Information Processing Letters 111 (2011) 262–268

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Communication-efficient failure detection and consensus in omission environments ☆

Iratxe Soraluze [a,*], Roberto Cortiñas [a], Alberto Lafuente [a], Mikel Larrea [a], Felix Freiling [b,1]

[a] University of the Basque Country, UPV/EHU, Spain
[b] Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

ARTICLE INFO

Article history:
Received 10 June 2010
Received in revised form 29 October 2010
Accepted 1 December 2010
Available online 7 December 2010
Communicated by A.A. Bertossi

Keywords:
Distributed computing
Fault tolerance
Consensus
Failure detector
General omission model
Communication efficiency

ABSTRACT

Failure detectors have been shown to be a very useful mechanism to solve the consensus problem in the crash failure model, for which a number of communication-efficient algorithms have been proposed. In this paper we deal with the definition, implementation and use of communication-efficient failure detectors in the general omission failure model, where processes can fail by crashing and by omitting messages when sending and/or receiving. We first define a new failure detector class for this model in terms of completeness and accuracy properties. Then we propose an algorithm that implements a failure detector of the proposed class in a communication-efficient way, in the sense that only a linear number of links are used to send messages forever. We also explain how the well-known consensus algorithm of Chandra and Toueg can be adapted in order to use the proposed failure detector.

1. Introduction

Consensus is one of the fundamental problems in fault-tolerant distributed computing [1]. However, it was proven in [2] that consensus cannot be solved deterministically in an asynchronous system where at least one process may fail by crashing. In order to circumvent this impossibility result, Chandra and Toueg proposed in [3] the concept of *unreliable failure detector* as a modular device attached to each process which provides (maybe erroneous) information about process failures. One of its main advantages is that it abstracts the notion of time, so that protocols built on top of it, e.g., consensus, can be designed as if they executed in an asynchronous system. Although failure detectors have been widely studied in systems where processes may only crash, there are few works about failure detectors in the general omission model, where, in addition to crashing, processes may also fail by omitting messages when sending and/or receiving [4]. The general omission model covers situations in which buffers overflow, processes have restricted capacity for resending, or even malicious behavior. Indeed, as it is shown in [5], the malicious behavior of processes in the Byzantine failure model can be reduced to a more benign model of omission failures by using tamper proof security modules, e.g., smartcards. A generalization of this idea can be found in [6].

Failure detection in omission environments was first addressed in [7], and more recently in [8–10], where several consensus algorithms based on failure detectors have been proposed. The failure detector algorithms of [8,9] rely on piggybacking to cope with transient omissions. The system model proposed in [10] allows processes to communicate indirectly so that more processes can participate actively in the consensus protocol. All these previously

☆ Research partially supported by the Spanish Research Council (MCeI), under grant TIN2010-17170, and the Basque Government, under grant IT395-10.

* Corresponding author.
E-mail addresses: iratxe.soraluze@ehu.es (I. Soraluze), roberto.cortinas@ehu.es (R. Cortiñas), alberto.lafuente@ehu.es (A. Lafuente), mikel.larrea@ehu.es (M. Larrea), freiling@uni-mannheim.de (F. Freiling).

[1] Work by Felix Freiling was performed while being affiliated to University of Mannheim, Germany.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2010.12.008

proposed algorithms have a permanent all-to-all communication pattern, which involves a high communication cost. The reduction of the communication cost has been widely studied in crash environments. In this regard, *communication efficiency* was introduced in [11] as using at most n links to send messages forever (being n the number of processes in the system).

In this work, we propose a new failure detector class for the general omission model together with a communicaion-efficient implementation. The properties of the failure detector need to be redefined in the general omission model due to the following issue related to failure detection in this model. Roughly speaking, if there is a message omission in the communication between two processes, it is impossible to reliably blame the faulty process since the omission could be due to either a send-omission of the sender or a receive-omission of the receiver and both cases are indistinguishable. For this reason, instead of trying to distinguish among correct and faulty processes, we follow a similar approach as the one in [8–10], looking for processes that are *correct enough* in the sense that they are able to (1) compute (i.e., they do not crash), and (2) communicate with a majority of processes in order to reach consensus [3]. The main difference is that in this paper we present a communication-efficient algorithm based on a bidirectional communication pattern between pairs of processes. In order to achieve communication efficiency, we propose a mechanism to pause the communication in some links as long as processes keep their communication ability to solve consensus. This communication ability is represented by the connectivity degree of a process, and is measured by the number of processes a process is able to communicate with without omissions.

The proposed communication-efficient failure detector can be used by the well-known consensus algorithm of Chandra and Toueg [3] with some adaptations. However, in the general omission model the implementation of the failure detector cannot be independent of the messages sent by the algorithm using it [12]. Indeed, in this model incorrect processes could arbitrarily stop sending consensus messages while they continue sending failure detector messages. To cope with such a behavior, we propose to insert consensus messages into failure detector messages, as it is done in [10]. A different approach, followed in [13], consists in abandoning the modular design provided by failure detectors and in using instead timers directly in the consensus algorithm.

1.1. Contributions

In this paper we first give a new definition of the completeness and accuracy properties of a failure detector class for the general omission model, based on process connectivity. We then propose an algorithm implementing the new failure detector class, and show that it is communication-efficient in the sense that at most $n-1$ bidirectional links are used to send messages forever. We also describe how the classical consensus algorithm of Chandra and Toueg for the crash model can be adapted to work with the proposed failure detector.

2. System model

We model a distributed system as a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$[2] where every pair of processes is connected by a bidirectional communication link. Concerning timing assumptions, we consider a partially synchronous model in which there are bounds on relative process speed and message transmission times [14]. Moreover, these bounds are not known and they hold only after some unknown but finite time (called GST for Global Stabilization Time). The communication links supporting this model are also called eventually timely links [15]. However, we consider that communication links are reliable, i.e., every message put into a link is eventually received at the destination (although potentially omitted by the receiving process). We assume that every process has a local clock that can measure real-time intervals, although clocks are not synchronized.

We consider the general omission model, where processes can fail either by permanently crashing or by omitting messages. Omission failures can occur either while sending or while receiving messages, and can be transient, i.e., a process may temporarily omit messages and later on reliably deliver messages again. A process is *correct* if it does neither crash nor omit any message. Informally, we say that a process is *correct enough*, later on defined as *well-connected*, if it satisfies the following two conditions: (1) it does not crash, and (2) it is able to communicate in both directions and without omissions, either directly or indirectly, with a majority of processes. Observe that we are assuming the existence of a majority of *well-connected* processes.

2.1. Bidirectional communication: b-link

We use the concept of *b-link* to represent the state of a bidirectional link. Given a pair of processes $(p, q) \in \Pi \times \Pi$, we denote by $b\text{-}link_{p\leftrightarrow q}$, equivalent to $b\text{-}link_{q\leftrightarrow p}$, the state of the bidirectional communication between processes p and q. At a given time, $b\text{-}link_{p\leftrightarrow q}$ can be in one of the following three possible states: *Active*, *Paused* and *Blocked*. When $b\text{-}link_{p\leftrightarrow q}$ is *Active*, p and q exchange messages periodically (in both directions). Instead, when $b\text{-}link_{p\leftrightarrow q}$ is *not Active*, i.e., it is either *Paused* or *Blocked*, p and q do not exchange messages periodically. An *Active* $b\text{-}link_{p\leftrightarrow q}$ becomes *Blocked* when either the communication between p and q is not timely or a message is omitted. An *Active b-link* that behaves timely and where the processes have not omitted any message can be paused in order to reduce the communication cost. Reciprocally, a *Paused b-link* can be activated in order to increase the process connectivity. Fig. 1 shows the state diagram and state transitions for a *b-link*.

2.2. Well-connectedness

Let $G = (V, E)$ be the undirected graph representing the system, where vertexes are processes, i.e., $V(G) = \Pi$, and edges are *Active b-link*s, i.e., $E = \{\{p, q\}$ such that $b\text{-}link_{p\leftrightarrow q}$

[2] We will also use p, q, r, etc. to denote processes.

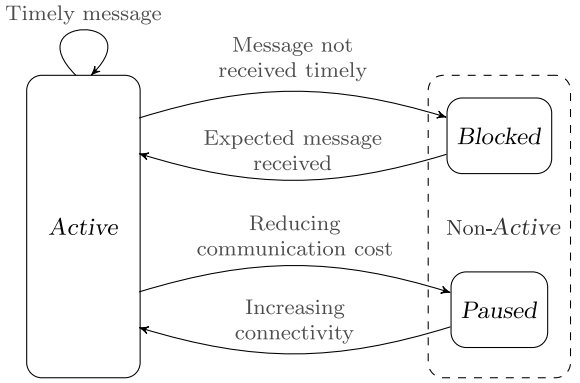

Fig. 1. State diagram of a *b-link*.

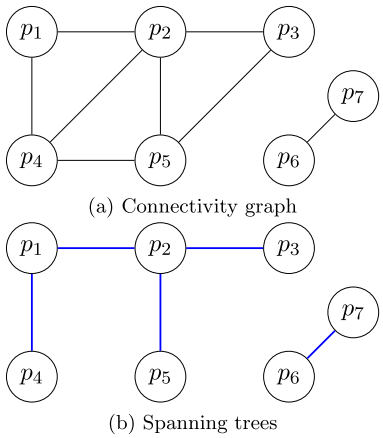

Fig. 2. Undirected graph of *b-links* (a) and subsequent spanning trees (b).

is *Active*}. Due to crashes and omissions, G can be a disconnected graph with several connected components. All the processes belonging to a connected component $S \subseteq G$ can communicate through a path of *Active b-links*. Thus, we say that two processes $p, q \in V(S)$ are *connected*.

We assume in our model that every system contains a connected component S such that $|V(S)| \geqslant \lceil \frac{(n+1)}{2} \rceil$. We say that every process $p \in V(S)$ is *well-connected*.

2.3. Failure detector definition

The failure detector definition we propose for the general omission model is close to that of an Eventually Perfect failure detector for the crash model, denoted $\diamond\mathcal{P}$ [3]. Indeed, we adapt the *correct/faulty* classification of processes in the crash model to the *well-connected/not well-connected* classification in the general omission model. This failure detector satisfies the following completeness and accuracy properties:

- Strong Completeness: eventually every *not well-connected* process is permanently considered as *not well-connected* by every *well-connected* process.
- Eventual Strong Accuracy: eventually every *well-connected* process is permanently considered as *well-connected* by every *well-connected* process.

2.4. Communication efficiency

We say that an algorithm is *communication-efficient* in the general omission model if it uses at most $n-1$ bidirectional links to send messages forever. In the previously defined graph G, the minimum set of edges that connect all the vertexes, i.e., a spanning tree of G, has $n-1$ edges. If G is a disconnected graph, in each connected component $S \subseteq G$ with m processes, a spanning tree of S will have $m-1$ edges, so that in the disconnected graph G there will be less than $n-1$ edges.

3. Implementing the failure detector

In this section, we present a communication-efficient failure detection algorithm satisfying the properties defined in the previous section. First we describe how the connectivity of processes is managed in order to get communication efficiency. Then, we get into details to explain how state transitions are implemented and formal properties satisfied.

3.1. Achieving communication efficiency

Our failure detection algorithm eventually uses at most $n-1$ *Active b-links* in order to maintain the n processes connected, and thus is communication-efficient. To achieve communication efficiency, every process p locally computes a spanning tree T for the connected component $S \subseteq G$ that p belongs to, using a deterministic version of the well known breadth-first search (BFS) algorithm [16]. The input to the BFS algorithm is the connectivity matrix of p, so every process in the same connected component eventually computes the same spanning tree. We assume that our BFS implementation starts with the process with the smallest identifier within each connected component and traverses the network in the order of increasing process identifier values.

Once a process p obtains the spanning tree T, p looks up which of its *Active b-links* should be paused according to T, i.e., if a $b\text{-}link_{p\leftrightarrow q}$ is in S but not in T its state is set to *Paused*. The process in charge of pausing a $b\text{-}link_{p\leftrightarrow q}$ is always the process with the smallest identifier between p and q. This way, for each connected component an overlay network with $m-1$ *Active b-links* is built, being m the number of processes in the connected component.

Fig. 2a shows an example of an undirected graph representing the connectivity of the system in a given execution. Fig. 2b shows the result of applying the spanning tree algorithm to the graph. Paused links are not shown. Note that the following *b-links* are paused: $b\text{-}link_{2\leftrightarrow 4}$, $b\text{-}link_{3\leftrightarrow 5}$ and $b\text{-}link_{4\leftrightarrow 5}$. According to the policy for pausing *b-links*, process p_2 will pause $b\text{-}link_{2\leftrightarrow 4}$, process p_3 will pause $b\text{-}link_{3\leftrightarrow 5}$ and process p_4 will pause $b\text{-}link_{4\leftrightarrow 5}$.

3.2. The algorithm

Fig. 3 presents the proposed failure detection algorithm. For every process p, a set $connected_p$ stores p's perception

```
1  Procedure main()
2      connected_p ← Π;
3      forall q ∈ Π do
4          link_p[q].state ← ACTIVE;
5          link_p[q].buffer ← ∅;
6          link_p[q].send-seq ← 1;        {next message to be sent to q}
7          link_p[q].recv-seq ← 1;        {next expected message from q}
8          link_p[q].timeout ← default time-out interval;
9          forall u ∈ Π do M_p[q][u] ← 1;
10         V_p[q] ← 0;                    {version number for every row of M_p}

11 || Task 1: repeat periodically         {Sending heartbeats}
12     forall q ∈ Π − {p} do
13         if link_p[q].state = ACTIVE then
14             send-message(ALIVE, link_p[q].send-seq++, M_p, V_p) to q;

15 || Task 2: repeat periodically         {Checking time-outs}
           if ( link_p[q].state = ACTIVE and
                p has not received (ALIVE, link_p[q].recv-seq,
                M_q, V_q) from q ≠ p during the last
16              link_p[q].timeout time units of p's clock ) then
17         link_p[q].timeout++;
18         change-link-state(q, BLOCKED);

19 || Task 3: when receive a message m (type, id, M_q, V_q) from q
   {Processing received messages in order}
20     insert m into link_p[q].buffer;
21     while ( ∃ a message m′ with (type, id, M_q, V_q) in
               link_p[q].buffer such that id = link_p[q].recv-seq ) do
22         if link_p[q].state = BLOCKED then
23             change-link-state(q, ACTIVE);
24         if type = START and link_p[q].state = PAUSED then
25             change-link-state(q, ACTIVE);
26         if type = PAUSE and link_p[q].state = ACTIVE then
27             change-link-state(q, PAUSED);
28         forall u ∈ Π − {p} do
29             if V_q[u] > V_p[u] then
30                 forall v ∈ Π do M_p[u][v] ← M_q[u][v];
31                 V_p[u] ← V_q[u];
32         remove m′ from link_p[q].buffer;
33         link_p[q].recv-seq++;

34 || Task 4: when M_p changes do        {Check connectivity}
35     connected_p ← calculate-set-of-connected-processes(M_p);
36     if |connected_p| < ⌈(n+1)/2⌉ then
37         q ← process r with the smallest id such that
           (link_p[r].state = PAUSED) and (r ∉ connected_p);
38         if q ≠ null then
39             change-link-state(q, ACTIVE);
40             send-message(START, link_p[q].send-seq++, M_p, V_p) to q;
41     candidates_p ← get-redundant-b-links(M_p);
42     forall q ∈ candidates_p do
43         change-link-state(q, PAUSED);
44         send-message(PAUSE, link_p[q].send-seq++, M_p, V_p) to q;

45 Procedure change-link-state(q:process-id; newState:state-type)
46     link_p[q].state ← newState;
47     M_p[p][q] ← (newState = ACTIVE);
48     V_p[p]++;
```

Fig. 3. The failure detector algorithm.

of the connected component p belongs to, and henceforth provides the properties of the failure detector. In this regard, if $|connected_p| < \lceil \frac{(n+1)}{2} \rceil$, then process p does not consider itself as *well-connected*. On the other hand, if $|connected_p| \geqslant \lceil \frac{(n+1)}{2} \rceil$, then process p considers itself as *well-connected*, as well as all the processes in $connected_p$.

A $b\text{-}link_{p \leftrightarrow q}$ is implemented by a pair of variables, $link_p[q].state$ at process p, and $link_q[p].state$ at process q. For the sake of brevity, in the rest of this section we will use $link_p[q]$ to refer to $link_p[q].state$. A Boolean matrix M_p is used by every process p to represent the connectivity of the whole system. M_p represents the adjacency matrix of an undirected graph where $M_p[p][q]$ shows if there is an edge between p and q. If $M_p[p][q] = M_p[q][p] = 1$ then $b\text{-}link_{p \leftrightarrow q}$ is *Active*, while if $M_p[p][q] = M_p[q][p] = 0$ then $b\text{-}link_{p \leftrightarrow q}$ is non-*Active* (either *Paused* or *Blocked*).

The algorithm is based on the periodical exchange of heartbeat messages. Every message sent by p carries M_p as well as a version number $V_p[q]$ for each row q of the matrix. A process p updates M_p whenever a $b\text{-}link_{p \leftrightarrow q}$, for any q, changes. Also, p updates M_p from the matrix M_q carried by every message received from q (in this case, vectors V_p and V_q are compared in order to get the most up-to-date information regarding the connectivity of the system).

We now describe the four main tasks that the algorithm executes. In Task 1, every process p periodically sends an ALIVE message through each one of its *Active b-links*. In order to detect message omissions, every message that p puts into a link has an associated sequence number $link_p[q].send\text{-}seq$.

In Task 2, every process p waits for the next ALIVE message from each one of its *Active b-links*. If p does not receive the next expected message from a process q timely, then $link_p[q]$ is changed from ACTIVE to BLOCKED and the corresponding timeout value is incremented. When $link_p[q]$ changes to BLOCKED, p stops sending ALIVE messages to q, and therefore $link_q[p]$ will eventually change to BLOCKED too. Observe that if the timeout has expired due to a message omission, the *b-link* will remain *Blocked* forever. Otherwise, i.e., if the timeout was premature, the increment of the timeout value guarantees that eventually the expected message, unless omitted, will always be received timely.

In Task 3, received messages are delivered following their sequence number. If the message is received through a BLOCKED $link_p[q]$, it consequently becomes ACTIVE, and by Task 1 process p starts sending ALIVE messages to q again. The *b-link* state transitions between *Active* and *Paused* of Fig. 1 are implemented too, and some rows of matrix M_q are copied into M_p if needed.

Finally, Task 4 is executed by a process p whenever M_p changes (either in Task 2 or in Task 3) in order to update $connected_p$. Furthermore, Task 4 adjusts p's connectivity if necessary. In case p needs to increase its connectivity, it will send START messages to processes with which it has a *Paused b-link*. This task also checks if there are redundant *Active b-links* (by the *get-redundant-b-links* procedure, which implements the previously described BFS algorithm). To pause an *Active* $b\text{-}link_{p \leftrightarrow q}$, process p sends a PAUSE message to q.

3.3. Correctness proof

We now show that the presented algorithm satisfies the failure detector properties defined in Section 2. Furthermore, we also show that the algorithm is communication-efficient.

The proof is divided in three parts. First, Lemmas 1 to 5 show that, for every two processes p and q, the variables $link_p[q]$ and $link_q[p]$ represent consistently the behavior defined for $b\text{-}link_{p\leftrightarrow q}$. Then, Lemmas 6 to 8 show that the system eventually converges, i.e., eventually all the processes in each connected component agree permanently on the same set of *Active b-link*s. Finally, Lemmas 9 and 10 show that the $connected_p$ variable at each *well-connected* process p satisfies the completeness and accuracy properties defined in Section 2.

Lemma 1. *If an* ACTIVE $link_p[q]$ *is set to* BLOCKED*, then eventually either* $link_q[p]$ *is set to* BLOCKED *or* PAUSED*, or* $link_p[q]$ *will not remain* BLOCKED*.*

Proof. By Task 2, a process p that does not receive the next expected message from q ($link_p[q].recv\text{-}seq$) in time by an *Active* $b\text{-}link_{p\leftrightarrow q}$ sets $link_p[q]$ to BLOCKED (Line 18). If the next expected message from q has been omitted, $link_p[q]$ is set to BLOCKED permanently. As a consequence, p stops sending ALIVE messages to q (by Task 1 of p) permanently, and if $link_q[p]$ is ACTIVE it will be set to BLOCKED permanently too (Line 18). If the next expected message from q has not been omitted, this message will eventually arrive to p and $link_p[q]$ will be set to ACTIVE again (Lines 22–23). If when p receives the expected message from q, sets $link_p[q]$ to ACTIVE again and sends by Task 1 an ALIVE message to q which is received in q before its timeout on p triggers, then $link_q[p]$ will remain ACTIVE. Otherwise, if the timer on p triggers at q, $link_q[p]$ will be set to BLOCKED temporarily. Finally, note that if q concurrently has set $link_q[p]$ to PAUSED, $link_q[p]$ will remain PAUSED, since no messages from p are waited by Task 2 of q if $link_q[p]$ is PAUSED, and thus the transition to BLOCKED does not apply. □

Lemma 2. *If an* ACTIVE $link_p[q]$ *is set to* PAUSED*, then eventually* $link_q[p]$ *is set to* PAUSED *or* BLOCKED*.*

Proof. By the procedure *get-redundant-b-links*, a process p tries to pause $b\text{-}link_{p\leftrightarrow q}$ by sending a PAUSE message to q and changing $link_p[q]$ to PAUSED (Lines 41–44). Observe that, according to our policy for pausing *b-link*s, an *Active* $b\text{-}link_{p\leftrightarrow q}$ is paused only by the process with the smallest identifier between p and q.

When a $link_p[q]$ is set to PAUSED, if the PAUSE message sent by p is not eventually received in q, by Task 2, q sets $link_q[p]$ to BLOCKED permanently. If the PAUSE message sent by p is eventually received in q and $link_q[p]$ is ACTIVE, by Task 3, q will set $link_q[p]$ to PAUSED (Lines 26–27). Otherwise, if $link_q[p]$ is BLOCKED due to a premature timeout, $link_q[p]$ will be set first to ACTIVE (Lines 22–23 of Task 3 of q) and then to PAUSED (Lines 26–27). □

Lemma 3. *If a* BLOCKED $link_p[q]$ *is set to* ACTIVE*, then eventually either* $link_q[p]$ *is set to* ACTIVE*, or* $link_p[q]$ *will not remain* ACTIVE*.*

Proof. By Task 3 (Line 23) a process p sets a BLOCKED $link_p[q]$ to ACTIVE when the next expected message, $link_p[q].recv\text{-}seq$, is received. Consequently, by Task 1, p starts to send ALIVE messages to q. If $link_q[p]$ is BLOCKED and q eventually receives the next expected message from p, q will set $link_q[p]$ to ACTIVE and, by Task 1, q will send ALIVE messages to p. Otherwise, if the next expected message from p to q is omitted, by the reasoning followed in Lemma 1, $link_q[p]$ is set to BLOCKED permanently, and hence $link_p[q]$ will not remain ACTIVE. Finally, note that if $link_q[p]$ is set to PAUSED concurrently, by Lemma 2 $link_p[q]$ is set to PAUSED or BLOCKED, i.e., it will not remain ACTIVE. □

Lemma 4. *If a* PAUSED $link_p[q]$ *is set to* ACTIVE*, then eventually either* $link_q[p]$ *is set to* ACTIVE*, or* $link_p[q]$ *will not remain* ACTIVE*.*

Proof. By Task 4 (Lines 36–40), a process p tries to activate $b\text{-}link_{p\leftrightarrow q}$ (in order to increase its connectivity) by sending a START message to q and changing $link_p[q]$ to ACTIVE. If $link_q[p]$ is PAUSED and q receives the START message, by Task 3, q sets $link_q[p]$ to ACTIVE (Lines 24–25) and consequently, by Task 1, q starts to send ALIVE messages to p. Otherwise, if q does not receive the START message from p, $link_q[p]$ will not be set to ACTIVE. Thus, q will not start to send ALIVE messages to p and therefore, $link_p[q]$ will be set to BLOCKED. Finally, note that if $link_q[p]$ is permanently BLOCKED (due to an omission from p to q), then by the reasoning followed in Lemma 1, $link_p[q]$ will be set to BLOCKED too. □

Lemma 5. *The communication between every pair of processes* $p, q \in \Pi$ *corresponds to the behavior defined for* b-link*s.*

Proof. By Lemmas 1, 2, 3 and 4 the communication from p to q, determined by the variable $link_p[q]$, and the communication from q to p, determined by the variable $link_q[p]$, will eventually have a consistent behavior, which is the one defined for the corresponding $b\text{-}link_{p\leftrightarrow q}$. Observe that the variables $link_p[q]$ and $link_q[p]$ will not have permanently the combination of values (ACTIVE, non-ACTIVE), where non-ACTIVE is either BLOCKED or PAUSED. □

Lemma 6. *Eventually,* b-link*s will block only due to omissions and they will be blocked permanently.*

Proof. By Lemma 1, a $b\text{-}link_{p\leftrightarrow q}$ switches to the state *Blocked* when process p (or q) does not receive an expected message in time. This may occur either because the timer $link_p[q].timeout$ triggered (in Task 2 of p) before the message was received, or because the message was omitted. Since the system is partially synchronous, and since, by Task 2 (Line 17), the timeout associated with the $link_p[q]$ variable ($link_p[q].timeout$) is incremented whenever the link state is set to BLOCKED, eventually no

expected message will be received after the timer triggers. Thus, eventually, the only reason for any *b-link* to be *Blocked* will be a message omission, and henceforth the *b-link* will be *Blocked* permanently. □

Observation 1. At every process p, the matrix M_p is updated with its own connectivity information and with the matrices M_q received in the ALIVE messages. Every time p changes matrix M_p in the procedure *change-link-state* (Lines 45–48) the version number $V_p[p]$ is incremented. The updated M_p and its version number V_p are sent with p's next ALIVE message at least to one process in the same connected component. Observe that when p delivers a message in Task 3, M_p is updated with M_q comparing the version numbers V_p and V_q, and therefore, copying only the last version of the connectivity information (Lines 28–31). Besides, the local delay in process p for relaying M_p and V_p, δ, is bounded in the algorithm by the period of Task 1 of p, which is finite if p is eventually synchronous and has not crashed. This way, implicitly, a relaying mechanism of the last version of M_p and V_p is obtained among the processes in the same connected component.

Lemma 7. *Eventually every process p will not activate more* b-link*s.*

Proof. Eventually, by the definition of connected component and by Lemma 6, for every two processes p and q that belong to different connected components, $b\text{-}link_{p\leftrightarrow q}$ will be *Blocked* permanently. Observe that eventually, by Observation 1, all the processes in the same connected component will share the same matrix M_p with the connectivity information. In Task 4 (Lines 36–40), processes activate *b-link*s trying to connect to a majority of processes. If a process p is *well-connected*, it will eventually be in a connected component with at least $\lceil \frac{(n+1)}{2} \rceil$ processes. After that, p will not activate more *b-link*s (condition in Line 36). If a process q is not a *well-connected* process, by definition, it will belong to a connected component with less than $\lceil \frac{(n+1)}{2} \rceil$ processes. However, eventually q will not have any *Paused b-link* with processes outside its connected component, so q will not be able to activate more *b-link*s (Lines 37–38). □

Lemma 8. *Processes in the same connected component will calculate eventually and permanently the same set of Active* b-link*s* (*eventually the state of a* b-link *does not change from* Active *to* Paused *or vice versa*).

Proof. From Lemma 5, the communication between two processes p, q is always bidirectional, in the sense that both processes exhibit a consistent behavior regarding $b\text{-}link_{p\leftrightarrow q}$. According to Observation 1, when p activates a *b-link*, all the processes in the same connected component will receive this information and will have the same connectivity matrix, M_p; therefore they will share the same set of *Active b-link*s. Observe that the algorithm used to pause *b-link*s in Task 4, procedure *get-redundant-b-links*, is deterministic. Henceforth, all the processes in the same connected component will calculate the same set of *Active b-link*s. Since by Lemma 7 eventually no *b-link* will be activated, and consequently no more *b-link*s will be paused, the set of *Active b-link*s will be permanently identical for every process p that belongs to the same connected component. □

Lemma 9. *Eventually and permanently, for every* well-connected *process p every* not well-connected *process* $q \notin connected_p$.

Proof. If q is a *not well-connected* process, by Lemma 8 and by the definition of *not well-connected* process, eventually q is permanently in a connected component with less than $\lceil \frac{(n+1)}{2} \rceil$ processes. Being p a *well-connected* process, by definition, p is in a connected component with at least $\lceil \frac{(n+1)}{2} \rceil$ processes. Therefore p and q will belong to different connected components, and by the algorithm and Lemma 8, $q \notin connected_p$ permanently. □

Lemma 10. *Eventually and permanently, for every* well-connected *process p every* well-connected *process* $q \in connected_p$.

Proof. If q and p are *well-connected* processes, they will activate *b-link*s to connect to a majority of processes, and by Lemma 8, eventually they will calculate permanently the same set of *Active b-link*s and they will be connected with a majority of the processes permanently. If both p and q are connected to a majority of the processes they must be in the same connected component, and by the algorithm and Lemma 8, $q \in connected_p$ permanently. □

Theorem 1. *The algorithm of Fig.* 3 *implements the properties of Strong Completeness and Eventual Strong Accuracy defined in Section* 2.

Proof. Directly from Lemmas 9 and 10. □

Theorem 2. *Eventually the number of* b-link*s used in the system is permanently* $n-1$ *or lower.*

Proof. By Lemma 8, eventually every connected component will not change, and *Active b-link*s will express the connectivity in the connected component, i.e., the number of *b-link*s used permanently. The procedure *get-redundant-b-links* executes a BFS algorithm that guarantees a spanning tree of *Active b-link*s in a connected component. By the definition of connected component and by Lemma 6, eventually every $b\text{-}link_{p\leftrightarrow q}$ for p and q in different connected components will be *Blocked* forever. Therefore, since the spanning tree of a graph with k nodes has $k-1$ edges, the number of *Active b-link*s in the system will be at most $n-1$. □

3.4. Using the failure detector to solve consensus

We describe now how the proposed failure detector can be used to solve consensus. Following the line of [10], instead of designing a new consensus algorithm from scratch we propose an adaptation of the well-known consensus algorithm of Chandra and Toueg for crash-prone systems [3].

The adaptation is close to the one proposed in [10], but differs in what concerns the use of the overlay network provided by the underlying failure detector.

As in [10], consensus messages are inserted into messages of the failure detector, in order to cope with omissions in the consensus algorithm. Process connectivity, and specifically the component containing *well-connected* processes, provides the necessary relaying framework for consensus messages to be delivered. Therefore, consensus messages follow a path of *Active b-link*s to reach their destination. Proceeding in this way, the communication cost of the consensus algorithm remains linear too. Nevertheless, contrary to [10], where the underlying failure detector uses a permanent all-to-all communication pattern which eases the adaptation, the fact that our failure detector is communication-efficient forces to check if the overlay network has changed during the execution of the current round of the algorithm, in which case the round is skipped in order to avoid blocking.

References

[1] M.C. Pease, R.E. Shostak, L. Lamport, Reaching agreement in the presence of faults, Journal of the ACM 27 (2) (1980) 228–234.
[2] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM 32 (2) (1985) 374–382.
[3] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM 43 (2) (1996) 225–267.
[4] K.J. Perry, S. Toueg, Distributed agreement in the presence of processor and communication faults, IEEE Trans. Software Eng. 12 (3) (1986) 477–482.
[5] Z. Benenson, M. Fort, F.C. Freiling, D. Kesdogan, L.D. Penso, TrustedPals: Secure multiparty computation implemented with smart cards, in: D. Gollmann, J. Meier, A. Sabelfeld (Eds.), ESORICS, in: Lecture Notes in Computer Science, vol. 4189, Springer, 2006, pp. 34–48.
[6] P. Veríssimo, Travelling through wormholes: a new look at distributed systems models, SIGACT News 37 (1) (2006) 66–81.
[7] D. Dolev, R. Friedman, I. Keidar, D. Malkhi, Failure detectors in omission failure environments, in: PODC, 1997, p. 286.
[8] C. Delporte-Gallet, H. Fauconnier, F.C. Freiling, Revisiting failure detection and consensus in omission failure environments, in: D.V. Hung, M. Wirsing (Eds.), ICTAC, in: Lecture Notes in Computer Science, vol. 3722, Springer, 2005, pp. 394–408.
[9] C. Delporte-Gallet, H. Fauconnier, F.C. Freiling, L.D. Penso, A. Tielmann, From crash-stop to permanent omission: Automatic transformation and weakest failure detectors, in: A. Pelc (Ed.), DISC, in: Lecture Notes in Computer Science, vol. 4731, Springer, 2007, pp. 165–178.
[10] R. Cortiñas, F.C. Freiling, M. Ghajar-Azadanlou, A. Lafuente, M. Larrea, L.D. Penso, I. Soraluze, Secure failure detection in TrustedPals, in: T. Masuzawa, S. Tixeuil (Eds.), SSS, in: Lecture Notes in Computer Science, vol. 4838, Springer, 2007, pp. 173–188.
[11] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, Stable leader election, in: J.L. Welch (Ed.), DISC, in: Lecture Notes in Computer Science, vol. 2180, Springer, 2001, pp. 108–122.
[12] A. Doudou, B. Garbinato, R. Guerraoui, A. Schiper, Muteness failure detectors: Specification and implementation, in: J. Hlavicka, E. Maehle, A. Pataricza (Eds.), EDCC, in: Lecture Notes in Computer Science, vol. 1667, Springer, 1999, pp. 71–87.
[13] C. Delporte-Gallet, H. Fauconnier, A. Tielmann, F.C. Freiling, M. Kilic, Message-efficient omission-tolerant consensus with limited synchrony, in: IPDPS, IEEE, 2009, pp. 1–8.
[14] C. Dwork, N.A. Lynch, L.J. Stockmeyer, Consensus in the presence of partial synchrony, Journal of the ACM 35 (2) (1988) 288–323.
[15] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, On implementing omega in systems with weak reliability and synchrony assumptions, Distributed Computing 21 (4) (2008) 285–314.
[16] D.E. Knuth, The Art of Computer Programming, vol. I: Fundamental Algorithms, 3rd edition, Addison–Wesley, 1997.