Broadcasting Messages in Fault-Tolerant Distributed Systems: the benefit of handling input-triggered and output-triggered suspicions differently

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

This paper investigates the two main and seemingly antagonistic approaches to broadcasting messages in fault-tolerant distributed systems: the approach based on Reliable Broadcast, and the one based on View Synchronous Communication (or VSC for short). We discuss both communication primitives in a system model with fair-lossy channel, which leads us to introduce the "time-bounded buffering" *problem: VSC addresses this problem, but not Reliable Broadcast. Moreover, we show that VSC solves Reliable Broadcast in a system model with* "program-controlled crash".

However, VSC does more than Reliable Broadcast, and this has a cost. We analyse this cost by distinguishing between two types of failure suspicions: input-triggered *failure suspicions that are related to incoming messages, and* output-triggered *failure suspicions that are related to outgoing messages. We show that VSC has not managed to exploit the difference between these two types of failure suspicions, which has not allowed to solve the dilemma between (1) short fail-over time and (2) infrequent incorrect exclusion of processes from the membership. We show how to escape from this dilemma by replacing the standard VSC broadcast primitive by two broadcast primitives, one sensitive to input-triggered suspicions, and the other sensitive to output-triggered suspicions. This allows to get the best of two worlds.*

1. Introduction

Reliable Broadcast vs. View Synchronous Communication: *Reliable Broadcast* [17, 10] and *View Synchronous Communication* (*VSC*) [2, 16, 14, 7] are two communication abstractions that have been extensively considered in the context of asynchronous fault-tolerant distributed systems. Reliable Broadcast and VSC allow the broadcasting of messages while ensuring some sort of "atomicity" property: either all correct destination processes or none of them deliver some message. However, when taking a closer look at the specification of each primitive, one has on one side a simple and clear definition (Reliable Broadcast), and on the other side a complex definition (VSC), which moreover varies from one author to another.

Now, why would one consider VSC at all, rather than the well-defined Reliable Broadcast primitive only? A careful analysis of the literature shows that theoretical papers tend to consider Reliable Broadcast, whereas more practical papers favor VSC. The goal of the paper is to show that, although the specification of Reliable Broadcast is simple, it leads to implementation problems that are addressed by VSC.

A first observation is that, while VSC assumes a *dynamic* system model in which processes can join the computation after initialization of the system, Reliable Broadcast assumes a *static* system model (all processes are known initially). A dynamic system model is obviously more general, encompassing the crash and recovery of processes.[1] This is an important difference, but not the only one.

Reliable channels vs. lossy channels: Another difference is related to the underlying channel model. The implementation of Reliable Broadcast is usually described assuming reliable channels while the VSC approach considers the implementation of the VSC communication primitive over lossy channels. Obviously, assuming reliable channels is not realistic in practice. The implementation of Reliable Broadcast over lossy channels requires message retransmission; the same holds for VSC. In order for some process p to be able to retransmit message m, p needs to buffer m. This raises the question of how long p must buffer m? In this paper we argue that, unless the asynchronous system is augmented with a perfect failure detector (one that never makes mistakes), or equivalently is actually synchronous, the implementation of Reliable Broadcast over lossy channels requires m to be buffered for an unbounded duration. This is quite problematic in practice as it prevents garbage collection of messages. In contrast, implementations of VSC are able to get around this problem: they are based on a group membership service[2] which excludes slow processes from the membership and forces them to crash. This is the most likely reason why practical papers consider the VSC communication primitive rather than Reliable Broadcast.

Drawback of the VSC approach: However, the VSC approach has its own practical drawbacks. Processes that are excluded from the group might not have crashed. Thus the overhead of an incorrect failure suspicion is high in the VSC approach if, in order to keep the same degree of replication, every excluded process is replaced by a new process that *joins* the group. For this reason, systems based on VSC are usually configured with a high timeout value to suspect crashed processes. The problem is that choosing a high timeout value also has drawbacks, namely it leads to high fail-over time.[3]

Contribution of the paper: So, while the VSC approach addresses the issue of message buffering in the context of the implementation of reliable communications over lossy channels, this indirectly leads to privilege high timeout values for suspecting crashed processes. We show that the two issues of buffering and fail-over time can be decoupled, with significant advantages for the fail-over time of applications. We show that this decoupling can be achieved by distinguishing two "reliable broadcast" primitives instead of just one (i.e., VSC). We also show that these two primitives lead us to distinguish *input-triggered* suspicions from *output-triggered* suspicions. While output-triggered suspicions lead to exclusions from the membership, this is not the case with input-triggered suspicions. Moreover, we show that fail-over time is influenced only by

[1] In many systems, the recovery of a process is modeled by treating recovered processes as new processes.

[2] In the paper we consider the *Primary Partition Group Membership Service*. However, in Section 5 we briefly discuss the buffering issue in the context of the Partitionable Membership model [7].

[3] The fail-over time of an algorithm is the time elapsed between the crash of a process (t_1) and the time at which the algorithm has recovered from the crash (t_2). During the interval $[t_1, t_2]$ the algorithm is blocked.

input-triggered suspicions, and not by output-triggered suspicions. This allows aggressive input-triggered suspicions to coexist with conservative output-triggered suspicions.

Organization: The rest of the paper is organized as follows. Section 2 discusses Reliable Broadcast and introduces the *time-bounded buffering* problem. Section 3 discusses View Synchronous Communication and shows how this paradigm solves the *time-bounded buffering* problem. Section 4 introduces the distinction between *input-triggered* and *output-triggered* suspicions, and shows the drawback of the VSC approach in this context. Section 5 shows that the drawback of the VSC approach can be overcome by having two broadcast primitives, rather than just one. In Section 6, the use of the two broadcast primitive is illustrated by an example. Related work is discussed in Section 7. Finally, Section 8 concludes the paper.

2. Reliable Broadcast

In this section, we discuss the implementation of reliable communication over fair-lossy channels. This leads us to a discussion on message buffering issues, and to the introduction of the *time-bounded buffering* problem. Then, we argue about the impossibility of implementing Reliable Broadcast over fair-lossy channels with time-bounded buffering without the help of a perfect failure detector of class $\mathcal{P}$, i.e., the strongest class of failure detectors in Chandra-Toueg's hierarchy [5]. This highlights one of the major drawbacks of the Reliable Broadcast approach.

2.1. Processes, channels and Reliable Broadcast

The Reliable Broadcast approach assumes an asynchronous system model where the set of processes is fixed. Processes are only subject to crash failures (no Byzantine failures) without recovery. A *correct* process is a process that never crashes. Processes are completely connected by a set of unidirectional channels that cannot create, duplicate and garble messages. Concerning message losses, we consider the two following properties:

- *No loss:* If process p sends message m to process q, and q is correct, then q eventually receives m.
- *Fair loss:* 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 do not create, duplicate, or garble messages, and that satisfy either the no loss or the fair-lossy properties are respectively called *reliable channels* or *fair-lossy channels*. Fair-lossy channels adequately model links that temporary fail, or links that lose messages due to (router) buffer overflows.

Reliable Broadcast is specified in terms of two primitives R-BROADCAST and R-DELIVER, which satisfy the following properties [10]:

- **Validity:** If a correct process R-BROADCASTS m, then it eventually R-DELIVERS m.
- **Agreement:** If a correct process R-DELIVERS m, then all the correct processes eventually R-DELIVER m.
- **Integrity:** For any message m, every correct process R-DELIVERS m at most once, and only if m was previously R-BROADCAST.

2.2 Reliable Broadcast over reliable channels

Reliable Broadcast can be easily implemented in an asynchronous system with reliable channels: when a process p wishes to R-BROADCAST a message m, p sends m to all processes. When some process q receives m for the first time, then (1) q sends m to all processes and (2) q R-DELIVERS m.

2.3 Reliable Broadcast over fair-lossy channels

Clearly, the above implementation of Reliable Broadcast does not work with fair-lossy channels. For implementing Reliable Broadcast over fair-lossy channels, we first consider the implementation of point-to-point reliable communications over (point-to-point) fair-lossy channels. Let $SEND$ and $RECEIVE$ be the primitives providing reliable communications (see Figure 1): 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.

Then we can implement Reliable Broadcast over fair-lossy channels by combining the above implementation of reliable channels with the one described in Section 2.2.

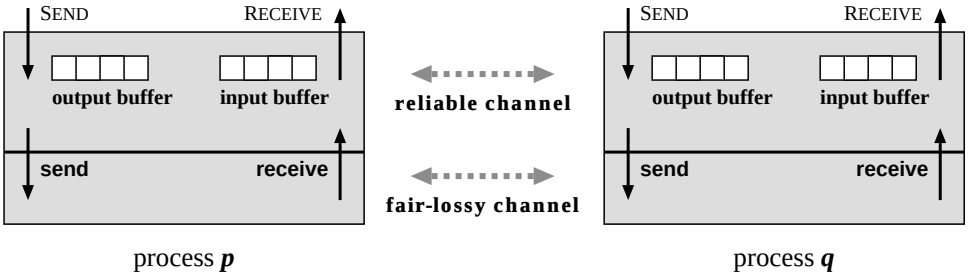

Figure 1. Providing reliable channels over unreliable channels

2.4 Reliable Broadcast over fair-lossy channels: the time-bounded buffering problem

In the implementation of Section 2.3, it may be the case that p keeps m in its output buffer forever: if q crashes, p might never receive $ack(m)$, and so might never delete m from its output buffer. This naturally leads to the following question: is there an implementation of $SEND$ and $RECEIVE$ in which p can safely delete m from its output buffer after a finite amount of time?

To formalize this issue, we introduce the *time-bounded buffering problem*: a time-bounded buffering implementation of reliable communications is an implementation wherein every message is eventually discarded from all the output buffers (Fig. 1).[4] Time-bounded buffering is related to the notion of *message stability*, a terminology used in the context of group communication. Let m be a message in the output buffer of some process p that must be broadcast to a set of processes $Dst(m)$. Message m is said to be

[4]Instead of time-bounded buffering, we might consider the problem of *space-bounded* buffering (every process needs only a bounded buffer size). However, while time-bounded buffering is a system issue, space-bounded buffering is an application issue (it cannot be analyzed independently of the application). This explains why we consider the time-bounded buffering problem. Space-bounded buffering is considered in [15] where the author assumes that the application executes repeated Reliable Broadcasts. The paper shows that in this context, space-bounded buffering requires a perfect failure detector (one that does not make any mistake).

stable at p, denoted by $stable_p(m)$, once p "knows" that all processes in $Dst(m)$ have either received m or have crashed. Once $stable_p(m)$ holds, p can safely delete m from its output buffer. Hence, solving Reliable Broadcast with time-bounded buffering is equivalent to ensuring that, for every process p and all messages m in p's output buffer, $stable_p(m)$ eventually holds.[5]

2.5 Reliable Broadcast over fair-lossy channels: time-bounded buffering and impossibility result

Failure detectors have been introduced to deal with the unsolvability of Consensus and Atomic Broadcast in asynchronous systems [5]. In such a model, each process may access a local failure detector module that provides (possibly partial or inexact) information about process failures. The property of failure detectors are defined in terms of their *completeness* and *accuracy*: completeness guarantees that crashes are eventually detected, whereas accuracy limits false suspicions. Chandra and Toueg [5] define a hierarchy of eight classes of failure detectors in terms of their completeness and accuracy properties. In the remainder of the paper, we consider failure detectors of the three strongest classes, namely class $\mathcal{S}$, class $\Diamond\mathcal{P}$, and class $\mathcal{P}$. Roughly speaking, given some failure detector $\mathcal{D}$ we have $\mathcal{D} \in \mathcal{P}$ if $\mathcal{D}$ perfectly detects failures, $\mathcal{D} \in \Diamond\mathcal{P}$ if $\mathcal{D}$ becomes perfect after some time, and $\mathcal{D} \in \mathcal{S}$ if $\mathcal{D}$ never suspects some correct process but eventually detects all the failures.

We argue that no implementation of Reliable Broadcast over fair-lossy channels can solve the time-bounded buffering problem, solely based on failure detectors of either class $\mathcal{S}$ or class $\Diamond\mathcal{P}$. For $\mathcal{S}$, the result follows directly from [1], which shows that quiescent communication cannot be implemented using $\mathcal{S}$.[6] For contradiction, assume that there exists an implementation of time-bounded buffering using $\mathcal{S}$. Trivially the implementation would be quiescent: once m is discarded by p, process p can no more send m!

For $\Diamond\mathcal{P}$ the argumentation is as follows. Consider the $\Diamond\mathcal{P}$ failure detector (which eventually becomes perfect). In any run R, no process can distinguish whether, at a given time t, the failure detector has already become perfect or not. Consider R-BROADCAST of message m, and the problem of deleting m from the output buffer. For at least one process p and one process q, p cannot delete m from its output buffer before p knows that either q has received m or q has crashed. The failure detector does not help, because when p suspects q it has no way to know whether q has actually crashed or is merely suspected because it is slow or too many messages are dropped by the channel.

So, neither failure detectors of class $\mathcal{S}$ nor those of class $\Diamond\mathcal{P}$ allow us to implement Reliable Broadcast with bounded-time buffering. However the problem can be solved with a perfect failure detector $\mathcal{P}$ (one that does not make any mistake) as follows: a process p that has some message m in its output buffer discards m once it knows that for every process q, either q has acknowledged m or q is suspected.

Note that even though time-bounded buffering ensure that all processes eventually discard m, they need to keep the identifier $id(m)$ forever in order to recognize duplicate messages, and avoid R-DELIVERING m more than once. This is a different issue, also addressed by View Synchronous Communication (see Sect. 3.2).

[5] Time-bounded buffering can be seen as a classical problem in the context of point-to-point communications over lossy channels. One of the goal of the paper shows that this is an issue that allows to understand Reliable Broadcast with respect to View Synchronous Communication.

[6] Informally, an implementation is quiescent if the invocation of one $SEND$ causes only a finite number of invocations of $send$ (see Fig. 1).

2.6 Reliable Broadcast over lossy channels: time-bounded buffering and program-controlled crash

The impossibility result for a time-bounded buffering implementation of Reliable Broadcast with a $\Diamond\mathcal{P}$ failure detector is quite a limiting constraint in practice. Systems based on View Synchronous Communication overcome this impossibility by relying on *program-controlled crash* [4]. Program-controlled crash gives the processes the ability to kill other processes or to commit suicide. It can be used to implement Reliable Broadcast over fair-lossy channels with time-bounded buffering. Indeed, consider process p with message m in the output buffer to q. If after some duration p has not received $ack(m)$ from q (directly or indirectly), p decides (1) to kill q, and (2) to discard m from its output buffer. Indeed, as q eventually crashes, there is no obligation for q to R-DELIVER m, i.e., p can safely discard m.

2.7 The overhead of program-controlled crash

However, program-controlled crash has a non negligible cost. To see that, consider some process q that is forced to crash. So, in order to keep the same degree of replication, another process q' will have to be created in order to replace q. This requires a dynamic system model. The management of the processes that are part of the system is handled by a *group membership service*. So, the suicide of q triggers a costly sequence of operations: (1) membership change to exclude q, (2) membership change to include q', which incorporates (3) the costly state transfer operation to bring q' to an up-to-date state. In other words, each exclusion of a correct process leads to an important overhead. From a practical point of view, this means that incorrect failure suspicions should be avoided as much as possible. This can be achieved by choosing a conservative timeout value in the implementation of the failure suspicion mechanism. Unfortunately the price is a high fail-over time. We come back to this issue later in the paper.

3. VSC vs. Reliable Broadcast

In this section we compare Reliable Broadcast with View Synchronous Communication. and show that View Synchronous Communication ensures Reliable Broadcast in the context of a view.

3.1. Group membership and View Synchronous Communication

View Synchronous Communication (or VSC for short) [7] assumes an asynchronous system model where processes may fail by crashing and may recover. Such systems can be modelled as a set of processes that is not fixed: processes can join the system during the computation (dynamic system model). The recovery of a process p is then modelled by the join of a new process p'.

View Synchronous Communication is based on a *group membership service*. This 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 view is provided to a process is called the *install* event. A process may *leave* the group, as a result of an explicit leave request, because it failed or because it is expelled by other members of the current view. Similarly, a processes 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 this paper, we only consider the primary-partition membership service (in Section 7 we show the advantage of the mechanisms proposed in the paper compared to partitionable membership). Let v_i^p denote the i^{th} view installed by p. The primary-partition membership service is defined by an agreement property on the view history: if p installs v_i^p and if q installs v_i^q, then we have $v_i^p = v_i^q$. This agreement property allows us to omit the process superscript and simply denote a view by v_i.

VSC allows processes to broadcast messages to the members of their current view with certain guarantees. Let V-BROADCAST^v denote the primitive by which a message is broadcast by a process in view v, and by V-DELIVER^v the primitive that delivers a message to a process in view v. The view superscript is sometimes omitted, if not relevant. There exist numerous specifications of VSC, which all share the following core properties [7]:

- **Validity:** If a correct process executes V-BROADCAST$^v(m)$, then it eventually V-DELIVERS m (in view v or in a subsequent view).
- **Termination:** If a process executes V-BROADCAST$^v(m)$, then eventually (1) every process in the view v V-DELIVERS$^v(m)$ or (2) every correct process in v installs a new view.
- **View Synchrony:** If process p belongs to two consecutive views v and v', and V-DELIVERS$^v(m)$, then every process q in $v \cap v'$ that installs v', also V-DELIVERS$^v(m)$, i.e., delivers m before installing v'.
- **Sending View Delivery:**[7] A message broadcast in view v, if delivered, has to be delivered in view v. In other words, if V-DELIVER$^{v'}(m)$ and V-BROADCAST$^v(m)$ occur, then $v' = v$.
- **Integrity:** For any message m, every correct process V-DELIVERS m at most once, and only if m was previously V-BROADCAST.

According to the Termination property, if a process executes V-BROADCAST$^v(m)$, the installation of a new view v' can replace the delivery of m. The View Synchrony property is a kind of atomicity property: before installing the new view v', either all processes deliver m or none of them deliver m. From the Sending View Delivery property, it follows that in the second case, m is never delivered.

3.2 VSC implementation ensuring time-bounded buffering

VSC over fair-lossy channels can be implemented with time-bounded buffering by relying on program-controlled crash. Consider the following implementation. At the beginning, the implementation for V-BROADCAST$^v(m)$ is similar to the one for R-BROADCAST(m) described in Section 2.3. In the context of this implementation, assume that at some point message m is in the output buffer of some process p (see Figure 1). If p receives $ack(m)$ from all the processes in view v, then p can discard m from its output buffer. If p does not received $ack(m)$ from some process r, then p eventually triggers a view change in order to exclude r. Upon installation of a new view v' from which r is excluded, p can discard m from its output buffer. If r discovers that it has been (incorrectly) excluded from the new view v', r commits suicide.

[7] Some specification consider a property called "Same View Delivery" instead of "Sending View Delivery" [7].

VSC also solves the problem of time-bounded buffering of message identifiers needed to prevent duplicate delivery of messages (see Section 2.5). All the messages sent in view v are tagged with v. In view v, all the messages that are received and tagged with a view prior to v are discarded. Upon installation of a subsequent view v', all the identifiers of messages received in view v can thus be discarded.

3.3 VSC ensures Reliable Broadcast in the context of a view

We now show that VSC ensures the three properties of Reliable Broadcast in the context of a view. The Validity and Integrity properties of VSC clearly enforce the Validity and Integrity properties of Reliable Broadcast. The Agreement property of Reliable Broadcast holds for the following reason:

For the Agreement property, consider a correct process p that executes V-DELIVER$^v(m)$ (in view v). By the Integrity property of VSC, some process must have executed V-BROADCAST(m). The Sending View Delivery property guarantees that m was V-BROADCAST in view v. If all the processes in view v V-DELIVER m, then the Agreement property of Reliable Broadcast holds in v. Otherwise, the Termination property of VSC implies that every correct process eventually installs a new view v'. By the View Synchrony property, every process in v that installs v' has V-DELIVERED m in view v.

So, VSC ensures the properties of Reliable Broadcast in the context of a view with time-bounded buffering. However, this comes with a price: the overhead of program-controlled crash, discussed in Section 2.7.

4. Flaw of the VSC approach

4.1 The two roles of the group membership service

In this section, we go over the role played by the group membership service, or *GMS*, in the context of View Synchronous Communication. We point out that the group membership service solves two problems with quite different timing constraints.

We first observe that the time-bounded buffering property allows the implementation of VSC to recover buffer space. This means that with infinite memory there is no need to provide the property. In the context of VSC, this means that no process would ever have to be excluded. However, if VSC never excludes processes, existing algorithms based on VSC would block. The reason is that view changes that exclude processes are used by existing algorithms as a *failure detection mechanism*: if some process p is in view v, but not in the subsequent view v', then all processes in $v \cap v'$ learn by installing view v' that p has crashed (or will crash). So, when looking closer at the role of GMS in the context of VSC one can make the following observation:

1. GMS ensures the time-bounded buffering property.
2. As a failure detection mechanism, GMS prevents blocking in (application) algorithms: if $q \in v$ waits for a message broadcast by p in view v, then a view change that excludes p allows q to stop waiting for m.

The VSC approach handles these two different aspects uniformely. This may have some bad effects since timing constraints are quite different in (1) and (2). With respect to (2), detecting failures quickly is crucial for reducing blocking periods of algorithms, and so for reasonable fail-over time. In other words, the group membership service must be prompt to change the view when a failure has actually occurred. On the other

hand, one tolerates longer delays for forcing time-bounded buffering. Indeed, these longer delays have no direct impact on the timing behaviors of algorithms as long as buffer resources are available. In (1), timing constraints are much more flexible (they derive from space constraints).

In the VSC approach, we cannot thereby take advantage of the timing flexibility allowed for enforcing time-bounded buffering since the VSC approach artificially links this problem to the one of preventing blocking. There is a clear dilemma between short timeout values and high timeout values. We show below show how to overcome the dilemma by distinguishing *output-triggered* suspicions from *input-triggered* suspicions, and handling each of these suspicions differently (by considering two broadcast primitives instead of just one).

4.2 Output-triggered $vs.$ input-triggered suspicions

The failure detection mechanisms can be based on timeouts on *input* channels: if a process q is supposed to send messages to p, and if p's input buffer from q is empty for some time, p suspects q. Another way for p to get information on the operational status of q is to determine—by having p look at its output buffer to q (see Fig. 1)—whether q receives or not messages it has sent. In other words, p can suspect q with respect to the fact that either (1) messages in its output buffer to q are never received, or (2) because its input buffer from q is empty. We refer to *output-triggered* and *input-triggered* suspicions respectively. Note that for output-triggered suspicions, *space* constraints rather than *time* constraints can be considered: as long as there is enough space to hold outgoing messages for retransmission, there is no reason to exclude any process based on timeouts.

While the two suspicion mechanisms have been used in implementations, clever use of these their specificities has not been been exploited. Depending on the problem addressed, output-triggered suspicions could be more appropriate than input-triggered suspicions, or *vice-versa*. We suggest a clever exploitation of these differences by introducing two broadcast primitives, instead of just one.

5 Two broadcast primitives instead of just one

In order to exploit the difference between output-triggered and input-triggered suspicions, we split the features of V-BROADCAST into two broadcast primitives that we call V-R-BROADCAST and V-FD-BROADCAST, (R stands for *Reliable* and FD for *Failure Detection*). Both alike satisfy the specification of VSC given in Section 3, but with a different GMS: the views used by V-R-BROADCAST are different from the ones used by V-FD-BROADCAST. As the issue pointed out in Section 4 is non-functional, it is not surprising that our two new broadcast primitives have the same specification. Typically, V-R-BROADCAST would be used to reliably broadcast a message in a view while ensuring time-bounded buffering; on the other hand, V-FD-BROADCAST should be used whenever view changes are needed to prevent blocking.[8]

[8]It might be argued that the application programmer should not be required to choose between two broadcast primitives based on considerations external to the application. First, the considerations that lead to the selection of one or the other primitives are not unrelated to the application: they provide different QoS. Second, if the application programmer does not choose between the two primitives, i.e., between two implementation options, who else could choose, based on rational argument?

5.1 Membership views $vs.$ ranking views

Two GMS define two types of views. We call them *membership views* (or simply *views*), and *ranking views* (or *rk-views*). Ordinary views are identical to the views of View Synchronous Communication, and they are denoted similarly by $v_0, v_1, \ldots, v_i, \ldots$ Intermediate views are installed between membership views. Processes agree on the sequence of both membership and rk-views. The rk-views between v_i and v_{i+1} are noted as follows:

$$v_i^0,\ v_i^1, \ldots, v_i^j, \ldots, v_i^{last_i}$$

View v_i^0 is equal to v_i, and the last ranking view $v_i^{last_i}$ is equal to v_{i+1}. *The membership of all ranking views* $v_i^0, \ldots, v_i^{last_i-1}$ *is the same as the membership of* v_i. Only the order of the processes differ (the reason will be explained below), e.g., $v_i = v_i^0 = [p, q, r]$, $v_i^1 = [q, r, p]$, $v_i^2 = [r, p, q]$, $v_i^3 = [p, q, r]$, etc.

During the existence of the rk-views $v_i^0, \ldots v_i^{last_i-1}$ the membership view remains v_i.

Referring to the discussion of Section 4, membership views are generated by suspicions resulting from conservative timeout values, while rk-views are generated by suspicions resulting from aggressive timeout values. As all the ranking views v_i^j, except $v_i^{last_i}$, are composed of the same set of processes as v_i, they do not force the crash of processes. So, the role of rk-views is to contribute to a short fail-over time, while membership views ensure time-bounded buffering of messages.

As mentioned above, the specification of the two broadcast primitives is identical to the specification of VSC given in Section 3.1, but with different views. This affects only the Sending View Delivery property, which becomes:

- **Sending View Delivery:**
  If V-R-DELIVER$^v(m)$ and V-R-BROADCAST$^{v'}(m)$ occur, then $v = v'$ is the same membership view (rk-view changes could have occurred between V-R-BROADCAST(m) and V-R-DELIVER(m)).
  If V-FD-DELIVER$^v(m)$ and V-FD-BROADCAST$^{v'}(m)$ occur, then $v = v'$ is the same rk-view, i.e., no view change and no rk-view change occurred between V-FD-BROADCAST(m) and V-FD-DELIVER(m).

5.2 The two broadcast primitives and output-triggered $vs.$ input-triggered suspicions

The two broadcast primitives, V-R-BROADCAST and V-FD-BROADCAST, introduced above allow us to take advantage of the two types of suspicions: input-triggered vs. output-triggered. As explained in Section 4, exclusions (resulting from suspicions) ensure message stability (i.e., time-bounded buffering), whereas ranking views (resulting from suspicions, too) prevent algorithms from blocking. Message stability is an issue related to output buffers, while blocking is an issue related to input buffers. Thus it is better to base message stability (i.e., process exclusion) on output-triggered suspicions. On the other hand, prevention of blocking (i.e., delivery of ranking views) ought to be based on input-triggered suspicions. The GMS related to V-R-BROADCAST should define membership views based on output-triggered suspicions (the suspicion of some process p leads to the exclusion of p, and the definition of a new membership view), whereas the GMS related to V-FD-BROADCAST should define rk-views based on input-triggered suspicions.

5.3 Defining ranking views

In this section we will discuss the definition of rk-views. Various options are possible, the simplest one being the *rotating coordinator rk-views*.[9] In a given group communication system, various rk-view paradigms could be predefined. The choice of the paradigm would have to be specified as a parameter upon creation of a group. In the rotating coordinator rk-views, the first process in some rk-view v,[10] and only this process, is monitored by all other processes in the rk-view. If this process is suspected (input-triggered suspicions), the GMS is invoked to install a new rk-view v', where v' is obtained from v by a permutation that moves the head of the v list to the tail of the v' list, e.g., if $v = [p, q, r]$, then $v' = [q, r, p]$. This corresponds to a coordinator change, from coordinator p in the rk-view v, to the coordinator q in the rk-view v'. The monitoring of the first process of some rk-view v can be implemented using heartbeat messages: the first process of the rk-view v periodically sends *I am alive* messages to the other processes of v.

5.4 Difference in the implementation of the two broadcast primitives

Even though the two broadcast primitives have the same specification, they cannot be implemented in the same way. The difference lies in the following facts:

- Let V-R-BROADCAST$^v(m)$ be executed by p. Upon reception of m by $q \in v$, process q can immediately V-R-DELIVER m.
- Let m be V-FD-BROADCAST$^v(m)$ by p. Upon reception of m by $q \in v$, process q *cannot* immediately V-FD-DELIVER m.

The difference is related to the program-controlled crash feature, which can help in the first case, but not in the second. Consider in the second case the current rk-view $v = [p, q, r]$, process p that V-FD-BROADCASTS message m, immediately V-FD-DELIVERS m and crashes: neither q nor r have V-FD-DELIVERED m. The specification of V-FD-BROADCAST does not allow the installation of the rk-view $v' = [q, r, p]$ without violating the View Synchrony property: p belongs to v' and has V-FD-DELIVERED m, so both q and r have to V-FD-DELIVER m before installing v'.

Note that the installation of a membership view, e.g., $v' = [q, r]$ is possible, without violating the View Synchrony property: as p is not in the membership of v', processes q and r can install v' without having previously V-FD-DELIVERED m.

5.5 Implementation of the two broadcast primitives

The implementation of the two broadcast primitives does not contain any new ideas: it is based on known techniques (see below). For this reason the section is kept short.

First it should be mentioned that, since the group membership problem is not solvable in an asynchronous system [4], the same result holds for our two broadcast primitives. To overcome the impossibility we assume the existence of failure detectors of a type that allows us to solve consensus, e.g., the failure detector $\Diamond\mathcal{S}$ [5]. We also assume a majority of correct processes (needed to solve consensus with $\Diamond\mathcal{S}$).

[9] The rotating coordinator paradigm is well known in the context of fault-tolerant computing, e.g., [6, 11, 8, 5].

[10] An rk-view is a "sequence" of processes. The "coordinator" is the first process in the rk-view.

a) V-R-BROADCAST: The implementation of V-R-BROADCAST is similar to the one of the classical VSC primitive: this can be done by reduction to a consensus problem [12, 9]. Here the difference is only in the way suspicions are generated: the implementation relies on output-triggered suspicions.

b) V-FD-BROADCAST based on the rotating coordinator rk-views: The implementation of V-FD-BROADCAST is almost similar to the implementation of the uniform VSC primitive [16]. To understand the reason, consider the View Synchrony property (Sect. 3.1):

- **View Synchrony:** If process p belongs to two consecutive views v and v', and V-DELIVERS$^v(m)$, then every process q in $v \cap v'$ that installs v', also V-DELIVERS$^v(m)$, i.e., delivers m before installing v'.

In the case of two consecutive rk-views v and v' (that have by definition the same membership), process p necessarily belongs to v and v', i.e., the View Synchrony property becomes *uniform*:

- **View Synchrony:** Let v and v' be two consecutive rk-views. If process p (correct or not) V-FD-DELIVERS$^v(m)$, then every process q in $v \cap v'$ that installs v', also V-FD-DELIVERS$^v(m)$, i.e., delivers m before installing v'.

If we assume the "rotating coordinator" rk-views, the first process in the current rk-view needs to be monitored: suspicions of this process are input-triggered. If the current coordinator is suspected, then the rk-view change protocol is executed.

5.6 Performance issues

Most of the time, the performance of group communication is measured in "nice" runs, i.e., in runs with no crashes and no incorrect failure suspicions. The reason is that the performance of group communication in the case of a crash is dominated by the timeout value used for failure detection, which leads to large figures (that people usually do not like to show). As an example, if the timeout value is around 10 seconds, then the cost of VSC is the case of a crash will be on average around 10 seconds.

Assume that V-FD-BROADCAST is implemented with a small input-triggered timeout value (e.g., 1s), and V-R-BROADCAST is implemented with a large output-triggered timeout value (e.g., 100s). This means that the cost of V-FD-BROADCAST in the case of a crash will be on average around 1 second (i.e., better than VSC with a timeout of 10s), and the cost of V-R-BROADCAST will be on average around 100 seconds (i.e., worst than VSC). To understand that we gain in both cases compared to VSC with a timeout value of 10 seconds, the reader must understand that the crash of a process — in the context of reliable broadcast — impedes the group only whenever the rest of the group is blocked waiting for a message from the crashed process:

- If the group is blocked in the case of a crash, then V-FD-BROADCAST*s*hould be used (in order to have a small blocking period).
- If the crash of a process does not block the group, then V-R-BROADCAST should be used: V-R-BROADCAST*i*nstead of VSC (which has a smaller timeout value) reduces the probability of incorrectly excluding processes. This also has a positive impact on the overall performance.

6 Example: VSC and primary-backup replication

Here, we illustrate the use of the two broadcast primitives V-R-BROADCAST and V-FD-BROADCAST in the context of the primary-backup replication technique.

6.1 Primary-backup replication

The primary-backup replication technique [3] consists in having one *primary* server and one or more *backup* servers ready to take over if the primary fails (see Fig. 2). Client requests are sent to the primary. Once the primary has processed some request, it makes sure that each backup is up-to-date with respect to the new state. For that, the primary sends to the backup an *update* message representing the state change induced by the processing of the request.

After broadcasting the *update* message, the primary waits for an acknowledgement from a majority[11] of servers in the current view v and returns the reply to the client. The primary is then ready to handle the next request. Further details can be found in [13].

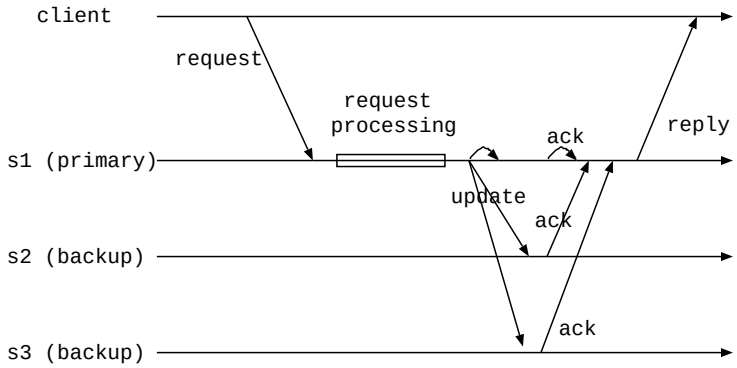

Figure 2. Primary-backup replication

For the purpose of illustrating the use of the two broadcast primitives we transform the scheme of Figure 2 as follows (see Fig. 3): (1) the roles of the clients are played by the servers, i.e., the servers can issue requests (e.g., s_2 in Fig. 3), and (2) requests are sent to all servers, instead of just to the primary (so that the clients do not need to worry about the crash of the primary, and send their requests to the new primary). There are thus two different messages that are broadcast among the servers: the "requests" messages and the "update" messages.

6.2 Illustration of the two broadcast primitives

The use of the two broadcast primitives for implementing primary-backup replication is shown in Figure 4: V-R-BROADCAST is used for broadcasting *request* messages, while V-FD-BROADCAST is used by the primary for broadcasting *update* messages. On Figure 4, the leftmost membership view v_k and rk-view v_k^0 is $[s_1, s_2, s_3]$: this rk-view defines s_1 as the primary (i.e., the first process in the sequence). A new rk-view $v_k^1 = [s_2, s_3, s_1]$ is later installed, which defines s_2 as the new primary: the membership view v_k remains unchanged, i.e., though s_1 has been suspected, s_1 remains in the membership view

[11] It is not necessary to wait for the acknowledgement from all servers.

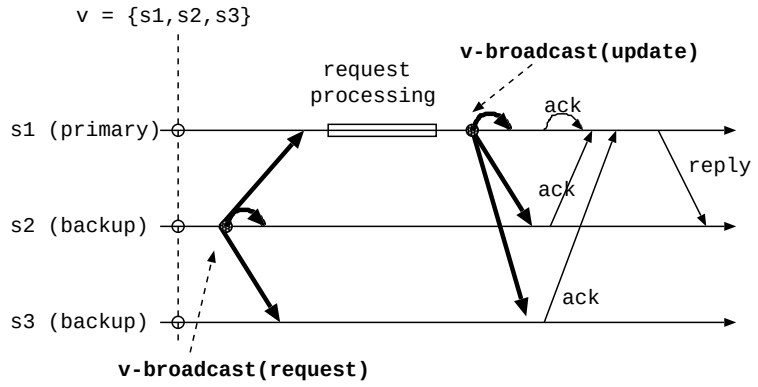

Figure 3. Primary-backup replication and VSC (role of the client played by s_2)

v_k. The Sending View property of V-FD-BROADCAST ensures that V-FD-BROADCAST$(update)$ and V-FD-DELIVER$(update)$ occur in the same rk-view $v_k^0 = [s_1, s_2, s_3]$. The Sending View property of V-R-BROADCAST also ensures that V-R-BROADCAST$(request)$ and V-R-DELIVER$(request)$ occur in the same membership view $v_k = \{s_1, s_2, s_3\}$ (but not necessarily in the same rk-view). Point-to-point messages (ack, $reply$) are transparent to view changes.

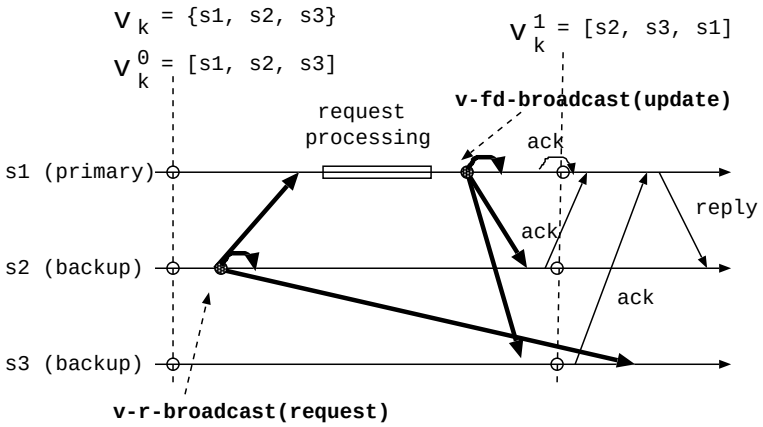

Figure 4. Primary-backup replication with V-R-BROADCAST **and** V-FD-BROADCAST

In the light of this example, we can figure out the benefit of having two broadcast primitives instead of just one (as in the classical VSC context). The crash of s_2 (which broadcasts a request) and the crash of s_1 (which processes requests and broadcasts updates) do not have the same impact on the system: the crash of s_1 should be quickly detected (it blocks the group), whereas fast detection of the crash of s_2 is not essential (the crash of s_2 does not block the group, sinc the primary waits for a majority of ack messages). With only one broadcast primitive, it is impossible to handle the broadcast of s_1 differently from the broadcast of s_2.

7 Membership and ranking views compared to partitionable group membership

Wrong suspicions related to V-FD-BROADCAST do not lead to the exclusion of processes. This can be seen as similar to a partitionable membership service, where processes in a minority partition are not forced

to crash [7]. Apart from this similarity, our proposal differs from VSC in a partitionable group membership (called *extended* VSC) [7].

In Figure 4, consider the membership view $v_k = \{s_1, s_2, s_3\}$, and the rk-view $v_k^0 = v_k$. Let processes s_1, s_2, s_3 be correct, but consider a temporary link failure inducing the formation of two temporary partitions $\pi_1 = \{s_1\}$ and $\pi_2 = \{s_2, s_3\}$. Assume that this partition leads to the definition of a new rk-view $v_k^1 = [s_2, s_3, s_1]$, installed on s_2 and s_3 (and on s_1 after the partition is repaired). With extended VSC, the messages broadcast by processes in partition π_1 are sent to the processes in π_1, while the messages broadcast by processes in π_2 are sent to the processes in π_2. This is not the case with our broadcast primitives. If no (membership) view changes occurs, then all messages V-R-BROADCAST or V-FD-BROADCAST (1) before the partition, (2) during the partition, or (3) after the repair of the partition, are eventually delivered to $\{s_1, s_2, s_3\}$. The layer implementing VSC has thus the responsibility to buffer messages during the existence of the partition, and to transmit these messages outside the partition, once the partition is repaired. In other words, *the occurrence of the partition is totally transparent*. This is not guaranteed by extended VSC: if a partition occurs, the application has the responsibility to forward messages broadcast within one partition to the processes outside of the partition, during the merge of the partitions. *The occurrence of the partition is not transparent to the application.*

8 Conclusion

The paper has introduced the *time-bounded buffering* problem in the context of the implementation of reliable communication over fair-lossy channels. Specifically the paper has discussed the problem of solving Reliable Broadcast over fair-lossy channels while ensuring time-bounded buffering, and has shown how VSC addresses the issue thanks to the *program-controlled crash* feature.

The paper has also shown that, while VSC provides more than Reliable Broadcast with time-bounded buffering, it has failed to do it adequately. This is related to the fact that VSC has overlooked the fundamental difference between output-triggered and input-triggered failure suspicions. The paper has shown the benefit that results from distinguishing between these two types of failure suspicions. The paper has also shown how this difference can be exploited by replacing the single VSC broadcast primitive by two broadcast primitives, called V-R-BROADCAST and V-FD-BROADCAST. In addition, instead of considering only the usual time-based suspicions, space constraints rather than time can be considered for output-triggered suspicions: as long as there is enough space to hold outgoing messages for retransmission, there is no reason to exclude any process based on timeouts.

We believe that the novel approach to building fault-tolerant distributed algorithms introduced in the paper is an important step to improve the fail-over time of applications built on top of a VSC infrastructure.

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] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.

[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, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC-15)*, pages 322–330, Philadelphia, Pennsylvania, USA, May 1996.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

[6] J. M. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, August 1984.

[7] G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *Computing Surveys*, 4(33):1–43, December 2001.

[8] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.

[9] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. on Software Engineering*, 27(1):29–41, January 2001.

[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] L. Lamport. The part-time parliament. TR 49, Digital SRC, September 1989.

[12] C. Malloth and A. Schiper. View synchronous communication in the internet. Technical Report 94/84, EPFL, Dept d'Informatique, October 1994.

[13] R. Oliveira, J. Pereira, and A. Schiper. Primary-backup replication: From a time-free protocol to a time-based implementation. In *Proc. of the 20th IEEE Symp. on Reliable Distributed Systems (SRDS-20)*, pages 14–23, New Orleans, LA, October 2001.

[14] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A Dynamic View-Oriented Group Communication Service. In *Proc. of the 17th ACM Symposium on Principles of Distributed Computing*, Puerto Vallarta, Mexico, June 1998.

[15] A. Ricciardi. Impossibility of (repeated) reliable broadcast. TR PDS-1996-003, Univ of Texas, Austin, April 1996.

[16] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proc. of the 13th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS-13)*, pages 561–568, May 1993.

[17] F. Schneider, D. Gries, and R. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–15, 1984.