Time $vs.$ Space in Fault-Tolerant Distributed Systems

Bernadette Charron-Bost* charron@lix.polytechnique.fr
Xavier Défago† defago@jaist.ac.jp
André Schiper‡ andre.schiper@epfl.ch

*LIX, École Polytechnique, 91128 Palaiseau Cedex, France
†Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan
‡École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland

Abstract

Algorithms for solving agreement problems can be classified in two categories: (1) those relying on failure detectors that we call FD-based*, and (2) those that rely on a Group Membership Service that we call* GMS-based*. The paper discusses the advantages and limitations of these two approaches, and proposes an extension to the GMS-approach that combines the advantages of both approaches, without their drawbacks. This extension leads us to distinguish between* time-triggered *suspicions of processes and* space-triggered *exclusions.*

1 Introduction

The detection of failures necessarily plays a central role in the engineering of dependable systems. This is especially true in the context of time-critical applications, where the occurrence of a failure is not masked if it may result in long blackout periods during which the system blocks. In most legacy distributed real-time systems, the detection of failure uses time and a communication infrastructure with a fully deterministic behavior. This requires to build the systems on dedicated ad-hoc hardware, which has a significant impact on the overall cost of the solution. In order to reduce this cost, whenever possible, real-time distributed systems (with soft real-time constraints) are now implemented using cheaper commodity hardware and software. As a result, the communication infrastructure becomes less predictable (e.g., transmission delays are not guaranteed, messages may be lost), and a perfect detection of failures is no more possible.

Fault-tolerant services in a distributed system are usually implemented using replication. There are two basic approaches to replication: active and passive replication. In active replication (also called the state machine approach) [12], the requests are processed by all server replicas. This technique requires that all requests are processed in the same order, and hence relies on *Atomic Broadcast*. Basically, Atomic Broadcast can be solved either using (1) unreliable failure detectors [5] (called *FD-based* solutions hereafter), or (2) a group membership service (called *GMS-based* solution). While GMS-based solutions force the crash of processes that have been incorrectly suspected, this is not the case with FD-based solutions. As discussed below, this difference has an important consequence: the overhead due to an incorrect suspicion is considerably higher in the former case than in the latter. The Atomic Broadcast algorithm proposed by Chandra and Toueg [5] is an example of FD-based algorithms, while the Isis sequencer-based algorithm [3] is an example of GMS-based algorithms. A survey presenting many other examples of both categories can be found in [7].

With passive replication (also called *primary-backup* replication) [4, 9], only one replica (the primary) processes each request and sends update messages to the other replicas to update their state. Classical implementations of passive replication are all GMS-based. Semi-passive replication [8] defines a variant of passive replication that can be implemented using an FD-based algorithm.

As mentioned earlier, the overhead due to an incor-

 Appeared in *Proc. of the* 6^{th} *Int'l Workshop on Object-oriented Real-Time Dependable Systems (WORDS'01).*

rect suspicion is considerably higher in GMS-based algorithms than in FD-based ones. This allows the failure detection timeout of FD-based algorithms to be more aggressive, compared to GMS-based solutions. In a LAN, this typically means a timeout for FD-based solutions in the order of 100 ms, compared to a timeout for GMS-based solutions in the order of 30 seconds. The difference is huge for real-time applications, as the timeout value has a direct impact on the fail-over time:[1] in the order of 100 ms in the first case and in the order of 30 seconds in the second case.

The goal of this paper is to clarify the advantages and the limitations of these two classes of approaches—FD-based $vs.$ GMS-based—and to propose an extension to the GMS-based approach. More precisely, the paper introduces a new communication primitive in the context of View Synchronous Communication.[2] This new primitive allows us to introduce the distinction between time-triggered suspicions and space-triggered exclusions. From a practical point of view, this leads to a more powerful paradigm that combines the advantages of GMS-based and FD-based approaches, without having the drawbacks of FD-based approaches (see below, Section 4).

The rest of the paper is organized as follows. Section 2 presents the system model. Section 3 introduces the distinction between genuine and provoked failures, and between short-term and long-term requirements in the context of failure detection. Section 4 explains the limitations of FD-based approaches. Section 5 is devoted to the GMS-based approach, and discusses the cost of an incorrect failure suspicion. Section 6 defines a new broadcast primitive in the context of View Synchronous Communication. Section 7 shows the benefit of the new broadcast primitive. Section 8 illustrates the use of the new broadcast primitive on an example. Finally Section 9 concludes the paper.

[1]The fail-over time is the time elapsed between the crash of a process (t_1) and the time at which the system has recovered from the crash (t_2). During the interval $[t_1, t_2]$ the system is not operational.

[2]*View Synchronous Communication* is a paradigm that consists of a *Group Membership Service*, together with the *View Synchronous Communication* primitive.

2 System model

We consider an asynchronous distributed system where processes communicate by message passing. Processes are only subject to crash failures (no Byzantine failures). A crashed process, if it recovers, takes a new identity (so formally, a crashed process never recovers). A correct process is a process that never crashes. Moreover, we assume *program controlled crash* [6], which is the ability given to processes to kill other processes or to commit suicide.

Channels are *fair-lossy*, i.e., defined by the no-creation and no-duplication properties, together with the following *fair-loss* property: if process p sends an infinite number of messages to process q, and q is correct, then q receives an infinite number of messages from p. Channels that are considered in the paper are neither *reliable*[3] nor *quasi-reliable*.[4] The reason for this choice is that real channels are neither reliable, nor quasi-reliable. This leads us to consider the problem of implementing reliable communications over unreliable channels.

Since we consider in the paper agreement problems, which are not solvable in an asynchronous system, we extend our asynchronous system model with failure detectors that make agreement problems (e.g., consensus) solvable [5].

3 Important dichotomies

3.1 Genuine $vs.$ provoked failures

We distinguish between two kinds of crash failures: genuine and provoked failures. *Genuine failures* are failures that naturally occur in the system, without any intervention from a process. Conversely, *provoked failures* are failures that are deliberately caused by some process (murder or suicide). Provoked failures are considered in the *program controlled crash*

[3]A *reliable channel* [2] between processes two p and q is a channel that neither duplicates messages nor creates spurious ones, and that satisfies the following property: (No Loss) if p sends a message m to q, and q is correct, then q eventually receives m.

[4]A *quasi-reliable channel* [1] between two processes p and q is a reliable channel in which the "No Loss" property is replaced by the following weaker property: (Quasi-No Loss) if p sends a message m to q, and p and q are both correct, then q eventually receives m.

model.

A fault-tolerant algorithm typically tolerates the crash of a bounded number of processes. So each provoked failure actually *decreases* the number of genuine failures that can be tolerated. In other words, program controlled crash reduces the actual fault-tolerance of the system. Provoked failures can occur in GMS-based solutions, but are excluded in FD-based solutions.

3.2 Short-term $vs.$ long-term requirements

In fault-tolerant systems, failures can have different consequences, and these consequences have implication on the failure detection requirements. In some cases, the detection of a failure is a *short-term* requirement, while in other cases it is a *long-term* requirement. We say that the detection of failure is a short-term requirement when the crashed process blocks the system at least until the crash is detected. Conversely, if the crashed process does not prevent the system from functioning, then failure detection is a long-term requirement.

In practice, it is important to distinguish between short-term and long-term requirements. Indeed, in the first case, failures must be detected in a timely manner even though this may lead to mistakes (i.e., wrong failure detection). Conversely, in the case of long-term requirements, the detection of failures is not urgent. We come back to this issue below.

4 Limitations of FD-based solutions

FD-based solutions do not rely on provoked crashes. However, to the best of our knowledge, all FD-based solutions to agreement problems described in the literature assume either reliable or quasi-reliable channels. While reliable channels are adequate for proving impossibility results, reliable or quasi-reliable channels inadequately model real network links. Moreover, implementing quasi-reliable channels over lossy channels require either infinite storage[5] or a perfect failure detector. Indeed, consider the channel between p and q. Process p, upon sending message m to q, must buffer m until receiving an acknowledgement $ack(m)$ from q. As long as no $ack(m)$ is received, p retransmits m from time to time. If no $ack(m)$ is received, p must buffer m forever, unless a perfect failure detector notifies p of the crash of q.

In most environments, both assumptions (infinite storage or perfect failure detector) are unrealistic. This explains that systems that have been built have considered GMS-based approaches, and not FD-based approaches.

[5]We use the term “storage” with the meaning *act of keeping* and not *place for keeping*.

5 GMS-based solutions: specification of VSC

Contrary to FD-based solutions, GMS-based solutions do not require neither a perfect failure detector, nor infinite storage. The View Synchronous Communication paradigm [13] consists of two parts: (1) a Group Membership Service, together with (2) the View Synchronous Communication primitive. We start with the specification of the Group Membership Service.

5.1 Group membership service

The Group Membership Service provides the notion of *views* to processes. A view v is a pair $(viewId, members)$, where $viewId$ is the id of the view, and $members$ is the set of processes that are part of the view v. The view v of p at time t should ideally be the set of processes that p considers to be alive at time t. The view evolves when processes crash or leave the system, or when new processes join the system. The literature distinguishes between two flavors for the membership service: the *primary partition* membership service and the *partitionable* membership service. Roughly speaking, the primary partition membership service defines a totally ordered sequence of views, while with a partitionable membership service views are only partially ordered (concurrent views can coexist). Here, we concentrate only on the primary partition membership service. The event by which a new view is provided to a process is called the *install* event. The primary partition membership is specified by the following agreement property [3, 13]:

Agreement on the View History: Consider two processes p, q. Let v_i^p denote the view $\#i$ of p, and v_i^q

the view $\#i$ of q. Agreement on the view history holds if for all i, if p installs v_i^p and if q installs v_i^q, then $v_i^p = v_i^q$.

This property allows us to denote a view simply by v_i, without mentioning the process superscript.

The membership service also satisfies additional properties that can be summarized as follows: (1) a crashed process is eventually removed from the membership (*completeness* property), and (2) if a process p belongs to a view v_i but not to the view v_{i+1}, and p has not asked to leave the group, then p is not correct. The second property can trivially be ensured in a model with program controlled crash as follows: whenever a process p that belongs to v_i detects that it does not belong to v_{i+1}, it commits suicide (i.e., crashes).

The membership service consists also of a *join* operation allowing the addition of a process to the view, and of a *leave* operation allowing the removal of a process from the view.

5.2 View Synchronous Communication (VSC)

The multicast service allows processes to broadcast messages to the members of their current view with certain guarantees. Let us denote by $\text{BROADCAST}^v(m)$ the primitive by which m is broadcast by a process in view v, and by $\text{DELIVER}^v(m)$ the primitive by which m is delivered by a process in view v. The view superscript is sometimes omitted, if not relevant. We consider the following specification for the multicast service (see also [11, 13]):

Termination: If a correct process executes $\text{BROADCAST}^v(m)$, then (1) every process in the view v eventually executes $\text{DELIVER}^v(m)$, or (2) every correct process in v eventually installs a new view.

View synchrony: If process p belongs to two consecutive views v and v', and has executed $\text{DELIVER}^v(m)$, then every process $q \in v \cap v'$ executes $\text{DELIVER}^v(m)$, i.e., delivers m before installing v'.

Sending View Delivery: A message broadcast in view v, can only be delivered in view v. In other words, if $\text{DELIVER}^v(m)$ and $\text{BROADCAST}^{v'}(m)$ occur, then $v = v'$.

5.3 Discussion

Interestingly, the implementation of the VSC paradigm over lossy channels requires neither a perfect failure detector nor infinite buffer space. This can be explained as follows. Let p execute BROADCAST^v. Every process $q \in v$, upon reception of m, buffers m and sends $ack(m)$ to all processes in v. As long as q has not received $ack(m)$ from some process r, q retransmits m to r from time to time. Once $q \in v$ has received $ack(m)$ from all processes, it can discard m. However, assume that q never receives $ack(m)$ from r. By the assumption of fair-lossy channels, r must have crashed. By the Completeness property of the membership service, eventually a view that does not contain r is installed by q. At that point, q can discard m without violating the Termination property.

Unfortunately, the exclusion of r can lead to a new problem: the exclusion of a correct process. Indeed, if r is just slow, we end up with the exclusion of a correct process, and a process that has been excluded is forced to crash. Thus the cost of incorrect suspicions in the VSC paradigm is high.[6] From a practical point of view, this means that incorrect suspicions should be avoided as much as possible, e.g., by considering a conservative timeout value in the implementation of the failure detection mechanism. Unfortunately this leads to a high fail-over time. The GMS paradigm fails to take into account the short-term/long-term dichotomy (Sect. 3.2).

6 Extending VSC: Terminating Broadcast

To avoid the problem of high fail-over time we propose to extend the VSC paradigm with a new broadcast primitive, denoted by T-BROADCAST, where T stands for *Terminating*.[7] Roughly speaking, contrary to $\text{BROADCAST}(m)$, $\text{T-BROADCAST}(m)$ does not require the delivery of m or the installation of a new view. A third option exists, which consists in delivering a value denoted by S_p (S stands for *suspicion*, and

[6] To keep the same degree of replication, the crashed process must be replaced by a new process that must execute the *join* operation.

[7] The primitive has some similarities with *Terminating Reliable Broadcast*, or *TRB* [10]. However, since the two primitives are not identical, we have deliberately avoided using the same name.

p is the sender). Formally, T-BROADCAST(m) is defined by the same properties as BROADCAST(m), with the Termination property replaced by the following *T-Termination* property:

T-Termination: If a correct process p executes T-BROADCAST$^v(m)$, then (1) every process in the view v eventually execute T-DELIVER$^v(m)$, or (2) every process in the view v eventually execute T-DELIVER$^v(S_p)$, or (3) every correct process in v eventually installs a new view.

The T-Termination property requires the processes to agree on the delivery of m or S_p. The delivery of S_p does not lead to the exclusion of p. We have thus two broadcast primitives in the extended VSC paradigm: BROADCAST and T-BROADCAST.

7 Time-triggered suspicions $vs.$ space-triggered exclusions

Before illustrating the use of BROADCAST $vs.$ T-BROADCAST, we show in this section the benefit of T-BROADCAST over BROADCAST.

Consider $q \in v_i$ awaiting message m that $p \in v_i$ is supposed to BROADCAST. Process q knows that it will never deliver m upon installation of view v_{i+1}, from which p is necessarily excluded.[8] The exclusion of p from view v_{i+1} forces p to crash. In other words, the following two events are coupled: (1) the decision of no-delivery of m, and (2) the forced crash of p.

Instead of BROADCAST$^{v_i}(m)$, consider that p is supposed to execute T-BROADCAST$^{v_i}(m)$, let q await m, and assume that p crashes before executing T-BROADCAST$^{v_i}(m)$. Process q knows that it will never deliver m upon delivery of S_p. However, if p is incorrectly suspected, the delivery of S_p does not force p to crash. So the two events (1) decision of no-delivery of m, and (2) forced crash of p, are now decoupled. Decoupling these two events has one big advantage. It allows the decoupling of the two corresponding policies: (1) the policy related to the decision of no-delivery of m, from (2) the policy related to the decision to exclude p (or some other process). Moreover, it is important to understand that (1) is a policy about the *sender* of a message, while (2) is a policy about a *destination* process. In fault-tolerant distributed applications, a policy about a sender process has usually an impact on the fail-over time, while a policy about a destination process has usually no impact on the fail-over time. Therefore, decoupling these two policies seems totally natural.

We propose the following two policies: (i) *suspicion based on timeouts* for the decision of no-delivery of m, and (ii) *exclusion based on space constraints* for the exclusion of a destination process. Policy (i) is a standard *time-triggered* policy, and the timeout can be aggressive, considering that a false suspicion does not lead to an exclusion. Moreover, an aggressive timeout with respect to a sender process is usually important for a small fail-over time.

The exclusion policy (ii) is a new policy, based on the following observation. Consider T-BROADCAST$^{v_i}(m)$ executed by p, and the decision to deliver m. The Termination property forces m to be buffered until it is known that all processes in v_i have delivered m, or that some process q that may not have delivered m is excluded. How long should m be buffered? It seems stupid to discard m if memory space is available at p. The natural policy is thus to buffer m as long as the memory resources allow to do so. However, as soon as p runs out of buffer space, the exclusion of q allows to regain memory resources. This is what we call *space-triggered exclusion*: the decision to exclude q from v_i is taken by some process p as soon as p's memory space allocated to messages for q is full.

Note that the time elapsed between the execution of T-BROADCAST$^{v_i}(m)$ and the space-trigger exclusion of some process in v_i can be arbitrary long without any impact on the fail-over time of the application. This is because the space-triggered policy is about a *destination* process. On the contrary, the time elapsed between the execution of T-BROADCAST$^{v_i}(m)$ by p, and the time-triggered decision to deliver S_p rather than m, has usually an impact on the fail-over time of the application. This is because this policy is about a *sender* process. So, while T-BROADCAST allows us to separate the policies related to the sender process from

[8] If p is correct, it must have executed BROADCAST$^{v_i}(m)$, and so DELIVER$^{v_i}(m)$. By the View synchrony property if $p \in v_{i+1}$ delivers m, then all processes in $v_i \cap v_{i+1}$, including q, must have deliver m in view v_i. So p cannot be in v_{i+1}.

those related to the destination processes, this is not the case with the BROADCAST primitive.

8 Illustration of BROADCAST and T-BROADCAST

We illustrate in this section the use of BROADCAST and T-BROADCAST on a sequencer based Atomic Broadcast algorithm. Atomic Broadcast is defined by the two primitives A-BROADCAST and A-DELIVER. The role of the sequencer is to attach a sequence number $sn(m)$ to every message m. Messages are delivered in the order of the sequence number.

To determine the sequencer process, the run of the algorithm is decomposed in *epochs*. Every process p has an integer variable k_p, that represents its current epoch. The epoch changes whenever the current sequencer seq is suspected. The epoch variable, together with the current view of p, defines the process that p considers to be the sequencer process. The algorithm can be sketched as follows:

- Upon execution of A-BROADCAST(m) by process p, p executes BROADCAST$^v(m)$.

- Upon DELIVERY of m, the sequencer process s executes T-BROADCAST$^v(id(m), sn(m))$, where $id(m)$ is the identifier of m, and $sn(m)$ an integer that is the sequencer number attached to m.

- Upon T-DELIVERY of m and $(id(m), sn(m))$, every process A-DELIVERS m once all messages m' such that $sn(m') < sn(m)$ have been A-DELIVERED.

- Upon T-DELIVERY of S_{seq}, where seq is the sequencer, every process p changes epoch by executing $k_p \leftarrow k_p + 1$.

Notice the use of BROADCAST $vs.$ T-BROADCAST: the message m is BROADCAST, while the sequence number $sn(m)$ is T-BROADCAST. To ensure a short fail-over time, the time-triggered suspicion related to T-BROADCAST must be aggressive. However, the space-triggered suspicion related to T-BROADCAST or BROADCAST has no impact on fail-over time.

9 Conclusion

The paper has extended the Virtually Synchronous Communication paradigm with the Terminating Broadcast primitive. This makes it possible to adequately decouple two antagonistic aspects of failure detection: fast reconfiguration for a short fail-over time, and slow exclusion of failed processes for housekeeping. The reconfiguration relies on the Terminating Broadcast primitive which uses an aggressive time-triggered mechanism to *suspect* processes, thus guaranteeing a short fail-over in case of failure. Conversely, housekeeping issues are satisfied by relying on the (not time-critical) exclusion of crashed processes. Moreover, this exclusion is not triggered by timeouts, but rather by the lack of buffer space for keeping unacknowledged messages.

This result is important because it shows that it is possible to combine the positive aspects of both FD-based and GMS-based systems in order to build fault-tolerant distributed systems that do not block for long periods in the event of a process crash.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 126–140, Saarbrücken, Germany, September 1997.

[2] A. Basu, B. Charron-Bost, and S. Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, USA, September 1996.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.

[4] 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.

[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] X. Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, Federal Institute of Technology, Lausanne (EPFL), 2000.

[8] X. Défago, A. Schiper, and N. Sergent. Semi-passive R eplication. In *17th IEEE Symp. on Reliable Distributed Systems (SRDS-17)*, pages 43–58, West Lafayette, USA, October 1998.

[9] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[10] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[11] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 1993.

[12] F.B. Schneider. Replication Management using the State-Machine Approach. In Sape Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.

[13] R. Vitenberg, I. Keidar, G.V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical report, Dept. of Computer Science, Technion, Israel, 99.