Analyzing RBP, a Total Order Broadcast Protocol for Unreliable Channels

Luiz Angelo Barchet-Estefanel

Swiss Federal Institute of Technology, Lausanne (EPFL)
School of Computer and Communication Systems (I&C)
Graduate School in Communication Systems 2001−2002

Analyzing RBP, a Total Order Broadcast Protocol for Unreliable Channels

Project Report

Luiz Angelo Barchet Estefanel
July 3rd 2002

Table of Contents

1 Introduction

RBP is the acronym for Reliable Broadcast Protocol. It was published by Chang and Maxemchuck [CHA 84] in 1984, and 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 and implementations still make use of its concepts (for example, RMP [MON 94], Pinwheel [CRI 95] and TRMP [MAX 2001]).

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.

In order to provide "high−performance characteristics" [MON 95], the acknowledgements are piggybacked with extra functionalities: at the same time RBP assigns a sequence number to a message, the token is passed. This approach orders the data packets consistently across all sites, and provides the means to pass the token to the next process in the ring.

When a site gets the token (i.e., it becomes the sequencer), it broadcasts an acknowledgement if and only if it has seen all data messages since the last acknowledgment it received. When a process detects that it has lost a message (using the message identifier), it uses negative acknowledgements to request retransmission of any missing message. When all messages since the last acknowledgement were received by the current sequencer, it can broadcast its acknowledgement and thus pass the token. Hence, when a process sends an acknowledgement, it is guaranteed that it has all previous sequenced messages.

RBP also should deal with process failures, and for that, its definition considers a "Reformation Phase", where crashed process are excluded from the membership. Nevertheless, RBP is an old protocol, that was proposed in a time when most of distributed system "building blocks" (like Consensus, Reliable Broadcast, Failure Detectors, etc.) did not exist. By this reason, the Reformation Phase cannot ensures most of the group communication properties.

We started our analysis by comparing RBP with another Total Order solution, the Primary−Backup+VSC replication model. As View Synchronous Communication (VSC) is widely studied, it presents many techniques that can be used in order to update RBP. One of these techniques, called "two views" model [CHB ??] is specially interesting because it allow us to take advantage from failure detectors with aggressive timeouts while minimizing the drawbacks from incorrect suspicions. Based on the analysis we've done, we propose an improved version from the RBP protocol that uses the techniques studied in this work.

In Chapter 2 we present the system definitions and models considered in this work, as well as a review from the properties of Total Order Broadcast that RBP must ensure.

Chapter 3 describes the RBP protocol. We decided to present it describing its operational model in a failure−free environment, and then in a the presence of process failures. We believe that this "step−by−step" description allow a better analysis from the protocol, its characteristics and drawbacks.

Chapter 4 analyses the Primary−Backup replication model. We could identify many similarities between this approach for Total Order Broadcast and RBP. However, as Primary−Backup is widely known and studied, it present many solutions that can be applied on RBP.

Chapter 5 quickly describes how the improved RBP was implemented, as well as the problems we found and the tests we have done with it.

Finally, Chapter 6 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. By the RBP definition, the set of processes is dynamic: a process that fails can be removed from the set, as well as new processes can join the set. Thus, 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.

The communication channel definition that is more adequate to model the existing transport layers is the following [AGU 97]:

- **Fair−lossy channel** − if *p* send 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).

The reason why we consider the definitions of reliable and unreliable communication is that, in opposition to many agreement protocols presented in the literature, RBP was designed to use unreliable communication. In fact, RBP implements a reliable communication within the protocol (using a mix of ACKs, NACKs and retransmissions). If we consider that channels "behave well", i.e., they do not lose too many messages, the mechanism constructed by RBP can reduce the cost of a transmission.

2.1 Total Order Broadcast Specification

While RBP was initially designed to use broadcast primitives, it can easily be adapted to multicast, i.e., the messages are sent to *Dest(m)*, the set of destinations of a message *m.* According to its objectives, RBP belongs to the set of Total Ordered Multicast protocols. The problem of Total Order Multicast can be defined by four properties: Validity, Agreement, Integrity, and Total Order [DEF 2000].

- **VALIDITY** − If a correct process broadcasts a message *m* to *Dest*(*m*), then some correct process in *Dest(m)* eventually delivers *m*.
- **UNIFORM AGREEMENT** − If a process in *Dest(m)* delivers a message *m*, then all correct processes in *Dest(m)* eventually deliver *m*.
- **UNIFORM INTEGRITY** − For any message *m*, every 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)*.
- **UNIFORM TOTAL ORDER** − If 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.

We should remark that the Agreement, Integrity and Total Order properties presented above are "Uniform". An Uniform property applies not only to correct processes but also to faulty processes. While the basic definition of RBP suppose 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 4.2 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:

- **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'*.

3 The RBP Protocol

While the original definition of RBP [CHA 84] describes how Reliable Broadcast and Total Ordering can be achieved, no precise algorithm is given. Additionally, the protocol mixes many structures (like reliable communication and total ordering) in order to minimize the number of control messages. This interleaving makes difficult the analysis of each functionality, and is the main reason why the properties for Reliable Broadcast and Total Order were not yet prove for RBP.

To help the analysis and the understanding of the protocol and its peculiarities, we decided to present the protocol step by step, showing each functionality separately, merging them and increasing the environment complexity, to reach the RBP definitions.

3.1 Operation Without Failures

In this section, we present the basic structure of the RBP protocol, in an environment where only message losses can occur (process failures will be described in Section 3.2). Here, we describe how to achieve Reliable Broadcast and Total Ordering, and how they can be implemented together, as in the RBP protocol.

3.1.1 Reliable Communication and Reliable Broadcast

Reliable Broadcast (or Multicast) can be easily implemented in asynchronous systems with reliable channels: when a process *p* wants to R−Broadcast a message *m*, *p* sends *m* to all processes. When some process *q* receives *m* for the first time, then *q* sends *m* to all processes and after this, R−Delivers *m*.

However, providing Reliable Broadcast in an environment subjected to messages losses (as the fair−lossy channels that we consider in this work) is not so easy. Even if processes do not crash, we can't be sure that a process correctly received a message unless we have some feedback from this process. Hence, a common solution is to implement point−to−point reliable communication over (point−to−point) fair−lossy channels, and implement Reliable Broadcast on the top of it.

To transform an unreliable channel, the basic principle is to retransmit a message until the reception of an acknowledgment. Consider ***SEND*** and ***RECEIVE*** as the primitives that provide reliable communication, and ***send*** and ***receive*** as the primitives for unreliable communication. To execute ***SEND(m)*** to *q*, a process *p* copies *m* into an output buffer (Figure 1) and executes ***send(m)*** repeatedly until it receives and 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 removes *m* from the output buffer.

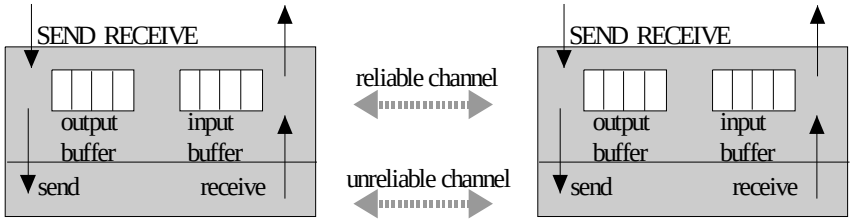

Figure 1 − Providing reliable channels over unreliable channels

Based on the implementation of reliable communication presented above, it is possible to implement Reliable Broadcast as if the channels were reliable. In order to make a Reliable Broadcast, each process must wait for the *ack* from all processes in the destination set.

However, simply retransmitting a message until receiving and acknowledgement has a great drawback. If each process that receives the broadcasts sends back an acknowledgement, soon the network will be flooded with *ack* messages. As scalability must always be a concern in distributed systems, this is not a good solution.

A better solution implies reducing the number of exchanged messages. For example, RBP considers a ring structure. If we ensure that the token is reliably passed, processes can use this token to check that messages were delivered by all processes (the only restriction is that all receives should be members of the ring).

Consider a process *p* that wishes to R−Broadcast a message *m* to all processes along a ring (logical or physical). If only the token passing is reliable, *p* can do the following: once it receives the token, it appends the message *m* to the token contents, and passes the token to the next process. If each process *q* that receives the token delivers the message *m* attached to the token, at the end of the token passing, when *p* gets the token again, *p* can be sure that all processes delivered *m*. Instead of receiving an *ack* from every process, the sender *p* has just to wait the return of the token.

This method, however, does not use well the network resources. As only one process each time can append a message to the token, processes that need to send many messages are forced to wait until all processes get the token, even if they have no messages to send (actually, the token behaves like a "permission" to send messages).

A better possibility considers that each sender unreliably broadcasts the message *m* to all processes. The process that has the token appends to the token all messages that are in its buffers (including the messages received from other processes), and delivers them. After, when the token is passed to a process *q*, it verifies if the messages appended to the token were already delivered or not. If they were not, *q* delivers the "missing" messages, append new ones to the token and passes it away. This ensures that all processes that receive the token will deliver a message, and at the same time, allows processes to use better the network resources (the token is used only to ensure that a message is received by all processes, not to restrict message transmission).

A simple algorithm for the "token−based reliable broadcast" can be seen in Figure 2. It express the algorithm for the senders (lines 1−3), token holders (lines 4−13) and receivers (lines 14−21). While the role of senders is only execute a "send to all", and receivers deliver messages as they arrive, the token holder has more responsibilities. Every time a process receives the token, it uses the token to check if some message was missing (comparing the *token.delivered* list with its own delivered queue). If this is the case, the token holder can acquire the missing messages from the *token.delivered* list, and delivers them.

This way, processes use the token to verify the "history" of delivered messages, and by this reason, all processes eventually deliver a message, according to the Reliable Broadcast properties.

While simple, this solution has a problem: this algorithm is not optimal in relation to the token size. As the *token.delivered* list should include all delivered messages, soon the token message becomes very large. In order to reduce this problem we can append to the token only a message ID’s, instead of the full message.

```
1: Sender:
2:   procedure RBroadcast(m)                    {To RBroadcast a message m}
3:     send (m) to all

4: Token holder (code of process s):
5:   Initialisation:
6:     token = {delivered[ ] ← ε}                {token, composed by  the history of all messages delivered }

7:   Upon reception of token from s−1 mod n      {when becomes the token holder}
8:     if ∃ m' in token.delivered  ∉  deliveredQs   {m' was not yet delivered}
9:       deliver (m')
10:    deliveredQs ← deliveredQs ∪ {m'}           {adds m' to the delivered queue}
11:  for ∀ m' in deliveredQs ∉ token.delivered    {pick all messages not yet in token.delivered}
12:    token.delivered ← token.delivered ∪ {m'}   {add the delivered message to the history}
13:  SEND token to s+1 mod n                      {reliably passes the token to the next process}

14: Destinations (code of process p):
15:  Initialisation:
16:    recvQp[ ] ← ε                             {set of received messages (received queue)}
17:    deliveredQp[ ] ← ε                        {set of delivered messages (delivered queue)}

18:  Upon  reception of m                        {when receive a message sent by a sender}
19:    recvQs ← recvQs ∪ {m}                     {add the message to the received queue}
20:    deliver (m)                               {delivers m}
21:    deliveredQs ← deliveredQs ∪ {m}           {adds m to the delivered queue}
```

Figure 2 − Token−based reliable broadcast

The drawback of this new solution is that a process can no more get "missing" messages from the token. Instead, it has to request retransmission of the message. This situation is presented in Figure 3. If a message is broadcasted at time *t=0*, but process *r* did not receive it, it will detect that a message is missing when it received the token at time *t=1*, and thus, requires retransmission of that message. This retransmission request can be considered as a negative acknowledgement (*nack*), and if the network behaves well most of the time (as it normally happens in a LAN), the number of *nack* is insignificant if we compare with an *ack* solution. If we require that a process only passes the token away after having delivered all previous messages (that means, after having acquired the missing messages), this protocol provides Reliable Broadcast as well.

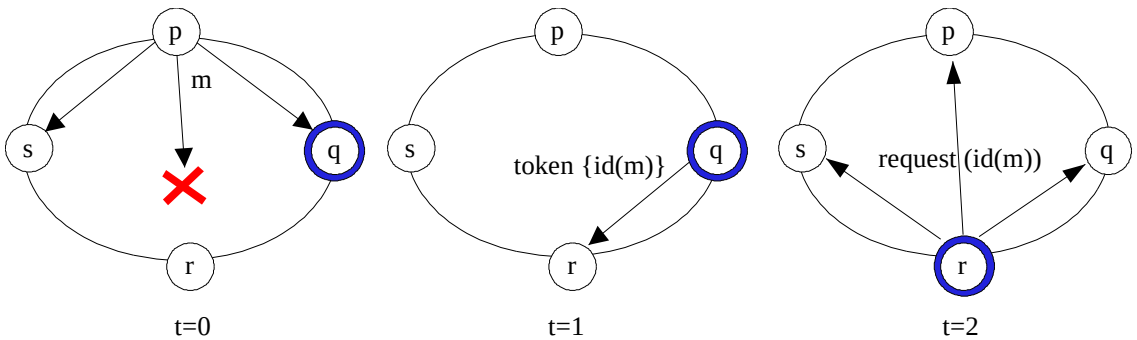

Figure 3 − Process r has missed the message *m*

A variant of this solution, considers that the token holder sends the the message ID from delivered messages to all processes (Figure 4). This solution can help to speed−up the acquisition of missing messages: even if a process does not receive the token, by receiving the message ID from delivered messages (at time *t=1*) it can verify if something is missing, and thus, it can request lost messages concurrently with the token passing (for example, at time *t=2*). When a process finally

becomes the token holder, it likely has all messages. Therefore, if correctly applied, the retransmission mechanism compensates eventual message losses. As consequence, there is no need for "reliable" channels, because the protocol can implement itself the necessary mechanisms to acquire missing messages.

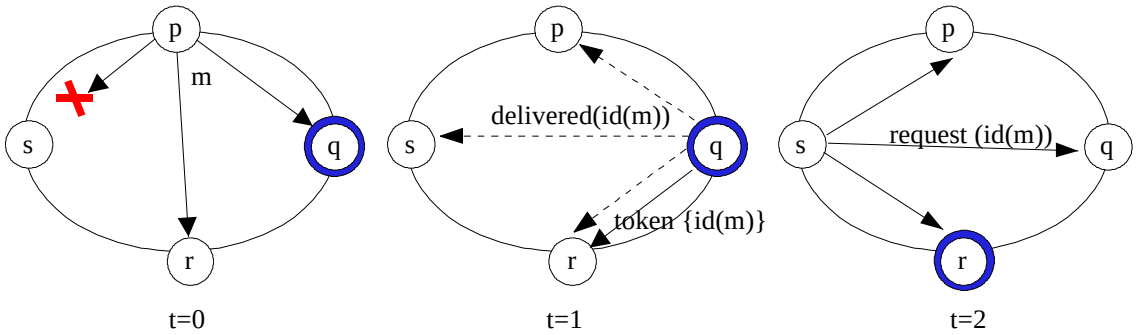

Figure 4 − Process s requests retransmission concurrently to the token passing

In Figure 5, an algorithm for this new solution is presented. The role of senders is still a simple "send to all", but receivers (lines 19−32) and token holder (lines 7−18) need to reflect the use of message IDs and retransmission requests.

As consequence, when a process receives the token, it first checks if all messages in *token.delivered* were already delivered. If the token holders has missed a message *m*, it sends a retransmission request, and waits until this message is received (otherwise, we cannot ensure that a message was delivered by all processes).

Once the token holder has delivered all messages from *token.delivered*, it appends to *token.delivered* new messages it has delivered, and passes the token to the next process. Furthermore, it sends the message ID from these new messages to all processes.

Upon the reception of a message "delivered(id(m))", a process can verify if this message was already received, otherwise it requests its retransmission. However, this process does not block suntil receiving the answer: this request has the only objective to speed−up the processing, not to guarantee delivering (what is done by the token holder).

```
1: Sender:
2:   procedure RBroadcast(m)                          {To RBroadcast a message m}
3:     send (m) to all

4: Token holder (code of process s):
5:   Initialisation:
6:     token = {delivered[ ] ← ε}                     {token, composed by the history of all messages delivered }

7:  Upon reception of token from s−1 mod n            {when becomes the token holder}
8:    if ∃ id(m') in token.delivered s.t. m' ∉ deliveredQs   {m' was not yet delivered}
9:      if m' ∉ recvQs                                {m' was not received}
10:       do
11:         send request(id(m)) to all                {request the retransmission of m'}
12:       while m' ∉ recvQs                           {do this until receiving m'}
13:     deliver (m')
14:     deliveredQs ← deliveredQs ∪ {m'}              {adds m' to the delivered queue}
15:   for ∀ m' in deliveredQs ∉ token.delivered        {pick all messages not yet in token.delivered}
16:     token.delivered ← token.delivered ∪ {m'}      {add the delivered message to the history}
17:     send delivered(id(m')) to all                 {informs that a message m' was delivered}
18:   SEND token to s+1 mod n                         {reliably passes the token to the next token holder}

19: Destinations (code of process p):
20:   Initialisation:
21:     recvQp[ ] ← ε                                 {set of received messages (received queue)}
22:     deliveredQp[ ] ← ε                            {set of delivered messages (delivered queue)}

23:   Upon reception of m                             {when receive a message sent by a sender}
24:     recvQs ← recvQs ∪ {m}                         {add the message to the received queue}
25:     deliver (m)                                   {delivers m}
26:     deliveredQs ← deliveredQs ∪ {m}               {adds m to the delivered queue}

27:  Upon reception of delivered(id(m))               {indicates that this message was delivered by someone}
28:      for ∀ m' in token.delivered ∉ recvQs         {if the message was missed}
29:        send request(id(m)) to all                 {request retransmission of that message}

30:   Upon reception of request(id(m)) from q
31:    if m ∈ recvQs
32:      send (m) to q
```

Figure 5 − Token−based reliable broadcast using message IDs

3.1.2 Total Ordering

In distributed algorithms, Total Ordering 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, specially 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. There are many classes of totally ordered broadcast and multicast algorithms [DEF 2000]. According to its definition, RBP can be classified as a "moving sequencer" algorithm.

In the "moving sequencer" strategy, the process that assigns sequence numbers is not fixed, but moves among the processes. This algorithm can be implemented using a token: the process that holds the token is the only one that can assign sequence numbers to the messages (that's why the sequencer is usually called "token site" or "token holder"). If there are no failures, the token passing is enough to provide Total Order.

Once sequence numbers are assigned, all receivers eventually obtain them (due to the Reliable Broadcast properties). To simplify the modeling of our protocol, we can make strong assumptions, requiring that each token site assigns a sequence number to a single message, and then must pass the token to the next process.

This algorithm is very similar to the algorithm for reliable broadcast presented before in Figure 5. The main difference is that now the token does not contain the identifiers of all messages, just the last sequence number assigned to a message.

Basically, each "sequencing round" of the protocol can be described in three steps. First, someone (a sender) broadcasts a message *m* to all processes (Figure 6). The token holder picks this message, assigns a sequence number to it, and broadcasts the message ID and the sequence number to all processes (Figure 7). Finally, the token is passed to the next process in the ring (Figure 8).

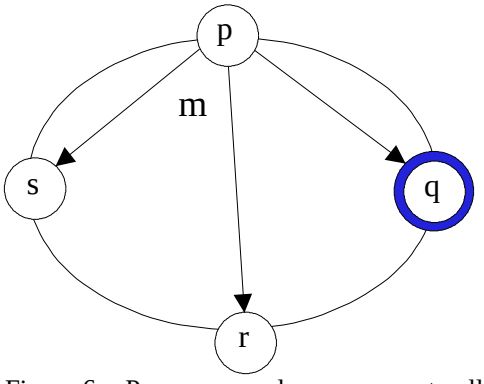

Figure 6 − Process p sends message m to all

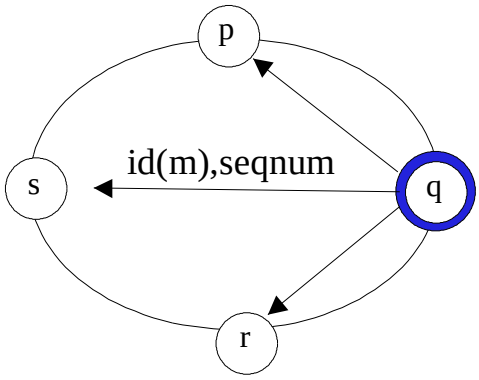

Figure 7 − Sequencer q sends {id(m),seqnum} to all

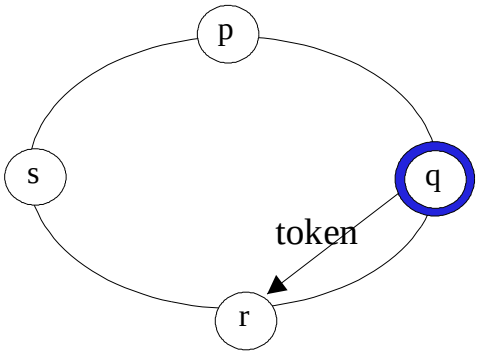

Figure 8 − Sequencer q passes the token

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 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 similar to the one used in Section 3.1.1: request retransmissions. Figure 9 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 9).

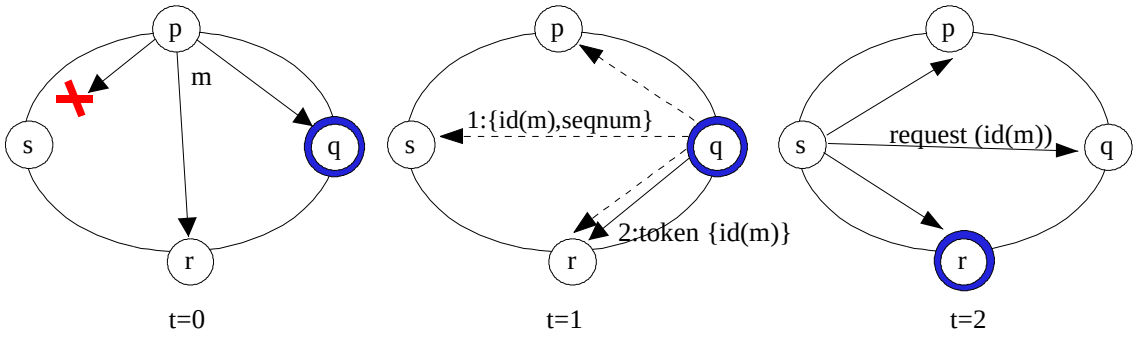

Figure 9 − Process s requests retransmission concurrently to the token passing

Another situation that requires message retransmission is when the sequence number is not received. To illustrate this, we show the following scenario. Starting in Figure 10, we see that at time *t=1* the message containing {id(m),1} is not received by process *s*. As processing continues, *q* send a message *m'* at time *t=2*.

At time *t=2*, process *s* has no way to know that a message was lost, because it didn't get enough information to detect the lost. Later, as presented in Figure 11, *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).

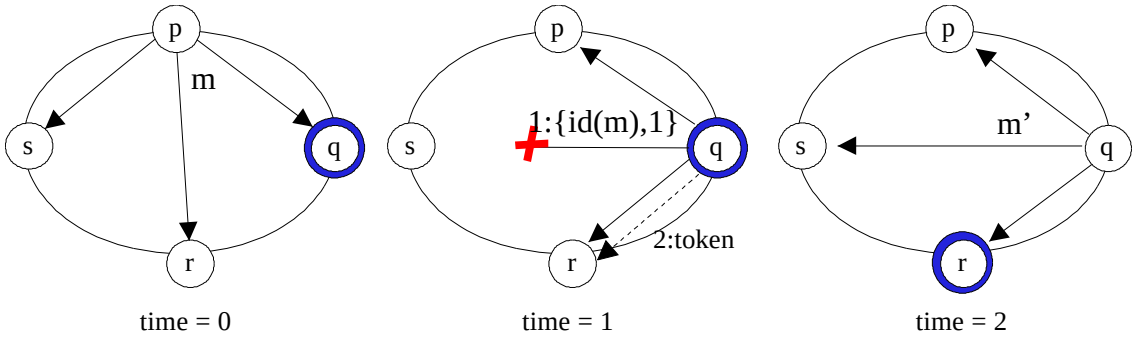

Figure 10 − Process s loses a message

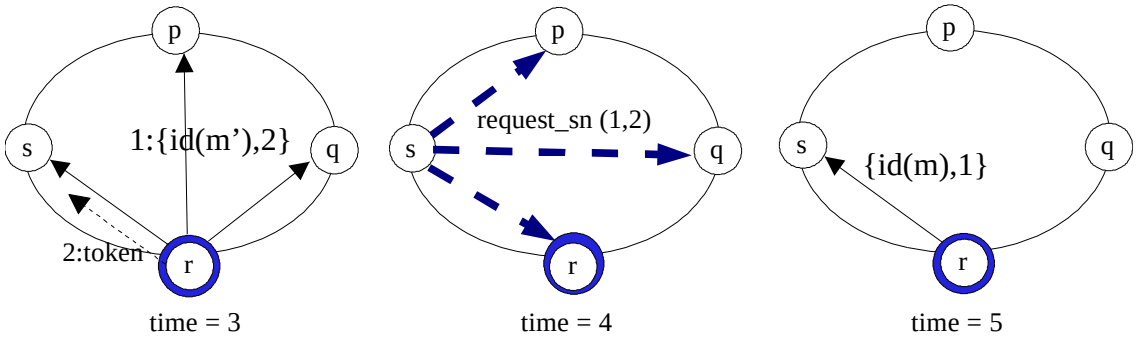

Figure 11 − Process s requests retransmission

Summing up, detection of lost messages and retransmission requests are essential operations for the moving sequencer algorithm if the network is unreliable. In Figure 12 we present an

algorithm that provides total order and reliable broadcast using the moving sequencer strategy. To understand better this algorithm, suppose a set of processes {*p*,*q*,*r*,*s*} organized around a ring, similarly to the set used on the previous examples. As in Figure 6, when *p* calls the procedure *TO−multicast(m)*, it will send the message *m* to all processes (line 3). All processes that receive this message will then add the message *m* to their "receive queue" *recvQ* (line 27).

If process *q* is the token holder, it will first check if there is no missing messages (lines 8−11) or missing sequence numbers (lines 12−16). If these verifications fail, then *q* requests the lost messages until the verification is successful.

Next, process *q* picks the message *m* from its *recvQ* (line 17), increases the value of *token.seqnum* (line 18), and assigns this sequence number to *m* by broadcasting the ID of message *m* and the sequence number *token.seqnum* (line 19). This was already shown in Figure 7. After that, process *q* sends the token to the next sequencer (line 20, or Figure 8).

If the process is not a token holder (for example, process *r*), it will wait to receive a message containing *id(m)* together with a sequence number *seqnum*. When this message is received (line 28) the process will check if *m* is in its recv*Q* (or request it), and add the tuple (id(*m)*, *seqnum*) to the sequenced queue *seqQ* (line 31).

Finally, the processes must deliver messages. In order to do this, they wait until a tuple in seqQ corresponds to the next expected message (line 35), and only then deliver that message (lines 36). As messages are ordered by a global sequence number, to deliver according this order ensures total order in the algorithm.

```
1: Sender:
2:    procedure TO−multicast(m)                    {To TO−multicast a message m}
3:          SEND (m) to all

4: Sequencer (code of process s_i):
5:    Initialisation:
6:       token = {integer seqnum ← 0}                          {token, composed by a sequence number}

7:    Upon reception of token from s−1 mod n                   {when becomes the token holder}
8:      if token.seqnum > netxdeliver                  {a seq. num. was missed, and is blocking the deliver of other messages}
9:        do
10:         send request_sn(nextdeliver,token.seqnum) to all    {request all  missing seq. num. messages}
11:       while token.seqnum > netxdeliver             {a seq. num. was missed, and is blocking the deliver of other messages}
12:     while ∃ {id(m'),num} in  seqQ_s  s.t. m' ∉ recvQ_s          {I missed the message m'}
13:       do
14:         send request(id(m)) to all                          {request the retransmission of  m'}
15:       while m' ∉  recvQ_s                                   {do this until receiving m'}
16:       seqQ_s ← seqQ_s ∪ {id(m'),num}                        {adds m' to the sequenced queue}
17 :    pick the first m' in recvQ_s ∉ token.sequenced          {pick the first message not yet sequenced}
18:     token.seqnum ← token.seqnum + 1                       {increases the sequence number (to ensure unique ordering)}
19:     send (id(m');token.seqnum) to all
20:     SEND token to s+1 mod n                               {passes the token to the next sequencer}

21: Destinations (code of process p):
22:     Initialisation:
23:       recvQ_p[ ] ← ε                                      {sequence of received messages (received queue)}
24:       seqQ_p[ ] ← ε                                     {sequence of ordered messages (sequenced queue)}
25:       integer nextdeliver_p ← 1                         {the sequence order of the next message to be delivered}

26:     Upon  reception of m                                  {when receive a message sent by a sender}
27:       recvQ_s ← recvQ_s ∪ {m}                             {add the message to the received queue}

28:     Upon reception of  (id(m);seqnum)                     {when received a message with a sequence number}
29:       if m ∉  recvQ_s                                   {m' was not received}
30:         send request(id(m)) to all                        {request the retransmission of  m'}
31:       seqQ_p ← seqQ_p ∪ {(id(m);seqnum)}           {add the message and its sequence number to the sequenced queue}

32:   Upon reception of request(id(m)) from q
33:     if m ∈  recvQ_s
34:       send (m) to q

31:   Upon reception of request_sn(first,last) from q
33:     for all {id(m'),seqnum} in seqQ_p s.t. first ≤ seqnum < last
34:       send {id(m'),seqnum} to q

35:       while ∃(m' ; seqnum') ∈ seqQ_p s.t. seqnum' = nextdeliver_p do     {while there are messages to deliver in order}
36:         deliver (m')
37:         nextdeliver_p ← nextdeliver_p + 1                 {increases the value of the next expected sequence number}
```

Figure 12 − Moving sequencer algorithm

3.1.3 A better Use of Message Passing

While the algorithm presented in Figure 12 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. Other example is the use of the SEND primitive for the sender and the token passing. Once SEND is expanded in "keep sending until receive *ack*", this represents many extra messages (perhaps, even more than all messages sent by the rest of the protocol), what is not desirable.

It is interesting to see that the definition of the RBP protocol has already a solution for these problems. In fact, if a single message piggybacks extra information, we can overload its functions and reduce drastically the number of messages. According to RBP, the messages broadcasted by the sequencer aggregate four separate functions as follows:

- acknowledgement to source *s* that message *m* has been received by the ring members;
- informs 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 in RBP is reduced to two messages, as presented in Figure 13. There, the message sent by the sequencer at time *t=1* 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*.

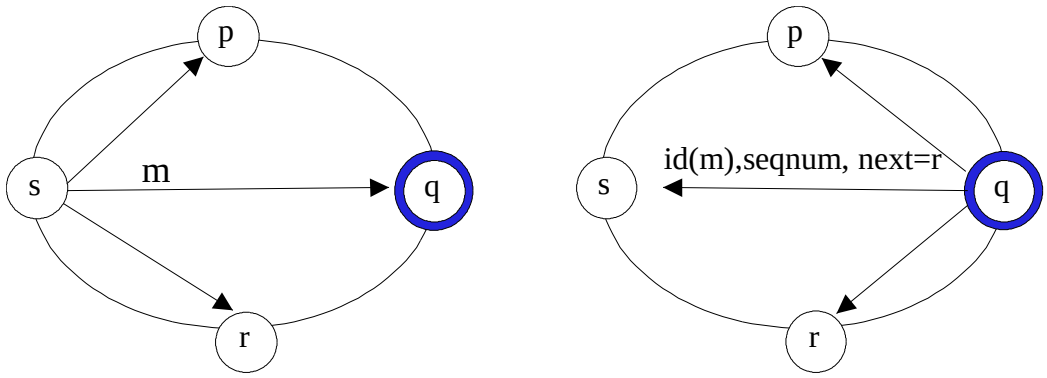

Figure 13 − process p sends message m to all

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 trigged. 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. 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 token holder must retransmit the acknowledgement to that source. An example of this situation is shown in Figure 14, where at time *t=0* the *ack* is not delivered to process *p*. For this reason, process *p* continues to retransmit the message *m*. When the token holder receives again the message *m* at time *t=1*, it sends back to *p* a copy of the token message, that was awaited by *p*.

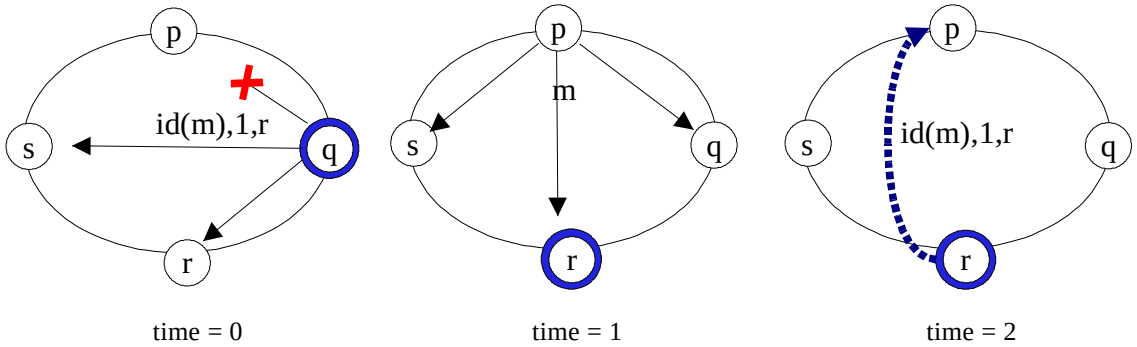

Figure 14 − Ack retransmission when the source lost the message

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. In Figure 15 we illustrate an example of this situation. At time *t=0*, process *q* sends the *ack* for a message *m*, passing the token to process *r*. At time *t=1*, r accepts the token, assigns a sequence number to a message *m'* and sends this sequence number to all processes.

This message sent by *r* should be the *ack* for process *q*, in order to stop sending the token to *r*. However, *q* did not receive this message, and thus, keeps retransmitting the token to *r* (time *t=2*). At time *t=4*, process *s*, that is the current sequencer, reacts to the retransmissions from *q*, by sending back to *q* the missing message, in order to notify *q* that its token was correctly passed.

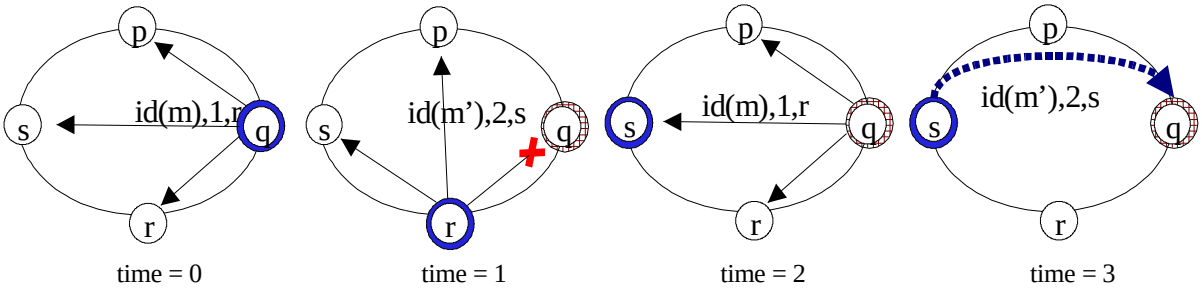

Figure 15 − Retransmitting a lost ack to an older sequencer

Figure 16 presents an algorithm that was improved in order to incorporate the remarks presented above. It is important to note that while this algorithm is based on that one from Figure 12, it was rewritten according to the model and notation presented in [DEF 2000]. This model presents some advantages, specially by allowing the expression of concurrent task, and not only an "event−driven machine". The main difference with the algorithm presented in Figure 12 is that there is no more a specific code for the receivers and the sequencers. Due to the ring structure, the processes must execute both roles, and thus, we decided to define a "single" code where the role of the process (line 24) determines the actions it must execute (for example, if it simply requests retransmissions or if it must keep requesting retransmissions until receive the answers).

"Old" messages are handled in two places of the algorithm. First, if the sender keeps sending a message that was already acknowledged, the algorithm sends back the respective *ack* (lines 16−19). If it is and "old" token, the current token holder detects it by comparing the sequence number received, and thus, retransmits the lost *ack* (lines 35−36).

```
1   Initialization:
2     recvQ_p ← ε                                   {sequence of received messages (receive queue)}
3     seqQ_p ← ε                                    {sequence of messages with a seq. number}
4     lastdelivered_p ← 0                           {sequence number of the last delivered message}
5     toknext_p ← p + 1(mod n)                      {identity of the next process along the logical ring}
      message id_p ← 0                              {each process keeps a "local" message id, in order to differentiate messages from
6                                                   the same source}
7     ts_p ← 0                                      {the last timestamp (sequence number) received}
8     if p=p1 then                                  {virtual message to initiate the token rotation}
9       send ACK(⊥,ts_p + 1, toknext_p) to p1       {format: (B, ts_p, toknext_p); B = (source id, message id) }

10  procedure TO−broadcast(m)                       {To TO−broadcast a message m}
11    do                                            {resent the message until be ACKed}
12      message id_p ← message id_p + 1             {increments the local message identifier}
13      send(m, B) to all
14    until receive ACK (B,−,−,−)                   {resent the message until be ACKed}

15  Task 1:
16    when receive (m, B) from q
17      if ∃ (B, seqnum) in seqQ_p                  {if is an "old" message}
18          if p = toknext then                     {only the token holder resends the ack}
19            send (B, seqnum) to q                 {retransmit the corresponding ack}
20      else
21        recvQ_p ← recvQ_p + (m, B)                {include the message (m,B) in the receive queue}

22    when receive ACK(B, seqnum, toknext) from q   {when receive an ACK}
23      seqQ_p ← seqQ_p + (B, seqnum)               {add the mesg Id and seq num in the seq. queue}
24      if p ≠ toknext then                         {if the process is not the next token holder}
25        if seqnum > ts_p+1 then                   {p expected seq num is lesser that the seqnum received}
26            request_sn (ts_p+1, seqnum)         {requests retr. of the ack messages from (ts_p +1 to seqnum}
27        if B ∉ recvQ_p then                       {if message (m,B) is not in the receive queue}
28          request_retransmission (B)              {requests retransmission of the message identified by B}
29      else                                        {if p is the next token site}
30        if  seqnum > ts_p+1 then                  {p expected seqnum is lesser that the seqnum received}
31          do
32            request_sn (ts_p+1, seqnum)           {requests retr. of the ack messages from (ts_p+1  to seqnum}
33          until (ts_p = seqnum)
34        else
35          if  ts_p > seqnum then                  {receiving an old ACK}
36            retransmits_acks (seqnum, nts_p)      {retransmits acks from seqnum to ts_p}
37          else  ts_p ← seqnum                     {if is the expected message}
38        if B ∉ recvQ_p then                       {if message (m,B) is not on the receive queue}
39          do
40            request (B)                           {requests retransmission of the message identified by B}
41          until (B ∈ recvQ_p)
42        wait until (recvQ_p \ seqQ_p) ≠ ε         {if there are new messages to sequence}
43            msg ← select first msg s.t. (m,B) ∈ recvQ_p and (B) ∉ seqQ_p   {selects a msg that was not yet sequenced}
44            do
45              send ACK(B,ts_p+1,toknext_p) to all   {assigns a seqnum ts_p+1 and sends the ACK for the sequenced msg}
46            until receive ACK from toknext_p

47    when receive request (B) from q               {resent requested messages}
48      if ∃ (m',B) in recvQ_p do
49        send m' to q

50    when receive request_sn (first, last) from q  {resent requested acks}
51      for each (B,seqnum) in seqQ_p s.t. first ≤ seqnum ≤ last
52        send (B,seqnum) to q

53  Task 2:
54    do forever
55      while ∃ m' s.t. (B, lastdelivered_p +1) ∈ seqQ_p  do   {check if there are messages to deliver}
56        deliver (m')
57        increment (lastdelivered_p)               {allows to delivers messages in order and only once}
```

Figure 16 − RBP algorithm (without failures)

3.2 Operation in Case of Failures

In the previous section we studied the RBP definitions and properties, and constructed the RBP protocol in an environment without processes failures. In this section we analyze the failure situations that the algorithm can face and how the protocol should react.

3.2.1 Someone has Failed

The mechanism of the RBP protocol 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 restart the token passing, the token list must be reconstructed (in a procedure called "Reformation Phase"), removing suspected processes and restarting the token passing [CHA 84]. 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. In the RBP definition, dated from 1984, failure detectors as proposed by Chandra and Toueg did not exist yet. In fact, it is mentioned that a process suspects a failure if it tries to send a message for a certain number of times, without obtaining any answer. This makes the failure detection very specific: a process only suspects process that it is trying to communicate. As the roles executed by the processes differ in importance (a sequencer worth more than a sender) this "ad−hoc" failure detection can help us to better understand the impact of a failure in the system.

In the following section we will identify some failure scenarios that illustrate possible problems, and we will analyzing the relation between the role of the failed process, the process that detects its failure and the impact into the system.

a. Simple process

If a process is just a sender, i.e., it is not the current sequencer nor the next sequencer, its fail does not affect immediately the protocol. We can assume this because no action is immediately dependent of this process. In Figure 17 there is an example of this situation. When process *q* sends an *ack* message, the only process expected to reply is *r*, and thus, the failure of *s* is not important at that moment.

As consequence, when a "simple" process fails, the detection of its failure (and specially, the actions to circumvent it) can be delayed. The moment where its failure needs to be handled is when the failed process should become the next sequencer.

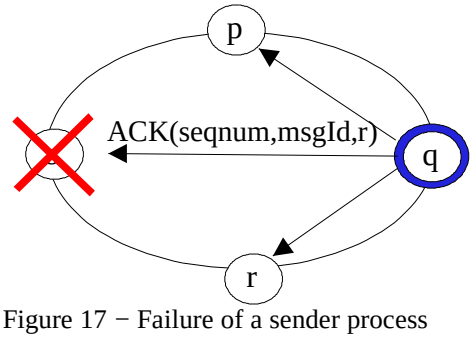

Figure 17 − Failure of a sender process

b. Next sequencer

While in the previous scenario the Reformation Phase can be delayed, if the next sequencer fails the token passing is blocked (Figure 18), forcing the reconstruction of the token list. Initially, is the sequencer (in the example, process *q*) that must detects the failure and start the Reformation Phase, as it is in directly communication with the next sequencer.

Unfortunately, this is not a simple scenario. Failure detection as defined in RBP does not allows to determine if the next sequencer was already failed or if it failed after receiving the token from *q*. If it had time to do some computations (or it was a false suspicion), a new and complex situation can occur: as the communication is not reliable, the next sequencer actually can accept the token, but the *ack* message is not received by all processes (including the sequencer), as shown in Figure 19.

This "misunderstanding" must be solved by the Reformation Phase. Actually, to ensure that consistency of the sequence number assignment is not violated (i.e., a new token does not assign the same sequence number to a different message), the Reformation Phase must be aware of both possibilities, and thus, act to compute all sequenced messages received by correct processes, no matter if it was a real failure or a communication failure.

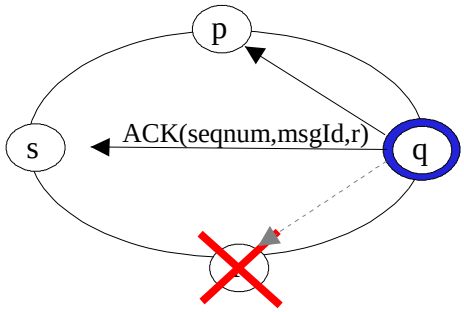

Figure 18 − Next sequencer fails

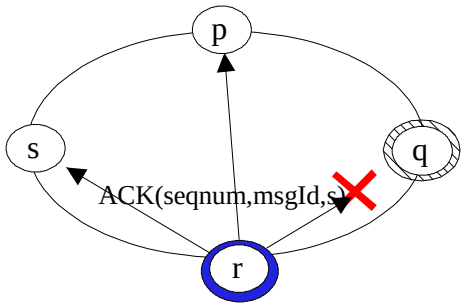

Figure 19 – ACK is not received

c. Sequencer

When the sequencer fails, there are also two possible situations that can block the token passing. In the first situation, the sequencer fails but the next sequencer has not received the token. In the second situation, the sequencer fails and the next sequencer has not yet all previous messages. Just to remember, a process is formally considered as a sequencer if it has accepted the token, i.e., it transmitted its own ACK message (we consider here that the previous sequencer has received that message, and by consequence, has no more obligation to detect the failure of the sequencer).

The first failure scenario occurs when the sequencer sends the token and fails, but the message is not received by the next sequencer (due to the unreliable communication or to the failure itself). In this situation, presented in Figure 20, the previous sequencer (process *p*) has already received the ACK. More, as the next sequencer *r* is not trying to communicate with the sequencer, it does not detect its failure.

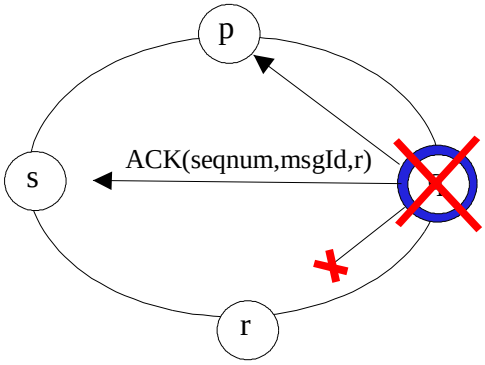

Figure 20 − Token message is lost

When this situation occurs, no process is directly monitoring *q*, and thus, no process knows that the **token is lost**. The failure detection based on "some message retransmission without answer" does not help, as no one is actively communicating with the failed process.

Due to this possibility, RBP requires that after some time interval sequencers must pass away the token, even if there is no message to sequence. In this case a sequence number is assigned to a NULL message "$ACK(\perp, ts_p + 1, toknext_p)$". Thus, when the sequencer has waited for messages to sequence but nothing has arrived, it should send an "empty" *ack* to transmit the token. Based on this "forced" token passing, Maxemchuck [CHA 84] proposed another kind of failure detection that circumvents the problem presented before: any process can suspect a failure of the sequencer and call the Reformation Phase if it does not detect activity from the sequencers for a large enough time. Informally, as sequencers are forced to pass the token from time to time, the absence of a token passing for a time too long can indicate that the token passing is blocked, even if it is impossible to know which process has failed.

In the second situation, the sequencer fails and the next sequencer has not yet all previous messages (Figure 21). Here, the token passing can be blocked because the next sequencer (process *r*) is unable to recover all lost messages before accepting the token (and perhaps, the failed process is the only one that could retransmit the missing messages). According to the failure detection presented in [CHA 84], the "next sequencer" is the process that shall raise a suspicion and call the Reformation Phase, as it cannot receive any answer for the retransmission requests.

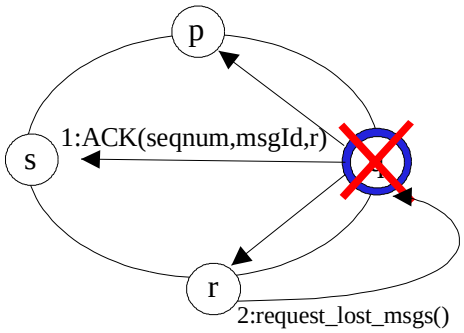

Figure 21 − Next sequencer has not all messages

d. Multiple Failures

As with simple failures, the impact of multiple failures is related to the token passing. If only simple processes fail, there is no immediate need of Reformation. If a sequencer or a next sequencer fails together with other processes (but not the two at the same time), we also fall in the previous situations.

However, if both sequencer and next sequencer fail at the same time, the **token is lost** and no alive process is directly concerned in monitoring the failures. In this scenario as well, the absence of token passing must be used as a failure trigger, and the Reformation Phase must be called by any alive process, as presented above.

3.2.2 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 happens 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 (defined at the beginning of Section 3.1.2), 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 $(r-1) \bmod n$ (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 $(r-n+1) \bmod n$ 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 specially apt to detect message stability, and thus, can execute a very efficient garbage collection.

3.2.3 Reformation Phase

According to the RBP definition, a failed process must be removed from the token list. When a failure is detected, the correct processes enter in a different execution mode, called "Reformation Phase", that aims to create a new token list. When a new token list is accepted, 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 [MAX 2001]. Figure 22 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 chose 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.

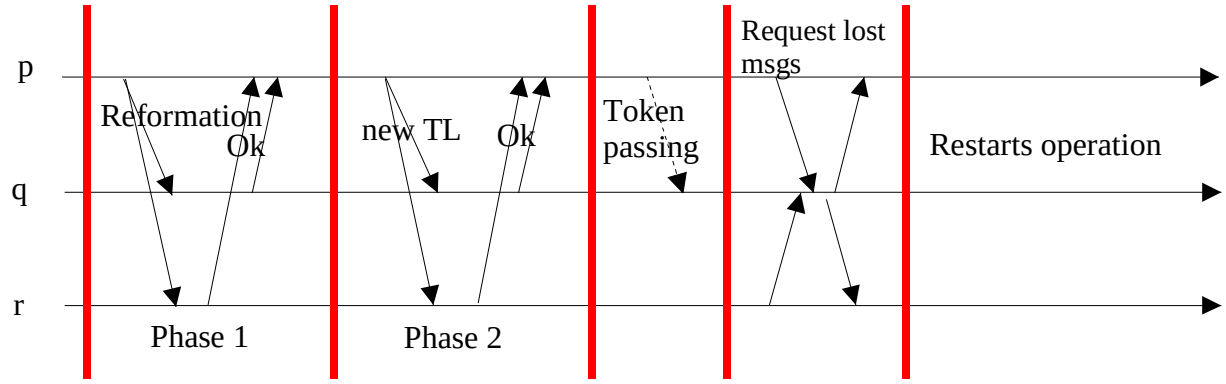

Figure 22 − Reformation phase in a good run

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:

- 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.

According to the RBP definition, these constraints can be grouped in three tests that a list must pass during the reformation phase [CHA 84]:

- **Majority test:** A site can only join only one list each time. 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 (RBP does not specify how these processes could recover messages in order to keep the application consistent).

Chang and Maxemchuck [CHA 84] present the algorithms for the originator (Figure 23), the "slaves" and the sequencer (Figure 24). These algorithms have different elements, as each participant has different roles in the reformation phase.

```
Reformation protocol for originator site i

Phase I: When a failure or recovery is detected, start Reformation;
         Broadcast an invitation to all sites in the broadcast group.
Phase II: Wait until either all responses received or TIMEOUT_i;
         If (all responses = "yes" and pass Majority and Resiliency Tests)
           New TL ← {all sites that responded};
           Announce New TL to all sites in the New TL;
         Else
           Announce "Reformation Abort" to all sites in the New TL;
           Modify TL version number, Wait and Restart;
Phase III: Wait until either all responses received or TIMEOUT_i;
         If (All responses from sites in New TL = "Yes")
           Generate "a New Token" and pass to New Token Site;
           Commit New TL;
           End of Reformation Phase;
         Else
           Announce "Reformation Abort" to New Token site;
           Wait and Restart;
```

Figure 23 − Reformation protocol for the originator site [CHA 84]

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 allow a process in the "normal phase" to discard messages coming from excluded processes.

```
Reformation protocol for slave j including new token site

Phase I: Wait until "Reformation Invitation" is received;
         If (Sequence Test passes and j does not belong to any reformation list)
           Vote "Yes";
         Else
           Vote "No";
Phase II: Wait until either "New TL", "Abort" is received, or TIMEOUT_j;
         If ("New TL" is received)
           If (j still belongs to this list)
             Recover all missing messages and then Vote "Yes";
             Commit the New TL;
             End of Reformation Phase (except the new token site);
           Else
             Vote "No";
         If ("Abort" is received or TIMEOUT_j)
           Leave the list that previously joined.
           Wait and Restart;

Authorization phase for new token site k

Phase III: Wait until either "New Token," "Abort," or TIMEOUT_k;
         If ("a New Token" is received and j still belongs to this list)
           Accept the token and Starts Acknowledging Messages;
           End of Reformation Phase;
         If ("Abort" is received or TIMEOUT_k)
           Wait and Restart;
```

Figure 24 − Reformation protocol for the slaves and new token site [CHA 84]

Updating the algorithm from Section 3.1.3 to deal with processes failures, we can define an algorithm as presented in Figure 25.

```
1   Initialization:
2     recvQ_p ← ε                                   {sequence of received messages (receive queue)}
3     seqQ_p ← ε                                    {sequence of messages with a seq. number}
4     stableQ_p ← ε                                 {sequence of stable messages (stable queue)}
5     lastdelivered_p ← 0                           {sequence number of the last delivered message}
      message id_p ← 0                              {each process keeps a "local" message id, in order to differentiate messages
6                                                   from the same source}
7     toknext_p ← p + 1(mod n)                      {identity of the next process along the logical ring}
8     ts_p ← 0                                      {the last timestamp (sequence number) received}
9     TL_p ← 0                                      {the token list version }
10    if p=p1 then                                  {virtual message to initiate the token rotation}
11      send ACK(⊥,ts_p + 1, toknext_p) to p1       {format: (B, ts_p, toknext_p); B = (source id, message id) }

12  procedure TO−broadcast(m)                       {To TO−broadcast a message m}
13    do
14      message id_p ← message id_p + 1             {increments the local message identifier}
15      send(m, B, TL_p) to all
16    until receive ACK (B, seqnum, toknext, TL_p)  {resent the message until be ACKed}

17  Task 1:
18    when receive (m, B, TL_q) from q
19      if (TL_q=TL_p) then
20        if ∃(B,seqnum') ∈ seqQ_p then             {if m was already sequenced, and the sender didn't receive the ack}
21          if p = toknext then                     {only the token holder resends the ack}
22            send ACK(B,seqnum',⊥) to q            {retransmits the corresponding ack}
23        else
24          recvQ_p ← recvQ_p + (m, B)              {include the message (m,B) in the receive queue}

25    when receive ACK(B, seqnum, toknext, TL_p) from q   {when receive an ACK}
26          seqQ_p ← seqQ_p + (B, seqnum)           {add the mesg Id and seq num in the seq. queue}
27      if (TL_q=TL_p) then
28        reset activityTimeOut                     {the sequencers are active}
29      if p ≠ toknext then                         {if the process is not the next token holder}
30        if seqnum > ts_p+1 then                   {p expected seq num is lesser that the seqnum received}
31          request_sn (ts_p+1, seqnum)             {requests retr. of the ack messages from (ts_p +1 to seqnum}
32        if B ∉ recvQ_p then                       {if message (m,B) is not in the receive queue}
33          request_retransmission (B)              {requests retransmission of the message identified by B}
34      else                                        {if p is the next token site}
35        if seqnum > ts_p+1 then                   {p expected seqnum is lesser that the seqnum received}
36          do
37          request_sn (ts_p+1, seqnum)             {requests retr. of the ack messages from (ts_p+1 to seqnum}
38          until (ts_p = seqnum) or failure(q)
39          if failure(q) then
40            Reformation()
41        else
42          if ts_p > seqnum then                   {receiving an old ACK}
43          retransmits_acks (seqnum, nts_p)        {retransmits acks from seqnum to ts_p}
44          else ts_p ← seqnum                      {if is the expected message}
45          if B ∉ recvQ_p then                     {if message (m,B) is not on the receive queue}
46            do
47            request (B)                           {requests retransmission of the message identified by B}
48            until (B ∈ recvQ_p) or failure(q)
49            if failure(q) then
50              Reformation()
51          wait until (recvQ_p \ seqQ_p) ≠ ε or timeout   {if there are new messages to sequence, or timeout}
52            if (recvQ_p \ seqQ_p) = ε then        {if there are no messages to sequence}
53              do
54                send ACK (⊥,ts_p+1,toknext_p, TL_p) to all   {passes the token to the next holder − acks a null message}
55              until receive ACK from toknext_p or failure(toknext_p)
56              if failure(toknext_p) then
57                Reformation()
58              else
59                msg ← select first msg s.t. (m,B) ∈ recvQ_p and (B) ∉ seqQ_p   {selects a mesg that was not yet sequenced}
60                do
61                  send ACK(B,ts_p+1,toknext_p) to all   {sends the ACK for the sequenced msg}
62                until receive ACK from toknext_p or failure(toknext_p)
63                if failure(toknext_p) then
64                  Reformation()

65    when receive request (B) from q               {resent requested messages}
66      if ∃ (m',B) in recvQ_p do
67        send m' to q

68    when receive request_sn (first, last) from q  {resent requested acks}
69      for each (B,seqnum) in seqQ_p s.t. first ≤ seqnum ≤ last
70        send (B,seqnum) to q

71    when detects ActivityFailure then             {Detects inactivity on the protocol }
72      Reformation()                               {starts reformation }

73  Task 2:
74    do forever
75      while ∃(B,seqnum') ∈ seqQ_p s.t. (m',B) ∈ recvQ_p do   {search through all sequenced messages}
76        if seqnum ' < (ts_p −n+1) then            {detects messages received by all processes (stability) }
77          stableQ_p ← stableQ_p + (m',seqnum')    {in practice, the protocol waits the roundtrip of the token}
```

```
77          seqQ_p ← seqQ_p \ {B,seqnum'}                    {remove msg from the recv queue and seq queue}
78          recvQ_p ← recvQ_p \ {m',B}
79      while ∃ m' s.t. (m', lastdelivered_p +1) ∈ stableQ_p do   {deliver the stable messages}
80        deliver (m')
81        stableQ_p ← stableQ_p \ {(m',−)}
82        increment (lastdelivered_p)                        {lastdelivered makes the delivering ordered}
```

Figure 25 − RBP algorithm with process failures

3.3 Other Protocols Based in the RBP

In the literature there are some examples of protocols that rely in the same principles of RBP. Actually, most of them (RMP [MON 94], TRMP [MAX 2001]) are evolutions from RBP, and were developed by the same research group.

Anyway, it is interesting to identify what was changed in the original RBP across the time. Here, we present only a simple review on the features of RMP and TRMP that we found interesting.

3.3.1 RMP

RMP was the first evolution 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"). Nevertheless, RMP includes many other changes in relation to the RBP protocol, as for example, acknowledgement of multiple messages in a single ack, management of multi−groups and selection of the resiliency level.

Most of these innovations were proposed with the specific objective to increase the performance of the protocol. It was also extensively verified (after all, the research group was based in the NASA Independent Verification and Validation Facility − IV&V) [WHE 95].

Even if RMP has extra functionalities in relation to RBP, it kept most of the structure from its RMP. An example is the failure detection. Failure detection is still based in the number of retransmissions a process executes without receiving answers. The difference is that RMP utilizes a dynamic analysis of the network connectivity, in order to set the number of retransmissions a process must execute before raising a suspicion.

The Reformation Phase was changed, but only to a limited degree. While RMP's Reformation is compliant with the virtually synchronous execution model (RBP does not provide virtual synchrony), it still uses a Three−Phase Commit (3PC) protocol in order to define the membership.

3.3.2 TRMP

TRMP [MAX 2001] 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 trustness to the processes.

These hierarchical rings are organized such that processes of the higher ring is also a member 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.

As the main ring can be considered as the "core" of all operations in the stock market used as example, the Reformation Phase must be extremely fast. For these reasons, the authors assume that instead of distributed, the Reformation is done by a centralized "reformation server". There are redundant reformation servers in case of failures.

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 "evolutions" of RBP.

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

According to the RBP definition, failure detection is mainly executed based in a number of message retransmissions without answer. On the other hand, failure detection has evolved considerably since the RBP publication, and the model of failure detection specified by Chandra and Toueg is essential to most distributed applications today.

Examining again the failure scenarios proposed in Section 3.2.1, 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 a "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".

An analysis on the scenarios presented in Section 3.2.1 show that 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 specially 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 Maxemchuck [CHA 84] 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 only 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 A Lower Bound on RBP Delivering

In the RBP definition [CHA 84], the delivery and the garbage collection are strictly related to *k*−resiliency. In fact, due to its construction around a logical ring, RBP can provide *n*−resiliency (all processes) easily, and by that reason, lower levels of resiliency are not discussed in the original paper. As *n*−resiliency also ensures message stability, these concepts are interchangeable in RBP.

In RMP [MON 94], however, the application can define the level of resiliency it wants (in [MON 95] 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. In order to analyze this, we suppose the failure situation presented in Figure 26.

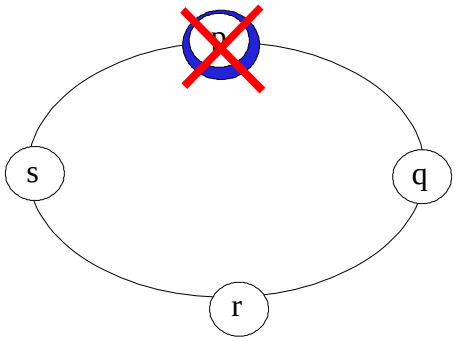

Figure 26 − Crash of the sequencer p

First, suppose that *p* assigns a sequence number to a message *m*, broadcasts it and crashes (real failure). Suppose also that once a process receive 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 provided by RMP can only be executed if the application does not require consistency among the processes.

4.3 Primary−Backup and View Synchronous Communication

In order to provide Total Order and Reliable Broadcast, a technique usually present in the literature [CHB ??] is the use of Primary−backup replication together with the View Synchronous Communication. While Primary−backup provides total ordering (as the RBP "normal phase"), View Synchronous Communication deals with the membership changes in a dynamic group (as the RBP Reformation Phase).

Because this is a widely studied technique, it presents interesting solutions that can be used to improve the RBP protocol. A short definition of these techniques is provided below, and in Section 4.4 we suggest some innovations for the RBP protocol, using the solutions presented here.

4.3.1 Primary−Backup Replication

The primary−backup replication technique consists in having one *primary* server and one or more *backup* servers (Figure 27). 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 acknowledgement 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 replicates 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 [SCH 2002]. 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.

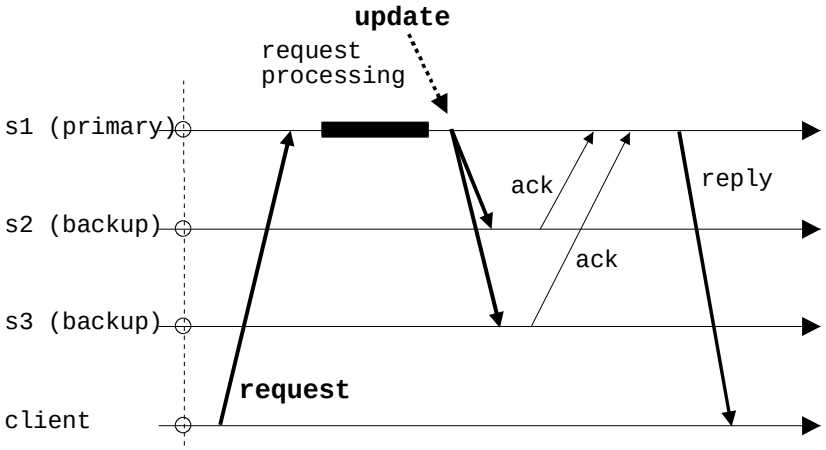

Figure 27 − Primary−backup replication

4.3.2 Group Membership and View Synchronous Communication

View Synchronous Communication (VSC, for short) 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. Let VSCast denote the primitive by which a message is broadcasted by a process in view v, and by VS−Deliver the primitive that delivers a message to a process in view v. The properties of the VSC can be considered as [CHB ??]:

- **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 [VIT 99], 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). Figure 28 illustrates how to use the VSCast primitive together with the primary−backup replication.

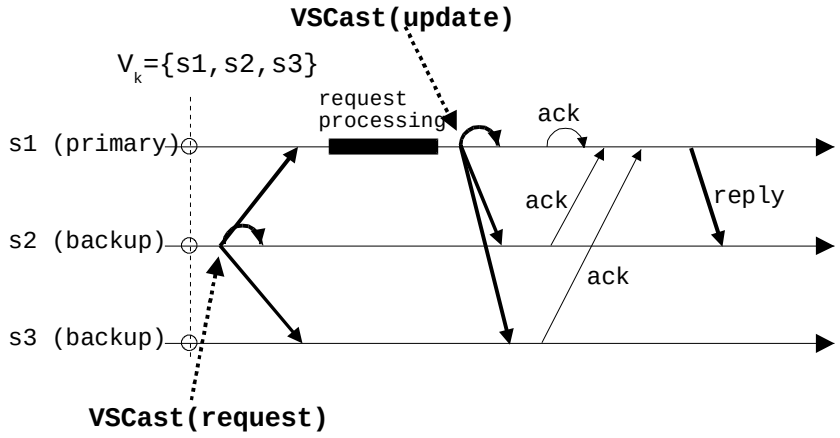

Figure 28 − Primary−backup replication and VSC

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 [SCH 2002] (Figure 29):

VSCast(m) executed by p_k:
  send (i,m) to all in $v_i(g)$

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. $\forall p_l \in v_i(g)$: wait until receive $unstable_l$ from p_l or p_l 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 $unstable_l$ was received)
    − Msg_k is the union of the sets $unstable_l$ received
  4. execute consensus among $v_i(g)$ with $init_k$ as the initial value
  5. let (View, Msg) the decision of consensus
  6. VS−deliver all messages in Msg not yet VS−Delivered
  7. if $p_k \in$ View, then install View as the next view $v_{i+1}(g)$

Figure 29 − Solving VSC view change [SCH 2002]

4.3.3 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, as discussed in Section 3.1.1: 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*.

In Section 3.1.1 we also observed that this implementation does not work with fair−lossy channels. A possible solution was presented in Figure 1, and basically 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 [CHB ??] 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 the message safely.

[CHB ??] 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 $\Diamond\mathcal{P}$.

The only way to solve the time−bounded buffering problem is to use a perfect failure detector $\mathcal{P}$: it is the only failure detector that allows removing a message with safety, because it does not make incorrect suspicions. As this is a strong requirement, View Synchronous Communication addressed this problem by inducing process−controlled crash.

4.3.4 Program−Controlled Crash and View Changes

The impossibility of solving the time−bounded buffering problem with a $\Diamond\mathcal{P}$ 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 this 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.

4.3.5 Two views

In a 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, [CHB ??] propose the use of two levels of GMS.

On the propose from [CHB ??], 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 $v_0, v_1, ..., v_i, ...$, the i−views between v_i and v_{i+1} are denoted as $v_i^0, v_i^1, ..., v_i^j, ..., v_i^{last}$. 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 $v_i^0,... , v_i^{last-1}$ 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 $v_i^0...v_i^{last-1}$ the ordinary view remains v_i.

This model in two layers allows us to solve the problem from Section 4.3.4. 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 $v_i^0 ... v_i^{last-1}$ 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 [CHB ??] (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.

4.4 Revisiting the Reformation Phase

As presented in Section 3.2.3, the reformation phase proposed in [CHA 84] is a three−phase commit protocol. If in a good run the reformation can be solved as in Figure 22, 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 [CHA 84]), the definition of a new token list is delayed until some list obtains majority of processes. However, this is not so simple, because according to the algorithms for the slave processes (Figure 24), if the "majority test" took too much time, any process is allowed to leave that list and join/start another list. Together, this 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, contrarily to the View Synchronous Communication model, RBP does not force excluded process to crash. Actually, the definition from [CHA 84] is too vague, and it supposes that if some correct process does not belong to a valid token list, it will start another reformation, until return 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 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 "period" too short, 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 its definition 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.

4.4.1 Similarities and Differences between Primary−Backup and RBP

As Primary−Backup and RBP are designed to solve the Total Order Broadcast problem, they present many similarities. 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 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.

When we compare the view change technique, however, we observe that the Reformation algorithm presents many drawbacks in comparison with the VSC+Membership model. First, the Reformation 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 processes 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*.

If RBP Reformation Phase does not provide *Same View Delivery*, it also lacks another important property, the *Sending View Delivery*. As this property confines the messages into the same view, it simplifies the task of controlling message delivery and garbage collection. The absence of this property usually forces the protocols to tag each message with extra information, like the view number [VIT 99], and to filter each message in order to eliminate messages from an "invalid" view. As presented in Figure 25, RBP needs to include a "Token list version" in all messages exchanged, and to verify this version every time a message is received. Even if these fields can represent only a small part of each message, the cost associated with the verification reduces the efficiency of this protocol, that was designed to have high performance levels [WHE 95].

Finally, 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 reduce 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 algorithm from Figure 29, we can define a new algorithm adapted to the RBP protocol (Figure 30).

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. ∀ 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 ← Msg, seqQ ← {} /* as all processes get Msg, these messages are stable */
  7. if p_k ∈ TL, then "install" TL as the next view TL_{i+1}

Figure 30 − First sketch of the "View Change" reformation protocol

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

A run in this algorithm will be similar to Figure 31.

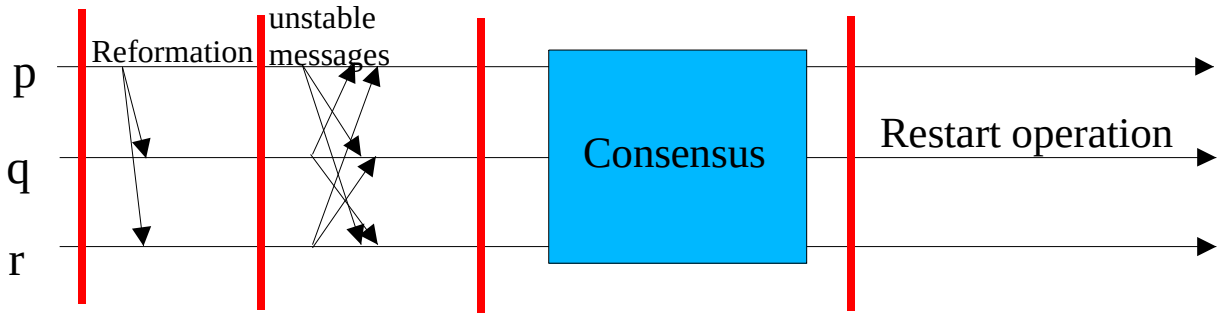

Figure 31 − a RBP view change run

Another important aspect from using a membership layer that provides *Sending View Delivery* as the algorithm from Figure 30 is that we can implement a VSCast−like primitive that provides a broadcast to the members of the group, taking from the RBP "normal phase" the responsibility to manage the group and its features.

4.4.2 RBP Reformation and Process−Controlled Crash

According to the examples presented in the previous section, the original RBP Reformation algorithm does not 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 [CHA 84].

By modifying "View Change Reformation" algorithm presented in Figure 30, we solve this problem. As presented in Section 4.3.4, 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 done, as presented in Figure 31. Here, if a process was excluded from the new view (line 7), it commits suicide.

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 ← Msg, seqQ ← {} /* 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

Figure 32 − "View Change" reformation protocol with program−controlled crash

4.4.3 I−Views on RBP

As Program−Controlled Crash has a non−negligible cost, we can also apply the technique of "two views" [CHB ??] 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 [CHB ??], 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 **{"token list core","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 in the definition of TRMP [MAX 2001], 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 belong 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 Figure 33).

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.

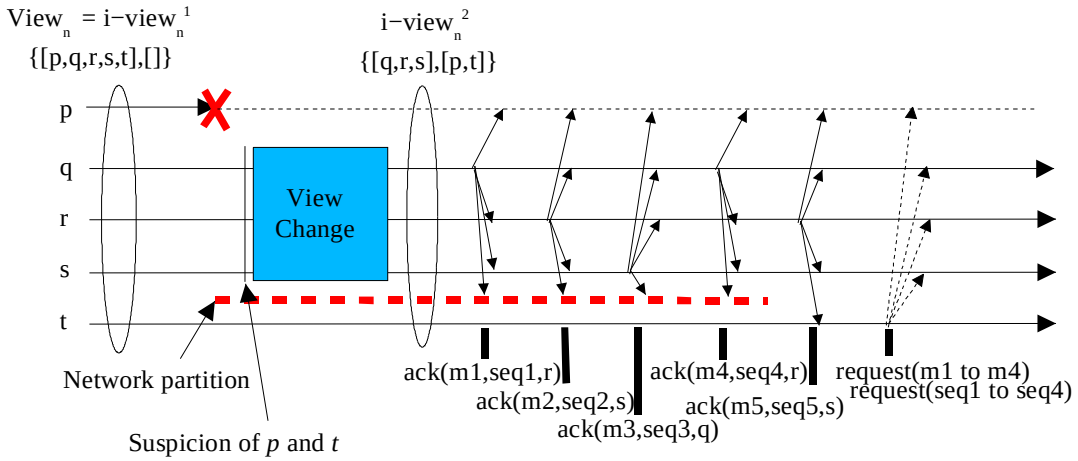

Figure 33 − I−view change and external members

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).

4.4.4 Optimizing the I−View Change

According to the i−view definition, its aim is to avoid blocking the protocol, not 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, this means that 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 Figure 34:

Upon suspicion of some process in TL_i by an aggressive Failure Detector
  RBroadcast (i−view, i)

Upon R−Deliver (i−view, i) by p_k for the first time
  1. send ack to all
  2. ∀ pi, wait until receive ack from p_i or p_i suspected
  3. let $initial_k$ contain (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}

Figure 34 − Optimized i−view change

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.

4.4.5 When to use Regular Views in RBP

In the previous sections we focused exclusively on i−views. There, we 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 more adequate: 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 iRBP

Based in the techniques presented on Chapter 4 to update the Reformation algorithm and circumvent the main problems from the original definition of RBP, we propose iRBP, the improved RBP protocol.

The main advantages from iRBP in relation to the original RBP protocol are:

- substitution of the original failure detection model by *heartbeat* failure detectors;
- a better separation between Total Order protocol and membership control;
- a view change protocol based in [SCH 2002] is used for membership control;
- use of program−controlled crash to solve the *time−bounded buffering problem*;
- use of the "two views" technique [CHB ??] and delayed garbage collection to minimize the cost of program−controlled crash due to wrong failure suspicions;
- optimization of the i−view changes, in order to reduce its cost;

In order to test this new specification, we implemented the protocol using the Neko/Java environment. Our main interest was to validate the operation of the RBP protocol together with the "two views" membership model. In the next sections we briefly describe the implementation, the main problems that were found, and some comments on the environment we used.

5.1 The Environment

iRBP was implemented in Java, using Neko, a framework for the development of distributed algorithm [URB 2001]. The main advantages to use Neko is its modular approach and its abstraction from the network level. In fact, the network abstraction from Neko allows the construction of a single program code that can be used for both simulations or real application.

The modular approach of Neko makes easy the development of a distributed algorithm: we can structure the application as a stack of layers, each one corresponding to a specific "building block". Processes communicate through message passing, but most of the complexity from the communication layers is masked by Neko (for example, there is no need to provide IP address and ports, like *sockets* require). As the layers are independent and communicate through well−defined interfaces (the basic interface comprises *send* and *deliver*), we can reuse layer and plug them to construct applications with different requirements.

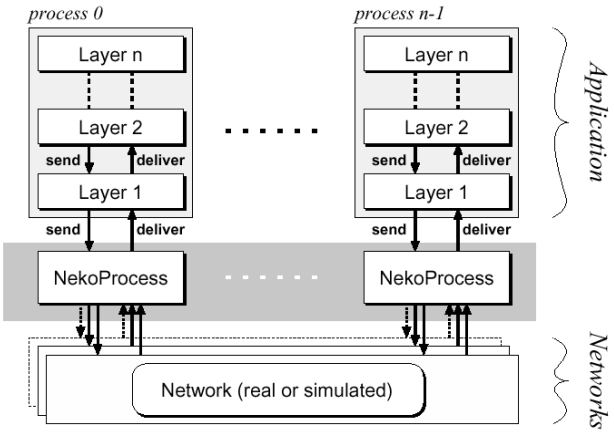

Figure 35 − Architecture of Neko [URB 2001]

As Neko was already used to implement many applications and experiments, like:

- Atomic Broadcast algorithms
- Failure detector components
- Consensus algorithms
- Reliable broadcast algorithms
- Evaluation of the cost of distributed algorithms using performance metrics
- Illustration of the FLP impossibility results
- Replicated servers

These previous experiments, together with the modularity and reusability of Neko, allowed us to profit from most pre−existent layers: we had only to implement the protocol−specific layers iRBP and MShip, as presented in Figure 37.

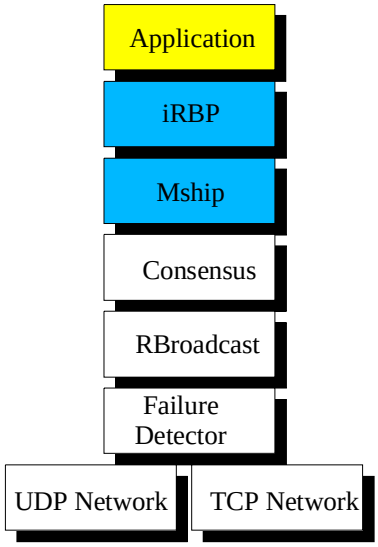

Figure 36 − iRBP Protocol Stack

Following the basic definitions from RBP, we implemented the token passing using unreliable communication primitives. Thus, UDP was chosen as the communication protocol for the "normal phase". However, as the we use Consensus and Reliable Broadcast, we were forced to use also TCP because these layers were designed for reliable communication. Thus, a better representation from the interrelation among the layers would be the following (Figure 37):

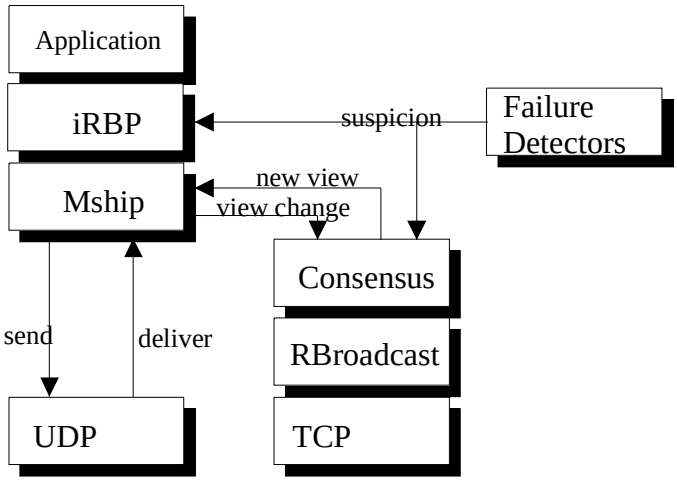

Figure 37 − Interconnections between iRBP Layers

5.2 Problems Found While Implementing the iRBP Protocol

5.2.1 Total Order Layer − iRBP Class

The implementation of the total order algorithm is very similar to the "normal phase" presented in Figure 25. In fact, it was quite easy to implement it. The main problem we had found was to implement the "send until" instructions. This problem comes from the fact that the active layer from Neko has a thread (and a message queue) only for incoming messages, not for outgoing ones (Figure 38).

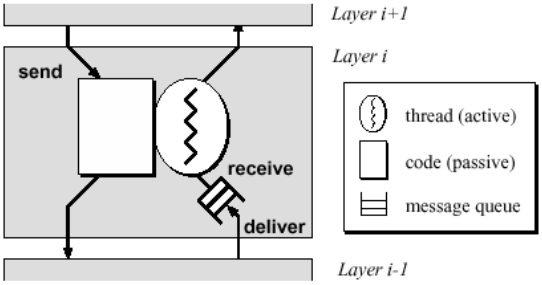

Figure 38 − Active Layer [URB 2001]

Unless we have a thread also for to send messages, we are forced to execute a loop until the reception of an *ack*. As this is not an elegant solution, we decided to add a thread also for the send primitive: if an acknowledgement is received, we remove the send message from the outgoing queue, similarly to what happens in Figure 1.

We also had to restructure the queues (recvQ, seqQ, stableQ), in order to optimize them. Actually, the instructions from the algorithm of Figure 25 hide much of the complexity of manipulating messages in a queue. If we can speed−up the access to the data on the queues, we can reduce the cost of the protocol. Hence, these queues were constructed as classes of objects, each one with different methods for storage and search adapted to their specific needs.

Figure 39shows the basic organization of internal components from the iRBP layer.

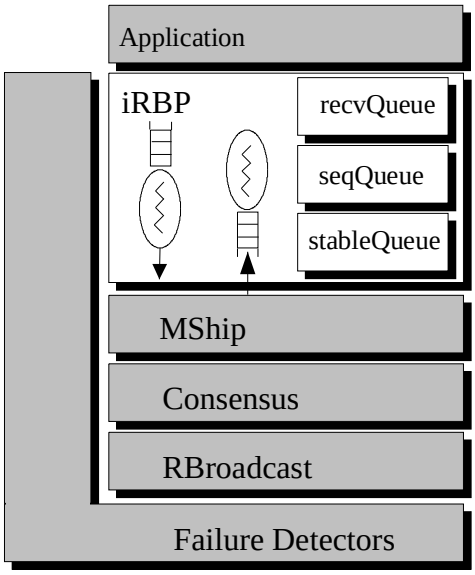

Figure 39 − iRBP internal structure

Nevertheless, the main modification from the original definition of RBP is that membership control is no more executed by the total order broadcast protocol, but for a specific layer (called Mship) that implements group communication primitives, membership control and view changes.

5.2.2 Membership − MShip Class

Compared with the total order protocol, the membership control was harder to implement. Actually, this happens because the membership control must be present in every communication among processes. It is responsibility from the membership layer to control the group membership, to provide primitives for group communication as well as to support membership changes. In addition, the membership layer needs to provide a notion of "virtual ring" for the token−based total order layer (iRBP): we tried to hide the complexity on manipulating this ring by providing methods like *nextProcess()* and *previousProcess()*.

Unfortunately, we were obligate to violate the encapsulation of the membership layer: as token passing is a critical operation in iRBP, the total order layer receives the failure suspicions, and thus, it is responsible for starting view changes. In the same way, the sequenced queue and the stable queue from the iRBP layer had to be shared with the MShip layer, because during view changes the content of the sequenced queue should be exchanged with other processes, and both stable queue and sequenced queue should be manipulated when installing a new view.

At least, implementing the view change protocol was easy. Thanks to the well−defined algorithm for VSC view changes and Neko's building blocks, this operation was fast implemented. In our tests, we used the Chandra and Toueg consensus, together with the Reliable Broadcast layer.

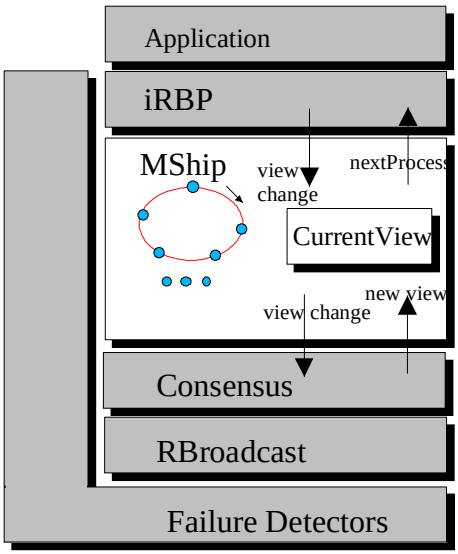

Figure 40 − MShip structure

Figure 40 shows the basic structure from MShip internal components.

5.2.3 Testing

As one of our main objectives was to verify if views and i−views could coexist on the group membership control, we executed various tests in order to simulate failures and to verify the behavior of the "two views" model. A specific layer, MuteLayer, was developed to help simulating failures. Its main function is to prevent a process to communicate with other processes. This is done by dropping messages, instead of sending them as expected. As the MuteLayer is located below the Failure Detector layer, "correct" processes cannot receive any message from the "failed" one, and

thus, it is suspected to have failed. As this kind of control could be programmed, it was easy to simulate crashes, temporary partitions, etc.

We executed many runs with different failure behaviors, in order to test the following scenarios:

- a simple "crash": i−view and regular view changes;
- multiple crashes: i−view and regular view changes;
- temporary network partition: i−view changes, and retransmission requests from the external process;
- reintegration of external processes in a new i−view;

All these runs were successfully executed if concerning the view changes and acquisition of missing messages. However, as the Neko framework does not allow us to test a real suicide, we had to simulate a suicide by muting a process indefinitely. This restriction from Neko comes from the fact that if connections are broken when a process really crashes, the network layer raises some exceptions (like "connection refused") that Neko does not handle yet.

While this is a critical concern if we think on real applications, it does not reduces the merit from Neko. In fact, the ability to deal with broken connections requires a group membership in the lower levels of the system, and this is not suitable for all applications (for example, simulations do not need it, and other applications can be concerned with the additional cost of this service). Anyway, we think that in future versions of Neko we can try to include this network membership control as an optional layer for Neko.

6 Conclusions and Future Works

In this work, we studied RBP, a distributed protocol for Total Order Broadcast. One of the main characteristics of the RBP protocol is that it can operate even in environments subjected to message losses, but still providing good performance rates and low network traffic.

If RBP has an interesting mechanism for Total Order Broadcast, it lacks in the management of groups under failure situations: as RBP was published in 1984, many techniques for group membership did not exist yet, and unfortunately, this protocol was not brought up to date.

In this work, we try to compensate all these years. We compared RBP with another protocol for Total Order that was widely studied in the later years. By identifying common problems to both protocols, we could select some techniques, which were employed to update the RBP group membership control.

By replacing the original membership with a more recent protocol, we could provide all properties that characterize today a group membership service. In addition, we experimented a new technique used to minimize the problems generated by wrong failure suspicions. This technique is is subject of recent publications, and to the best of our knowledge, it has not yet been used in any implementation.

Summing up, the main result from this work was that a good but old protocol can be upgraded using top−of−art techniques without losing its characteristics. More, we shown that these techniques, can be implemented in a real application and even optimized.

However, this work is far from the end. The next work that should be finished is to compare the performance from the RBP protocol with other Total Order protocols: due to time restrictions, we could not execute these comparisons for the present work, and the performance is the "flagship" from RBP−based protocols. Later, one can apply the knowledge from this work to update the RMP protocol, which provides many extra features to the application, but suffers from similar problems in the membership layer.

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

7 Bibliographical References

[AGU 97] Aguilera, M., Chen, W., Toueg, S.: "On the Weakest Failure Detector for Quiescent Reliable Communication". Ithaca:Cornell University, Jul. 1997.

[BIR 96] Birman, K.: "Building Secure and Reliable Network Applications". Manning Publications, Greenwich, 1996.

[CHA 84] Chang, J., Maxemchuck, N.: "Reliable Broadcast Protocols", ACM Trans. on Computer Systems, vol. 2, no. 3, pp. 251−273, Aug. 1984.

[CHB ??] Charron−Bost, B., Défago, X., Schiper, A.: "A New Look at Broadcasting Messages in Fault−Tolerant Distributed Systems", ??

[CRI 95] Cristian, F., Mishra, S.: "The Pinwheel asynchronous atomic broadcast protocols". In 2nd Intl. Symposium on Autonomous Descentralized Systems, pp. 215−221, April 1995.

[DEF 2000] Défago, X.: "Agreement−related Problems: from semi−passive replication to tottally ordered broadcasts", Thesis, EPFL, 2000.

[MAX 2001] Maxemchuck, N., Shur, D.: "An Internet Multicast System for the Stock Market", ACM Trans. on Computer Systems, vol. 19, no. 3, pp 384−412, Aug. 2001.

[MON 94] Montgomery, T.: "Design, Implementation and Verification of the Reliable Multicast Protocol", Thesis, West Virginia University, 1994.

[MON 95] Montgomery, T., Callahan, J., Whetten, B.: "Fault Recovery in the Reliable Multicast Protocol", draft, Nov. 1995.

[SCH 2002] Schiper, A.: "Distributed Algorithms: lecture notes for the Graduate School in Computer Science", EPFL, 2002.

[URB 2001] Urbán, P., Défago, X., Schiper, A.: "Neko: A single environment to simulate and prototype distributed algorithms.", In Proc. of the 15th Int'l Conf. on Information Networking (ICOIN−15), Beppu City, Japan, Feb. 2001.

[VIT 99] Vitenberg, R., Keidar, I., Chockler, G., Dolev, D.: "Group Communication Specifications: A Comprehensive Study", 1999.

[WHE 95] Whetten, B., Montgomery, T., Kaplan, S.: "A High Performance Totally Ordered Multicast Protocol". In K. P. Birman, F. Mattern and A. Schiper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33−57, Springer Verlag, 1995.