Failure Detection vs Group Membership in Fault-Tolerant Distributed Systems: Hidden Trade-Offs

André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
andre.schiper@epfl.ch

Abstract. Failure detection and group membership are two important components of fault-tolerant distributed systems. Understanding their role is essential when developing efficient solutions, not only in failure-free runs, but also in runs in which processes do crash. While group membership provides consistent information about the status of processes in the system, failure detectors provide inconsistent information. This paper discusses the trade-offs related to the use of these two components, and clarifies their roles using three examples. The first example shows a case where group membership may favourably be replaced by a failure detection mechanism. The second example illustrates a case where group membership is mandatory. Finally, the third example shows a case where neither group membership nor failure detectors are needed (they may be replaced by *weak ordering* oracles).

1 Introduction

Fault-tolerance in distributed systems may be achieved by replicating critical components. Although this idea is easily understood, the implementation of replication leads to difficult algorithmic problems. A distributed algorithm requires the specification of a system model. Two main models have been proposed: the *synchronous* model and the *asynchronous* model. The synchronous model assumes (1) a known bound on the transmission delay of messages, and (2) a known bound on the relative speed of processes — while the slowest process performs one step, the fastest process performs at most k steps where k is known). The asynchronous system does not assume any bound on the transmission delay of messages, and on the relative speed of processes. Obviously, the asynchronous model is more general. If some algorithm $\mathcal{A}$ is proven correct in the most general model (e.g., in the asynchronous model), $\mathcal{A}$ is also correct in a more restricted model (e.g. in the synchronous model). Clearly, it is advantageous to develop algorithms for the most general system model.

Unfortunately, it has been proven that a very basic fault-tolerant problem, the *consensus* problem, cannot be solved by a deterministic algorithm in the asynchronous model when a single process may crash [15]. The same problem may be solved by a deterministic algorithm in the synchronous system model. However, the synchronous system model requires that bounds be defined, which leads to a dilemma. If the bounds are chosen too small, they may be violated: in this case, the algorithm might behave erroneously. If the bound is too large, this has a negative impact on the performance of the algorithm if there is a crash: the crash detection time will be long, and the algorithm will be blocked in the meantime.

Two other system models have been defined, which are between the asynchronous and the synchronous system models: the partially synchronous system model [11,14] and the asynchronous model augmented with failure detectors (which we will simply refer to as the *failure detector* model) [5].[1] Consensus is solvable in these two system models. The partially synchronous model assumes that bounds exist, but they are not known and hold only eventually. The failure detector model specifies the properties with regard to failure detection in terms of two properties: *completeness* and *accuracy*. Completeness specifies the behaviour of the failure detectors with respect to a crashed process. Accuracy specifies the behaviour of failure detectors with respect to correct processes. For example, the $\Diamond\mathcal{S}$ failure detector is defined (1) by *strong completeness* — which requires that each faulty process is eventually suspected forever by each correct process — and (2) by *eventual weak accuracy* — which requires that eventually there exists some correct process that is no longer suspected by any correct process. The failure detector model has allowed a very important result to be established: $\Diamond\mathcal{S}$ is the weakest failure detector that allows us to solve consensus [4].

The results for failure detectors, and other work performed over the last 10 years, have contributed to providing a good understanding of the algorithms related to replication, e.g., consensus, atomic broadcast, group membership. The main open problem that remains is understanding the various algorithms from a quantitative point of view. This means not only comparing the cost of these algorithms in failure-free runs, but also in runs with process crashes. For crash detection, most existing infrastructures rely on a group membership service, whereas algorithmic papers rely on failure detectors. As shown below, this has an important impact on performance, and leads to the following questions: when is a membership service really needed, and when is a failure detection mechanism preferable?

Before addressing these questions, Section 2 introduces the group membership problem, and discusses solutions to this problem. Section 3 illustrates a case where group membership can favourably be replaced by a failure detection mechanism. However, failure detection alone is not enough: Section 4 gives an example where membership is necessary. Finally, Section 5 shows that it is sometimes possible to do without failure detection and group membership. Section 6 concludes the paper.

2 The Group Membership Problem

2.1 Specification

Roughly speaking, a *group membership service* manages the formation and maintenance of a set of processes called a *group*. The successive memberships of a group are called *views*, and the event by which a new is provided to a process is called the *install* event. A process may *leave* the group as a result of an explicit leave request

[1] Other system models have been defined, e.g., [10,18].

or because it failed. Similarly, a process may *join* the group, for example to replace a process that has left the group. One distinguishes two types of group membership services: *primary-partition* and *partitionable*. Primary-partition group membership services attempt to maintain a *single* agreed view of the current membership of the group. On the contrary, partitionable group membership services allow *multiple* views of the group to coexist in order to model network partitions. In the paper we only consider the primary-partition membership service.

2.2 Solving Group Membership

Many algorithms have been proposed to solve the group membership problem. These algorithms have in common to be complex. This is the case of the protocol in [24], but there are two more recent examples. In [17] Lotem et al. describe a membership protocol that requires the introduction of notions such as quorums, sub quorums, ambiguous sessions, last formed sessions, resolution rules, learning rules. In [7], and in its recent version [20], an Atomic Broadcast algorithm is described, which is based on a ring of processes. The protocol requires a *reformation* phase if one of the processes in the ring is suspected. The reformation phase decides on the processes that form the new ring, i.e., it solves the membership problem. The authors of [20] propose a complex protocol, based here on a three-phase commit protocol.

Understanding these protocols is not easy and takes time. However, the membership problem becomes trivial using consensus, which is a well understood problem [5,25]. Consider the current membership (also called *view*) v_i and the problem of defining the next membership v_{i+1}. This is can be seen as a consensus problem to be solved among processes in v_i, where the initial value of each process is a proposal for the next view (e.g., the set of processes not suspected), and the decision is the next view v_{i+1} (see Algorithm 1) [19].[2] Algorithm 1 completely hides the complexity of group membership in the consensus black box. In [17] the authors claim that the solution based on consensus is more costly in terms of communication rounds. However, the figures given (i.e., five communication rounds) is not correct: the right number is one plus the cost of consensus, i.e., the protocol can terminate in three communication rounds. [3] It is doubtful that [17] requires less that three communication rounds.

3 Failure Suspicions Instead of Membership Exclusion

Developing complex membership protocols — instead of reducing membership to consensus — had an indirect consequence. It has hidden the benefit of decoupling “failure detection” from “membership exclusion”. While a group membership service gives a consistent information about the state of processes (correct or not), failure detection

[2] One part of the algorithm is missing here. If one correct process starts the protocol, all other processes have also to start the protocol (otherwise consensus might not terminate). This can be done using Reliable Broadcast [16].

[3] Consensus can be solved in two rounds, e.g., [25].

Algorithm 1 Solving group membership among *current-view* by reduction to consensus (code of process p)

```
1: $v_p \leftarrow current\text{-}view \setminus suspected\text{-}processes$ ;
2: $decision_p \leftarrow consensus(v_p)$ ;
3:       {execute consensus among current-view; $v_p$ is the initial value for consensus}
4: $new\text{-}view \leftarrow decision_p$ ;
```

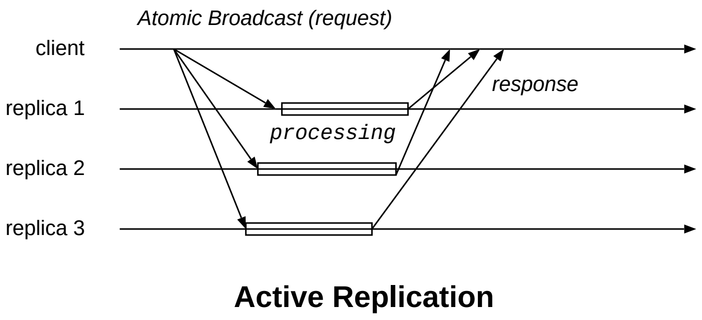
provides an inconsistent information. It may sometimes be sufficient and less costly to rely on inconsistent failure detection information rather than on consistent group membership information. Consider the following example. Let $v_i = \{p, q, r\}$ be the current view of a group (information known to p, q and r) and let p wait for a message from q. If only membership information are accessible to p, then p waits for the message until a new view v' is installed from which q is excluded. If failure detection information is accessible to p, than p waits until it suspects q to have crashed: the view is still v, other processes not necessarily suspect q, process p might later change its mind about q, and it is possible that q is never excluded from the membership. Failure detection is a lightweight service, compared to a membership service — which relies on a failure detection service. We show below a concrete example of the benefit of relying on failure suspicions instead of membership exclusion.

3.1 Replication Techniques

There exists two main classes of replication techniques that ensure strong consistency: *active* and *passive* replication (Fig. 1). Both replication techniques are useful since they have complementary features. With active replication [26], each request is processed by all replicas. This ensures a fast reaction to failures, and sometimes makes it easier to replicate legacy systems. However, active replication uses processing resources heavily and requires processing of requests to be deterministic.[4] With passive replication (also called *primary-backup* replication) [3] only one replica (the primary) processes the request, and sends update messages to the other replicas (the backups). This uses less resources than active replication does, without the requirement of operation determinism. However, passive replication is known to have a slow reaction to failures. The reason is related to failure detection. Passive replication is usually based on a group membership, which excludes the primary whenever it is suspected to have crashed [2,21].

Excluding a process from the membership has a high cost, which leads a group membership to avoid excluding processes that have not crashed. This requires a high failure detection timeout value, which leads to a slow reaction to the crash of the primary, and a high response time for the client. High response time can be prevented by decoupling failure suspicions from membership exclusion, as shown by the semi-passive replication technique.

[4] Determinism means that the result of an operation depends only on the initial state of a replica and the sequence of operations it has already performed.



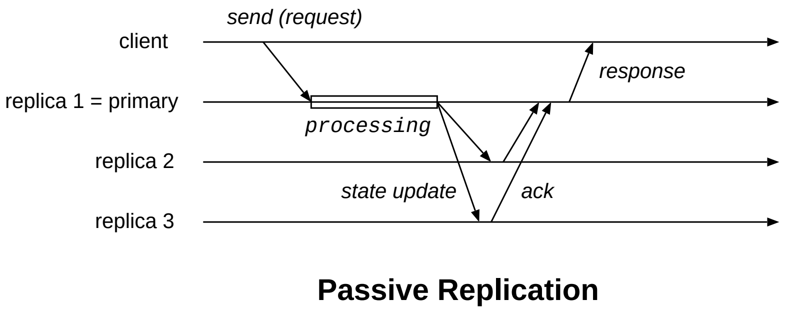


Fig. 1. Principle of active *vs* passive replication

3.2 Semi-passive Replication

Semi-passive replication [13,12] is a variant of passive replication: it retains its major characteristics (e.g., allows for non-deterministic processing). The main difference between passive and semi-passive replication is the selection of the primary. In semi-passive replication the selection of the primary is based on the rotating coordinator paradigm [7,14]; in passive replication the selection of the primary is based on a group membership service. The rotating coordinator paradigm allows the primary to be suspected without being excluded. This has a big advantage: it reduces the overhead of an incorrect suspicions. Consider the two cases: (1) the correct primary has been suspected and excluded from the membership, and (2) the correct primary has been suspected but not excluded from the membership. In case (1), in order to keep the same degree of replication, the excluded process needs to join again the membership, which leads to an new execution of the membership protocol (*join* operation), followed by the costly state transfer.[5] In case (2) no special action needs to be taken. In other words, an incorrect failure detection is costly in case (1), while it costs almost nothing in case (2). This allows in case (2) the failure detection mechanism to be much more aggressive, while in

[5] A correct process that is excluded from the membership is forced to commit suicide, and has to take a fresh copy of the state shared among the members.

case (1) it needs to be conservative. An aggressive failure detection time reduces the response time in case of the crash of the primary, i.e., corrects one of the major limitation of the class of passive replication techniques compared to the class of active replication techniques. Semi-passive also allows us to keep one of the major advantage of the class of passive replication techniques: parsimonious processing.

3.3 Semi-passive Replication and Lazy Consensus

With semi-passive replication the client sends its request to all replicas (see Fig. 2), but a single process handles the request, the primary (unless suspected). Processing the request provides the state *update* information to the primary. The primary then starts an instance of consensus to decide on the *state update value*. Upon decision, all replicas — the primary and the backups — apply the update to their current state. In other words, the initial value for consensus is a "state update value".

If all replicas need to have an initial value before starting consensus, then each replica would have to process the client request, which would be costly. To prevent this, semi-passive replication relies on a variant of consensus called *lazy consensus* [12]. With consensus, process p calls the procedure that solves consensus with its initial value v_p as a parameter. With lazy consensus, the parameter is a function called giv (which stands for *get initial value*). This function is called by p within the consensus algorithm whenever p needs an initial value. Lazy consensus is solved by a variant of the Chandra-Toueg $\Diamond\mathcal{S}$ consensus algorithm based on the rotating coordinator paradigm [5]. If the first coordinator c is not suspected, then only c calls the giv function to get the update value. In other words:

– Semi-passive replication technique leads to a sequence of lazy consensus.
– Lazy consensus is based on the rotating coordinator paradigm, where the coordinator for consensus is the primary from the point of view of the replication algorithm.
– The initial value for consensus is obtained by calling the giv function, which processes the client request.
– If the first coordinator is suspected (Fig. 2, Scenario 2), a new process takes over the coordinator role for consensus, i.e., becomes the primary from the point of view of the replication algorithm. Changing the primary does not exclude the previous primary from the membership!

To summarise, group membership is a nice abstraction, but needs be used with care. Group membership transforms failure suspicions into process exclusion. There are cases where failure suspicions should not lead to process exclusion.

4 Failure Suspicion with Membership Exclusion

The previous section has illustrated a case where failure suspicion should not lead to process exclusion. In this section we give an example of failure suspicion that requires process exclusion.

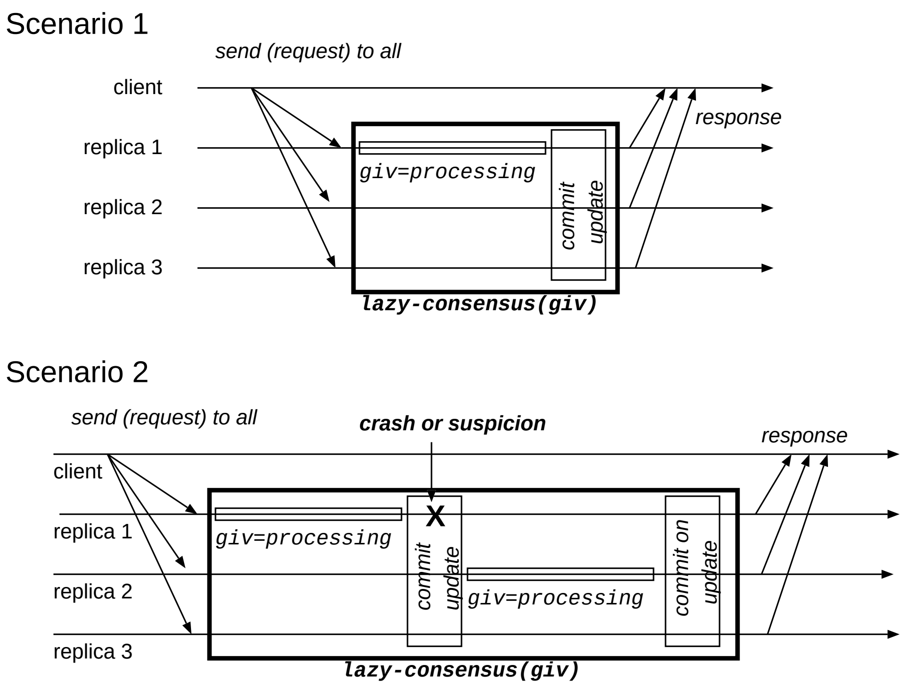


Fig. 2. Semi-passive replication. In Scenario 1, only the first coordinator (replica 1) calls the *get initial value (giv)* function. In Scenario 2, the first coordinator crashes or is wrongly suspected, replica 2 takes over the role of the coordinator, and calls the *giv* function.

4.1 Reliable Channels?

In the context of fault-tolerance, theoretical papers usually assume that channels are *reliable* — if p sends a message to q, and q is correct,[6] then q eventually receives m — or *quasi-reliable* — if p sends a message to q, and the two processes are correct, then q eventually receives m. However, real channels are neither reliable nor quasi-reliable. Lossy channels (and finite memory) lead to the exclusion of processes from the membership (see below). This explains that group membership is always considered in real systems, but is mostly absent from theoretical papers (apart from papers solving the group membership problem).[7]

[6] A correct process is a process that never crashes.

[7] This explains also the difficulty to come to a convincing specification for the group membership problem: a convincing specification requires to understand exactly when membership is needed, and when membership is not needed.

4.2 Lossy Channels and the Time-Bounded Buffering Problem

Consider the implementation of the quasi-reliable channel between p and q over fair-lossy channels.[8] Let $SEND$ and $RECEIVE$ be the primitives providing quasi-reliable communication, and $send$, $receive$ the primitives of the low-level lossy channel (Fig. 3). To execute $SEND(m)$ to q, process p copies m into an output buffer and executes $send(m)$ repeatedly until it receives an acknowledgement of m from q, denoted by $ack(m)$. The first time q receives m, it executes $RECEIVE(m)$. Each time q receives m, it sends $ack(m)$ back to p. When p receives $ack(m)$, it deletes m from its output buffer.

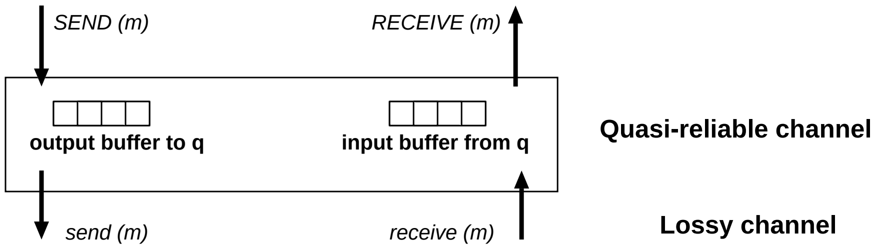


Fig. 3. Implementation of quasi-reliable channels over lossy channels

In this implementation, if q crashes, p might never receive $ack(m)$, and so might never delete m from its output buffer. This issue can be formalised by the *time-bounded buffering* problem [8]. Let m be a message in the output buffer of process p that must be sent to process q: time-bounded buffering ensures that p eventually deletes m from its output buffer. The problem cannot be solved in an asynchronous system model, neither in an asynchronous system model augmented with failure detectors of class $\mathcal{S}$ or class $\Diamond\mathcal{P}$ [5,8]. The same holds for Reliable Broadcast over fair-lossy channels. Real system overcome this impossibility by relying on *program-controlled crash* [6], which gives processes the ability to kill other processes. Consider process p with message m in its output buffer to q. If after some duration p has not received $ack(m)$ from q, it decides (1) to exclude q from the membership (i.e., to kill q), and (2) to discard m from its output buffer: as q eventually crashes, there is no obligation for q to deliver m.

There is a better solution than using timeouts to kill q. Process p kills q if, upon execution of $SEND(m)$ to q, *p's output buffer to q is full.* The murder of q is here the consequence of lack of resources, and not time-related. This is the best solution: it makes sense for p to kill q iff p has not enough space to buffer messages for q.

[8] A fair-lossy channels do not create, duplicate and garble messages, and ensure that if p sends an infinite number of messages to q, and q is correct, then q receives and infinite number of messages from p.

4.3 Process Suspicion vs Process Exclusion

In the above example, the exclusion of q from the membership is different from the exclusion of the primary r in the context of passive replication. With passive replication, if the primary r crashes at time t, then the replication service is immediately blocked. In the example of Section 4.2, if q crashes at time t, then p is blocked much later (depending on the size of its output buffer to q). This shows that when the blocking time an issue, suspecting the primary r fast is important, whereas suspecting q fast in the example of Section 4.2 is not important. In one case (primary r) the suspicion is *input-triggered*, whereas in the other case the suspicion is *output-triggered* [8]:

– *Input-triggered suspicions:* p suspects q because p waits a message from q, and its input buffer from q is empty. Process p is blocked until it suspects q.
– *Output-triggered suspicions:* p suspects q because some message remains for a long time in its output buffer to q, or because its output buffer to q is full.

There is no reason for input-triggered suspicions to lead to process exclusion: semi-passive replication is a good example. On the other hand, if the output buffer of p to q is full, p does not have many options: block or exclude q. In other words, input triggered suspicions *should never* lead to exclusion, while output-triggered suspicions *should always* lead to exclusion. This gives a clean picture. A group membership service should not react to input-triggered suspicions, but only to output-triggered suspicions.

Most existing systems handle input-triggered and output-triggered suspicions in the same way, i.e., by excluding processes. This is a poor choice from a performance point of view. If p waits a message from q, the timeout to detect the crash of q should be short (in order to reduce the blocking period). If p sends a message to q, there is no need to detect the crash of q quickly. A single failure detection mechanism, requires a compromise: the suspicion should not be too fast (do avoid too many wrong suspicions) and not too slow (do reduce the blocking time). Having input-triggered suspicions (without exclusions) on one hand side, and output-triggered exclusions on the other hand side allows to escape from the dilemma.

4.4 Input/Output-Triggered Suspicions vs Partitionable Membership

The distinction between input and output triggered suspicions is orthogonal to the distinction between primary partition and partitionable membership. Partitionable membership [9] does not distinguish between input-triggered and output-triggered suspicions: it relies on one single failure detection mechanism.

The difference between (1) input and output-triggered suspicions in the context of primary partition membership, and (2) partitionable membership can be clarified on the following example. Consider a system of five processes (Fig. 4): two client processes c_1, c_2, and three server processes s_1, s_2, s_3 (which implement semi-passive replication). Consider the following scenario, where the exclusion of the servers s_i is output-triggered:

– At time t_0 all processes are reachable from all processes. The server membership is $\{s_1, s_2, s_3\}$. No process is suspected, and all client requests are handled by the s_1. The server s_1 broadcasts the “update” message (Sect. 3) to s_2 and s_3.

- At time t_1, a link failure occurs, which partitions the system in two components: $\Pi_1 = \{c_2, s_1\}$ and $\Pi_2 = \{c_1, s_2, s_3\}$: processes s_2, s_3 suspect s_1 (and s_1 suspects probably s_2 and s_3). The requests of c_1 are handled by another server, say s_2. The server s_2 broadcasts the update message to s_1 and s_3. If the output buffer of s_2 to s_1 is large enough, s_1 is not excluded.
- At time t_2, the link failure is repaired. No special action needs to be taken. The update sent by s_2 to s_1 during the link failure have been buffered, and can now reach s_1. The occurrence of the partition is transparent to the servers.

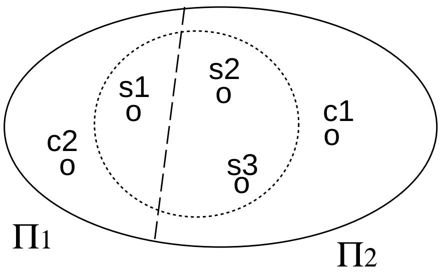


Fig. 4. Partition of the processes in two components Π_1 and Π_2

With partitionable membership, the link failure is not transparent. In the interval $[t_1, t_2]$ a partitionable membership defines two concurrent views. Once the link failure is repaired, the states of the two partitions would have to be merged (at the application level).

5 Doing without Failure Suspicion

In Section 3 we have shown the benefit of decoupling failure suspicion from membership exclusion. In Section 4 we have introduced the distinction between input and output-triggered suspicions, and explained the relationship between output-triggered suspicions and membership exclusion. We address now the following question: can input-triggered suspicions be avoided? Indeed the following dilemma remains: what timeout value should be chosen for input-triggered suspicions? On one hand the timeout value should be large in order to avoid wrong suspicions (they still have a negative impact on performance[9]). On the other hand timeout values should be small in order to ensure fast reaction to failures. Avoiding input-triggered suspicions avoids the dilemma of fine tuning the failure detection mechanism.

Randomisation is one solution, since it allows us to solve consensus in asynchronous systems [1,23]. However, randomisation leads to slow algorithms. Another solution is to augment the asynchronous system with *weak ordering oracles* [22], which order

[9] Even if wrong suspicions have less impact on performance than process exclusion, they still have a negative impact.

messages frequently, but might also deliver messages out of order. Weak ordering oracles capture the behaviour of network multicast in state-of-the-art local area networks: if messages are multicast in a local area network, there is a good chance that some of them will be received by all processors in the same order. Experiments have shown that if the interval between broadcast is around 0.15 ms, then very few messages are received out of order (about 5%) [22].

5.1 Weak Ordering Oracles

Weak ordering oracles are defined by two primitives, *W-ABroadcast(r,m)* and *W-ADeliver(r,m)*. The first primitive asks the oracle to broadcast m. The second primitive corresponds to the delivery of m by the oracle. The parameter r groups messages with the same r value (they can be seen as *round* numbers). The weak ordering property holds for round r if there exists some message m such that all processes deliver m before the other messages of round r. To illustrate this property, consider three processes p_1, p_2, p_3 executing the following queries to the oracle:

– p_1 executes W-ABroadcast$(0, m_1)$; W-ABroadcast$(1, m_2)$; W-ABroadcast$(2, m_3)$
– p_2 executes W-ABroadcast$(0, m_4)$; W-ABroadcast$(1, m_5)$; W-ABroadcast$(2, m_6)$
– p_3 executes W-ABroadcast$(0, m_7)$; W-ABroadcast$(1, m_8)$; W-ABroadcast$(2, m_9)$

and assume the following sequences of W-ADeliver(r, m) (for brevity, we denote next W-ADeliver(r, m) by D(r, m)):

– on p_1: D$(0, m_1)$; D$(1, m_2)$; D$(0, m_4)$; D$(2, m_3)$; D$(0, m_7)$; $\ldots$
– on p_2: D$(0, m_4)$; D$(0, m_1)$; D$(1, m_5)$; D$(0, m_7)$; D$(2, m_3)$; $\ldots$
– on p_3: D$(0, m_4)$; D$(0, m_7)$; D$(2, m_3)$; D$(1, m_8)$; $\ldots$

The weak ordering property holds for round $r = 2$ (m_3 is the first message with $r = 2$ delivered by p_1, p_2, p_3), but does not hold for either $r = 0$ or $r = 1$. The oracle can make mistake: it does not have to satisfy the weak ordering property for all rounds.

The definition of the oracle assumes that each process executes W-ABroadcast(r, m) sequentially for rounds $r = 1, 2, \ldots$. Let $first_p(r)$ denote the first message of round r delivered by p. The *k-Weak Atomic Broadcast (or k-WAB)* Oracle is defined by the following properties:

– **Validity:** If a correct process executes W-ABroadcast(r, m), then all correct processes eventually execute W-ADeliver(r, m).
– **Uniform Integrity:** For every par (r, m), W-ADeliver(r, m) is executed at most once, and only if W-ABroadcast(r, m) was previously executed.
– **Eventual Uniform k-Order:** If all processes execute an infinite sequence of W-ABroadcast(r, m), for $r = 1, 2, \ldots$, then there exist k values $r_1, \ldots, r_k$ such that, for all $i \in [1, k]$ and all processes p, q, we have $first_p(r_i) = first_q(r_i)$.

Algorithm 2 Ben-Or binary consensus algorithm: code of process p $(f < n/2)$

```
1: Consensus ($initVal$):

2:    $estimate_p \leftarrow initVal$
3:    $decided \leftarrow false$
4:    $r_p \leftarrow 0$

5: while $true$ do

6:    send $(\text{FIRST}, r_p, estimate_p)$ to all
7:    wait until received $(\text{FIRST}, r_p, v)$ from $n - f$ processes
8:    if $\exists\ v$ s.t. received $(\text{FIRST}, r_p, v)$ from $n - f$ processes then
9:       $estimate_p \leftarrow v$
10:   else
11:      $estimate_p \leftarrow \perp$

12:   send $(\text{SECOND}, r_p, estimate_p)$ to all
13:   wait until received $(\text{SECOND}, r_p, v)$ from $n - f$ processes
14:   if not $decided_p$ and ($\exists\ v \neq \perp$ s.t. received $(second,\ r_p, v)$ from $f + 1$ processes) then
15:      decide $v$          {continue the algorithm after the decision}
16:      $decided_p \leftarrow true$
17:   if $\exists\ v \neq \perp$ s.t. received $(\text{SECOND}, r_p, v)$ then
18:      $estimate_p \leftarrow v$
19:   else
20:      $estimate_p \leftarrow coin()$          {toss the coin}

21:   $r_p \leftarrow r_p + 1$
```

5.2 Solving Consensus with 1-WAB Oracles

Consensus can be solved in an asynchronous system augmented with a 1-WAB oracle [22]. The algorithm is inspired by Ben-Or's randomised binary consensus algorithm [1] (Algorithm 2). Ben-Or's algorithm executes a sequence of rounds, where each round has two phases (n is the number of processes, $f < n/2$ is the maximum number of processes that can crash):

– *Phase I, lines 6-11*: process p sends its current $estimate_p$ (0 or 1) of the decision value to all, and waits to receive the estimate value from $n - f$ processes. If the same value v is received from $n - f$ processes, then $estimate_p$ is updated to the value received, else $estimate_p$ is set to $\perp$.
– *Phase II, lines 12-20*: process p sends again its $estimate_p$ to all, and waits to receive the estimate value from $n - f$ processes. If the same value v is received from $f + 1$ processes, then p decides v. If some value different from $\perp$ is received, then $estimate_p$ is set to v, otherwise $estimate_p$ is updated with random value (0 or 1) .

Algorithm 3 is the 1-WAB consensus algorithm. Lines 6-22 of Algorithm 2 are identical to lines 6-19 of Algorithm 3. Line 20 in Algorithm 2 (random coin toss) is replaces with line 23 in Algorithm 3. Lines 6-8 in Algorithm 3 are new (query and response of the oracle): the weak ordering oracle replaces the coin toss. It is interesting to note that, contrary to Ben-Or's algorithm which solves only the binary consensus problem, the 1-WAB consensus algorithm solves the non-binary consensus problem. Proving that the Algorithm 3 satisfies the safety properties of consensus is not very different from the proof of Ben-Or's algorithm. Proving termination relies on the eventual uniform 1-WAB property of the 1-WAB oracle.

Algorithm 3 Consensus alg. using 1-WAB oracles: code of process p $(f < n/2)$

```
 1: Consensus ($initVal$):

 2:    $estimate_p \leftarrow initVal$
 3:    $decided \leftarrow false$
 4:    $r_p \leftarrow 0$

 5: **while** $true$ **do**

 6:    W-ABroadcast$(r_p, estimate_p)$
 7:    **wait until** W-ADeliver of the first message $(r_p, v)$
 8:    $estimate_p \leftarrow v$

 9:    send $(\text{FIRST}, r_p, estimate_p)$ to all
10:    **wait until** received $(\text{FIRST}, r_p, v)$ from $n - f$ processes
11:    **if** $\exists\, v$ s.t. received $(\text{FIRST}, r_p, v)$ from $n - f$ processes **then**
12:       $estimate_p \leftarrow v$
13:    **else**
14:       $estimate_p \leftarrow \perp$

15:    send $(\text{SECOND}, r_p, estimate_p)$ to all
16:    **wait until** received $(\text{SECOND}, r_p, v)$ from $n - f$ processes
17:    **if not** $decided_p$ **and** ($\exists\, \overline{v} \neq \perp$ s.t. received $(second,\ r_p, \overline{v})$ from $f + 1$ processes) **then**
18:       decide $\overline{v}$                    {*continue the algorithm after the decision*}
19:       $decided_p \leftarrow true$
20:    **if** $\exists\, \overline{v} \neq \perp$ s.t. received $(\text{SECOND}, r_p, \overline{v})$ **then**
21:       $estimate_p \leftarrow \overline{v}$
22:    **else**
23:       $estimate_p \leftarrow initVal$

24:    $r_p \leftarrow r_p + 1$
```

5.3 Solving Atomic Broadcast with WAB Oracles

A Weak Atomic Oracle (or WAB Oracle) is a k-WAB Oracle where $k = \infty$. The 1-WAB consensus algorithm can be extended to an atomic broadcast algorithm, which requires a WAB oracle [22]. Contrary to the classical solution, in which atomic broadcast is reduced to consensus [5], the solution directly relies on the oracle. This shows that consensus can sometimes be bypassed. Moreover, contrary to failure detection based solution, the algorithm does not suffer from the failure detection dilemma. There is no timeout to tune, and no notion of *reaction time to failures*. The performance is as good in the presence of failures as in the absence of failures.

6 Conclusion

While fault-tolerant distributed algorithms in the context of replication are nowadays well understood, the important trade-off related to the reaction to process failures has not attracted the attention that it deserves. The trade-off is between (1) fast reaction to crashes, and (2) infrequent wrong failure suspicions. We have seen how to escape from this trade-off using semi-passive replication (which relies on consensus) rather than passive replication (which relies on group membership). More generally, we have seen how to escape from this trade-off by distinguishing on one hand input-triggered suspicions that do not lead to process exclusions, and on the other hand output-triggered suspicions that lead to process exclusion. Finally, while the timeout trade-off remains for input-triggered suspicions, we have seen that it can be avoided by using weak ordering oracles instead of failure detectors.

These trade-offs are important in the context of quantitative evaluation of consensus and atomic broadcast algorithms, and more generally of group communication algorithms. Such evaluations represent an important challenge. Some preliminary results have been obtained, but much more needs to be done. Understanding group communication algorithms from a quantitative point of view is mandatory, before considering that group communication is a solved problem.

Acknowledgements

I would like to thank Bernadette Charron-Bost, Xavier Défago, Fernando Pedone, Péter Urbán, Matthias Wiesmann, David Cavin and Luiz Barchet Estefanel for their contributions to the results presented in the paper.

References

1. M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *proc. 2nd annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.
2. K.P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.
3. N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The Primary-Backup Approach. In Sape Mullender, editor, *Distributed Systems*, pages 199–216. ACM Press, 1993.
4. T.D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of ACM*, 43(4):685–722, 1996.
5. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

6. Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, Pennsylvania, USA, May 1996.
7. J.M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, August 1984.
8. B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting Messages in Fault-Tolerant Distributed Systems: the benefit of handling input-triggered and output-triggered suspicions differently. TR IC/2002/020, EPFL, May 2002.
9. G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *Computing Surveys*, 4(33):1–43, December 2001.
10. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel & Distributed Systems*, 10(6):642–657, June 1999.
11. D.Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of ACM*, 34(1):77–97, January 1987.
12. X. Défago and A. Schiper. Specification of Replication Techniques, Semi-Passive Replication and Lazy Consensus. TR IC/2002/007, EPFL, February 2002.
13. X. Défago, A. Schiper, and N. Sergent. Semi-passive Replication. In *17th IEEE Symp. on Reliable Distributed Systems (SRDS-17)*, pages 43–58, West Lafayette, USA, October 1998.
14. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.
15. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
16. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.
17. E.Y. Lotem, I. Keidar, and D. Dolev. Dynamic Voting for Consistent Components. In *Proc. 17th Annual ACM Symposium on Principles of Distributed Computing (PODC-97)*, 1997.
18. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
19. C. Malloth and A. Schiper. View Synchronous Communication in Large Scale Networks. In *ESPRIT Basic Research BROADCAST, Third Year Report, Vol 4*, July 1995.
20. N.F. Maxemchuk and D.H. Shur. An Internet multicast system for the stock market. *ACM Trans. on Computer Systems*, 19(3):384–412, August 2001.
21. L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, C.A. Lingley-Papadopoulis, and T.P. Archambaud. The Totem system. In *IEEE 25th Int Symp on Fault-Tolerant Computing (FTCS-25)*, pages 61–66, 1995.
22. F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. TR IC/2002/010, EPFL, March 2002. Appears also as Technical Report HPL-2002-44, Hewlett-Packard Laboratories, March 2002.
23. M. Rabin. Randomized Byzantine Generals. In *Proc. 24th Annual ACM Symposium on Foundations of Computer Science*, pages 403–409, 1983.
24. A.M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.
25. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.
26. F.B. Schneider. Replication Management using the State-Machine Approach. In Sape Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.