to automatically recover following transient faults, namely to be self-stabilizing, is an essential property that can be integrated into the design and implementation of systems.

The goal of this work is to present a core self-stabilizing building block for message passing systems. Namely, we present a self-stabilizing implementation of a single writer multi-reader atomic register by passing messages. The implementation uses new bounded time-stamping that can be started in an arbitrary state. We also present a FIFO data-link algorithm over bounded capacity non-FIFO link.

Giving our implementation one can realize a self-stabilizing version of replicated state machines [10] using [7].

Related work and overview. [1] presented an implementation of atomic register by the use of message passing. Two algorithms that use unbounded and bounded time-stamps were presented in [1]. An unbounded solution may fit settings in which the system is initialized in a consistent configuration (say all time-stamps are zero) and consistency is maintained by following the steps of the algorithm; in such a case if one uses a register of, say, 64 bits the number of writes before all time-stamps are used is practically infinite. Certainly longer than the life time of the system. However, in the scope of self-stabilization where (initial) consistency is not an assumption but one of the goals (the converges goal) of the steps of the self-stabilizing algorithm registers can

∗Sackler School of Mathematics and Blavatnik School of Computer Science, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv, 69978, Israel. Email: nogaa@tau.ac.il. Research supported in part by an ERC advanced grant, by a USA-Israeli BSF grant, by the Israel Science Foundation.
†Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel. Email: dolev@cs.bgu.ac.il. Research supported in part by Rita Altura trust chair in computer science, and the US Air-Force.
‡LIP6, Universite Pierre et Marie Curie, Paris 6....
§CS Technion
be started with the maximal time-stamp value at once. There is no need to specify the reasons (and in fact typically it is hard to list all unexpected scenarios) leading to the inconsistency, still just to motivate the goal, assume that the (probabilistic mechanism of) error correcting/detecting codes used in transmitting messages resulted in accepting a corrupted message, message with a big enough time-stamp.

The bounded time-stamp solution presented in [1] does not fit either. The solution is based on the bounded time stamp combinatorial solutions in [9, 8]. These constructions do not fit a self-stabilizing solution, since they assume an initial set of time stamps and a way to introduce a time stamp bigger than the existing set. However, if the set of time stamp is illegal, say there are time stamps in a cycle then there is no definition for the biggest among them and there is no procedure to introduce a new bigger time stamp. Here we introduce a constructive procedure to define a time stamp bigger than any set (of a give maximal size) of time stamps.

The use of a majority quorum is another major technique in the solutions. The most recent value is written to the (virtual) register, with the biggest time stamp. To cope with the asynchronous nature of the system and the atomicity requirement readers write to the majority quorum, the value they are going to return with the greatest time stamp they found. Thus, as long as no newer write exists, later reads will return an identical value.

In the self-stabilizing settings any label coupled with any value (and in particular the label that the writer may introduce next) may be in the process of (very slow) transmission of one reader to another. This transmission attempt can be contributed to the inconsistent first configuration, where any state of any process is possible. When the transmission is completed the reader apodt the (very old) value as new. Thus, the stabilizing solution may return wrong information for a while, before respecting the atomicity requirements for (practically) infinite execution. We prove that after a bounded number of writes the results of reads and writes can be totally casual ordered.

The original design of [1] coped non-fifo links. We introduce a new stabilizing data link algorithm that works over bounded capacity non-fifo links. The data-link assists in fast convergence reducing the effective number of labels in the system.

Paper organization. Section 2 describes the combinatorial structure used for implementing a bounded time-stamp. An abstract game that demonstrates the way the writer eventually manage to produce a time-stamp greater than any time-stamp in the system is described in Section 3. In Section 4 we turn to the design of an important building block, FIFO queue over bounded non-FIFO networks. Section 5 presents the implementation of the self-stabilizing multiple reader single writer atomic register, using (majority) Quorum operations.

2 Uninitializezable Bounded Labels

Let $k$ be some integer greater than 1. Let $n = k^2 + 1$, let $X = \{1, 2, \ldots, n\}$ and let $L$ (the set of labels) be the set of all ordered pairs $(s, A)$ where $s$ is an element (called in the sequel sting) of $X$ and $A$ is a subset (called in the sequel Antistings) of $X$, $|A| = k$. Therefore $|L| = \binom{n}{k} \cdot n = k^{(1+\omega(1))k}$.

The comparison operator $\prec_{b}$ for bounded labels denotes:

$$(s_i, A_i) \prec_{b} (s_j, A_j) \equiv (s_j \in A_i) \land (s_i \not\in A_j)$$

Note that this operator is antisymmetric by definition, yet may not be defined for all pairs of $(s_i, A_i)$ and $(s_j, A_j)$ in $L$ (e.g. $(s_j \in A_i)$ and $(s_i \in A_j)$).

We define now a function to compute, given a subset $S$ of at most $k$ labels of $L$, a new label which is greater (with respect to $\prec_{b}$) than every label of $S$. This function, called Next$_b$ (see Figure 1) is implemented using the following scheme. Given a subset of $k$ labels $(s_1, A_1)$, $(s_2, A_2)$, $\ldots$, $(s_k, A_k)$, we construct a label $(s, A)$ which satisfy:

- $s_i$ is any element of $X$ that is not in the union $A_1 \cup A_2 \cup \cdots \cup A_k$ (as the size of each $A_i$ is $k$, the size of the union is at most $k^2$, and since $X$ is of size $k^2 + 1$ there is always such an $s_i$).
- $A$ is any subset of size $k$ of $X$ containing all values $(s_1, s_2, \ldots, s_k)$ (if they are not pairwise distinct just add to the set of all of them arbitrary elements of $X$ to get a set of size exactly $k$).
Theorem 1 Given a subset $S$ of $k$ labels of $L$, $(s_i, A_i) = \text{Next}_b(S)$ satisfy:

$$\forall (s_j, A_j) \in S, (s_j, A_j) \prec_b (s_i, A_i)$$

**Proof:** Let $S$ be a subset of $k$ labels of $L$ and $(s_i, A_i)$ be $\text{Next}_b(S)$. Let $(s_j, A_j)$ be an element of $S$. Then, by construction, we have: $s_j \in A_i$ and $s_i \notin A_j$. Thus, by definition of $\prec_b$, we have the result.

Note also that the process of computing $A_i$ and $s_i$ given the $k$ labels ($\text{stings}, \text{Antistings}_s$) is very simple (can be done in $O(k^2)$ time, that is, in time linear in the total length of the given labels). As the number of labels $|L|$ is $k^{1+o(1)}k$ it follows that $k$ is $\frac{1+o(1)}{\log \log |L|}$.

**r-enumerable bounded labels.** We now introduce extended version of the labels described in section 2, that are called $r$-enumerable bounded labels. Each label is now a 2-tuple $(l, i)$ where $l$ is a bounded label and $i$ is an integer (round) ranging from 0 to some constant $r$ ($r \geq 1$). We use notation $\tilde{L}$ to denote the set of bounded labels contained in $L$, a set of extended bounded labels. The comparison operator $\prec_e$ for $r$-enumerable bounded labels denotes:

$$(x, i) \prec_e (y, j) \equiv x \prec_b y \vee (x = y \wedge i < j)$$

The **Next** protocol is described in Figure 1. The use of the extended version allows us to change the original label (epoch) seldomly, and therefore reduce the memory used in the data structures and let the writer a practically infinite time period to collect the set of basic labels in the system; this set can be used for coordinating the next change of the basic label.

### 3 Single Finder Label Guessing Game — Single Writer Multiple Reader Register

In this section we present the core challenges for the design of the self-stabilizing multiple-reader single writer register. The writer needs to introduce a new label without knowing the exact set of labels that are currently stored in the memory of the readers. Note that the number of labels stored in the system is bounded and therefore when new labels are stored other labels are eliminated. The writer may have a totally wrong view (a corrupted view, due to transient faults) on the existing set of labels in the system. Thus, when the writer introduces a new label the label may not be greater than existing labels. In such a case the value of the introduced label will be coupled with an evidence on the existence of a non-smaller label, and the writer will need to eventually re-choose a label. The writer uses two queues in a way that ensures an eventual choice of a label greater than any label in the system.

**Game.** Consider labels with a single $\text{Sting}$ value and $2m$ $\text{Antistings}$ values all distinct from a range of 0 to $(2m + 1)^2$.

Consider a two players game the $\text{finder}$ that would like to introduce a label greater than the set of (at most) $m$ labels unknown to the $\text{finder}$ that the $\text{hider}$ has. The game starts with an arbitrary set of labels held by the $\text{hider}$. The $\text{finder}$ may introduce a label $L$ and if a label $L'$ that
is not smaller than \( L \) exists in the set of the hider, then the hider exposes one such label to the finder, otherwise the finder wins. Every time the hider exposes an evident the hider may choose to include \( L \) in the set, in case the hider decides to include \( L \) and the size of the set is greater than \( m \), then the hider must remove one label from the set. This last choice is not exposed to the finder.

We now propose a winning strategy for the finder. Assume an arbitrary initialization of two FIFO queues \(\text{StingQueue} \) of length \( m + m + 1 = 2m + 1 \) and \(\text{AntiStingQueue} \) of length \((2m + 1)2m\) the values in each queue are all distinct. An enqueue of an existing value is interpreted as a removal of the value from (any position in) the queue, if the value exists in the queue, and then enqueue; an enqueue that is about to cause an exceed of the upper bound of the queue is interpreted as: dequeue and then enqueue. The first \( m \) of the (above \( m + m + 1 \)) elements of \(\text{StingQueue} \) are due to the need to record the \( m \) recent proposals that turned to be proposals of an existing Sting, and the second \( m \) are due to the need to introduce a new Sting whenever the chosen sting exists in an unknown label, as a member of an (not yet queued) Antisting; we show that the additional one of these \( m + m + 1 \) must be a successful choice. The number of labels with at least one unknown element of Antistings is \( m \), each time such an Antisting, that has an element equals to the current Sting is exposed, the Antistings of the unknown label are enqueued to AntiStingQueue. This can happen at most \( m \) times before the finder is aware of all the existing Antistings. Thus, the finder has a winning strategy, finding a label with a Sting not in both queues. Note that here are enough values for the choice of a new sting since the domain of a sting is \((2m + 1)^2 + 1\) and there are at most \((2m + 1 + (2m + 1)2m\) distinct values in the queues.

**Observations.** In our implementation the finder/writer may decide to introduce new labels even when the hider does not introduce an evidence. We use the \( r \)-enumerable bounded labels, and consider a label \((l, i)\) to be an evidence for label \((l', j)\) iff \( l' \not\approx_l l \). Using large enough \( r \) (e.g., a 64 bits round value) for the \( r \)-enumerable bounded label we are sure that either there is a practically infinite execution in which the finder/writer introduces new labels with no epoch change, and therefore with a growing round counter, and well defined label ordering, or a new epoch is frequently introduced due to the exposure of hidden unknown epoch in the system. The last case yields the winning strategy above. Note that if the game continues, after the finder is aware of (a superset including) all existing Stings and Antistings, and introduces a greater label, there is a practically infinite execution before a new epoch is introduced. In the scope of implementing the multiple reader single writer register, following the first write of a label greater than any other label in the system, with a round label 0, to a majority quorum, any read in a practically infinite execution, will return the last label that has been written to a quorum. In particular, in case a reader finds a label introduced by the writer that is larger than all other labels but not yet completely written to a majority quorum, the reader assists in completing the write to a majority quorum before returning the read value.

In the scope of the atomic register implementation the writer/finder may stop operating while the set of labels does not include a label greater than the rest, in such a case read operations may be repeatedly aborted until the writer writes new labels. Moreover, a slow reader may store a label unknown to the rest (in particular to the writer) and introduce the label to the rest following any number of steps taken by others. Thus, on one hand postponing the convergence of the system to the stage in which the writer is aware of a superset of the existing labels; but at the same time, lets the system operate correctly, implementing read and write operations, until the label unknown to the rest is introduced.

The algorithm can be started in a safe configuration just as defined in [1], the choice of labels with the extra restriction implied by the queues will not falsify the requirement for the label choice in [1] and therefore we have the safer than safe property [4]. Following a transient fault, and several writes of the writer, the system reaches a practically infinite time to establish the viable label property for each label in the system and thus to operate correctly forever.

**Coping with non-fifo links.** In the sequel we present a stabilizing data link algorithm that is based on the assumption of bounded non-fifo link capacity. The data-link reduces the number of labels that can be stored in the system by a factor proportional to the link capacity. Thus,
reducing the convergences time by a similar factor.

4 Stabilizing data link over bounded non-FIFO network

Problem and specification. We assume in the following that each edge \((i, j)\) is composed of two virtual directed edges \((i, j)\) and \((j, i)\) and the capacity of each link is \(c\). The links are non-FIFO and the delivery time is unbounded. Initially, the links may contain ghost packets (packets that have never been sent). The links are weakly fair in the sense that if both the sender and the received are not crashed, if the sender sends an infinite number of packets, then the receiver receives an infinite number of packets. Sending a packet to a link whose capacity is exhausted results in losing a packet (either a packet in the link or the packet being sent). Therefore the specification of reliable communication between two neighboring nodes in self-stabilizing settings becomes similar to the specification of reliable broadcast. In the following we introduce the \((\alpha, \beta, \gamma)\)-Stabilizing Data Link communication over \(c\)-bounded channels expressed formally as follows: if \(p_i\) and \(p_j\) are correct processes, then

- **\(\alpha\)-Validity** be not delivered to \(p_j\). Every message (sent from \(p_i\) to \(p_j\)) but the first \(\alpha\) is eventually delivered to \(p_j\).
- **\(\beta\)-Duplication** Every message (sent from \(p_i\) to \(p_j\)) but the first \(\beta\) is delivered at most once to \(p_j\).
- **\(\gamma\)-Creation** At most \(\gamma\) ghost messages are delivered to \(p_j\).
- **Fifo-Delivery** Messages sent from \(p_i\) to \(p_j\) are delivered following the FIFO order.

Note that it is impossible to achieve \(\gamma = 0\) in a stabilizing setting since the program counter may be corrupted in the initial configuration. Indeed, the first step of the receiver may be to deliver a message whatever the algorithm is. Then, this message is a ghost message. Assume now that the first message sent by \(p_i\) is identical to this ghost message and that this message is eventually deliver. Then, \(p_j\) receive two copies of this message. Consequently, it appears that it is impossible to achieve \(\beta = 0\).

\((1, 1, 1)\)-Stabilizing Data Link communication. In the following we propose a communication between two neighboring nodes that verifies the \((1, 1, 1)\)-Stabilizing Data Link Communication. Note that in self-stabilizing setting there is a tradeoff between the degree of verification of the validity property and the verification of the no duplication property. Our implementation prefers the no duplication at the price of violating the validity for a short fragment of execution.

Proof of correctness.

Theorem 2) Algorithm 2 verifies the specification \((1, 1, 1)\)-Stabilizing Data Link Communication.

Proof: Let \(p_i\) and \(p_j\) be two neighboring nodes that execute Algorithm 2, \(p_i\) as sender and \(p_j\) as receiver. Let \(e\) be an execution starting from an arbitrary configuration. In the following we prove \(e\) verifies the four properties of the \((1,1,1)\)-Stabilizing data-link communication.

1-Validity proof. In the following we show that every message sent from \(p_i\) to \(p_j\) is delivered by \(p_j\) when started in a configuration where \(ab\) and \(last\_delivered\) have not the same value. Then we show that starting in a configuration where \(ab = last\_delivered\) only the first message from \(p_i\) to \(p_j\) is not delivered.

Consider a configuration where \(ab = last\_delivered\) and let \(m\) be a message sent by \(p_i\) with the value \(ab\) for the alternated bit champ. Assume \(m\) is never delivered. That is, \(p_j\) never executes line 5 in the receiver’s code. That is, the tests in line 3 or line 4 never evaluates to true for \(m\).
Assume first \(Q[m, b] \geq c + 1\) never evaluates to true. 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.

Assume last\(_{delivered} \neq ab\) never evaluates true for \((m, b)\). Impossible from the hypothesis on the starting configuration. Note that after each deliver \(ab\) and last\(_{delivered}\) have the same value after the execution of the line 6 of the algorithm and hence the next invocation of the send primitive by \(p_i\) will make the values \(ab\) and last\(_{delivered}\) different.

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 4 of the receiver code evaluates to false, the deliver of \(m\) is not executed. However, since \(p_i\) keeps sending messages and \(p_i\) acknowledges these messages the send procedure returns and the next invocation of this procedure will execute line 1. It follows that \(ab \neq last\(_{delivered}\) and every message sent after the first message is delivered.

1-duplication proof. Assume that a message have been already sent by \(p_i\) 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 5 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\(_{delivered} = ab\) (due to the hypothesis and the discussion on the 1-validity). Even if the send keeps invoking SendPacket \((m, ab)\) until it receives the \(3c + 2\) acknowledgments, none of these messages will not be received since for each of them the test in line 4 returns false.

1-creation proof. Initially the link \((p_i, p_j)\) may contain at most \(c\) ghosts. 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 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 will be delivered.

Fifo-delivery 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 \((m, b))\). \qEDA

\((0, 1, 1)\)-Stabilizing Data Link communication.

---

**Queue operations:**
The \([\cdot] \) 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.

---

### Theorem 3
Algorithm 3 verifies the specification \((0, 1, 1)\)-Stabilizing Data Link Communication.

**Proof:** Let $p_i$ and $p_j$ be two neighboring nodes that execute Algorithm 3, $p_i$ as sender and $p_j$ as receiver. Let $e$ be an execution starting from an arbitrary configuration. In the following we prove $e$ verifies the four properties of the \((0, 1, 1)\)-Stabilizing data-link communication.

0-Validity proof. In the following we show that every message (either \(<\text{SYNCHRO}>\) message or normal message) sent from $p_i$ to $p_j$ is delivered by $p_j$ when started in a configuration where $ab$ and last\_delivered have not the same value. Then we show that starting in a configuration where $ab = \text{last\_delivered}$ only the first message (either \(<\text{SYNCHRO}>\) message or normal message) from $p_i$ to $p_j$ is not delivered.

Consider a configuration where $ab = \text{last\_delivered}$ and let $m$ be a message sent by $p_i$ with the value $ab$ for the alternated bit champ. Assume $m$ is never delivered. That is, $p_j$ never executes line 5 in the receiver’s code. That is, the tests in line 3 or line 4 never evaluates to true for $m$.

Assume first $Q[m, b] \geq c + 1$ never evaluates to true. 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

---

<table>
<thead>
<tr>
<th>Send</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input:</strong>&lt;br&gt;$m$: message to be sent&lt;br&gt;<strong>persistent variables:</strong>&lt;br&gt;$ab$: alternating bit value&lt;br&gt;$ack$: integer denoting the number of acknowledgments received for the current ab value</td>
<td><strong>persistent variables:</strong>&lt;br&gt;$last_delivered$: boolean that states the alternating bit value of the last delivered message&lt;br&gt;$Q$: 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.</td>
</tr>
<tr>
<td>1: $ab := \lnot ab$</td>
<td>1: upon ReceivePacket ((m, ab))</td>
</tr>
<tr>
<td>2: $ack := 0$</td>
<td>2: $Q[m, ab] := \min(Q[m, ab] + 1, c + 1)$</td>
</tr>
<tr>
<td>3: while $ack &lt; 3c + 2$</td>
<td>3: if $Q[m, ab] \geq c + 1$ then</td>
</tr>
<tr>
<td>4: SendPacket (&lt;\text{SYNCHRO}&gt;, ab);</td>
<td>4: if last_delivered (\neq ab) then</td>
</tr>
<tr>
<td>6: if ReceivePacket ((ack; (&lt;\text{SYNCHRO}&gt;, ab)))</td>
<td>5: if $m \neq &lt;\text{SYNCHRO}&gt;$ then</td>
</tr>
<tr>
<td>7: $ack := ack + 1$</td>
<td>6: DeliverMessage ((m))</td>
</tr>
<tr>
<td>8: $ab := \lnot ab$</td>
<td>7: last_delivered := $ab$</td>
</tr>
<tr>
<td>9: $ack := 0$</td>
<td>8: $Q := \perp$</td>
</tr>
<tr>
<td>10: while $ack &lt; 3c + 2$</td>
<td>9: SendPacket ((ack; (m, ab)))</td>
</tr>
<tr>
<td>11: SendPacket ((m, ab));</td>
<td></td>
</tr>
<tr>
<td>12: if ReceivePacket ((ack; (m, ab)))</td>
<td></td>
</tr>
<tr>
<td>13: $ack := ack + 1$</td>
<td></td>
</tr>
<tr>
<td>14: DeliverAck $m$</td>
<td></td>
</tr>
</tbody>
</table>

---

Figure 3: \((0, 1, 1)\)-Stabilizing Data Link communication
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 $(m,b,x)$ tuple remains. Now, if the receiver sends $c + 1$ packets $(ack,(m,b))$, it implies that the receiver’s queue for entry $(m,b,\ast)$ was incremented $c + 1$ times which invalidates the assumption.

Assume $last\_delivered \neq ab$ never evaluates true for $(m,b)$. Impossible from the hypothesis on the starting configuration. Note that after each deliver $ab$ and $last\_delivered$ have the same value after the execution of the line 6 of the algorithm and hence the next invocation of the send primitive by $p_i$ will make the values $ab$ and $last\_delivered$ different.

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 4 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 1. It follows that $ab \neq last\_delivered$ and every message sent after the first message is delivered.

Now 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). Then, we know by the above discussion that we have $ab = last\_delivered$. When $p_i$ starts to send $m$, it has executed line 8. Hence, we have $ab = last\_delivered$. By the above discussion, we are ensured that $m$ is eventually delivered to $p_j$ and we obtain the result.

**1-duplication proof.** Assume that a message have been already sent by $p_i$ 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 5 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 $last\_delivered = ab$ (due to the hypothesis and the discussion on the 0-validity). Even if the send keeps invoking $SendPacket (m,ab)$ until it receives the $3c + 2$ acknowledgments, none of these messages will not be received since for each of them the test in line 4 returns false.

**1-creation proof.** Initially the link $(p_i,p_j)$ may contain at most $c$ ghosts. 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 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 will be delivered.

**Fifo-delivery 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 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 $(m,b)$).

4.1 Simultaneous implementation of two anti-directed data-links.

To have two data links for a particular pair of nodes, $i$, $j$, we may logically separate each packet into two portions. The first portion of the packet transmitted from node $i$ to node $j$ serves the data link in which $i$ is the sender and $j$ is the receiver, and the second portion of the packet serves the data link in which $i$ is the receiver and $j$ is the sender. In fact it is possible to define
independent tracks for each anti-directed data-link, such that each track is using a designated field in the appropriate portion of the message.

5 Tracking the last write

In this section we list the modifications needed to ensure that the algorithms presented in [1] become practically stabilizing. The modifications needed are:

- Labels are replaced by a composite label, a tuple consisting of: (i) one \(r\)-extended bounded label and \(\perp\) (non canaled composite label) or (ii) two \(r\)-extended bounded labels (the second \(r\)-extended bounded label is the canceling label)

- The writer (just as the finder in the game) maintains a Stings and an AntiStings queues to support the choice of a label larger than all labels in the system, where \(m\) is twice (to reflect composite labels) the maximal number of labels maintained in the data structures used in [1]

- Whenever the writer writes a new value, the second label of the composite label is chosen to be \(\perp\)

- Whenever a reader finds a non canaled composite label, for which there exists a non smaller extended label in its data structure, the reader replaces the \(\perp\) with the non smaller extended label and stores the new obtained composite label (instead of the read composite label) in the appropriate data structures; thus propagating the fact that the label was canceled to other reader and the writer.

The proof follows the correctness proof for the game.

Acknowledgments. We thank Ronen Kat and Eli Gafni for helpful discussions.

References


