# Fast and Scalable Total Order Broadcast for Wide-area Networks

Luiz Angelo Steffenel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Fast and Scalable Total Order Broadcast for Wide-area Networks*

Luiz-Angelo Steffenel

**N° 6037**

Novembre 2006

Thème NUM


ISSN 0249-6399 ISRN INRIA/RR--6037--FR+ENG

# Fast and Scalable Total Order Broadcast for Wide-area Networks

Luiz-Angelo Steffenel*

Thème NUM — Systèmes numériques
Projet AlGorille

Rapport de recherche n° 6037 — Novembre 2006 — 27 pages

**Abstract:** Fault tolerant protocols such as Total Order Broadcast are key aspects on the development of reliable distributed systems, but they are barely supported on large-scale systems due to the cost of traditional techniques. This paper revisits a class of Total Order Broadcast protocols called *moving sequencer*, known by its communication efficiency. Indeed, we evaluate RBP, one of the most known implementations of *moving sequencer* protocols. We demonstrate how RBP can be used with wide-area systems, and we propose new techiques to improve its resiliency and consistency properties under failures, as well as improving its scalability aspects.

**Key-words:** Total Order Broadcast, Group Membership, View Synchronous Communication, Scalability

* IUT Nancy Charlemagne - Université Nancy 2

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

# Modèle de Diffusion avec Ordre Total extensible et efficace

**Résumé :** Des protocoles tolérants aux fautes comme la Diffusion avec Ordre Total (*Total Order Broadcast*) sont des eléments-clés dans le développement de systèmes distribués fiables. Malheureusement, ces protocoles sont très peu utilisés avec des systèmes de grande taille à cause du coût des techniques traditionnelles. Ce travail revisite une classe d'algorithmes pour la Diffusion avec Ordre Total appelée *moving sequencer*, connue par son efficacité. Nous étudions notamment les caractéristiques du protocole RBP, en soulignant ses avantages et faiblesses. Ainsi, nous proposons des techniques pour améliorer les aspects de résilience, consistance et extensibilité de RBP, techniques qui peuvent être aussi utilisées par d'autres algorithmes avec des caractéristiques similaires.

**Mots-clés :** Diffusion avec Ordre Total, Membership de Groupe, Communication avec Vues Synchrones, Extensibilité

# 1 Introduction

Total Order Broadcast, also known as Atomic Broadcast [14] is one of the essential building blocks for a fault tolerant distributed system and therefore is the subject of a considerable amount of literature, as observed by Defago [10]. In spite of the different implementation strategies to assign a global sequence number to the messages, we observe that there are two main implementation methods, namely the "agreement-based" and the "sequencer-based" strategies. In the first method, a process is assigned with the special role of "sequencer", i.e., this process assigns the delivery order of the messages (the role of sequencer may be fixed or may move among the processes). In the second method, instead of relying in a single sequencer, the processes engage on a consensus operation [5] to ensure a global delivery order.

When dealing with large-scale systems (such as computational grids or P2P networks), the performance of an algorithm (and consequently the number of exchanged messages) becomes a key aspect. Agreement-based algorithms usually do not scale well, therefore we are invited to evaluate different implementations of the Total Order Broadcast, looking for an algorithm that minimizes the number of coordination messages and the number of communication steps required to deliver a message. Further, as large-scale systems are specially prone to node volatility and network problems, we also need to minimize the cost induced by a process failure.

In this paper we address the problem of efficient and scalable Total Order Broadcast through the study of the Reliable Broadcast Protocol, also known as RBP. Proposed by Chang and Maxemchuk [6] in 1984, RBP aims to provide total order and reliable broadcast in a distributed system subjected to process failures and fair-lossy links. While the original definitions are not recent, new protocols still rely on its concepts (for example, RMP [16], Pinwheel [9] and TRMP [15]). Basically, RBP is structured around a token ring, used to distribute responsibility for acknowledgements. A single token is passed from site to site around the ring, and only the holder of the token (also called the *sequencer*) can acknowledge the messages, and assign sequence numbers to them. The acknowledgement contains the identifiers (source id, for example) of the sequenced message, and is broadcasted to all members of the ring.

While RBP is quite efficient when there are no failures, it also should deal with process failures. Hence, its definition considers a quite obsolete membership management that cannot ensure most of the group communication properties. Therefore, we start our analysis by comparing RBP *Reformation Phase* with another group membership technique, the View Synchronous Communication - VSC [4, 13]. Indeed, we explore one variant from the VSC model, called *two views* model [7], that is especially adapted to the needs of RBP, as it allow us to take advantage from failure detectors with aggressive timeouts while minimizing the drawbacks from incorrect suspicions. Based on this analysis, we propose a Group Membership Service based on the VSC model that provides efficient and adapted group membership in a wide-area network while respecting the properties of the VSC model.

The rest of this paper is organized as follows: in Section 2 we present the system definitions and models considered in this work, as well as the properties a Total Order Broadcast

must ensure. Section 3 describes the RBP protocol operational model first in a failure-free environment, and then in a the presence of process failures. In Section 4 we revisit the RBP algorithm, in the light of new techniques. Indeed, we could identify many similarities between RBP and the Primary-Backup replication model, which present many solutions that can be applied to improve RBP. In Section 5 we present an improved solution for RBP *Reformation Phase,* based in the VSC Group Membership Service. Finally, Section 6 reviews some works related to RBP, and Section 7 presents the conclusions of this work.

# 2 System Model and Definitions

Distributed systems are modeled as a set of processes $\Pi = \{p1; p2; ...; pn\}$ that interact by exchanging messages through communication channels. As the set of processes is dynamic, i.e, a process that fails can join or be removed from the set, a better representation for the set of processes is $\pi(t)$, that means, the set of processes can vary over time. A process that does not crash during all the execution is called *correct*. However, as we assume an asynchronous system, even a correct process can be suspected of crashing. To differentiate a wrongly suspected process from a crashed process is a problem in asynchronous distributed systems, and thus, the protocols should consider this possibility.

We also consider Fair-lossy channels [1], that ensures only that if $p$ sends a message $m$ to $q$ an infinite number of times and $q$ is correct, then $q$ receives $m$ from $p$ an infinite number of times. If we have *fair-lossy* channels, we can construct *reliable* communication primitives ***SEND(m)*** and ***RECEIVE( )*** using *unreliable* ***send(m)*** and ***receive( )***, by ensuring that the message $m$ is retransmitted until its successful reception (the receiver can send an *ack*, for example).

## 2.1 Total Order Broadcast Specification

The problem of Total Order Broadcast problem can be defined by four properties, namely Validity, Agreement, Integrity, and Total Order [10]. However, we should remark that the Agreement, Integrity and Total Order properties may be *Uniform* or *Non-Uniform*. A Uniform property applies not only to correct processes but also to faulty processes. While the basic definition of RBP supposes Uniform deliver, i.e., a process can only deliver messages after all process have received it, we can suppose possible scenarios where this requirement can be weakened. In Section 3.3 we discuss what is the lowest level of *Non-Uniform* deliver that the protocol can use without violating the consistency of the application. Due to this possibility, the Non-Uniform properties are presented below and considered all through the document:

**VALIDITY** - If a correct process broadcasts a message $m$ to $Dest(m)$, then some correct process in *Dest(m)* eventually delivers $m$.

**AGREEMENT** - If a ***correct*** process delivers a message $m$, then all correct processes in *Dest(m)* eventually deliver $m$.

**INTEGRITY** - For any message $m$, every ***correct*** process $p$ delivers $m$ at most once, and only if (1) $m$ was previously broadcast by *sender(m)*, and (2) $p$ is a process in *Dest(m)*.

**TOTAL ORDER** - If ***correct*** processes $p$ and $q$ both deliver messages $m$ and $m'$, then $p$ delivers $m$ before $m'$ if and only if $q$ delivers $m$ before $m'$.

To reliably transmit messages over unreliable channels (as in the case of *fair-lossy* channels) RBP uses explicit retransmission requests. Message losses are detected by finding discrepancies between the expected messages and the received ones. Total ordering is achieved in RBP by assigning an unique sequence number to each message sent to the group, and delivering messages in this order. Most of the complexity of the protocol resides in providing these sequence numbers, ensuring that for each sequence number corresponds a single message.

# 3 Total Order and the Moving Sequencer Strategy

In distributed algorithms, Total Order Broadcast is usually provided by defining a global agreed sequence of messages (for example, by assigning sequence numbers to the messages), and then delivering them to the application following that order. If we use sequence numbers, this implies that no two messages receive the same sequence number, and by this reason, the sequence number must be globally unique. A simple solution to provide Total Order would be to centralize the distribution of sequence numbers in a fixed process (the sequencer). This approach, however, has many drawbacks, especially when the environment is subject to process failures (it introduces a single point of failure).

To ensure the liveness of the algorithm, the protocol must use some fault tolerant mechanism. In the *moving sequencer* strategy, as defined by [10], a single process assigns the sequence number to the messages. However, the role of sequencer is not fixed, but moves among the processes. Hence, this algorithm can be implemented using a token-passing strategy: the process that holds the token is the only one that can assign sequence numbers to the messages (the sequencer is sometimes called *token site* or *token holder*). If there are no failures, the token passing strategy is enough to provide Total Order, as once sequence numbers are assigned all receivers eventually obtain them (the token holder must obtain all previous sequenced messages before assigning a new sequence number).

To simplify the description of the protocol we can make strong assumptions such as requiring that each token site assigns a sequence number to a single message, and then must pass the token to the next process. Therefore, each *sequencing round* of the protocol can be described in three steps, as presented by Fig 1: first, someone (a sender) broadcasts a message $m$ to all processes; the token holder picks this message, assigns a sequence number to it, and broadcasts the message ID and the sequence number to all processes; finally, the token is passed to the next process in the ring.

This scheme ensures that the sequence number assigned to each message is globally unique because the last sequence number assigned is transmitted with the token, and only

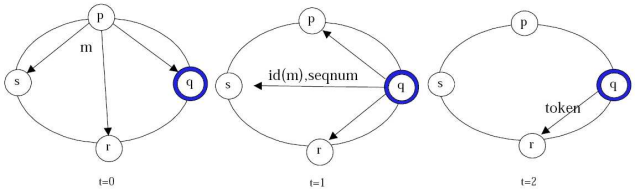


Figure 1: Process p sends message m to all

the process that has the token can assign a new number. If messages are delivered according to this order, Total Order is ensured.

Of course, this is not enough if the channels are not reliable. It can happen, for example, that some processes do not receive the message $m$ from the sender. In this case, the solution is to request retransmissions. Fig. 2 illustrates an example of such a situation. At time *t=0*, process $s$ did not receive the message $m$. Later, it receives a message containing the message ID and a sequence number assigned to that message. As message $m$ was not received, process $s$ cannot deliver it, and thus, must request a retransmission, which is done at time *t=2*.

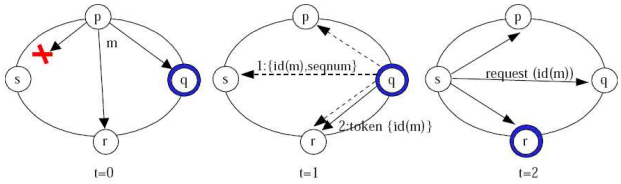


Figure 2: Process s requests retransmission concurrently to the token passing

Another situation that requires message retransmission is when the sequence number is not received (Fig. 3). When a process $s$ receives a message that assigns a sequence number=2 to the message *m'* (at time *t=3*). As $s$ did not receive before a message assigning the number 1, at time *t=4* it requests the retransmission of all *sequence messages* between 1 (the number it was expecting) and 2 (the last number it received). Only when someone answer to its requests, with a copy of the lost message ($\{id(m),\ 1\}$), $s$ can go ahead with message processing (and for example, accept the token).

While this strategy provides total order and reliable broadcast, it still can be improved in relation to the number of messages exchanged. On example can be observed in the algorithm: the sequencer sends two different messages, the sequence number and the token. A simple solution for these problems can be found by piggybacking extra information on the sequence message, reducing the number of exchanged messages. Indeed, the messages broadcasted by the sequencer may aggregate four separate functions as defined by RBP[6]:

- acknowledgement to source $s$ that message $m$ has been received by the ring members;

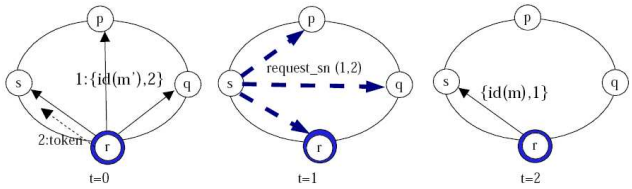


Figure 3: Process s requests retransmission

- inform all receivers that message $m$ is assigned to the global sequence number *seqnum*;
- acknowledgement to the previous sequencer (acceptation of the token);
- transmission of the token to the next sequencer.

Aggregating such functions in a single message (the token), a *step* of communication is reduced to only two messages. There, the message sent by the sequencer has the following roles: it sends an *ack* to sender $s$, it indicates to all the sequence number assigned to $m$, it sends an *ack* to the previous sequencer $p$, and it sends the token to $r$.

A problem that occurs when we overload all functions in a single message is that if this message is lost, many events dependent on its reception are triggered. A good example, which must be handled by the protocol, is the phenomenon of *old* messages. Old messages are in fact messages that have already been acknowledged, but continue to arrive. This can occur when a source or a sequencer did not receive the *acks* it was waiting (due to communication failures, for example), and thus, keeps sending a message (cf. Fig. 4). If a source retransmits a message that has already been acknowledged, we can suppose that the source failed to receive the acknowledgement (and by definition, will continue to retransmit the message until receiving the *ack*). In this case, the current token holder must retransmit the acknowledgement to that source despite the original token holder. Another situation where old messages can occur is when a former sequencer keeps sending the token. If it keeps sending the token, we can suppose that the process failed to receive the token-passing acknowledgement. When this situation happens, the acknowledgements must be retransmitted by the current sequencer.

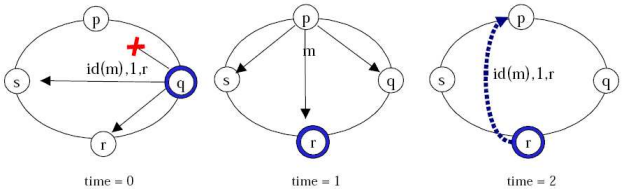


Figure 4: Ack retransmission when the source lost the message

### 3.1 Number of Messages

Due to the ability to piggyback acknowledgements and token passing in a single message, RBP uses very efficiently the network resources. Indeed, if no message is lost, RBP needs only two broadcasts: the first one, sent by the message source, submits the message; the second one comes from the sequencer, which assigns a sequence number to that message at the same time that passes the token to the next sequencer. Thus, if no messages are lost, the protocol requires only $2 \times (n-1)$ messages (assuming that broadcasts transmit $n-1$ point-to-point messages) for each message to be delivered.

Comparatively, the Atomic Broadcast based on the Consensus [5] may exchanges up to $(4+2n) \times (n-1)$ messages in a single execution step. There are other consensus algorithms that try to minimize the number of messages, as for example the Early Consensus [18], but their main focus is on the reduction of communication steps, which leads to a the number of exchanged messages still high (generating $O(n^2)$ messages against $O(n)$ from RBP).

### 3.2 Operation in Case of Failures

As presented in the previous section, the mechanism of the *moving sequencer* strategy is intrinsically connected to the token passing mechanism. As the token is passed in a logical ring, a single failure can block the protocol. To ensure *liveness* and restart the token passing, the token list must be reconstructed by removing suspected processes [6]. As the possession of the token gives an special role to a process, the impact of a failure is directly connected to the role of the failed process. To make this scenario even more complex, message losses can lead to inconsistent states, and thus, the set of messages received or sent by the failed process is also an important parameter to determine the impact of the problems caused by the failure.

An important aspect to be considered is how failures are detected. Current protocols usually employ Chandra and Toueg failure detectors as full-time services, a way to improve accuracy of the suspicion mechanism. Old protocols such as RBP, however, rely on simpler strategies to detect the failure of a process, and indeed, the RBP definition defines a suspicion of a process unsuccessfully tries to contact another process a certain number of times. This makes the failure detection very specific: as the roles executed by the processes differ in importance (a sequencer worth more than a sender), RBP establishes an *ad-hoc* failure detection.

In RBP, when a failure is detected, the failed process must be removed from the token list. Indeed, when a failure is detected the correct processes shall enter in a different execution mode, called *Reformation Phase*, which aims to create a new token list. Once a new token list is accepted by a majority of correct processes, a new sequencer must be defined to restart the sequencing. Note that the reformation phase is blocking, as none of the failure situations exposed in the previous section allow the processing to continue.

In a failure situation, each process that detects the failure (called *originator*) invokes the reformation phase and proposes a new token list. This new token list carries a version

number. As new failures and communication losses can occur, the own reformation phase must be tolerant to these failures.

In RBP, Reformation is a three-phase commit (3PC) protocol coordinated by the originator [15]. Fig. 5 presents a good run of the RBP Reformation Phase. It starts when the originator contacts each process in the new token list (the *slaves*), and ask them to join this list (Phase 1). If the list is accepted by all members that compose it (Phase 2), the originator chooses a process to be the new sequencer. This selection aims to choose the process with the greatest knowledge from the previous list, i.e., the process with the most recent committed message. Before restarting the message sequencing, the new sequencer must update the other processes, by retransmitting the lost messages. This last step is really important because when the reformation finishes, we assume that all processes have the same knowledge about the others.

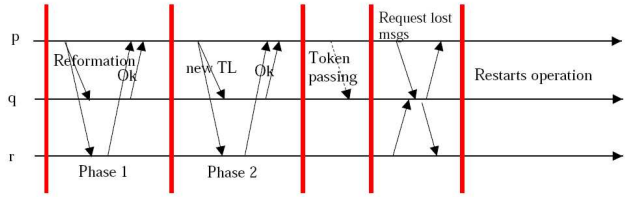


Figure 5: Reformation phase in a good run

However, this Reformation protocol has some drawbacks. As the detection is not simultaneous to all processes, multiple processes can invoke the reformation concurrently (multiple originators). To ensure that only a valid token list emerges from the reformation phase, RBP defines some constraints for the new token list and the processes:

- A site can join only one list each time;
- There must be only one valid token list;
- It must be composed by the majority of processes;
- It must be the most recent list proposed (having the higher version number);
- No message committed by the old token list can be lost.

These constraints can be grouped in three tests, as listed below [6]:

**Majority test:** The list must have a majority of the processes.

**Sequence test:** A site can only join a list with a version number greatest than its previous list

**Resiliency test:** No message committed by the old token list is lost. At least one site must have all committed (stable) messages.

Thus, a list must pass by all these tests before being accepted. One important point to remark is that a process excluded from the token list is allowed to join it in a future reformation (and [6] do not specify how these processes proceed to recover messages in order to keep the application consistent).

As some processes can be excluded from the token list (due to a false suspicion, for example), the protocol must ensure that messages sent by excluded processes do not interfere with the processing. Thus, each message is tagged by the token list version, which allows a process in the normal phase to discard messages coming from excluded processes.

### 3.3 Resiliency, Message Stability and Garbage Collection

Up to now, nothing was said about resiliency. However, as processes may crash (and the network may lose messages), it is important to specify the resiliency level that the protocol provides (or requires). Resiliency is intrinsically tied to message delivery. If we want to tolerate process failure, we cannot deliver messages immediately as they are received: if only one process receives a message, delivers it and crash, it can happen that other processes do not follow the same delivery order, which can lead to inconsistencies in the application.

Usually, we express resiliency levels by $k$-resiliency. This means that if we want to tolerate *k-1* failures, we should ensure that the message was received by at least $k$ processes. The most restrictive level that can be achieved in a system is the total resiliency, where all processes must receive the message before delivering it to the application layer. This way, using *total resiliency* we ensure the Uniform property.

In RBP, total resiliency is easy to be implemented. We just need to wait until the token be transmitted to all processes before delivering the message(s) (as each process that receives the token must acquire all previous messages). In our assumptions, each new sequence number broadcast means a new token passing. As result, when the sequence number $k$ is sent by sequencer $r$, this implies:

- receiver $r$ has all messages up to and including the $k^{th}$ sequenced message,
- receiver ($r$-1) mod $n$ (we use modulo operation because the processes are in a logical ring) has all messages up to and including the ($k$-1)$^{th}$ sequenced message,
- ...
- receiver ($r$-$n$+1) mod $n$ has all messages up to and including the ($k$-$n$+1)$^{th}$ sequenced message.

When a message was received by all processes, it can be considered stable. If we consider the relations above, when process $p$ sends a sequence number $k$, all messages with sequence number less or equal than $k$-$n$+1 can be delivered, because all processes have them. However, even if the application requires lower levels of resiliency, detecting message stability is an important issue.

The reason why message stability is so important comes from the following observation: if all processes have a message $m$, then $m$ will not be requested anymore for a retransmission.

If we can detect message stability, we can execute garbage collection safely, saving storage space. Due to its ring characteristic, RBP is especially apt to detect message stability, and thus, can execute a very efficient garbage collection.

In RMP [16], however, the application can define the level of resiliency it wants (in [17] this is considered a QoS level), and thus, RMP can provide $n$-resiliency, majority resiliency, and even unreliable deliver. As in distributed systems subjected to process failures the consistency of the application is a fundamental concern, we should analyze what is the lowest level of resiliency that RBP can provide without violating consistency.

First, suppose that $p$ assigns a sequence number to a message $m$, broadcasts it and crashes (real failure). Suppose also that once a process receives a message, it can deliver it immediately. Due to network links and the failure, it is possible that only a set of processes have received the message from p. If at least one process receives the sequenced message, it will deliver the message, and once the Reformation Phase is called, this process will share the message with the other processes, propagating the sequence number of $m$.

Now, suppose the same scenario but $p$ does not really crashes. Due to a network partition, $p$ is not able to send messages to the other processes. Now, if $p$ is the only process to have received the message, it will be the only process to deliver it. When the other processes detect its failure, the reformation shall generate a new token list **without** the message sequenced by $p$, and thus, that sequence number will be assigned to other message. As $p$ is not really crashed, this scenario leads the system to an inconsistent state.

Through this example, we show that there is a lower bound on the RBP resiliency. We cannot deliver messages immediately without risking an inconsistent state. In fact, if the protocol is tolerant to $f$ failures, it must wait $f+1$ token passings before delivering, to be sure that at least one correct process has all messages. Unreliable deliver such as provided by RMP can only be executed if the application does not require consistency among the processes.

# 4 Revisiting the RBP Protocol

In the previous section, we presented the RBP protocol trying to follow the original specification (actually, we just tried to present it step by step, in order to make easier the comprehension of its objectives and structures). We also presented some interesting characteristics from the variants of RBP listed above.

While we presented some remarks that could improve the protocol (or at least, update it with more recent techniques), no extra details were provided. In this section we will present our suggestions to improve the protocol, based in the evaluations we have done so far and in the techniques used nowadays to solve similar problems.

## 4.1 A Note in Failure Detection

Failure detection has evolved considerably since RBP was proposed, and the model of failure detection specified by Chandra and Toueg is essential to most distributed applications today.

Examining again the failure scenarios supported by RBP, it is possible to define two models of detection: detection on a process failure (process deterministically identified) and absence of activity from the sequencers.

The first model, similar to traditional failure detection, relies simply on monitoring another process. Due to this characteristic, it is easy to detach the detection from the RBP algorithm and use some other technique than the number of retransmissions. Indeed, when this kind of detection is required, monitoring is done on the next or on the previous process in the ring. If we consider that some detection techniques (like the Push and Pull detectors) have some scalability problems, this specificity on the monitoring can help to minimize the traffic on the network. In fact, this specific monitoring is similar to the ad-hoc failure detectors, proposed to solve the Consensus problem.

In the second model of detection, however, there is no specific monitoring on some process; the suspicion of a failure means that the protocol is not acting as usual. This detection is especially important when the token is lost, because it prevents the protocol from blocking forever (preserving the liveness property). A possible implementation of the model presented by Chang and Maxemchuk [6] is monitoring the token passing (the protocol specifies that there is a maximum time that a sequencer can hold the token before passing it). However, as this maximum time is a parameter related to the implementation of the RBP, this kind of detection cannot be easily detached from the protocol, as the suspicion must be tuned with the same parameters as the token passing. A good aspect, however, is that this model of detection does not generate any extra message.

It is clear that both detection techniques have different purposes. While the the first method leads to a fast detection of failed processes, the second method aims at preserving the liveness of the protocol, even if it does not provide aggressive detection. As token lost situations tend to be rare in comparison with the other failure situations, the second model of detection can be tuned with conservative timeouts. An active detection is good enough for most situations, and only when the token is lost the passive detection should raise a suspicion.

According to these remarks, we can replace the failure detection within the RBP by two Chandra and Toueg-like detectors, each one operating under different requirements:

**Active Detector:** a failure detector with aggressive timeouts that monitors a specific process.

**Passive Detector:** a failure detector with conservative timeouts (ideally, a detector that only uses application messages in the detection), which starts the Reformation Phase no matter the suspected process.

## 4.2 Weaknesses of the Reformation Phase

As presented in Section 3.2, the reformation phase proposed in [6] is a three-phase commit protocol. If in a good run the reformation can be solved as in Fig. 5, this is not necessarily the most usual situation, and the reformation can take much more time.

Let us consider the problem of defining a new token list. As each process that detects the failure can start the reformation phase (the "multiple originators" from [6]), the definition

of a new token list is delayed until some list obtains majority of processes. However, this is not so simple, because if the *majority test* takes too much time, any process is allowed to leave that list and join/start another list. Together, these events can force new rounds of the reformation phase, and when finally a new token list is agreed, all processes must acquire the messages (and the message numbers) committed in the previous token list before returning to the *normal* phase.

Another important point is that RBP definition of the *reformation phase* [6] supposes that if some correct process does not belong to a valid token list, it can call another reformation until it returns to the token list. The problem with this assumption is that while the process is not in the valid token list, garbage collection is running. Once a new token list is defined, messages become stable, and by definition we can consider that if garbage collection is being done, the buffers from the processes will have at most the messages sequenced in the last turn of the token, which is a too short interval, just enough to prevent an excluded process to comes back and acquire all previous messages.

In summary, the reformation phase is the most critical part from the RBP protocol: its definition is very old, and it lacks precision. In the next sections, we identify similarities between RBP and Primary-Backup+VSC replication model. These similarities can be exploited to improve the RBP protocol.

## 4.3 Similarities and Differences between Primary-Backup and RBP

In order to provide data replication, a technique usually present in the literature [4, 14, 7] is the use of Primary-backup replication together with the View Synchronous Communication (VSC). While Primary-backup provides total ordering conducted by the primary process (a *fixed sequencer* approach), View Synchronous Communication deals with the membership changes in a dynamic group (as the RBP Reformation Phase). Therefore, Primary-backup replication and the RBP protocol present many similarities that can be exploited in our work. First, they need to provide Total Order. Second, both techniques need dynamic groups, as failures can block the progress of the protocol.

To solve the first problem, both techniques are based in the sequencer approach, i.e., they ensure that only a single process can order the messages, and that this order is respected by all processes. However, they differ in the way a sequencer is chosen. For the Primary-Backup, a new sequencer (the primary) is only chosen when the precedent one has failed. In RBP, the role of sequencer moves among the processes, even if there is no failure.

Also for the second problem, the need for dynamic groups, these techniques have similar approaches. Both consider that when there is a failure, the role of the failed process is important to define the need for a view change. While in Primary-Backup this view change can be delayed indefinitely if the failed process is a backup, RBP eventually requires the view change, as failures block the token passing.

Because this is a widely studied technique, the Primary Backup+VSC presents interesting solutions that can be used to improve the RBP protocol. A short definition of these techniques is provided below, and in Section 5 we suggest some innovations for the RBP protocol, using the solutions presented here.

## 4.4 Primary-Backup Replication

The primary-backup replication technique consists in having one *primary* server and one or more *backup* servers (Fig. 6). If the primary fails, one of the backup servers is chosen to take the role of primary. By definition, client requests are sent to the primary. When the primary receives a request from a client, it performs the corresponding operation. Once the primary has processed the request, it makes sure that each backup server is up-to-date with respect to the new state. For that, the primary sends to the backup server 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 all backups. Once acknowledgements have been received, the primary returns the reply to the client, and then it is ready to handle the next request.

If there is no failure, it is clear that ordering is provided as all requests from the client are processed by the primary, and the *update* messages replicate this order in the backups. Atomicity is also ensured because the *update* is forwarded to all the backups and *ack* is awaited by the primary before sending the response.

However, if there are failures, some situations make difficult to ensure Atomicity without the use of additional techniques [19]. Specifically, a hard problem to solve occurs if the primary fails while sending the *update* and before sending the *reply*. Besides this problem of Atomicity, a new primary must be selected as the primary fails.

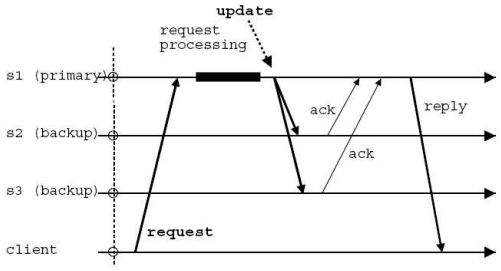


Figure 6: Primary-backup replication

## 4.5 Group Membership and View Synchronous Communication

View Synchronous Communication (VSC, for short) [3, 12] is an extension of the Group Membership specification. It assumes an asynchronous system model where processes may fail by crashing and may recover (joining the system under a new identity). VSC manages the creation and maintenance of a set of processes (called *group*) in a dynamic system model, i.e., processes can join and leave the system during the computation. 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 result of an explicit

request, because it failed or because it was excluded by other members of the current view. In the same way, a process can *join* the group.

Considering a primary-partition group membership, we can define an agreement property on the view history: if $p$ installs $v_i^p$ and if $q$ installs $v_i^q$, then $v_i^p = v_i^q$. In VSC, broadcasts to members of the current view are delivered with some guarantees. Therefore, *VSCast* denotes the primitive by which a message is broadcasted by a process in view $v$, and VSDeliver represents the primitive that delivers a message to a process in view $v$. The properties of the VSC can be considered as [7]:

**Validity :** If a correct process executes VSCast(m), then it eventually VS-Delivers $m$ (in view $v$ or in a subsequent view).

**Termination :** If a process executes VSCast(m), then (1) every process in view $v$ VS-Deliver(m) or (2) every correct process in $v$ installs a new view.

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

**Sending View Delivery :** A message broadcasted in view $v$, if delivered, ought to be delivered in view $v$.

**Integrity :** For any message $m$, every correct process VS-Delivers $m$ at most once, and only if $m$ was previously VSCast.

According to [8], some protocols allow a weaker version of the *Sending View Delivery* property. This weaker property, called *Same View Delivery* assumes that all processes deliver a message $m$ in the same view, even if this is not the view where the message was initially sent.

The use of VSC in the primary-backup replication ensures the Atomicity requirements even if there are failures, and allows the processes to select the primary server in a deterministic way (as the view is agreed by all processes).

Assuming that a message stability detector (i.e., a process that verifies if all processes in a view have received m) running concurrently, we can implement VSC as presented by [19] (Algorithm 1):

**Algorithm 1** Solving VSC view change [19]

VSCast(m) executed by pk:
  send (i,m) to all in vi(g)
Upon reception of (i,m) by $p_k$ while in view $v_i(g)$:
  VS-Deliver m add m to the set $unstable_k$
Upon suspicion of some process in $v_i(g)$:
  R-Broadcast(view-change, i)
Upon R-Deliver (view-change,i) by $p_k$ for the first time
  1. send $unstable_k$ to all
  2. $\forall p_i \in v_i(g)$: wait until receive $unstable_i$ from $p_i$ or $p_i$ suspected
  3. let $init_k$ be the pair ($View_k$, $Msg_k$) s.t.
    - $View_k$ is the set ($p_i \cup$ the processes from which $unstable_i$ was received)
    - $Msg_k$ is the union of the sets $unstable_i$ received
  4. execute consensus among $v_i(g)$ with $init_k$ as the initial value
  5. let (View, Msg) be the decision of consensus
  6. VS-deliver all messages in Msg not yet VS-Delivered

## 4.6 The time-bounded buffering problem

Reliable Broadcast, as used in the VSC algorithm presented above (including in the consensus) are easy to implement in asynchronous systems with reliable channels: when a process $p$ wishes to R-Broadcast a message $m$, $p$ sends $m$ to all process. When some process $q$ receives $m$ for the first time, $q$ sends $m$ to all processes and then R-Delivers $m$. However, this implementation does not work with fair-lossy channels. A possible solution consists on repeatedly execute *send(m)* until receiving an acknowledgement of $m$ from each process. Once a process $p$ receives *ack(m)* from all processes in *Dest(m)*, it can delete $m$ from its output buffer. This strategy, however, has a problem. If $q$ crashes, $p$ might never receive *ack(m)* from $q$, then it must keep $m$ in its output buffer forever.

This problem 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? This problem was formalized by [7] as the *time-bounded buffering problem*. A time-bounded buffering implementation of reliable communication is an implementation where every message is eventually discarded from all output buffers.

It is easy to see that time-bounded buffering is related to message stability. If process $p$ knows that all processes in *Dest(m)* have either received $m$ or crashed, then the message $m$ become stable and $p$ can remove it safely. However, Charron-Bost *et al.* claim that no implementation of Reliable Broadcast over fair-lossy links can solve the time-bounded buffering problem, based solely on failure detectors of either class $\mathcal{S}$ or class $\diamond\mathcal{P}$.

## 4.7 Program-Controlled Crash and View Changes

The impossibility of solving the time-bounded buffering problem with a $\mathcal{S}$ or $\diamond\mathcal{P}$ failure detector is a quite limiting constraint for practical systems. Systems based on View Synchronous Communication usually overcome this impossibility by relying on *program-controlled crash*. Program-controlled crash gives the ability to kill other processes or to commit suicide.

Program-controlled crash can be used in order to have a View Synchronous Communication implementation that ensures time-bounded buffering. Consider the following implementation: if after some time process $p$ has not received *ack* from $q$, $p$ can decide to kill $q$, and then to discard $m$ from its output buffer. Indeed, as $q$ eventually crashes, there is no obligation for $q$ to R-Deliver $m$, and thus, $p$ can safely discard $m$.

In fact, program-controlled crash is also used to ensure the view change properties. By the *Sending View Delivery* property, if a message is broadcasted in view $v$, all correct processes should deliver the messages broadcasted in the same view $v$. If a process is suspected, we are not sure that it is crashed. In addition, the *Termination* property says that if there is a view change, all correct processes eventually install view $v'$. Thus, by relying on program-controlled crash, processes excluded from the next view are forced to crash, ensuring VSC properties. For example, a simple way to force an excluded process to crash is to verify the new view. If the process does not belong to the new view, it commits suicide.

Of course, the use of program-controlled crash has a non negligible cost. Every time a process $q$ is forced to crash, a membership change is required to exclude $q$. If in addition we

should keep the same degree of replication, another process *q'* must replace *q*, this brings the extra cost of the state transfer to *q'*. Summing up, while program controlled crash is necessary to solve the time-bounded buffering problem, incorrect suspicions must be avoided as much as possible, to reduce the overhead caused by program-controlled crash. In order to reduce the occurrence of incorrect suspicions, a common solution is to choose a conservative timeout value for the implementation of the failure detectors. Unfortunately, the price of this choice is a high fail over time, which can lead to blocking situations.

### 4.8 A two-level view model

In typical group membership architecture, solving efficiently the time-bounded buffering problem and blocking prevention problem at the same time is not possible, because they are linked to the same failure suspicion scheme. In order to explore better both issues, Charron-Bost *et al.* proposes the use of two levels of GMS.

Therefore, each level of GMS defines different types of views. *Ordinary views* (or simply *views*) are identical to the views of View Synchronous Communication. *Intermediate views* (or *i-views*) are installed between ordinary views.

If ordinary views are denoted by $v_0, v_1, \ldots, v_i, \ldots$ , the i-views between $v_i$ and $v_{i+1}$ are denoted as $v_i^0, v_i^1, \ldots, v_i^j, \ldots, v_i^{last}$. The intermediate view $v_i^0$ is equal to $v_i$, and the last intermediate view $v_i^{last}$ is equal to $v_{i+1}$. One important point is that the membership of all intermediate views $v_i^0, v_i^1, \ldots, v_i^{last-1}$ is the same as the membership of $v_i$, that is, they only differ in the order that processes are listed in the view. For example, $v_i = v_i^0 = \{p, q, r\}$, $v_i^1 = \{q, r, p\}$, etc. During the existence of the i-views $v_i^0, \ldots, v_i^{last-1}$ the ordinary view remains $v_i$.

This model in two layers allows us to solve the problem from Section 4.7. Ordinary views are generated by suspicions resulting from conservative timeouts, while i-views are generated by suspicions resulting from aggressive timeouts. As all i-views from $v_i^0, \ldots, v_i^{last-1}$ are composed by the same set of processes, they do not force the crash of processes. This way, i-views contribute to avoid blocking situations, while ordinary views ensure time-bounded buffering of messages.

The only issue that must be determined is how i-views are elaborated. An example suggested by [7] (among various options) is a *rotating coordinator i-view*. In this example, the first process in some i-view is considered the coordinator (for example, the primary server in the *primary-backup* replication). When this coordinator is suspected, an i-view change moves it to the end of the processes list, and a new process is automatically selected as the coordinator.

## 5 A Group Membership Service adapted to Wide-Area Networks

When we compare the view change technique, however, we observe that the Reformation algorithm presents many drawbacks in comparison with the VSC model. First, the Ref-

ormation algorithm is not a "full" membership layer, as it does not keep mediating the communication between the processes. Instead, once a new token list is obtained from the Reformation algorithm, is the RBP protocol that should manage the membership.

Second, the RBP Reformation seems to not provide the *Same View Delivery* property, which all VSC primitives present. Suppose that a process is excluded from the token list, i.e., it is not more present in the token list *n*. Once excluded from the token list, it does not receive new messages from the ring, and thus, cannot detect the stability of the most recent messages. Due to the *k-resiliency* requirements, this process cannot deliver these remaining messages, as they seem unstable to it. If later this process rejoins the token list *n+1*, it can realize that those messages are already stable, and deliver them in view *n+1*, while the other processes have delivered in view *n*.

Therefore, the absence of the *Same View Delivery* property and the use of an efficient (but aggressive) garbage collection generate a contradiction. If a process is allowed to return in a later view, and deliver messages in this view, we suppose that it could recover lost messages (for example, the messages sequenced while this process was outside). As the garbage collection is too aggressive, likely this process will be unable to acquire the missing messages, remaining inconsistent with the other processes. As RBP does not consider program-controlled crash, this process cannot be killed to recover the consistency of the application.

The lack of these properties reduces considerably the power from the RBP Reformation Phase. Hopefully, we can replace the Reformation Phase algorithm. As the membership problem is similar to both Primary-Backup and RBP, we can use the VSC view change in order to upgrade the RBP Reformation algorithm.

The VSC view change presents many advantages in relation to the RBP Reformation. First, the VSC view change is a distributed algorithm without a central coordinator. Once a view change is called (through the broadcast of a *view change* message), all processes are invited to start the view change. If many processes detect the failure simultaneously, they will send the same *view change* message (as they belong to the same view), and thus, they will take part on the same agreement. The consensus itself has no need to be fully distributed (it can use rotating coordinators), because all processes agreed to start it. Based on the view change from Algorithm 1, we can define a new algorithm adapted to the RBP protocol (Algorithm 2).

Here, the "unstable" queue is the sequenced queue *seqQ*. Just remembering the RBP protocol, a sequenced message remains in the sequenced queue until it is able to be delivered. As we suppose uniform delivery, this means that messages in the sequenced queue are not yet stable. Once a process receives the sequenced queue from all process that are not suspected, it can compute$Msgs_k$, that is the union of all received *seqQ*. It also can suggest a new token list, based in the set of processes that answered it *reformation* message.

As the decision of the consensus comprise both the new token list and the set of all unstable messages, all processes that get this decision have the same knowledge on the unstable set. As this "uniform" knowledge can be considered as message stability, these messages are ready to be delivered. As all processes share the same sequenced queue, any

**Algorithm 2** First sketch of the "View Change" reformation protocol

Upon suspicion of some process in $TL_i$
  RBroadcast (reformation, i)
Upon R-Deliver (reformation, i) by $p_k$ for the first time
  1. send $seqQ_k$ to all  /* sends the "unstable" list of messages */
  2. $\forall p_i$, wait until receive $seqQ_i$ from $p_i$ or $p_i$ suspected
  3. let $initial_k$ be the tuple ($TL_k$, $Msgs_k$) s.t.
    - $TL_k$ is the new token list with all processes that sent their $seqQ$
    - $Msgs_k$ is the union of the $seqQ$ sets received
  4. execute consensus among $TL_i$ processes, with $initial_k$ as the initial value
  5. let ($TL$, $Msg$) be the consensus decision
  6. $stableQ \leftarrow Msg$, $seqQ \leftarrow \{\}$  /* as all processes get Msg, these messages are stable */
  7. if $p_k \in TL$, then "install" $TL$ as the next view $TL_{i+1}$

process can be the new sequencer, and we can select, for example, the first process in the token list.

## 5.1 RBP Reformation and Process-Controlled Crash

According to the examples presented in the previous section, the original RBP Reformation algorithm does provide neither *Same View Delivery* nor *Sending View Delivery* properties. Associated with an aggressive garbage collection, the absence of these properties can lead the protocol to a contradictory situation: the protocol allows a removed process to come back to the token list, but does not ensure that it will be able to acquire all the missing messages. If such situation occurs, a cyclic situation can happen: the inconsistent process blocks the token passing, it is removed from the token list, it tries to come back... In fact, the solution to avoid this problem is to force an excluded process to suicide, which is not considered by [6].

By modifying "View Change Reformation" presented in Algorithm 2, we solve this problem. As presented in Section 4.7, we can implement program-controlled crash by checking the new view. If a process is excluded from the new view (i.e., the new token list), it is forced to commit suicide, solving the time-bounded buffering problem and ensuring the VSC properties. This modification can be easily made, as presented in Algorithm 3. Here, if a process was excluded from the new view (line 7), it commits suicide.

**Algorithm 3** "View Change" reformation protocol with program-controlled crash

Upon suspicion of some process in $TL_i$
  RBroadcast (reformation, i)
Upon R-Deliver (reformation, i) by $p_k$ for the first time
  1. send $seqQ_k$ to all  /* sends the "unstable" list of messages */
  2. $\forall p_i$, wait until receive $seqQ_i$ from $p_i$ or $p_i$ suspected
  3. let $initial_k$ be the tuple ($TL_k$, $Msgs_k$) s.t.
    - $TL_k$ is the new token list with all processes that sent their $seqQ$
    - $Msgs_k$ is the union of the $seqQ$ sets received
  4. execute consensus among $TL_i$ processes, with $initial_k$ as the initial value
  5. let ($TL$, $Msg$) be the consensus decision
  6. $stableQ \leftarrow Msg$, $seqQ \leftarrow \{\}$  /* as all processes get $Msg$, these messages are stable */
  7. if $p_k \in TL$, then "install" $TL$ as the next view $TL_{i+1}$
    else suicide

## 5.2 i-Views on RBP

As Program-Controlled Crash has a non-negligible cost, we can also apply the technique of "two views" [7] in order to minimize this cost. While regular views force the crash of a process, i-views allow these suspected processes to keep alive. However, we cannot use the suggestion of i-view proposed by Charron-Bost, i.e., move the suspected process to the "end" of the membership list. As the token is periodically passed among the processes in the view, just moving a suspect process does not solve the problem, and soon the token passing is blocked again, requiring a new i-view change (that is also a costly operation).

In this work, we suggest a different approach to implement i-views, which is better adapted to RBP. If we consider excluded processes as an "external set of processes", we can create i-views by considering the group **{"core members","external"}**. While all broadcasts are sent to the whole group, the token is passed only among the "core" members, which are not suspected. The concept of external receivers was already used by TRMP [15], to allow hierarchical distribution of processes in a world wide network. As a suspected process does not participate in the sequencing process (it is not in the token ring), it does not block the token passing. As it still belongs to the view, it receives all broadcasts, and can send messages to the group.

Because external processes do not receive the token, we cannot use the token passing to ensure Reliable Broadcast and Total Order. In fact, as we don't know precisely if an external process is correct or not, no assumptions can be made about it. While Reliable Broadcast and Total Order properties must be ensured for the "core" processes, the only thing we can do for the external processes is to allow them to be kept up-to-date. This means that if an external process missed a message, is its own responsibility to detect this event and request message retransmissions, even if it remained unreachable for a long time (as illustrated in Fig. 7).

Now, suppose this scenario: a network partition causes a correct process to be unreachable. As it is suspected, it will be removed from the token list, and moved to the external list. The sequencing continues, messages become stable among the token list members, and garbage collection is performed. If the partition is repaired, the excluded process is unable to acquire lost messages, because they were already discarded. When this happens, it is obligated to commit suicide (for example, if core member sends a "no more in the buffers" message, as answer to the retransmission request).

Because garbage collection in RBP is executed simultaneously with message delivery, is highly probable that a correct process excluded due to a network partition or to its relative speed (too slow) will be forced to commit suicide. Through this fact, we observe that the aggressive garbage collection from RBP cancels one of the most important advantages in the use of i-views: avoid program-controlled crash on correct processes.

In order to minimize the program-controlled crash due to the absence of messages in the buffers, we can consider that one or more processes keep messages while there is available space, i.e., they postpone the garbage collection. This technique is completely feasible if we consider today's machines, and garbage collection can be executed when buffers are full or when there is a regular view change (note that both possibilities force the crash of processes).

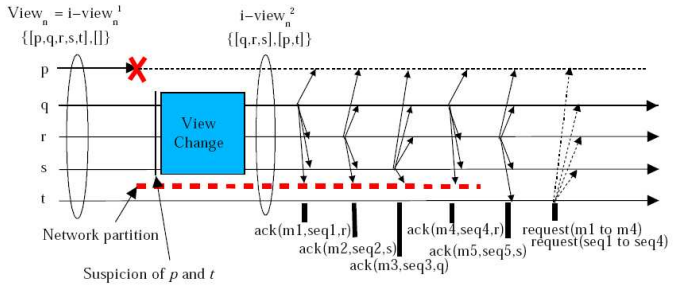


Figure 7: I-view change and external members

## 5.3 Optimizing the i-View Change

By definition i-views aim to avoid blocking the protocol, not necessarily to solve the time-bounded buffering problem. This means that the installation of an i-view does not require garbage collection. If we consider the suggestions from the previous section, where garbage collection is postponed the most possible, we can optimize the algorithm for i-view changes.

In fact, as i-views do not require garbage collection, message stabilization does not need to be implemented by the i-view change. As message stabilization during a view change is implemented by exchanging the set of sequenced messages from each process, we can optimize the i-view algorithm by skipping this step, as presented in Algorithm 4:

**Algorithm 4** Optimized i-view changes

Upon suspicion of some process in $TL_i$
  RBroadcast (i-view, i)
Upon R-Deliver (i-view, i) by $p_k$ for the first time
  1. send $ack$ to all
  2. $\forall p_i$, wait until receive $ack$ from $p_i$ or $p_i$ suspected
  3. let $initial_k$ contains $TL_k$ s.t.
    - $TL_k$ is the new token list with all processes that sent $ack$
  4. execute consensus among $TL_i$ processes, with $initial_k$ as the initial value
  5. let $TL$ be the consensus decision
  6. if $p_k \in TL$, then "install" $TL$ as the next view $TL_{i+1}$

Using this optimized algorithm, processes are no more force to manipulate lists of messages each time there is an i-view change, what makes the i-views a "light-weight" version of regular views. By reducing the overhead on the i-view changes, we reduce the impact of wrong suspicions due to aggressive failure detectors. Similarly, i-views do not force a process to suicide, reducing the overhead induced by the membership service.

## 5.4 When to use Regular Views

In the previous sections we focused exclusively on i-views. There, we have shown that, in order to reduce the cost of program-controlled crash, some techniques can be applied to

postpone the suicide of a correct process. However, there should be a time where regular views are required.

According to the "two views" model, a regular view should be installed when a failure detector with conservative timeouts suspects a process, while an i-view is installed each time a suspect is raised by a failure detector with aggressive timeouts. Fortunately, this model is very similar with the detection model presented in Section 4.1. Hence, as we have two "levels" of failure detection, it is possible to start the view change that most adapts to the situation: suspicions from the conservative failure detector generate regular view changes.

But regular view changes are not required only on the case of failures. For example, there are events that can trigger the definition of a new view. The most obvious events that require regular views are those who explicitly force the modification of the membership group, i.e., the *join* and *leave* operations. As i-views consider only permutations on the set of processes, when a member joins or leaves the group, no permutation on the i-view can reflect these changes.

We can also define another event that triggers a regular view change. The same way as the installation of a regular view forces processes to crash, we suggest that the suicide of a process due to the incapacity to acquire missing messages should force a view change. In fact, if a process knows that it should commit suicide, it can have a "fair" behavior and execute *leave* before committing suicide. We consider this a "fair" behavior because if a process execute *leave* before die, it informs the group that the replication level will decrease, and the system can take some actions to compensate this. Otherwise, a new view will be generated only when this process is suspected by a conservative failure detector.

## 5.5 *Core*-resiliency and Scalability

One important element on the performance analysis of a Total Order Broadcast algorithm is its *delivery latency*. We consider delivery latency as the time elapsed between the broadcast of a message $m$ by the source process and the first time it is delivered to the application. On RBP, latency depends on the resiliency level, because a message is only delivered after this resiliency level is achieved. Indeed, if we consider a $n$-resiliency and the number of processes grows up, the latency may reach undesirable levels. To prevent such effects, we may use the own structure of the two-level membership to bound the delivery latency to acceptable levels. Indeed, by defining a $k$-resiliency (where $k$ is the number of processes in the **core** group) and controlling the maximum number of process in the core group, it is possible to limit the latency levels even if the system scales up.

This solution is somehow related to the architecture proposed by Maxemchuk for its TRMP protocol [15], where nodes are hierarchically structured in order to reduce the number of messages exchanged at global scale. Our proposal, however, does not require specialized nodes as in the case of TRMP, but simply relies on the self-stabilization mechanism from the two-level membership. Indeed, we assume that eventually only a subset of the processes will be kept into the core group, keeping the delivery latency tied to a $k$-resiliency level where $k \leq n$. Further, due to the VSC properties, we can safely relax this mechanism by allowing at most $k' < n$ of processes simultaneously in the core-group.

Indeed, in a previous work [2] we observed that the delivery latency is closely related to the number of processes in the token list, or by extension, in the core group. These he experiments, conducted on the **ID/HP i-cluster** from the ID-IMAG laboratory[1], evaluated the delivery latency of the RBP protocol with 4, 8, 16, 32 and 64 machines (one process/machine), comparing different *core*-resiliency levels. Hence, Fig. 8 presents the protocol performance when messages arrive according to a Poisson process of rate 10 messages per second.

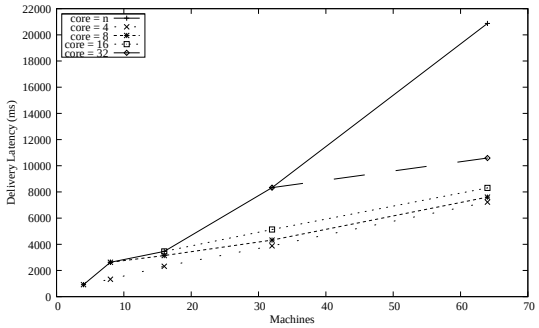


Figure 8: Comparison between the performance of different core-resiliency levels

We observe that by fixing the number of elements in the core group, we can reduce the delivery latency to a certain level (even if the overhead caused by the increasing network traffic still represents an important factor). Therefore, a careful choice of a small core-group allows the protocol to deliver messages with latency levels similar to conventional techniques (we should not forget that the Atomic Broadcast based on the Consensus requires from 6 to 8 communication steps), while taking advantage of the reduced number of exchanged messages that characterizes RBP.

# 6 Related Works

In the literature there are some examples of protocols that rely in the same principles of RBP. Actually, most of them (RMP [16], TRMP [15]) are evolutions from RBP, and were developed by the same research group.

RMP was the first successor of the RBP. The most evident change is the use of IP multicast, what allows the protocol to reduce even more the cost of a communication step (compared to an unicast *send to all*). Further, RMP includes some other features such as the acknowledgement of multiple messages in a single ack and the selection of the resiliency level. Most of these innovations were proposed with the specific objective to increase the performance of the protocol. Nevertheless RMP has kept most of the structure from RBP, such as the Three-Phase Commit (3PC) protocol in order to define the membership.

[1] http://www-id.imag.fr

TRMP [15] is a time driven version of the RMP protocol, and was presented as an Internet multicast protocol for the stock market. Due to the complexity of a world-wide distributed stock market system, TRMP has to deal with other problems like scalability, fairness and communication authentication, in addition to time constraints. To deal with scalability, TRMP proposes the use of a hierarchy of rings, instead of one single ring, as used by RBP and RMP. This avoids a long stabilization time, reduces the probability that the system stops due to the failure of a remote process (in particular, a *world-wide Reformation*), and allows the assignment of different levels of trustiness to the processes. These hierarchical rings are organized such that processes of the higher ring are also members of a lower ring, or at least, the lower ring receives retransmissions from the higher ring. In the same way, secondary sources can participate by sending messages to primary sources, which will insert them in the primary token ring. However, TRMP implements a centralized reformation server, with redundant reformation servers are available in case of failures.

Otherwise, few works try to adapt Total Order Broadcast to large networks. Recently, Guerraoui *et al.* [11] proposed FSR, an algorithm that intends to optimize the throughput of a Total Order Broadcast algorithm. FSR is an hybrid approach based on the fixed-sequencer strategy that do uses a token ring to ensure fairness among the nodes. Unfortunately, FSR uses an one-level VSC membership, which forces the delivery latency to be linear with the number of processes. We strongly believe that FSR can benefit from a two-levels membership, as we did with RBP, to improve its latency and scalability.

# 7 Conclusions and Future Works

In this paper we addressed the problem of Total Order Broadcast in the context of large-scale distributed systems. Traditional algorithms are not fit for those environments as they need to exchange too many messages to ensure a global delivery order. Hence, we study RBP, a distributed agreement protocol that can operate in environments subjected to message losses while still providing good performance rates and low network traffic. Due to its communication performance levels, RBP is an interesting alternative for a distributed application that should be deployed over a large area network such as a computational grid or a P2P network.

RBP presents, however, some conceptual flaws. Indeed, it relies in a group membership protocol that may induce inconsistencies, especially when dealing with large-scale systems. Most of these problems are related to the original group membership defined by the protocol, and therefore can be compensated by the use of up-to-date techniques. Hence, we worked on a group membership service that not only ensures consistency properties but is optimized to the operation in a wide-area network. For instance, we employed a new technique used to minimize the problems generated by wrong failure suspicions, reducing the membership overhead and minimizing the need for membership view changes. This technique is the subject of recent publications and to the best of our knowledge has not yet been used in any other implementation.

While studying RBP, we also observe some potentialities for the creation of an optimistic protocol: the way as it manages views and Total Order allows us to imagine possible solutions that do not block the processing of messages when there is a failure. We believe that the study of these situations can lead us to develop a family of protocols with high performance and optimized view change.

# References

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On the weakest failure detector for quiescent reliable communication. Technical report, Ithaca:Cornell University, July 1997.

[2] Luiz Barchet-Estefanel. iRBP - a fault tolerant total order broadcast for large scale systems. In *Proceedings of the EuroPar 2003*, LNCS Vol. 2790, pages 632–639, 2003.

[3] Kenneth Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 123–138, November 1987.

[4] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *The primary-backup approach*, chapter 8, pages 199–216. ACM Press Books, Addison-Wesley, second edition, 1993.

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

[6] Jo-Mei Chang and Nicholas Maxemchuk. Reliable broadcast protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, 1984.

[7] Bernardette Charron-Bost, Xavier Dfago, and Andr Schiper. Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently. In *Proceedings of the 21th International Symposium on Reliable Distributed Systems*, 2002.

[8] Gregory Chockler, Idith Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[9] F. Cristian and S. Mishra. The pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the 2nd International Symposium on Autonomous Descentralized Systems*, pages 215–221, April 1995.

[10] Xavier Dfago. *Agreement-related Problems: from semi-passive replication to totally ordered broadcasts*. PhD thesis, EPFL, Switzerland, 2000.

[11] Rachid Guerraoui, Ron Levy, Bastian Pochon, and Vivien Quma. High throughput uniform total order broadcast protocol for cluster environments. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2006.

[12] Rachid Guerraoui and Lus Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[13] Rachid Guerraoui and Andr Schiper. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS-17)*, pages 578–585, May 1997.

[14] Vassos Hadzilacos and Sam Toueg. *Fault-tolerant broadcasts and related problems*, chapter 5, pages 97–146. ACM Press Books, Addison-Wesley, second edition, 1993.

[15] Nicholas Maxemchuk and David Shur. An internet multicast system for the stock market. *ACM Transactions on Computer Systems*, 19(3):384–412, 2001.

[16] T. Montgomery. *Design, Implementation and Verification of the Reliable Multicast Protocol*. PhD thesis, West Virginia University, 1994.

[17] T. Montgomery, J. Calahan, and B. Whetten. Fault recovery in the reliable multicast protocol. Technical report, november 1995.

[18] Andr Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[19] Andr Schiper. *Distributed Algorithms: Lecture notes for the Graduate School in Computer Science EPFL-LSR*. 2002.

# Contents

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399