Using Failure Detection and Consensus in the General Omission Failure Model to Solve Security Problems

Carole Delporte-Gallet[1], Hugues Fauconnier[1], and Felix C. Freiling[2]*

[1] Laboratoire d'Informatique Algorithmique, Fondements et Applications (LIAFA), University Paris VII, France
[2] Laboratory for Dependable Distributed Systems, University of Mannheim, Germany

University of Mannheim
Department for Mathematics and Computer Science
Technical Report TR 2006-16
September, 2006

Abstract. It has recently been shown that fair exchange, a security problem in distributed systems, can be reduced to a fault tolerance problem, namely a special form of distributed consensus. The reduction uses the concept of security modules which reduce the type and nature of adversarial behavior to two standard fault-assumptions: message omission and process crash. In this paper, we investigate the feasibility of solving consensus in asynchronous systems in which crash and message omission faults may occur. Due to the impossibility result of consensus in such systems, following the lines of unreliable failure detectors of Chandra and Toueg, we add to the system a distributed device that gives information about the failure of other processes. Then we give an algorithm using this device to solve the consensus problem. Finally, we show how to implement such a device in a asynchronous untrusted environment using security modules and some weak timing assumptions.

1 Introduction

In systems with electronic business transactions, fair exchange is a fundamental problem. In fair exchange, the participating parties start with an item they want to trade for another item. They possess an executable (i.e., machine-checkable) description of the desired item and they know from which party to expect the desired item and which party is expecting their own item. An algorithm that solves fair exchange must ensure three properties: (1) every honest party eventually either delivers its desired item or aborts the exchange (*termination* property).

* Work by Felix Freiling was performed in part while visiting LIAFA and supported by the French Ministry of Research ACI project FRAGILE. A preliminary version was presented at ICTAC 2005 which appears as LNCS 3722.

(2) If no party misbehaves and all items match their descriptions then the exchange should succeed (*effectiveness* property). (3) If the desired item of any party does not match its description, then no party can obtain any (useful) information about any other item (*fairness* property). Fair exchange algorithms should guarantee these properties for mutually untrusted parties, i.e., even in the presence of arbitrary (malicious) misbehavior of a subset of participants. Therefore, fair exchange is usually considered a problem in the area of security.

It has recently been shown [4] that fair exchange, a security problem, can be reduced to a fault-tolerance problem, namely a special form of *consensus*. In the consensus problem, a set of processes must reach agreement on a single value out of a set of values, values which the individual processes have each proposed. The reduction from fair exchange to consensus holds in a model where each participating party is equipped with a tamper proof security module. Roughly speaking, the security modules are certified pieces of hardware (like a smart card) executing a well-known algorithm so they can establish confidential and authenticated channels between each other. Security modules in principle allow a vendor to execute an algorithm on a machine which is not in his posession anymore. They therefore form the basis for *trusted computing*, a vision shared by many key players in industry [21] to improve the security of computer system. Today, products exist which implement such trusted devices (for example the IBM 4758 Secure Coprocessor [10] or programmable Java Cards [13]). However, since these devices can only communicate by exchanging messages through their (untrusted) host parties, messages may be intercepted or dropped (see Fig. 1). Overall, the security modules form a *trusted subsystem* within the overall (untrusted) system. The integrity and confidentiality of the algorithm running in the trusted subsystem is protected by the shield of tamper proof hardware. The integrity and confidentiality of data sent across the network is protected by standard cryptographic protocols. These mechanisms reduce the type and nature of adversarial behavior in the trusted subsystem to message loss and process self-destruction, two standard fault-assumptions known under the names of *omission* and *crash* in the area of fault-tolerance. To summarize, problems from the area of security motivate us to revisit the consensus problem in omission failure environments.

A central assumption which is used in the reduction of fair exchange to consensus [4] is that the system be *synchronous*. A synchronous system has known upper bounds on all important timing parameters of the system like message delivery delay and relative process speeds. Synchronous systems are rare in practice. More common are asynchronous systems, i.e., systems with no or merely uncertain timing guarantees. This holds especially true for systems in which smart cards are used as security modules. Smart cards do not possess any device to reliably measure real-time since they are totally dependent on power supply from their host. If we would like to implement fair exchange using smart cards as security modules, we need an *asynchronous* consensus algorithm under the assumption of crash and omission faults.

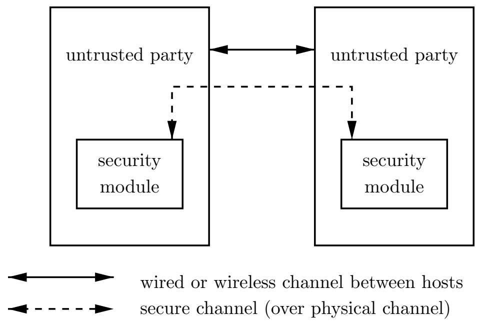
untrusted party
untrusted party
security module
security module
wired or wireless channel between hosts
secure channel (over physical channel)

Fig. 1. Untrusted parties and security modules.

The concept of *omission* faults, meaning that a process drops a message either while sending (*send* omission) or while receiving it (*receive* omission), were introduced by Hadzilacos [14] and later generalized by Perry and Toueg [20]. Under the assumption of omission faults, this paper investigates the feasibility of solving consensus in totally asynchronous systems. Since a result by Fischer, Lynch, and Paterson [11] states that solving consensus deterministically is impossible even if only crash faults can happen, we must strengthen the model so that solutions are possible. We do this using the approach of unreliable failure detectors pioneered by Chandra and Toueg [6]. In this approach, the asynchronous model is augmented with a device that gives information about the failures of other processes. Failure detectors have proven to be a very powerful abstraction of timing assumptions that can express necessary and sufficient conditions for the solvability of problems in the presence of failures. In practice, we want to build a system that solves a certain problem (like consensus). So interesting for practical purposes is the question: What type of failure detector is sufficient to solve that problem? If such a failure detector is found, we only need to implement the failure detector in a system with security modules to get an algorithm that works in practice, usually reducing the complexity of solving the overall problem substantially. Interesting from a theoretical standpoint is the question: What type of failure detector is necessary to solve a problem? Answers to this question point to the minimum level of timing information which is needed to solve that problem. If only less is available, the problem is impossible to solve.

Here, we concentrate on the sufficiency part of the question, i.e., we ask the following questions:

- What type of failure detector is sufficient to solve consensus in asynchronous systems in which crash and omission faults can occur?
- Under what realistic timing assumptions can such a failure detector be implemented?

We make the following two contributions in this paper:

- We define a new type of failure detector, which we call Ω in analogy to Chandra, Hadzilacos and Toueg [5], and give a protocol that solves consensus

  in omission failure environments as long as a majority of processes remains fault-free.
- We exhibit a set of weak timing assumptions in the spirit of earlier work [1, 3] that allow to implement Ω. More precisely, we show that the existence of some process with which every other process *eventually* can communicate in a timely way is sufficient to implement Ω. We argue that such timing assumptions can be made in practice.

Solving consensus in omission environments have been studied previously in other work [7, 8, 12]. Unpublished work by Dolev *et al.* [7, 8] also follows the failure detector approach to solve consensus. Their failure detector $\Diamond\mathcal{S}(om)$ is different but rather close in power to our definition of Ω. In contrast to Dolev *et al.* [7, 8], we focus on the implementability of that failure detector under weak synchrony assumptions. To the best of our knowledge, our consensus algorithm using Ω is also novel in this model.

Freiling *et al.* [12] study the question how randomization can be used to improve the performance of consensus algorithms in omission environments. In contrast to our work, they assume a synchronous algorithm. Synchronous consensus is also considered in work by Parvédy and Raynal [19] who give an algorithm which is adapted in the original work by Avoine *et al.* [4] which presents the reduction of fair exchange to a special form of consensus.

Concerning timeliness assumptions enabling to solve consensus, Dwork, Lynch and Stockmeyer [9] proved that consensus is solvable if all correct processes are eventually timely. Other work [2] obtained the same sufficient timeliness assumptions as here. Note that in both cases, the authors consider the *arbitrary* (or *Byzantine* [15]) failure model that is strictly stronger than omission faults. So their algorithms are necessarily more inefficient that ours. Also, these solutions do not use a modular approach with failure detectors. Overall, our results allow to implement consensus, and hence fair exchange, more efficiently and in a larger class of practical systems than before.

This paper is structured as follows: Section 2 introduces the system model and the reduction of fair exchange to a consensus-like problem using security modules. Section 3 specifies the new type of failure detector. Section 4 presents the algorithm to solve consensus using the failure detector from Section 3. Section 5 shows how to implement the failure detector under very weak synchrony assumptions. Finally, Section 6 concludes the paper.

2 Definitions and Model

2.1 Untrusted System

Security problems like fair exchange are considered within a distributed system which is modeled by a set of n parties that communicate using message passing over a network of channels in a fully connected topology. The communication primitives we assume are **send** and **receive**. Communication channels are reliable, i.e., every message sent is eventually received and every received message was previously sent. Processes in this system can behave arbitrarily and do not necessarily trust each other. Because of mutual distrust, we call this system the *untrusted system* (see Fig. 2).

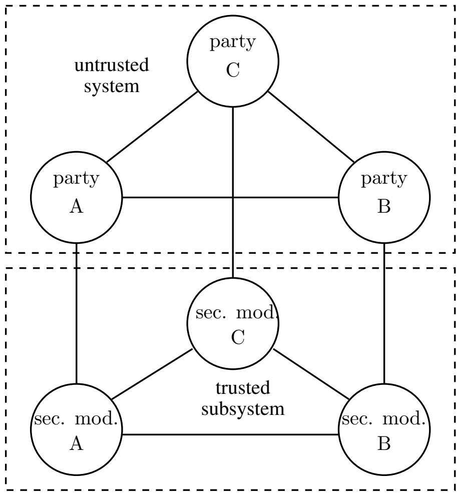

Fig. 2. The untrusted system and the trusted subsystem.

We assume that the network is asynchronous, i.e., there is neither a bound on the relative process speeds nor on the message delivery delays. This means that while one party takes a single step within the execution of its local algorithm, any other process can take an arbitrary (but finite) number of steps. Also, messages can take an arbitrary (but finite) amount of time to travel from the source to the destination.

2.2 Trusted Subsystem

We assume that every party contains a security module. A security module is able to execute its own algorithm independent of its host party. We call the entity executing the algorithm a *process.* Communcation between host party its security module process is done via (synchronous) invocations of local library routines.

The network of parties is reflected in the network of processes. The 1 : 1 correspondence between security modules and parties results in a second distributed system consisting of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that communicate using message passing over a network of channels in a fully connected topology. The communication primitives we assume are also **send** and **receive** which are also reliable. Due to the influence of their host parties, processes can be faulty, as we explain shortly. The network of processes is also asynchronous.

2.3 Failure Assumption

There are three ways in which processes can fail: (1) Processes can *crash*, i.e., they stop to execute steps of their local algorithm. Crashed processes never recover. (2) Processes can experience *send omission* failures, i.e., a message which is sent by a process is never placed into the communication channel. (3) Processes can experience *receive omission* failures, i.e., a message which arrives over the communication channel is never actually received by the algorithm of the process. Crash faults model, the usual hardware or operating system crashes, omission faults model overruns of internal I/O buffers within the operating system.

The types of failures result in three distinct failure assumptions:

– the *send omission model*, in which processes can crash and experience only send-omissions (and no receive omissions),
– the *receive omission model* (analogous to the send-omission model), and
– the *send/receive omission model* (sometimes also called *general omission*), in which processes can crash and experience either send-omissions or receive omissions.

A process p is *correct* if it does not make any failure at all, i.e., it never crashes and experiences neither send nor receive omissions. Process p is *crash-correct* if it never crashes. If process p crashes at some time we say it is *crash-faulty*.

In the following we assume that all correct processes send infinitely often messages to all processes.

Due to the omissions, some processes could be disconnected forever from correct processes. More precisely, we say that process p is *in-connected*, if infinitely often it receives messages from some correct processes. By analogy, we say that process p is *out-connected*, if an infinity of its messages are received by some correct processes. A process is *connected* if it is in-connected and out-connected. Note that a in-connected or out-connected or connected process is crash-correct because it makes an infinity of steps.

Clearly, in the send-omission failure model every process is in-connected, and in the receive omission failure model every process is out-connected. Transient omissions refer to cases when a process regularly omits a message but equally regularly sends/receives a message over the channel. Such omissions can be masked by piggybacking information about previous messages on every new message sent over a channel. In the following we assume such a piggybacking mechanism ensuring that if p receives an infinity of messages sent by q then p receives all messages from q.

Then with this assumption, if p is in-connected then p receives *all* messages from at least one correct process. In the same way, if p is out-connected at least one correct process receives *all* messages from p.

2.4 Relations to Crash Model

Since omissions introduce asymmetry in the communication relation, it is also an issue who can communicate with whom. For example, a process with receive

```
Code for $p$:
on receive $(m, d)$ from $q$
    if $d = p \wedge m$ not delivered before then Receive m
    else if $d \neq p$ then send $(m, d)$ to $d$

to Send$(m)$ to $d$:
    send $(m, d)$ to all
```

Fig. 3. Send/Receive with relay.

omissions may receive messages from a correct process p but may fail to receive messages from another correct process q. We can mask parts of this asymmetry by using the relay algorithm of Figure 3 which defines new primitives **Send** and **Receive**. These primitives ensure that if a process p is in-connected then it receives infinitely often messages from all correct processes. Correspondingly, if a process is out-connected, then infinitely many of its messages are received by all correct processes. However note that the relay algorithm is costly concerning the communication load (each message from p to q generates $2n - 1$ messages).

In the following algorithms we avoid to use this relay algorithm. But it shows that if all crash-correct processes are connected, then by piggybacking old messages and with the relay algorithm all omissions can be masked and the omission models become equivalent to the crash failure model. Interesting cases arise if not all crash-correct processes are always connected.

2.5 Consensus

We use the standard definition of Uniform Consensus in this paper. The problem is defined using two primitives called *propose* and *decide*, both taking a binary value v. An algorithm solving consensus must satisfy the following properties:

- (Termination) Every correct process eventually decides.
- (Uniform Agreement) No two processes decide differently.
- (Validity) The decided value must have been proposed.

2.6 From Fair Exchange to Consensus

We briefly sketch the basic idea of the reduction of fair exchange to consensus [4].

The consensus problem used is called *biased consensus* and can be regarded as uniform consensus with a veto right. The parties in the untrusted system pass their exchange items to their security modules. Now the items are in the trusted system and are exchanged using a simple broadcast protocol between the processes. The processes can now receive the expected items and check then against the specification (is the item as expected). The result of this check (1 or 0) is the input to a round of biased consensus. If any process found a mismatch between the item and the specification, consensus will result in 0. This will cause all processes to abort the exchange. Otherwise, the expected item item is released to the corresponding party.

Note that the processes run a certified piece of code, so the host party cannot interfere or reprogram the process. Also, communication between processes is confidential so no secret information is leaked to the parties unless the exchange succeeds. So the only method for a party to influence the trusted subsystem is by crashing the process or removing messages from the channel. Since the consensus protocol within the trusted subsystem tolerated crashes and omissions, fair exchange will terminate.

3 Failure Detectors for Omission Failure Environments

In this section we revisit failure detectors in crash environments and give a suitable definition for such a failure detector in omission failure environments.

The definition of failure detectors in the crash model are standard [6] and the literature contains a lot of definitions of failure detectors for crash failures. Among these, the failure detector Ω is particularly interesting: It has been proved to be the weakest failure detector to solve the consensus problem in the crash failure model with a majority of correct processes [5]. The output of Ω for each process p is the identity of one process, the assumed leader for p, such that eventually all correct processes have the same leader forever and this leader is a correct process. Hence Ω implements an *eventual leader election*.

We now extend the definition of failure detector Ω to omission models. In the omission model, the definition of Ω from the crash model would naively translate to an eventual leader election of a *correct* process (i.e., neither does it experience a crash nor any omission). This is generally too restrictive, because it could be impossible to ensure that the chosen eventual leader does not experience permanent omissions. In the same way it is too restrictive to force processes that may receive no messages from correct processes to have the eventual common leader. But, in the other hand, if some processes able to communicate to correct processes have a wrong leader it could be a problem. Then a trade-off is to ensure that all out-connected processes eventually have the good leader or know that they do not have. So we consider the following definition:

Definition 1. *Failure detector Ω for omission models is a failure detector that outputs at each time for each process one process, called the leader, or $\perp$ such that there is a connected process l and a time after which, Ω outputs l for each correct process p, and l or $\perp$ for each out-connected process.*

Note that in contrast to the definition of Ω in the crash model, our definition of Ω allows the eventual leader process to be faulty: The leader may experience send and receive omissions as long as it remains connected. We give implementations of Ω in partial synchrony models with weak synchrony assumptions in Section 5.

In the following algorithms the output of the failure detector Ω for process p is given by the value of local variable $Leader$.

4 Solving Consensus

We now show that the failure detector Ω introduced in the previous section is sufficient to solve consensus with a majority of correct processes in the send/receive omission model. Figure 4 depicts our consensus algorithm. It employs the well-known rotating coordinator paradigm, i.e., processes run through asynchronous rounds (counted using the variable r in task 1) and in every such round one process C is chosen as the coordinator. The processes start with v being their proposal value of consensus and spawn three concurrent tasks. In task 0, the coordinator is urged (by using $COORD$ messages) to "impose" its value on all processes by sending ONE messages (task 1). Processes then evaluate the value they receive from the coordinator (stored in $estfromC$). Unless it comes from the leader (referred to by Ω), a $\perp$ value is stored. In the second part of the algorithm, all processes broadcast their received value to all other processes (TWO messages). If such messages are received from a majority of processes, the non-$\perp$ value given in the messages is the decided value and an appropriate decision message is broadcast to all. Task 2 just ensures that eventually all processes who receive the decision message actually do decide.

Proposition 1. *Algorithm of Figure 4 implements consensus for a majority of correct processes in the send/receive omission model augmented with Ω.*

In the proofs of algorithms, by convention, given a variable x of process p, x_p^τ denotes the value of x in p at time τ; when time τ is clear we omit it.

To prove the proposition, we first state the two following lemmas:

Lemma 1. *If p and q end the first part (lines 13 to 18) of a round r, then:*

(1) if $estFromC_p = x$ for some $x \neq \perp$ then $estFromC_q \in \{\perp, x\}$,

If p and q end line 21 of a round r, then:

(2) if $L_p = \{x\}$ for some $x \neq \perp$ then $L_q = \{x\}$ or $L_q = \{x, \perp\}$,
(3) if $L_p = \{\perp, x\}$ for some $x \neq \perp$ then $L_q = \{x\}$ or $L_q = \{x, \perp\}$ or $L_q = \{\perp\}$.

Proof. (1): Notice first that for any process q, v_q is always a value proposed by some process and obviously $v_q \neq \perp$.

If $estFromC_p = x$ for some $x \neq \perp$ then p has received one message (ONE, x, r) from the coordinator $p_{1+r \bmod n}$. By the algorithm, the coordinator $p_{1+r \bmod n}$ sends only one message $(ONE, *, r)$ per round to all processes. Either the coordinator is not the leader for q ($p_{1+r \bmod n} \neq Leader_q$) and then $estFromC_q = \perp$, or the coordinator is the leader for q and q waits for the message ONE, and then $estFromC_q = x$.

(2) and (3): If $L_p = \{\perp, x\}$ or $L_p = \{x\}$ then at least one process, say u, ends the first part (lines 13 to 18) of round r, and $EstFromC_u = x$. By (1), at

```
Code for $p$:
1     Initialization:
2        $r := 0$                                            /* round number */
3        $v := \langle$proposed value$\rangle$
4     start Task 0 and Task 1 and Task 2
Task 0:
5        upon receive$(COORD, *, k)$ for the first time
6             let $(COORD, w, k)$ be such a message
7             send$(ONE, w, k)$ to all other processes

8        upon receive$(ONE, *, k)$ for the first time
9             let $(ONE, w, k)$ be such a message
10            send$(ONE, w, k)$ to all
Task 1:
11    loop forever
12       $C := 1 + r \bmod n$                                /* coordinator */
13       send$(COORD, v, r)$ to $p_C$
14       wait until (receive $(ONE, *, r)$ from $p_C$) or (($p_C \neq Leader$) and ($Leader \neq \perp$))
15            if $(ONE, w, r)$ is received then
16               $estfromC := w$
17            else
18               $estFromC := \perp$

19       send$(TWO, estFromC, r)$ to all
20       wait until $receive(TWO, *, r)$ from a majority of processes
21            let $L = \{w \mid (TWO, w, r)$ is received $\}$
22            if $L = \{rec\}$ for some $rec \neq \perp$ then
23               send $(DECIDE, rec)$ to all
24               $decide(rec)$
25               halt
26            else
27               if $L = \{rec, \perp\}$ for some $rec \neq \perp$ then
28                  $v := rec$
29       $r := r + 1$

Task 2:
30       upon received$(DECIDE, k)$ from $q$
31               send$(DECIDE, k)$ to all
32               decide$(k)$
33               halt
```

Fig. 4. Consensus algorithm for the send/receive omission model using Ω.

most two values, $\perp$ and x, could be sent by processes to all processes in line 19. And hence for any process q that ends round r either $L_q = \{x\}$ or $L_q = \{\perp, x\}$ or $L_q = \{\perp\}$. This concludes the proof of (3).

For (2), it remains to prove that if $L_p = \{x\}$ then $L_q \neq \{\perp\}$. As processes wait for a majority of processes, p and q get message from at least one common process s. By the algorithm s sends at most one message $(TWO, *, *)$ per round.

Then s sends message (TWO, y, r) with either $y = \perp$ or $y = x$. As p and q have waited for this message, this excludes the case $L_p = \{x\}$ and $L_q = \{\perp\}$. □

Lemma 2. *If every process begins some round r, with its variable v equal to the same value d then all processes ending this round either decide d or have their variable v equal to d at the end of this round.*

Now we show that the algorithm satisfies the properties of consensus.

Lemma 3. *The algorithm ensures the agreement property.*

Proof. Consider the first time a process, say p, sends a message $(DECIDE, d)$ for some d. By an easy induction, this sending occurs in task 1, say in round r. In this round, after line 21, L_p is $\{d\}$. Let q be any other process ending round r, by Lemma 1, in this round L_q is either $\{d\}$ and q decides in round r, or $\{d, \perp\}$ and q ends the round r with $v = d$.

By Lemma 2 and an easy induction, in every round $r' \geq r$, every process either decides d or ends the round with $v = d$. Hence, all processes which decide in task 1, decide d. If a process decides in task 2, by an easy induction, this decision is issued from a process which has decided in task 1. This proves the agreement property. □

Lemma 4. *The algorithm ensures the validity property.*

Proof. In the algorithm, all the processes send the values they have just received and by an easy induction they never insert in the algorithm a value of their own. □

Lemma 5. *The algorithm ensures the termination property.*

Proof. If there is no correct process, termination is trivial. If any correct process decides by task 2 or task 1 then clearly all correct processes decide.

Assume that no correct process decides, then we prove that all correct processes participate to an unbounded number of rounds. For this, assume the contrary and let r_0 be the minimal round number in which at least one correct process is blocked forever. Let p be such a process in round r_0:

– p cannot be blocked in Line 14: if the current coordinator p_C is not crash-correct or is not connected, there is a time after which it cannot be leader and then p cannot be blocked. Note that the ONE-message of the coordinator could be lost due to a send omission, but we assume that p_C sends infinitely many messages (e.g., by using a repeated send operation of that message). Otherwise it would not be a connected process. If the current coordinator is connected, by an easy induction it receives $COORD$ message from one correct process then p will eventually receive a ONE message (directly or after one relay) from the coordinator.
– p cannot be blocked in Line 20: by an easy induction all correct processes will reach round r and send a TWO message for this round. As there is a majority of correct processes, p will receive a majority of TWO messages.

By the property of the eventual leader election, there is a time τ after which all correct processes have the same leader p_l and this leader is connected and all out-connected processes have p_l or $\perp$ as leader. There is a time τ' such that after time τ' no correct process receives any messages from processes that are not out-connected. Let time τ_0 be $\max(\tau, \tau')$. Consider R the set of rounds in which correct processes are at time τ_0. Let r_0 be the first round number such that p_l is the coordinator for r_0 and r_0 is greater than all elements of R. If they do not decide before all correct processes will be in round r_0 at some time. All out-connected processes in round r_0 wait until receiving ONE message from p_l for round r_0 because either they do not suspect p_l or $Leader = \perp$. As p_l is correct it sends ONE message for this round and all out-connected processes in round r_0 adopt for $estFromC$ the value sent by p_l. As there is at least a majority of correct processes all out-connected processes in round r_0 receives TWO messages for round r_0 from at least a majority of processes. Moreover, in round r_0 no messages from processes that are not out-connected are received, then all the TWO messages for round r_0 received by out-connected processes in round r_0 contain the same value proposed by p_l and L set is reduced to one element which is different from $\perp$ then all correct processes decide. $\square$

This concludes the proof of the proposition.

Remark: It is possible to weaken the assumption that we have a piggybacking mechanism ensuring that every processes receiving an infinity of messages receives all messages.

For this, every process piggybacks in all its messages only its last $COORD$, ONE, TWO and $DECIDE$ messages. Then a process waiting in Line 14 for a $(ONE, *, r)$ message stops waiting and go to the next instruction (Line 15) if it receives or has already received at least one $(ONE, *, k)$ message with $k > r$.

5 Implementing Failure Detectors

In this section we give algorithms to implement eventual leader election in the case of send and send/receive omissions. All these algorithms make some additional assumptions [2, 6, 17], that are needed if we want to implement consensus deterministically [11]. We also assume that all processes are able to measure time.[3] Note that the weak forms of synchrony used in this section can be justified for a smart card setting in practice: Smart cards have a possibility to measure the passing of time by counting the number of invokations at their interface. Furthermore, it is reasonable to assume that there exists at least one honest party to which communication is at least eventually timely.

[3] In fact they can measure time with a very low accuracy: it is sufficient that (1) the time interval measure is not decreasing, (2) for each finite time interval I there is an integer n such that the measure for I is always less than n, and (3) if the measure of interval time I is less than n then I is a finite time interval.

5.1 Partially Synchronous Models and Eventual Leader Election

In the omission models, messages from p to q are not received by q only due to send omissions from p or receive omission from q. Hence all communication links are assumed to be reliable. There is no duplication of messages and every received message has been sent before.

Concerning timeliness, a communication link (p, q) is *eventually timely* if there is a Δ and time τ_0 after which every message sent at time τ by p to q is received by time $\tau + \Delta$. Following previous work [1, 3], we define eventual sources and bisources:

Definition 2. *Process p is an* eventual source *if and only if (1) p is a correct process and (2) for all correct processes q, communication link (p, q) is eventually timely. Process p is an* eventual bisource *if and only if (1) p is a source and (2) for all correct processes q, communication link (q, p) is eventually timely.*

Note that if we have at least one eventual bisource in the system, the system is eventually rather synchronous: If all messages are broadcast and relayed one time, as eventually all links from correct processes to the eventual bisource and all the links from this eventual bisource to every correct process are eventually timely, there is a time after which all messages sent by correct processes are received in a timely way by all correct processes. Nevertheless, note that in the partially synchronous model of Dwork, Lynch and Stockmeyer [9], it is assumed that eventually all links between processes are timely. This assumption is strictly stronger than the existence of an eventual bisource in the system. Having an eventual bisource does not exclude that the communication delay between two processes is unbounded if one of these processes is faulty but crash-correct. For example, the communication delays from (faulty but crash-correct process) p to (correct process) q are unbounded, if p makes infinitely often send omissions to all processes but q, the communication from p to q (or every other processes to which q could relay messages from p) is not timely.

5.2 Eventual Leader Election

In this section, when we consider send-omission models we assume that there is at least one eventual source and when we consider send/receive or receive-omission models, we assume that there is at least one eventual bisource.

In order to choose a leader, in the presented here algorithms every process monitors the timeliness of the communication links. For this each process sends "ping" messages regularly and verifies that the messages arrive within a bounded delay. If this is not the case, the origin of the message is suspected to be faulty.

Note that in the following algorithms there is no need to have a piggybacking mechanism ensuring that if p receives an infinity of message from q it receives all messages of q. Intuitively this follows from the fact that all correct processes send an infinity of messages to all other processes and that only the last received message is needed by the algorithms.

Eventual Leader Election in the Send Omission Model. The algorithm in Figure 5 implements Ω for the case of send omission faults under the assumption that there is one eventual source.

In the algorithm, $Timer[q]$ is a special variable that is decremented at each clock tick. When $Timer[q]$ achieves a value equal to zero, we say that $Timer[q]$ expires. The principles of the algorithm are rather simple. Each process maintains a variable δ that is the assumed communication delay. This variable is incremented each time a communication of a process exceeds the assumed communication delay. Each process sends periodically (every η clock ticks) a message to all other processes and maintains a vector V counting the number of times each process p exceeds the assumed communication delay δ. This vector is piggybacked in each message and each process updates its own vector V accordingly to the received vector (by taking the maximum of the two vectors). In this way, each vector V will evaluate the number of times a process exceeds the assumed communication delay. The leader will be the process having the minimal value in V (in case there is more than one such process, the process with the smallest identity is chosen).

Intuitively, if a process p makes an infinite number of send omissions to some out-connected process, then eventually, the $V[p]$ of every out-connected process will be unbounded. However, if $V[p]$ is bounded by b for some out-connected process, then it will be bounded by b for every out-connected process. Then eventually all the $V[p]$ of out-connected processes will be equal. Assuming that $V[p]$ is bounded for at least one process, choosing as leader the minimal p with the smallest value in vector V, ensures then that every out-connected process eventually chooses p forever.

Then if s is an eventual source, it is straightforward to verify that $V[p]$ is bounded for every crash-correct process ensuring that every crash-correct process eventually chooses forever the same leader.

Note that this leader is not necessarily a correct process: if p makes infinitely often send omissions to some process q that is not out-connected, it is possible that p is chosen as leader by all correct processes. In this case, the leader for q could be different from the one for p. However, if there is at least one eventual source in the system, this algorithm implements failure detector Ω:

Proposition 2. *In the algorithm of Figure 5, if there is at least one eventual source in the system then there is a crash-correct process l and a time after which every out-connected process gets process l as leader. Moreover, all correct processes receive infinitely often messages from l proving that l is connected.*

By an easy induction we get:

Lemma 6. *If p is out-connected and q is crash-correct, then for all τ there exists $\tau' \geq \tau$ such that $V_p^\tau \leq V_q^{\tau'}$.*

Consider $\lim_{\tau\to\infty} V_p^\tau[q]$, as $V_p^\tau[q]$ is a non decreasing sequence of integers, either $\lim_{\tau\to\infty} V_p^\tau[q] = k$ for some integer k or $\lim_{\tau\to\infty} V_p^\tau[q] = \infty$. In the first

```
Initialization:
 $\delta := 1$
 **for all** $q : V[q] := 0$
 **for all** $q : Timer[q] := \delta$
Task 1:
 **each** $\eta$ clock ticks
     **send** $V$ **to all**
Task 2:
 **on receive** $X$ **from** $q$
     **for all** $r : V[r] := \max\{V[r], X[r]\}$
     **set** $Timer[q]$ **to** $\delta$
Task 3:
 **on** $Timer[q]$ expired
     $V[q] := V[q] + 1$
     $\delta := \delta + 1$
     **set** $Timer[q]$ **to** $\delta$
Task 4:
 **forever do**
         $Leader := \min r$ such that $V[r] := \min\{V[q] | q \in \Pi\}$
```

Fig. 5. Implementation of Ω in a system with at least one eventual source and a majority of correct processes in the send-omission model.

case we say that $V[q]$ converges to k for process p, and in the second case that $V[q]$ does not converge for process p.

If p is crash-faulty or is not out-connected, for every correct process q, $Timer_q[p]$ will expire infinitely often and then $V_q[p]$ will be incremented infinitely often:

Lemma 7. *If p is crash-faulty or is not out-connected then for all crash-correct q, $\lim_{\tau\to\infty} V_q^\tau[p] = \infty$.*

Lemma 8. *If $V[p]$ converges to k for some integer k and for some out-connected process q, then $V[p]$ converges to k for all out-connected processes r.*

Let q out-connected such that $\lim_{\tau\to\infty} V_q^\tau[p] = k$ and r be a crash-correct out-connected process. By Lemma 6 it is impossible that $\lim_{\tau\to\infty} V_r^\tau[p] = \infty$. Then $V[p]$ converges to some k' for process r and then $k \le k'$. Conversely exchanging r and q we get in the same way $k' \le k$, proving the lemma.

Now consider an eventual source s, by definition there is a time τ_0 after which all messages sent by s arrive by some Δ, as each time $Timer_q[s]$ expires, δ_q is incremented, there is a time $\tau_1 > \tau_0$ after which $\delta_q \ge \Delta$ or $Timer_q[s]$ never expires. Proving that $V_q[s]$ is bounded for all process q. By the previous Lemma, we get:

Lemma 9. *If s is an eventual source then $V[s]$ converges to k for some integer k and for all out-connected processes.*

Hence, for at least one process q, $\lim_{\tau\to\infty} V_p^\tau[q] = k$ for all process p. By Lemma 7 and Lemma 8, let M be the max of all k such $V[r]$ converges to k for some r and p, there is a time τ_0 after which for all out-connected p we have $V_p[r] = k$ if $V[r]$ converges to k and $V_p[r] > M$ if $V[r]$ does not converge. Then all out-connected processes get the same leader forever. By Lemma 7, this leader is out-connected.

Eventual Leader Election for Send/Receive Omission Models. For the algorithm of Figure 6, we assume that at least a majority of processes are correct and that there is at least one eventual bisource. The principles of this algorithm are similar to the previous one: each process approximates in δ a bound on the communication delay. The main difference here is that processes maintain an array M to count the number of times messages from p to q exceeded the assumed bound. Moreover in order to ensure that the leader is in-connected it penalizes itself if it sees that it does not receive messages in a timely way from a majority of processes.

As processes may make receive omissions, the value of $M[p,q]$ does not necessarily mean that q has made $M[p,q]$ send omissions, and so the choice of the leader is more intricate. For this, for each process q, we consider all the sets containing a majority of processes and for each such set the maximum value of $M[p,q]$, then the estimate for q is the minimum of these values.

Each process maintains two variables *leader* and *Leader*, the first one is used to find *Leader*, the output of Ω. In fact, eventually all correct processes have the same process id in *leader*, but some processes that are not in-connected may have a different process id in *leader*. A process that is its own *leader* considers itself as *Leader* if it communicates in a timely way with a majority of processes, moreover a process chooses p as *Leader* if (1) it communicates in a timely way with p (2) *leader* is p and (3) p considers itself as *Leader* and if there is no such process *Leader* is $\perp$. By this way all out-connected processes eventually output the same leader l or $\perp$ and all correct processes eventually output l.

If there is at least one bisource in the system, this algorithm implements Ω:

Proposition 3. *In the Algorithm of Figure 6, if there is at least one eventual bisource there is a connected process l and a time after which every connected process gets l as leader. Moreover if p is crash correct and not in-connected then eventually p has $\perp$ or l as leader.*

In the following we say that p consider q as leader if the value of variable *leader* of p is q, and we say that p consider q as Leader if the value of variable *Leader* of p is q.

As not connected out-connected processes are not in-connected, this proposition proves that Algorithm of Figure 6 implements Ω.

As infinitely many messages from correct processes achieve in-connected processes and as infinitely many messages from out-connected processes achieve at

```
Process p:
Initialization:
  $\delta := 1$
  $IamLeader := False$
  for all $q : Timer[q] := \delta$
  for all $q, r : M[q, r] := 0$
  $GoodInputs := \emptyset$

Task 1:
  each $\eta$ clock ticks
    if $(|GoodInput| \leq n/2)$ then
      for all $q : M[q, p] := M[q, p] + 1$
      $IamLeader := False$
    else if $leader = p$ then $IamLeader := True$
    send $(M, IamLeader)$ to all

Task 2:
  on receive $A, b$ from $q$
    for all $x, y : M[x, y] := \max\{M[x, y], A[x, y]\}$
    if $leader = q$ then
      if $b$ then $Leader = q$
      else $Leader = \perp$
    add $q$ to $GoodInputs$
    set $Timer[q]$ to $\delta$

Task 3:
  on $Timer[q]$ expired
    remove $q$ from $GoodInputs$
    if $q = leader$ then $Leader = \perp$
    $M[p, q] := M[p, q] + 1$
    $\delta := \delta + 1$
    set $Timer[q]$ to $\delta$

Task 4:
  forever do
    for all $r$ do
      $V[r] := \min\{\max\{M[q, r] | q \in L\}$ such that $|L| = \lfloor \frac{n}{2} \rfloor + 1\}$
    $leader := \min r$ such that $V[r] := \min\{V[q] | q \in \Pi\}$
```

Fig. 6. Implementation of Ω in a system with at least one eventual bisource and a majority of correct processes.

least one correct process, eventually information from out-connected processes reaches all in-connected and crash-correct processes:

Lemma 10. *If p is out-connected and q is in-connected, then for all τ, there is a time τ' such that $M_p^{\tau} \leq M_q^{\tau'}$.*

If p is not in-connected and crash-correct, there is a time τ after which p does not receive any message from any correct process, as there is a majority of correct processes after time $\tau+\eta$ strictly less than $n/2$ processes belong to $GoodInputs_p$, and at each η, p increments for all q $M[q,p]$ and then $\lim_{\tau\to\infty} M_p[q,p] = \infty$ for all q. Then by Lemma 10:

Lemma 11. *If p is crash-correct and not in-connected then for all in-connected processes q and for all r $\lim_{\tau\to\infty} M_q^\tau[r,p] = \infty$.*

If p is crash-faulty or not out-connected, there is a time after which no messages from p are received by correct processes and then for every correct process q $Timer[p]$ expires infinitely often, and $M_q[q,p]$ is incremented infinitely often and $\lim_{\tau\to\infty} M_q^\tau[q,p] = \infty$. By Lemma 10:

Lemma 12. *If p is crash-faulty or not out-connected then for all in-connected q: $\lim_{\tau\to\infty} M_q^\tau[q,p] = \infty$.*

As at least a majority of processes is correct, any subset of more than $n/2$ processes contains at least one correct process, then if p is crash-faulty or not out-connected or not in-connected by the previous lemmas, $\max\{M_q^\tau[r,p] | r \in L \text{ s.t. } |L| = \lfloor \frac{n}{2} \rfloor + 1\}$ is unbounded for every in-connected process q:

Lemma 13. *If p is crash-faulty or not out-connected or not in-connected then $\lim_{\tau\to\infty} V_q^\tau[p] = \infty$ for every in-connected process q.*

By lemma 10:

Lemma 14. *If $\lim_{\tau\to\infty} V_q^\tau[p] = k$ for some out-connected q, then $\lim_{\tau\to\infty} V_r^\tau[p] = k$ for all in-connected process r.*

Now let s be an eventual bisource, then there a Δ and a time τ after which, (1) every message sent by a correct process to s and (2) every message sent by s to any correct process p is received within Δ. Then as δ_s is incremented each time a timer expires, there is a time $\tau_s > \tau$ after which every correct process are in $GoodInputs_s$, as there is a majority of correct processes, after time τ_s $|GoodInputs_s| > n/2$ and s will not increment $M_s[p,s]$ for any p. In the same way, there is a time $\tau' > \tau_s$ after which no messages from s will exceed δ_p for any correct process p and then $M_p[p,s]$ will not increase. Then:

Lemma 15. *If s is an eventual bisource then for all in-connected process p, $\lim_{\tau\to\infty} V_p^\tau[s] < \infty$.*

Hence, consider the set S of processes q such that for all in-connected processes p $\lim_{\tau\to\infty} V_p^\tau[q] < \infty$. From Lemma 13, S contains only connected processes. By the previous lemma, if there is at least one bisource this set is not empty. By Lemma 14, for every $q \in S$ all the $\lim_{\tau\to\infty} V_p^\tau[q]$ for in-connected p are equal to, say k_q. Let q_0 be the process belonging to S with minimal identity such that k_q is minimal. It is easy to verify that eventually all in-connected processes will chose q_0 as leader. As q_0 is itself connected all in-connected processes choose q as Leader.

Consider any process p that is not in-connected, there is a time after which $|GoodInputs_p| \leq n/2$:

- If p considers itself as leader its Leader is $\perp$.
- If p considers some other process q as leader then q considers itself as Leader and in this case $|GoodInputs_q| > n/2$ and q is in-connected and then q is q_0, or q does not consider itself as Leader and p has $\perp$ as Leader.

This concludes the proof.

6 Conclusion

In this paper we studied consensus in models where processes can crash and experience message omissions. This model was motivated from the area of security problems where omission models can be used to model security problems with smart cards. In this paper we were mainly interested in proving the feasibility of solving consensus in such models, i.e., finding solutions, we were not interested in their efficiency. Hence, most of the algorithms presented here can probably be improved to ensure better performance. For example, in the case of send-omissions and implementation of Ω by algorithm of Figure 5, this algorithm could be improved: In task 0, there is no need to relay the messages ONE because with send-omissions the eventual chosen leader is not only in-connected but already receives infinitely many messages from correct processes.

One interesting open problem is to define the weakest failure detector to solve consensus with omission models, i.e., asking the rather fundamental question on what failure detector is necessary. In particular it is not proved that really the existence of an eventual bisource is needed for receive (and send/receive) omissions models. This would also give a lower bound on the implementability of deterministic fair exchange.

The Ω implementation in the send omission model assumes only that there is at least one eventual source in the system, whereas for the receive or send-receive omission model we assume here that there is at least one eventual bisource. We conjecture that in the receive and send-receive omission models an eventual source is not enough.

Another line of future work is to make our "paper and pencil mathematics style" proofs more rigorous and verify them using machine-assisted tools. Previous and ongoing work in the area of fault-tolerant systems is very encouraging [16, 18].

Acknowledgments

Thanks to Mahir Kilic for pointing out a prior weakness in the proof of Lemma 5.

References

1. M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election (extended abstract). In *Proceedings of the 15th International Symposium on Distributed Computing*, LNCS 2180, pages 108–122. Springer-Verlag, 2001.
2. M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC: 23th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 328–337, St. Johns, Newfoundland, Canada, 2004.
3. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *22th ACM Symposium on Principles of Distributed Computing*, pages 306–314, 2003.
4. G. Avoine, F. C. Gärtner, R. Guerraoui, and M. Vukolic. Gracefully degrading fair exchange with security modules. In *In Proceedings of the 5th European Dependable Computing Conference(EDCC)*, pages 55–71, Apr. 2005.
5. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.
6. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.
7. D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Cornell University, Computer Science Department, Sept. 1996.
8. D. Dolev, R. Friedmann, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments (brief announcement). In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC97)*, 1997.
9. C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.
10. J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, Oct. 2001.
11. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
12. F. C. Freiling, M. Herlihy, and L. D. Penso. Optimal randomized fair exchange with secret shared coins. In *Proceedings of 9th International Conference on Principles of Distributed Systems (OPODIS)*, Dec. 2005.
13. S. B. Guthery. Java Card: Internet computing on a smart card. *IEEE Internet Computing*, 1(1):57–59, 1997.
14. V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, 1984. also published as Technical Report TR11-84.
15. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
16. Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing and scheduling. *ACM Transactions on Programming Languages and Systems*, 21(1):46–89, 1999.
17. A. Mostéfaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Dependable Systems and Networks (DSN)*, pages 351–360. IEEE Computer Society, 2003.
18. U. Nestmann and R. Fuzzati. Unreliable failure detectors via operational semantics. In *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference*, volume 2896

of *Lecture Notes in Computer Science*, pages 54–71, Mumbai, India, Dec. 2003. Springer-Verlag.

19. P. R. Parvédy and M. Raynal. Uniform agreement despite process omission failures. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*. IEEE Computer Society, Apr. 2003. Appears also as IRISA Technical Report Number PI-1490, November 2002.
20. K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, Mar. 1986.
21. Trusted Computing Group. Trusted computing group homepage. Internet: `https://www.trustedcomputinggroup.org/`, 2003.