The BCG Membership Service Performance Analysis

Fabíola Gonçalves Pereira Greve[+] (fabiola@ufba.br)
Raimundo José de Araújo Macêdo[*] (macedo@ufba.br)

Universidade Federal da Bahia
Laboratório de Sistemas Distribuídos - LaSiD
Prédio do CPD, Av. Adhemar de Barros, S/N, 40170-110. Salvador/BA.
http://www.lasid.ufba.br.

Resumo

BCG (Base Confiável de Comunicação em Grupo) é uma plataforma de comunicação em grupo confiável que difunde as mensagens de forma segura para os processos em grupos simples ou sobrepostos e garante a entrega ordenada das mesmas. Um serviço de *Group Membership* controla a visibilidade dos processos no grupo. Ele assegura que mudanças na configuração, devido à falhas, entradas ou saídas espontâneas, sejam entregues para todos os processes do grupo na mesma ordem lógica. Implementamos e avaliamos o serviço de *membership* para a BCG e neste trabalho apresentamos aspectos importantes dessa implementação, bem como fornecemos alguns resultados de análise de desempenho obtidos a partir de uma série de experimentos realizados com a introdução de falhas de maneira controlada. Além disso, generalizamos analiticamente os resultados obtidos para expressar o comportamento do protocolo diante de cenários de falha mais complexos.

Abstract

BCG (*Base Confiável de Comunicação em Grupo*) is a reliable group communication platform that safely multicasts messages to processes in single or overlapping groups and assures orderly message delivery. A Group Membership service controls the visibility of processes in the group, by ensuring that group configuration changes are delivered to all group members in the same order despite process failures, joining or departures. We have implemented and evaluated BCG´s membership service. In this paper we show important aspects of this implementation and present performance data collected from a series of experiments where faults have been introduced in a controlled manner. Furthermore, we analytically generalize our results to express more complex failure scenarios.

1.0 Introduction

The group communication paradigm has been used successfully for structuring many distributed applications, such as cooperative work applications, distributed data base systems, replication mechanisms, among others. This paradigm can be applied in all phases of distributed system architecture to model systems, resources and users [VRV93].

Basically, in a group communication, a message sent by a process has to be addressed to all members of a group. Message delivery must be atomic: either all processes receive the message or no one receives it. The group is an abstraction to which an application process refers without knowing the number and location of the members which form part of it.

[+] Profa. Assistente do DCC - Departamento de Ciência de Computação da UFBA.

[*] Pesquisador do CNPq, grant number 300013/87-6 (BCG Project).

Applications modeled as a group of processes frequently require an ordering mechanism on message delivery. Total ordering means that messages must arrive in the same global order to all processes in the group. In a replicating system, for example, messages for manipulating data must arrive in the same order to all replicas so as to avoid inconsistency.

Other applications require not only that processes belong to many groups but also that groups are overlapped, which means that two or more groups could have several processes in common. This is the case of newsgroups and teleconferencing systems.

In case of failures, application processes must agree on which ones have failed and in which order these events have happened. The part of a fault-tolerant communication protocol that controls the visibility of processes in the group is the *Group Membership Service*. Group view changes because members may want to join or leave the group, or processes crash. The group membership service must observe the order in which such events occur in a consistent way, so that distributed processes reach some kind of agreement on their local views of all reachable operational machines.

Several membership protocols have been proposed in the literature for asynchronous systems, i.e., systems where no assumption is made about message transmission and processing times. [ACMT95] distinguishes two types of protocols: primary-partition membership protocols [RB91] [KT91] [MPS91] [MSMA94] [HS95] and partitionable ones [ADKM92a] [ADKM92B] [JFR93] [RBC+93] [BDGB94] [EMS95].

Primary-partition membership services are designed for systems with no network partitions, or for systems that support only one partition in the group, the primary partition. In this case, processes at primary partition can continue, while processes in the other components of the network are blocked. The primary partition approach is not appropriate for modeling large and critical systems, where processes must continue in operation, despite partitions. For example, an airline reservation ticket system must continue to sell tickets despite remote failures.

Partitionable membership services allow multiple network partitions. Consequently, the group is split into multiple disjointed concurrent subgroups. Processes in one subgroup cannot communicate with processes in other subgroups, and they proceed as if they were the only ones in the group. Subsequently, when communication is restored, the service must provide some mechanism to reemerge subgroups.

Newtop is a partitionable fault-tolerant group communication protocol for asynchronous systems proposed in [Mac94] [EMS95]. It provides total ordering of messages for overlapping groups. We have implemented Newtop as part of the group communication platform called BCG (Reliable Base for Group Communication) under development at LaSiD/UFBA. In this paper we show the main aspects of BCG's membership service implementation and present performance analysis results collected from a series of experiments where faults have been introduced in controlled ways. Moreover, the results to express more complex failure scenarios have been analytically generalized.

Past published works have reported on evaluations in which message delivery times have been measured in the presence of failures [CBM94] [FR95] [MADK94]. As these measurements reflected somehow the efficiency of the associated ordering protocol (because time was taken only after message delivery), we defined new performance indicators in order to evaluate our membership protocol during its execution (and before message delivery) and run experiments to take these measures under different operational circumstances.

The rest of this paper is organized as follows. In section 2 the system model and fault-tolerant properties of Newtop are described. In section 3 generic aspects of BCG's architecture are presented. In section 4 the membership protocol developed for single groups is explained. In section 5 some performance measurement results are presented. Finally, conclusions are drawn in section 6.

2.0 The System Model and Main Properties of Newtop (an Overview)

We consider a set of processes P1, P2, ..., Pn, distributed in possibly distinct machines, communicating with each other only by exchanging messages through a network, which guarantees that messages are not corrupted nor lost and delivered in the sequential order they were sent (FIFO order). Processes form groups and groups can overlap.

We assume an asynchronous system, where message transmission and processing times cannot be accurately estimated. Processes fail only by crashing, i.e., stop functioning. The network can be partitioned in many non-intersecting concurrent subgroups. Processes in one subgroup multicast messages only to those processes in the same subgroup.

Each functioning process Pi has a *Group Membership Service* (GMi). When group G is initially formed, GMi installs an initial view V_i^1 = {P1, P2, ..., Pn}. As failures occur, GMi will install subsequent views V_i^1, V_i^2,..., V_i^r, until Pi crashes or leaves the group. Process Pi multicasts messages only to those processes in its current view Vi.

When failures occurs, GMi must promptly observe such an event and initiate an agreement protocol with functioning members to remove crashed processes from its view Vi. It is well known, however, that it is impossible to achieve consensus in finite time in an asynchronous system even if processes fail only by crashing [FLP85]. This is because (due to the uncertainties on message transmission time) a functioning process cannot distinguish between a faulty process or a slow one. To circumvent this impossibility result, asynchronous protocols can make use of a non-reliable *Failure Suspector* service to suspect processes crashes [CT91]. The failure suspector makes use of an inaccurate mechanism of suspicion based on arbitrary time-outs. In case of faults, a process Pi must reach agreement between those processes in its view Vi which it does not suspect have crashed and update its membership view, removing members which were confirmed to be faulty.

Sometimes, the *Failure Suspector* (FSi) of Pi can make a mistake and suspect erroneously that a process has crashed. If the suspicion is confirmed by all functioning members in Vi, a virtual partitioning will occur, where an initial group is split into functioning sub-groups. Processes in each sub-group think that they are the only members in the group. In the case of virtual or real partitioning, our membership protocol (which is based on Newtop) leaves the decision upon to continue or not of several functioning sub-groups to the application [Mac94] [EMS95].

Newtop supports symmetric and asymmetric protocols. The asymmetric version uses one of the members in the group (the sequencer) to control ordering of messages. A symmetric protocol does not make use of a sequencer and all members in the group are responsible for ordering. In this paper we describe the symmetric version. For details on the full version of the protocol see [Mac94] [EMS95].

A fault-tolerant protocol must monitor the liveness of each member in the group to promptly observe failure events. Thus, if an application process does not produce regular messages during a period of time, the underlying communication protocol must send control messages periodically to notify the group that it is still alive. This procedure is essential for detecting failures in a symmetric algorithm. In Newtop, each Pi has a *Time-Silence* mechanism that periodically sends a null message if, during a fixed interval of time (*t*) process Pi has not sent any message (regular or otherwise).

During membership agreement, an erroneous suspicion or a partial failure in the network can make a member recover missing messages of suspected processes from other functioning members in its view. In this case, the communication protocol must store messages until it is sure that all processes in the group have received them. Newtop has a safely mechanism to discard received messages called *message stability*. In this mechanism, a message *m* will only be discarded in Pi if Pi knows that all processes in its current view Vi have received *m*. In this case, we say that *m* is *stable*.

Newtop does not have an explicit mechanism for a member to *join* a group. If a process wants to join a group it will do so by forming a new group including itself and all the other members of the old set. Processes can maintain the old membership view while taking part in the formation of a new group. An algorithm for group formation is presented in [EMS95]. The implementation described in this paper does not support this facility.

2.1 Fault-Tolerant Properties of Newtop

We consider the events of sending, receiving and delivering a message *m*: *send(m)*, *receive(m)* and *deliver(m)*, respectively. Just before *m* is sent to a group (send event) *m* is timestamped with a logical clock value called the *block number* (*bn*). When *m* is received from the transport layer of BCG (receive event), it is first stored into local buffers and it is only after the delivery conditions for *m* have been satisfied, that *m* is passed on to the application (deliver event).

We shall now describe fault-tolerant properties of Newtop to show how the protocol provides group processes with a mutual consistent view on the order events take place (message delivery and view updates).

In Newtop, view updates which are performed by processes of a group G satisfy the following view consistency properties:

VC1 (VALIDITY): *The sequences of views installed by any two member processes of group G that never crash nor suspect each other are identical.*

In other words, the order of membership changes (between processes that never suspect each other) must be the same.

VC2 (LIVENESS): *If P_k in V^{r}_{i} leaves G or crashes or becomes disconnected from P_i and if P_i does not crash, then P_i will eventually install $V^{r'}_{i}$, such that r' > r and P_k is not in $V^{r'}_{i}$.*

This property assures that if a member leaves the group, the membership will observe that event and remove the absent member in a future view.

VC3 (ATOMICITY): *Any two member processes of G that never crash nor suspect each other deliver the same set of messages between two identical consecutive views.*

This property states that the delivery of a message to the members of a group must be atomic with respect to a view update by the members.

Property VC3 is called *virtual synchrony* in other fault-tolerant protocols. It ensures that processes perceive process failures and other configuration changes occurring in the same order. The notion of virtual synchrony was first defined by Birmann at ISIS system [Bir91] [Bir93]. Later, Transis [ADKM92a] [ADKM92b] and Totem [ADM+93] formulated the concept of *extended virtual synchrony* and most recently, Horus [FR95] has formalized the concept of *strong* and *weak virtual synchrony*.

Newtop has similarities with *weak virtual synchrony* as it does not guarantee in which view the message will be delivered. In contrast, *strong and extended virtual synchrony* assures that the message is always delivered in the same view as that in which it was sent. In order to preserve this requisite these protocols cause the system to stop sending messages during view installations. This degrades system performance. Horus [FR95] has shown that in every implementation of strong virtual synchrony, there will be an elapsed time in which the system cannot send messages before a view installation. Newtop can be modified to provide this *strong* property but at the same expense of blocking messages. For details see [Mac94].

In the presence of failures or crashes, Newtop obeys the following message delivery properties for all messages *m* multicast in group G:

MD1 (VALIDITY): *A process Pi will deliver a message m in view V^r_i only if the sender of m is in V^r_i.*

MD2 (LIVENESS): *If a process Pi sends m in view V^r_i, then provided it continues to function as a member of G_i it will eventually deliver m in some view $V^{r'}_i$, $r' \geq r$.*

Some systems provide *uniformity* (*safe delivery*) in message delivery. This means that if a member sent a message and other member, faulty or non–faulty, received that message, then every non–faulty process must also receive it. Systems that implement such facility show a significant reduction in performance. Newtop does not provide such safe delivery. Examples in section 4.2 illustrate this.

MD3 (ATOMICITY): (same as VC3) *Any two member processes of G that never crash nor suspect each other, deliver the same set of messages between two consecutive views that are identical.*

Below, we will define total ordering delivery, considering dynamic membership changes. The *happened before* relation [Lam78] (denoted as $\rightarrow$) is used on the send and delivery events.

MD4 (TOTAL ORDER DELIVERY): $\forall$ Pi, Pj s.t. $V^r_i \equiv V^r_j \wedge V^{r+1}_i \equiv V^{r+1}_j$:
(i) $\text{delivery}_i(m1, r) \rightarrow \text{delivery}_i(m2, r) \Leftrightarrow \text{delivery}_j(m1, r) \rightarrow \text{delivery}_j(m2, r)$
(ii) if $\text{delivery}_i(m1, r)$ and $\text{delivery}_i(m2, r')$ occur for a given Pi, then:
$m1 \rightarrow m2 \Leftrightarrow \text{delivery}_i(m1, r) \rightarrow \text{delivery}_i(m2, r')$.

Two processes that never suspect each other will deliver messages respecting total ordering, even in the presence of membership changes.

In the following sections we describe the symmetric membership protocol we have implemented for the platform BCG. The current prototype runs only for single groups.

3.0 The BCG Architecture

BCG is a group communication platform that provides programmers of distributed systems with the necessary mechanisms to develop reliable applications, ensuring ordered message delivery and dynamic group reconfiguration in presence of failures.

Each application process in the group has a BCG kernel[1] composed of three layers: *Application*, *Newtop,* and *Transport Multicast* layers (see figure 1). *Newtop* layer offers total ordering message service and membership service, which is responsible for maintaining visibility of processes in the group. All messages sent by the application process are dealt with *Newtop* protocol and then passed to the *Transport* layer that multicasts them reliably to all other members of the group. At the extreme end, *Transport* layer will transfer messages to *Newtop* where they will be stored in a pool waiting for the ordering condition to be satisfied before delivering. In the prototype currently running at LaSiD/UFBA the multicast is done through multiple point-to-point TCP/IP connections.

Newtop Layer is made up of six main processes which run concurrently and communicate to each other through a message-queue mechanism. The processes are: 1. *Membership*; 2. *Failure Suspector*; 3. *Deliver*; 4. *Transmitter*; 5. *Local Time-Silence* and 6. *Clock-Ticks*.

1. *Membership* process receives messages from *Transport Multicast* layer and distributes them to the other processes of Newtop. Membership controls the visibility of the group, based on failure events sent by *Failure Suspector.*

2. *Failure Suspector* controls liveness of remote processes based on time-out duration. Time-out is measured by a clock receiving ticks from the *Clock-Ticks* process.

3. *Deliver* process stores messages in a pool called Block Matrix (BM) and delivers messages to an application appropriately according to total ordering. It also guarantees that messages that are not *stable* will not be discarded from the BM pool.

4. *Transmitter* process is responsible for time-stamping messages with block-number (*bn*). It multicasts all user and control messages from Newtop to all other processes in the group (via *Transport* layer).

5. *Local Time-Silence* periodically requests the multicast of null-messages to ensure liveness of application process. Requests are based on time-outs which *Local Time-Silence* implements with the aid of the *Clock-Ticks* process.

6. *Clock-Ticks* process is responsible for generating clock ticks for the processes that maintain the clock (*Failure Suspector* and *Local Time-Silence*).

In figure 2 we show the interactions between these processes[2]. In the design of BCG system, special attention has been given to eliminate cycles in the graph (to avoid situations of

[1] In this paper we only consider the total order protocol Newtop. However, BCG architecture includes also the causal order protocol BCGcausal [Mac95] [LM97].

[2] In other publications, about BCG, we have denoted *Failure Suspector* process as *Remote-Time Silence* and we have also presented the *Receiver* process, which does not appear here because its functionalities are encompassed by the *Membership* process.

deadlock during execution). In spite of this, two remain. Fortunately, in both cases it was possible to communicate through a non-wait mechanism.

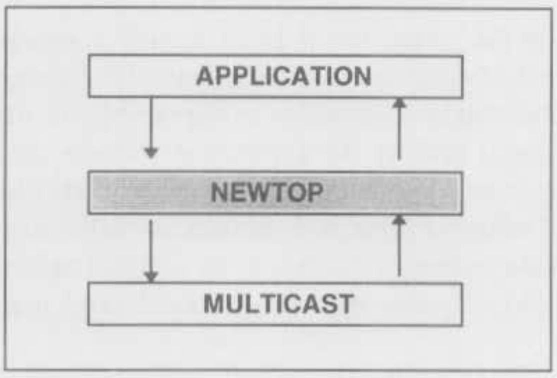


Figure 1: Layers of BCG

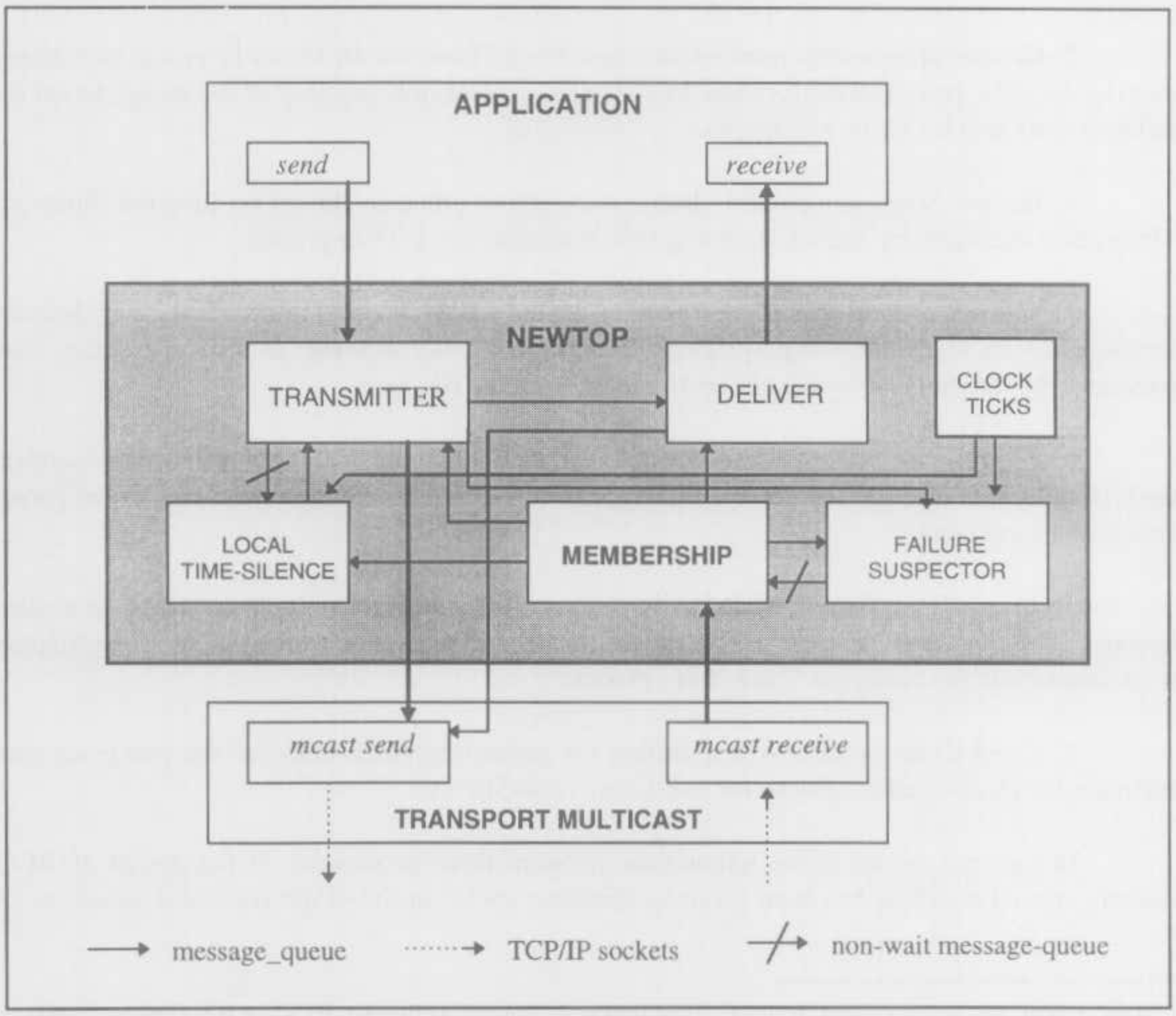


Figure 2: Newtop Processes

In the next section we describe the membership protocol and its relationship to other processes in Newtop.

4.0 Membership Protocol

In this section, we describe the behavior of the Membership protocol. Most of this section (except pseudocode and data structures) has been extracted from [Mac94] [EMS95].

To provide a mutually consistent view of membership changes due to processes crashes or departures, each *Group Membership* service (GMi) of each member in the group must execute the following main functions:

1. The *Failure Suspector* process (FSi) of Pi reports a failure of some remote process Pk, based on a time-out mechanism.

2. In such case, Pi will initiate a membership agreement on Pk's failure.

   2.1. If every functioning process at view Vi agrees to eliminate Pk then:
   - Pi will start a view installation protocol to exclude Pk of its view.

   2.2. Otherwise
   - Pi will start a recovery messages protocol for retrieving missing messages from Pk.

FSi sets a time-out for each new message block number *bn* it receives from a process Pj. During an estimated period of time, say, Φ, it expects the receipt of messages from all remote processes Pk, $k \neq j$, with block number $bn' \geq bn$. If some remote process Pk does not send any message during Φ, FSi will notify its suspicion of Pk failure to the group membership process (GMi) with message (suspect, Pk, *last-bn*), informing the last message block number (*last-bn*) sent by Pk. In current implementation, Φ was set to: $\Phi = t + 2\Delta$, where Δ is an estimated channel communication delay for transmitting messages and t is the time-out set by *Local Time-Silence* process [MS95].

The pseudocode of the main procedures of membership is given in figures 3-8. We shall describe these procedures through the trajectory of a message *msg* that arrives at group membership process of Pi (GMi). The *msg* will contain application information and will piggyback control information from the protocol, such as its *type*, its *block number*, etc. GMi uses a communication primitive *mcast* (*msg*) to multicast *msg* to all processes of view V^{r}_{i}. The multicast is done by the *Transmitter* process, which will time-stamp every message with a block-number. Each GMi process begins the protocol with a set *functioning* containing all processes in its current view V_i. GMi also maintains a set, called LRV (*Last Received Vector*), with the last message block-number received by every process in its current view Vi.

When GMi receives a suspect *msg* from its own FSi, it will store (Pk, *last-bn*) in set *my-suspect* and it will multicast its suspicion with message *mcast*(Pi, suspect, Pk, *last-bn*), which means that Pi suspects Pk with last message block-number *last-bn*. This message is also going to be received by GMi itself.

If GMi receives a suspect *msg* (Pj, suspect, Pk, *last-bn*) from a remote GMj, It will verify if it has a message block number from Pj greater than *last-bn*. If this is the case, it will refute the suspicion message. Otherwise, it will save suspicion of (Pk, *last-bn*) from Pj in set *other-suspect*. GMi will wait until it receives a suspect message from its own FSi, or until a new message from Pk arrives, with block-number greater than *last-bn*. In the first case, GMi will verify if the group has already reached agreement. In the second case, GMi will refute Pj suspicion of Pk and it will remove member (Pk, *last-bn*) from *other-suspect*. If a member suspects Pi, which means that k = i, GMi will take no action and it will expect another member to refute the suspicion or the partition of the group. This means that there was a network partition between Pi and some subset of group view Vi.

If GMi receives a refute message of a process Pk that it actually suspects in *my-suspect*, it will start the recovery of the missing messages from Pk. Messages that arrive at GMi from suspected faulty processes in *my-suspect* are stored in a pool called *pending-message*. If Pk's failure is later refuted, GMi catches all messages from Pk in this pool. If Pk's failure is confirmed, all messages are discarded. The *message stability* mechanism guarantees that if Pi does not have the latest messages from Pk, it can retrieve such messages from any functioning member in the group. In this case, Pi will *mcast* (Pi, recovery, Pk, *upbn*), where *upbn* is the last message block-number received by Pk. Upon the receipt of a recovery message, all active remote processes in group view Vi will multicast all messages from Pk with *msg.bn* > *upbn*.

Process Pi reaches agreement if it *confirms* every set of Pk processes under its suspicion in *my-suspect* set. To *confirm* a suspicion from some Pk, Pi must have suspect messages saved in *other-suspect*, from every set of active (*functioning* - *my-suspect*) members in its current view Vi. If consensus is achieved, GMi will multicast a confirmation message, informing the set of confirmed faulty processes (*detection*). After that, it will start the view installation protocol.

Functioning members that hold identical views and do not suspect each other will confirm identical *detection* sets in an identical order. However, it may happen that some remote GMj achieves consensus before GMi. In such case, GMi will receive a *msg* from GMj confirming a *detection* set of faulty processes. If GMi has not achieved consensus yet for the same set of faulty processes, the *detection* set from GMj will be a proper subset of *my-suspect* set of GMi. In this case, GMi is also able to achieve consensus. As a result, GMi will update *my-suspect* set for (*my-suspect* - *detection*); it will *mcast*(Pi, confirmed, *detection*) with the same *detection* set, and it will start a view installation protocol.

During a membership view change, the *Deliver* process blocks the delivery of messages until GMi notifies which processes are really faulty. This is because the *Deliver* process waits for the arrival of messages from suspect processes to preserve total order delivery. After achieving consensus, Newtop will treat messages from faulty processes in the following way. Among all messages from faulty processes, it will find those with minimum and maximum block numbers (*min-bn, max-bn*, respectively). Messages with *min-bn* are unblocked and delivered. Messages from faulty processes with block number between *min-bn* + 1 and *min-bn* are discarded by the protocol. This is a safety measure to preserve causality. Illustrating examples in section 4.2 clarify this.

If a member Pi wants to leave the group, GMi will instruct the *Local Time Silence* process to stop sending null messages. After a period of time, remote GMj services will reach agreement on the removal of Pi between processes in their current view.

Data Structures

1. *bn*:	block number of message (time-stamp).
2. *functioning*:	set of all processes in current view V.
3. *my-suspect*:	set of last *bn* of each process under my suspicion.
4. *other-suspect*:	set of last *bn* of processes that were declared faulty by others.
5. *pending-messages*:	last messages received from processes under my suspicion.
6. *detection*:	set of processes confirmed to be faulty.
7. *last-bn*:	last block number of suspected faulty process.
8. LRV:	Last message block-number Received by every process in view V.

Figure 3: Data Structures of the Protocol

```
Membership-Algorithm ()

endless loop {

        receive (msg);

        upon Failure Suspector msg (suspect, Pk, last-bn)
            my-suspect [Pk] = last-bn;
            mcast (Pi, suspect, Pk, last-bn);

        upon Transport Multicast msg (Pj, bn)
            if  (Pj is not in functioning)
                discard msg from Pj
            else if  (Pj is in my-suspect)
                save msg in pending-message;
            else  {
                if  (msg is a membership message)
                    Membership_Agreement (msg);
                LRV [Pj] = msg.bn;
                Distribute msg to internal processes of Newtop.
                if (Pj is in other-suspect  with  last-bn < msg.bn )
                    mcast (Pi, refute, Pj, msg.bn);
            }
}
```

Figure 4: Main loop of Membership Algorithm

```
Membership_Agreement (msg)

case (Pj, suspect, Pk, last-bn) : (note that Pi can receive this msg by its own GMi)
        if (Pk = Pi)
                do nothing, expect somebody refute or your departure
        if (Pk ≠ Pi) {
                bn = LRV [Pk];
                if (bn > last-bn) (Pi has msg.bn from Pk greater than last-bn )
                   mcast (Pi, refute, Pk, bn);
                else
                   other-suspect [Pk] [Pj] = last-bn;
                   Get-Consensus ();
                    if (consensus = TRUE) {
                         detection = my-suspect;
                         mcast (Pi, confirmed, detection);
                         Install-New-View ();
                    }
        }

case (Pj, refute, Pk, susp-bn):
        if (my-suspect [Pk] is set with last-bn < susp-bn)
                Recover_Messages (susp-bn);
                reset my-suspect [Pk];
                reset other-suspect[Pk][Pr], with last-bn < susp-bn,
                                              for all Pr in functioning ;

case (Pj, confirmed, detection) :
        if (Pi is in detection) (force my FSi to expurgate Pj from my view)
                my-suspect [Pj] = msg.bn;
                mcast (Pi, suspect, Pj, msg.bn);
        if (Pi is not in detection)        {
                if (detection is a proper sub-set of my-suspect)
                   (I've not achieved consensus yet)
                    my-suspect = my-suspect - detection;
                    mcast (Pi, confirmed, detection);
                   Install-New-View ( );
                }
        }

case (Pj, recovery, Pk, up-bn) :
        Instruct Deliver process to recover all msg from Pk with msg.bn > up-bn
        from block matrix BM and send them directly to Transport layer.
```

Figure 5: Membership Agreement Protocol

```
Get-Consensus ()
(membership must agree on the failure of all set of processes in my-suspect)

for all (Pk, last-bn) in my-suspect
        see if other_suspect [Pk] [Pr] = last-bn,
                for all Pr in functioning and not in my-suspect
                        consensus = TRUE;
```

Figure 6: Algorithm to Reach a Consensus

```
Recover-Messages (susp-bn)

Recover messages from Pk in pending-messages;
Set up-bn the last message block number recovered from Pk;
if  (up-bn < susp-bn)
    mcast (Pi, recovery, Pk, up-bn);
```

Figure 7: Recovery Messages Protocol

```
Install-New-View (detection)

Reset my-suspect, for all confirmed Pk in detection;
Reset other-suspect, for all confirmed Pk in detection;
Discard msg in pending-messages from all Pk in detection;

Detect min-bn and max-bn from all set of failed processes in detection.
Instruct Deliver process to transform all messages between min-bn + 1 and
        max-bn of failed processes in null messages to the application.
Expect until Deliver process delivers all messages with block number min-bn.
functioning = functioning - detection;   (update view)
LRV [Pk] = ∞,  for all Pk in detection;
Instruct all concurrent processes of Newtop to update their local functioning set.
```

Figure 8: Algorithm for View Installation

5.0 Performance Analysis

Most membership protocols have been evaluated by running them in conjunction with other services of the underlying communication protocol [CBM94] [FR95] [MADK94]. Thus, the measurements have been usually based on the average *message delivery delay*[3] of the communication protocol in absence of failures and sometimes in presence of simple ones, for example, one missing message in each multicast [CBM94]. Therefore, the figures shown also reflect the efficiency of the underlying associated services such as total order and transport protocols (because time was taken only after message delivery). Since our purpose was to give application programmers an idea of the precise cost incurred to recover from faults, we defined new performance indicators in order to evaluate our membership protocol during its execution (and before message delivery) and run experiments under different operational circumstances. We

[3]The *message delivery delay*, also called *delivery latency time*, is the time elapsed between the receipt of a message by communication protocol and the delivery of the same message to the application.

will also explain how such indicators can be related to the *message delivery delay* time of our total order delivery protocol.

In presence of failures, the group communication protocol blocks the delivery of messages to the application until the new view is installed or the missing messages are retrieved. Thus, the time taken to reach *agreement* upon failures will directly affect the average *message delivery delay time*. Therefore, the following indicators have been defined in order to evaluate our membership protocol.

1. *Agreement Time*: the delay between failure notification of a process by *Failure Suspector* and its removal from the group view. This measure represents the delivery delay time of failure events to the application and it will be used later in this work to define the message delivery delay in the presence of failures.

2. *Refute Time*: the delay between failure notification of a process by *Failure Suspector* and the arrival of a refute message.

3. *Recovery Message Time*: latency to receive the first missing message requested by a group member after a failure suspicion has been refuted (i.e. a refute message has been received). Note that the *refute* and *recovery message times* will directly contribute to the delivery delay time of missing messages for the application.

Based on the above indicators, several experiments have been carried out over a set of networked UNIX machines (six IBM RISC 6000 machines) connected by a lightly loaded 10 Mbps Ethernet.

5.1 Agreement Time Evaluation

The group membership protocol assures that processes perceive failure events at the same logical time. However, the uncertainties of asynchronous systems caused by several factors of the environment in which experiments were carried out (OS scheduling policy, communication channel overhead, network flow control, etc.) contribute to variations in the exact time in which processes detect the crash[4]. Frequently, as we have observed from our experiment, the machine which starts the agreement protocol will reach *agreement* slower compared to those machines that suspect later. Thus, we measured the *Agreement Time* as the average of agreement times taken by all processes in the group.

Many failure scenarios may be used to measure *agreement time*. However, we chose a simple one, in which only one process crashes in the group. This is the easiest scenario to reconfigure the group. Nevertheless, based on the values of the simple failure mode, we can generalize the protocol behavior for more complex failure scenarios, where it will certainly take a longer time to install the new view. This is explored below.

The Experiment: In order to measure the scalability of our protocol, we ran experiments for different group sizes, from two to six processes. For each run, we forced the crash of only one process in the group. Figure 9 shows the results with the average *agreement time* of 60 executions for each group size. As we expected, as group size increases, *agreement time* increases. The values vary from 10 to 90 milliseconds.

[4] We should add that the uncertainties of the environment raised difficulties to the carrying out of the experiments. For example, in the experiment active processes were frequently considered crashed by others when they started execution with an already significant delay.

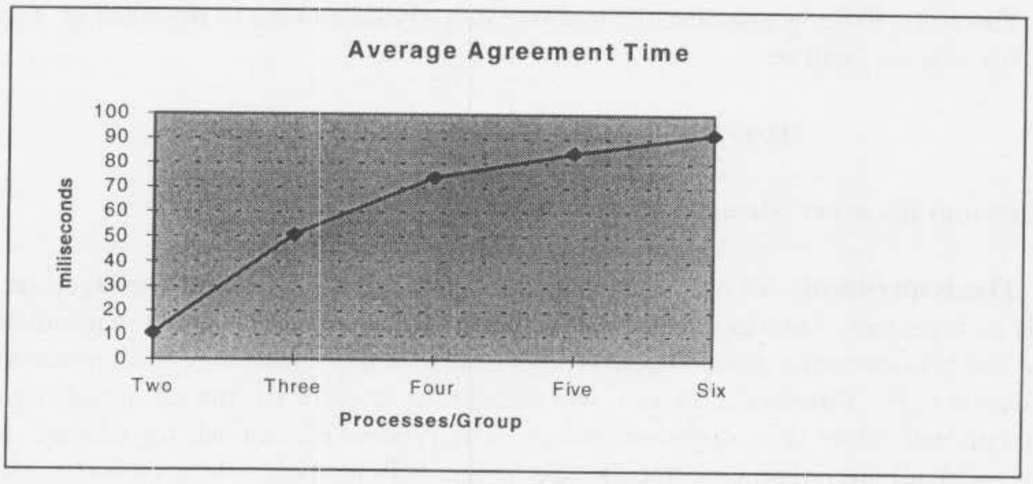


Figure 9: Average Delay for Consensus

We extended the results above to evaluate our membership performance in more complex situations, where more than one process crashes. Let δ_n be the *agreement time*, measured by our experiment, to get *agreement* upon the crash of one process in a group of size n. Let Φ be the remote time-out for *Failure Suspector*. We assume $\Phi > \delta_n$. Let T_f be the *agreement time* to achieve *agreement* upon the failure of f processes in a group of size n. We have $T_1 = \delta_n$.

Consider now that we want to evaluate *agreement time* for a group of n processes to agree on the failure of two processes, P1 and P2. In the worst case scenario, the following will occur: process P1 crashes. As a result, all processes in the group multicast suspect messages of P1´s crash, except P2, which has also crashed *immediately before* it multicasts the suspicion of P1. In this case, *Failure Suspectors* of active processes will only observe P2´s crash after a period of time Φ. After this, an elapsed time δ_{n-1} will be necessary, for $n - 1$ processes in the group (note P1 is crashed) to agree on P2´s crash. Thus, we can deduce that, in the worst-case, *agreement time* T_2 to achieve *agreement* upon the failure of two processes is: $T_2 = \Phi + \delta_{n-1}$.

Applying the above reasoning to obtain the time upon the failure of 3 processes, we have: $T_3 = 2\Phi + \delta_{n-2}$. Thus, we can generalize this result in the following way. For a group of size n to agree on the failure of f processes, in the worst-case, an *agreement time* T_f will be necessary.

$$T_f = (f - 1)\,\Phi + \delta_{n-(f-1)} \qquad \text{for } \Phi > \delta_n \text{ and } n > f$$

The average *agreement time* will increase according to the increase of Φ. On the one hand, Φ must be sufficiently long to avoid unfounded suspicions, on the other hand, Φ must be sufficiently short to allow quick reconfigurations of the group.

We can also extend the above results to evaluate the *message delivery delay in presence of failures* (*for total order*). Let D_f denote the *message delivery delay* in the presence of f process crashes in a group of size n. Consider that we want to evaluate this measure when one process, say P1, crashes ($f = 1$). We know that, in case of failures, the delivery of messages is blocked until the new view is installed. Let m be the first message that is blocked in the communication layer. According to the protocol, the *Failure Suspector* will expect an elapsed time Φ, after the arrival of m, to report P1's crash. After sending the suspicion, the protocol will wait an elapsed time T_1 to reach agreement and install the new view. Note that T_1 is the *agreement time* measured above, $T_1 = \delta_n$. Thus, we can say that in the worst-case in presence of one crash, a message *m* will be delivered to the application in a *delivery delay time* D_1, $D_1 = \Phi + T_1$.

Therefore, in the worst-case, the *message delivery delay time in presence of f crashes*, for a group of size *n*, will be:

$$D_f = \Phi + T_f \quad \underline{\text{or}} \quad D_f = f\Phi + \delta_{n-(f-1)}$$

5.2 Refute and Recovery Message Time Evaluation

The Experiment: All processes multicast messages to the group. Messages are sent at intervals of 5 seconds. Through the *Transport Layer* of BCG, we force a communication failure between two processes of a group. Therefore, a process P1 sends messages to all processes in the group, except to P2. Process P2, in turn, will suspect the crash of P1, but all the other processes in the group will refute such suspicion. In this case, process P2 will ask for missing messages from P1, and all the other processes in the group will multicast such messages. We have evaluated the performance of the protocol for groups from size three to six. We have also varied the local time-silence time-out value (LTS). The Time-Silence mechanism assures that processes send messages (regular or not) every LTS period of time. For all experiments, we measured the *Refute* and *Recovery Message Time*, for LTS equal to 50, 250 and 500 milliseconds. Figure 10 shows the average values of 700 tests carried out for each group size.

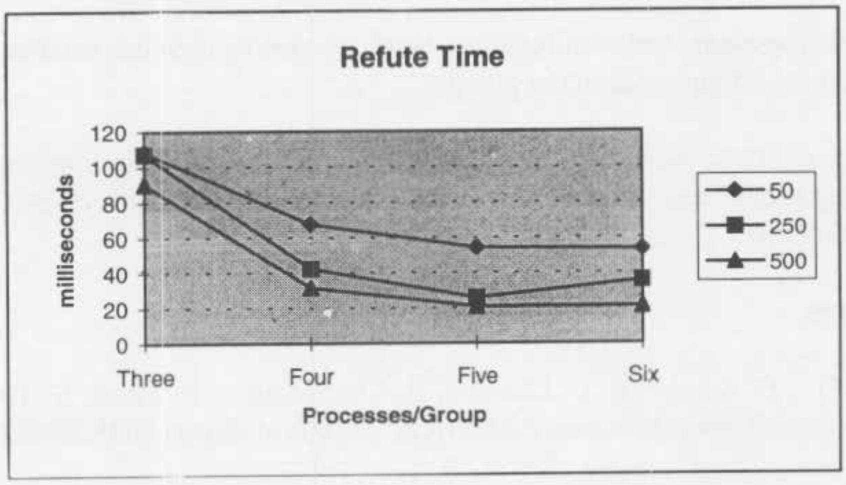
As expected, as group size increases, the number of refute messages increases and the average delay time to obtain the first refute message decreases. The variation of LTS did not cause any great impact on the overhead. The only exception was for LTS = 50. In this case, refute time becomes greater because many more messages are being transmitted in the channel. Surprisingly, however, the average *recovery message time* did not decrease with the increase in group size, it stabilized instead. This is due to the increase in the number of duplicate messages caused by the multiple replies of missing messages. The protocol states that for a group of size *n*, *O(n)* copies[5] of each message recovered will be transmitted. Thus, as group size increases, the communication channel overhead becomes worse. This has lead us to adopt another policy to proceed message recovery (to be implemented in the next version). In this new policy, we will use a combination of (1-to-1) and (1-to-N) communication in order to optimize message recovery time.

We can also make use of the same reasoning developed in the previous section to deduce *missing messages delivery delay* in a group of size *n*. Consider Φ, the time-out period of *Failure Suspector*. Consider also that *Rf* is the time to receive the first refute message, and *Rm*, is the elapsed time to recover the missing message in our experiment. The *missing message delivery delay time Dm* is:

$$Dm = \Phi + Rf + Rm$$

Correctness proofs for all the formulas presented in this section can be found in the complete version of this paper [GM98].

[5] The "Big Oh" notation denotes an upper limit on the number of messages.



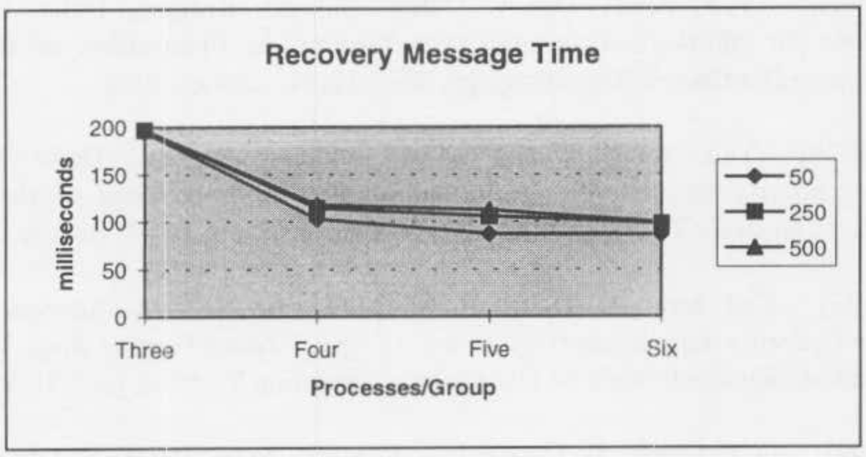


Figure10: Refute and Recovery Messages Measurements

6.0 Conclusions

We have presented the main aspects of the Group Membership Service implementation of BCG, a reliable group communication platform being developed at LaSiD-UFBA on a network of UNIX workstations. We have shown BCG´s architecture and the main interactions among its processes. The Group Membership protocol implemented provides programmers with the necessary mechanisms to develop fault-tolerant applications.

In order to evaluate the BCG's membership protocol, we have introduced some new performance indicators (*agreement*, *refute* and *message recovery* times) to measure the overheads incurred by the membership algorithm during the recovery phase. We have carried out experiments simulating simple failures (introduced in a controlled manner during execution time) and extended our results to express more complex failure scenarios. We have also extended our measurements to express the *message delivery delay in presence of failures*. The experiments were run for different group sizes and local time-silence time-outs. The collected data show that the variation of the local-time silence does not cause a great impact on the overall performance, and, as expected, agreement time increases as the group size increases. However, we have also noticed that *refute time* decreases as group size increases. Surprisingly, the average *recovery message time* did not decrease with the increase in group size, it stabilized instead. This is due to the increase in the number of duplicate messages caused by the multiple replies of missing messages. This has lead us to adopt another policy to proceed message recovery (to be implemented in the next version). In this new policy, we will use a combination of (1-to-1) and (1-to-N) communication in order to optimize message recovery time.

Finally, we have concluded from our experiments that the value chosen for remote time-silence time-out plays an important role in the performance of the delivery protocol as far as failures are considered. On the one ha !, remote time-out must be sufficiently long to avoid unfounded suspicions, while on the other hand, the remote time-out must be sufficiently short to allow rapid reconfigurations of the group.

In a future work we intend to extend our membership implementation to support overlapping groups and integrate it into the other delivery protocols of BCG (the causal and Δ-causal protocols).

References

[ACMT95] E. Anceaume, D. Chandra, B. Charron-Bost, P. Minet, S. Toueg. *On the Formal Specification of Group Membership Services.* Technical Report 2695. INRIA, November 1995.

[ADKM92a] Yair Amir, Danny Dolev, Shlomo Kramer, Dalia Malki. *Membership Algorithms for Multicast Communication Groups.* In Proceedings of the 6th International Workshop on Distributed Algorithms, pp. 292-312, November, 1992.

[ADKM92b] Yair Amir, Danny Dolev, Shlomo Kramer, Dalia Malki. *Transis: A Communication Subsystem for High Availability.* In Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22nd), pp. 76-84, Boston, July, 1992.

[ADM+93] Yair Amir, D. Dolev, P. M. Melliar-Smith, D.A. Agarwal, P. Ciarfella. *Fast Message Ordering and Membership Using a Logical Token Passing Ring.* In Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 551-560, May, 1993.

[BDGB94] Ö. Babaoglu, R. Davoli, L-A. Giachini, M. G. Baker. *RELACS: A communication infrastructure for constructing reliable applications in large-scale distributed systems.* BROADCAST project deliverable report, 1994. University of Newcastle upon Tyne, UK.

[Bir91] K. Birmann. *The Process Group Approach to Reliable Distributed Computing.* Technical Report TR91-1216, Department of Computer Science, Cornell University, July, 1991.

[Bir93] K. Birman. *The Process Group Approach to Reliable Distributed Computing.* Communications of the ACM, Vol. 9, No. 12. pp. 36-53, December 1993.

[CBM94] F. Cristian, R. de Beijer, and S. Mishra. *A Performance Comparison of Asynchronous Atomic Broadcast Protocols.* Distributed Systems Engineering, Vol. 1, No. 4, pp.177-201, June 1994.

[CT91] D. Chandra, S. Toueg. *Unreliable Failure Detectors for Asynchronous Systems.* Proc. 10th ACM Symposium on Principles of Distributed Computing, pp. 325-340, Montreal, August, 1991.

[DMS95] D. Dolev, D. Malki, R. Strong. *A Framework for Partitionable Membership Service.* Technical Report CS95-4, Institute of Computer Science, The Hebrew University of Jerusalem, 1995.

[EMS95] P. Ezhilchelvan, R. Macêdo, S. Shrivastava. *Newtop: A Fault-Tolerant Group Communication Protocol.* In Proceedings of the 15th International Conference on Distributed Computing Systems. Vancouver, Canada, June, 1995.

[FLP85] M. Fischer N. Lynch, M. Peterson. *Impossibility of Distributed Consensus with One Faulty Process*. J.ACM, 32, pp. 374-382, April 1985.

[FR95] R. Friedman and R. Renesse. *Strong and Weak Virtual Synchrony in Horus*. Technical Report 95-1537, Cornell University, Depto. of Computer Science, August, 1995.

[GM98] F. Greve, R. Macêdo. The BCG Membership Service Performance Analysis. Internal Technical Report RTI-001-98 of LaSiD, The Federal University of Bahia, January, 1998.

[HS95] M. Hiltunen, R. Schlichting. *Properties of Membership Services*. In Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems. Phoenix, April, 1995.

[JFR93] F. Jahanian, S. Fakhouri, R. Rajkumar. *Processor Group Membership Protocols: Specification Design and Implementation*. In Proceedings of the 12th IEEE Symposium on Reliable distributed Systems, pp. 2-11, Princeton, October, 1993.

[KT91] M. Kashoek, A. Tanenbaum. *Group Communication in the Amoeba Distributed Operating System*. In Proceedings of the International Workshop on Parallel and Distributed Systems, Vol.5, No.5, pp. 459-473, May, 1994.

[LM97] G. Lima, R. Macêdo. *Avaliação de Desempenho do Protocolo BCGcausal*. Internal Technical Report RTI-003-97 of LaSiD, The Federal University of Bahia, November, 1997.

[Mac94] R. Macêdo, *Fault-Tolerant Group Communication Protocols for Asynchronous Systems*. PhD Thesis, Newcastle upon Tyne University , August, 1994.

[Mac95] R. Macêdo, *Causal Order Protocols for Group Communication*. In Proceedings of XIII Simpósio Brasileiro de Redes de Computadores. pp. 265-283, Belo Horizonte, Brazil, May, 1995.

[MADK94] D. Malki, Y. Amir, D. Dolev, S. Kramer. *The Transis Approach to High Availability Cluster Communication*. Technical Report CS94-14, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[ME95] R. Macêdo, P. Ezhilchelvan. *Reliability Aspects of Multicast Protocols*. In Proceedings of VI Simpósio de Computadores Tolerantes a Falhas. pp. 239-259, Brazil, May, 1995.

[MS95] R. Macêdo, S. Shrivastava. *The Implementation and Performance Analysis of a Total Order Delivery Protocol for Group Communication*. In Proceedings of XV Congresso da Sociedade Brasileira de Computação, Vol I, pp. 287-299, July 1995.

[MPS91] M. Shivakant, L. Peterson, R. Schlichting. *A Membership Protocol based on Partial Order*. In Proceedings of the IEEE Interbational Working Conference on Dependable Computing for Critical Applications, pp 137-145, February, 1991.

[M-SMA94] M. P. Melliar-Smith, L. E. Moser, V. Agarwala. *Processor Membership in Asynchronous Distributed Systems*. IEEE Transactions on Parallel and Distributed Systems, 5(5):459-473, May, 1994.

[RB91] Aleta Ricciardi, K. Birman. *Using Processes Groups to Implement Failure Detection in Asynchronous Environments*. In Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing. pp. 341-352, 1991.

[RBC+93] R. Renesse, K. Birman, R. Cooper, B. Glade, P. Stephenson. The Horus System. In K. Birman e R. Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pp. 133-147. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[VRV93] P. Veríssimo, L. Rodrigues, W. Vogels. *Group Orientation: A Paradigm for Modern Distributed Systems*, BROADCAST Project deliverable report, vol I, October, 1993.