Non-blocking atomic broadcast with omission failures

Emmanuelle Anceaume, Pascale Minet

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1287

Programme 3
Réseaux et Systèmes Répartis

NON-BLOCKING ATOMIC BROADCAST WITH OMISSION FAILURES

Pascale MINET
Emmanuelle ANCEAUME

Octobre 1990

RR_1287

NON-BLOCKING ATOMIC BROADCAST WITH OMISSION FAILURES

DIFFUSION ATOMIQUE NON BLOQUANTE AVEC DEFAILLANCES PAR OMISSION

Pascale Minet, Emmanuelle Anceaume
INRIA, Reflecs Project, BP 105
Rocquencourt, 78153 Le Chesnay Cedex
France
minet@score.inria.fr, anceaume@score.inria.fr

ABSTRACT

A non-blocking atomic broadcast protocol meets the four properties of unanimity, order, uniform agreement and termination. The protocol presented here assumes weak fail-silent Network Attachment Controllers (NAC) subject to general omission failures or crash failures and a broadcast channel subject to omission failures or crash failures. This protocol uses the services provided by a Medium Access Control protocol. Transmission delays are assumed to be bounded. Concurrent broadcasts are not considered. In a failure-free environment it consists of two phases. In case of receivers failures, the number of phases is three. If the broadcaster fails either all the receivers ignore the broadcast or all the correct receivers deliver the broadcast message to their Host. The protocol described hereafter is non-blocking provided that the majority of the current correct NACs is present.

RESUME

Un protocole de diffusion atomique non bloquant satisfait les quatre propriétés d'unanimité, ordre, uniformité et terminaison. Les hypothèses suivantes sont adoptées : les contrôleurs réseau (notés NAC) et le canal de transmission ont deux modes de défaillance possibles : défaillance par omission ou défaillance par arrêt définitif. Pour le contrôleur réseau l'omission peut intervenir en émission ou en réception. Le protocole décrit ci-après utilise les services fournis par un protocole d'accès au médium. Les délais de transmission sont supposés bornés. Les diffusions concurrentes ne sont pas traitées. Ce protocole comprend deux phases en univers non défaillant et trois phases dans le cas d'une défaillance d'un récepteur. La défaillance du diffuseur entraine soit la validation du message par tous les récepteurs corrects, soit l'annulation du message par tous les récepteurs. Ce protocole est non-bloquant sous réserve que la majorité des contrôleurs corrects soient présents.

1. WHY AN ATOMIC BROADCAST ?

As the trend towards distribution continues, the role of atomic broadcast protocols in distributed operating systems of the future has been compared with those of message passing in the operating systems of today [1].
A broadcast is atomic if it meets the four following properties :

- **unanimity** : every message whose broadcast is initiated by a sender is either delivered to all correct receivers or to none of them. The latter case occurs only if the sender fails during its broadcast.

- **order** : all delivered messages from all senders are delivered in the same order at all receivers.

- **termination** : each correct receiver knows the outcome of a broadcast within some known time bound.

- **uniform agreement** : if any receiver (failed or non-failed) has delivered a message, then each non-failed receiver must deliver this message.

2. FAULT ASSUMPTIONS

The communication system consists of a number of Network Attachment Controllers (NACs) connected to a local area network (LAN) modeled as a possibly faulty broadcast channel. A Host is associated with each NAC.

2.1. Weak fail-silent NAC

The NACs are assumed to be **weak fail-silent** [2].

- a **non-faulty weak fail-silent** communication processor always sends correct messages and always delivers correct messages to its Host.

- a **faulty weak fail-silent** communication processor **omits to send or to receive messages** from the LAN. It is considered as permanently failed (crashed) as soon as the omission degree (number of omissions which are either consecutive or related to a given message) is higher than a given threshold.

- a **permanently weak fail-silent** communication processor does not send any messages, and does not deliver any messages to its Host : from the point of view of the Host it remains silent.

This assumption takes into account temporary faults like buffer overflow at the receiving end.

2.2. Channel with omission failure

The channel is subject to crash or omission failure. A faulty channel omits to deliver a frame to some or all NACs. The channel is considered as permanently failed (crashed) as soon as the omission degree is exceeded.

2.3. Communication system

Assuming that the omission degree is n for a NAC and c for the channel, the number of faults to be tolerated for a given message is equal to $2n+c$. Hence a message may be sent $2n+c+1=N$ times to be received by all correct NACs provided that no partition occurs. N is termed the maximum transmission number. A NAC is assumed to be able to receive the messages it sends.
Transmission delays are assumed to be bounded.

3. ATOMIC BROADCAST PROTOCOL

3.1. Main features

This atomic broadcast protocol uses the services provided by a Medium Access Control protocol, and does not consider concurrent broadcasts.

The non-blocking atomic broadcast protocol completes in two phases in a failure-free environment (see fig. 1). **The first phase** starts with the first transmission of the message M to be broadcast. Each correct NAC acknowledges the message receipt. A NAC is said to be correct as long as it belongs to the correct list maintained by each NAC. As soon as the broadcaster has collected at least one acknowledgement from each correct NAC, it enters the **second phase or commit phase** by broadcasting the commit message. Upon commit receipt, a correct NAC is allowed to deliver the committed message to its Host.

In a faulty environment, omission receive faults are tolerated by retransmissions. The broadcaster must retransmit its message until :

- either it has collected at least one acknowledgment from each correct NAC,
- or the maximum transmission number N is reached.

The first phase ends with one of these two conditions. If at the end of the first phase, some correct NACs have not acknowledged the message M, the broadcaster initiates a new phase with the broadcast of the new correct list excluding the silent NACs. This exclusion is allowed provided that this broadcaster has received at least a majority of acknowledgements from the correct NACs during the first phase. The new correct list contains all the correct receivers which have acknowledged M. The broadcaster completes its broadcast with the **commit phase**. In this case (receiver failures), the protocol completes in three phases (see fig. 2).

Majority-Rule : if after N transmissions, the broadcaster has not received the acknowledgements from at least a majority of correct NACs, it must halt.

The majority concept is introduced to avoid that :

- either a faulty broadcaster excludes correct NACs. Indeed with omission receive errors the broadcaster does not receive the acknowledgements of the correct NACs after N transmissions. It considers these correct NACs as failed and excludes them from the correct list ;
- or two or more partitions operate concurrently.

The majority required by this protocol is self-adaptive. It evolves dynamically according to the changes occurring in the system (failures, departures, and joins). For a given broadcast, the maximum number of receiver failures tolerated by the protocol is equal to the majority of the current correct list minus one.

3.2. Failure-free environment

Each NAC *n* maintains locally :

- the current correct list L,

- a variable *llast* = <*l*, *s'*> which is the timestamp of L, with *l* the sequence number of the correct list and *s'* the identifier of the list source,

- a variable *mlast* = <*m-1*, *s*> which is the most recent message timestamp it has received, with *m-1* the sequence number of the message and *s* the identifier of the message source.

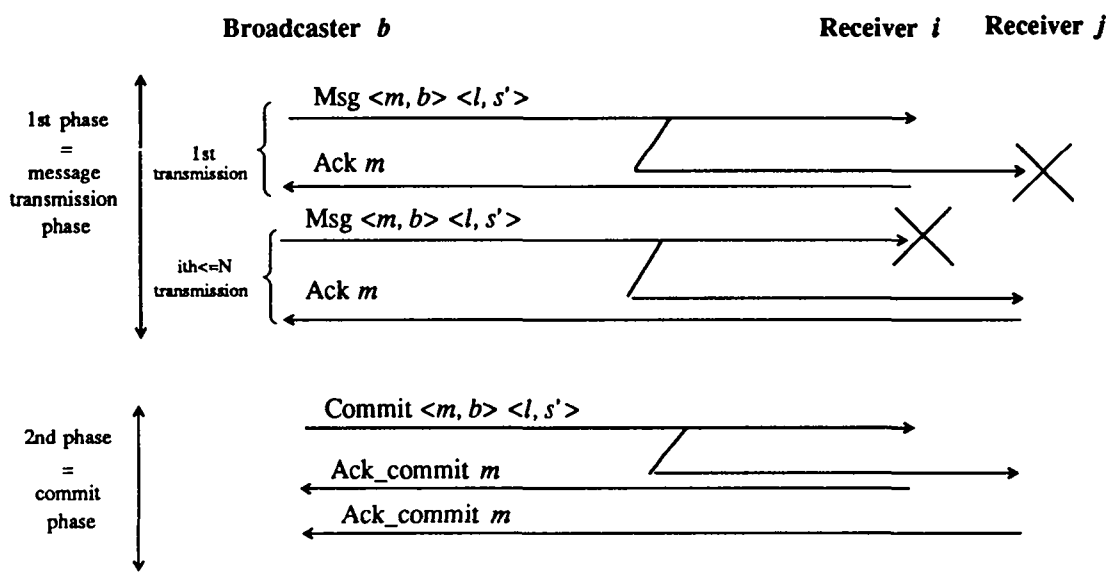

Figure 1 : Failure-free environment : broadcast in two phases

A broadcaster *b* belonging to the correct list L is allowed to broadcast a new message only if it has committed or aborted the previous message. This broadcaster *b* enters the first phase of the non-blocking atomic broadcast protocol. It broadcasts its message M with a message timestamp <*m*, *b*> and a list timestamp <*l*, *s'*>.

To tolerate omission faults, the broadcaster is allowed to transmit up to N times its message. In the following, these transmissions are called retries.

When a correct NAC *R* receives a valid message, it acknowledges it. A message is considered valid by a NAC whose the variables *mlast* and *llast* are equal to <*m-1*, *s*> and <*l*, *s'*> respectively if :

- the broadcaster belongs to the list timestamped by *llast*,

- and the list timestamp received is equal to *llast*,

- and :

  - either the message sequence number is equal to m (normal case),

  - or the message timestamp received is equal to mlast (it is a retry)

  - or the message sequence number is equal to m-1 (crash of the sender of the message timestamped mlast, (see sender failure)).

When a broadcaster has received at least one acknowledgement of all the correct NACs, it initiates the commit phase by broadcasting the commit message. The commit message contains the timestamp of the message to commit with the list timestamp associated. Each list member delivers the committed message to its Host. Like the first phase, all the correct NACs must acknowledge the commit message which can be transmitted up to N times.

3.3. Receiver failure

If a broadcaster has not received at least one acknowledgement of each correct NAC after N transmissions of the message, it must enter a new phase by broadcasting a new correct list excluding the silent receivers. This new list is retransmitted until either the maximum transmission number is reached or this list is acknowledged by all its members. As a consequence of the Majority Rule, a broadcaster is allowed to broadcast a new list if and only if this new list contains a majority of members of the previous list. Otherwise it halts.

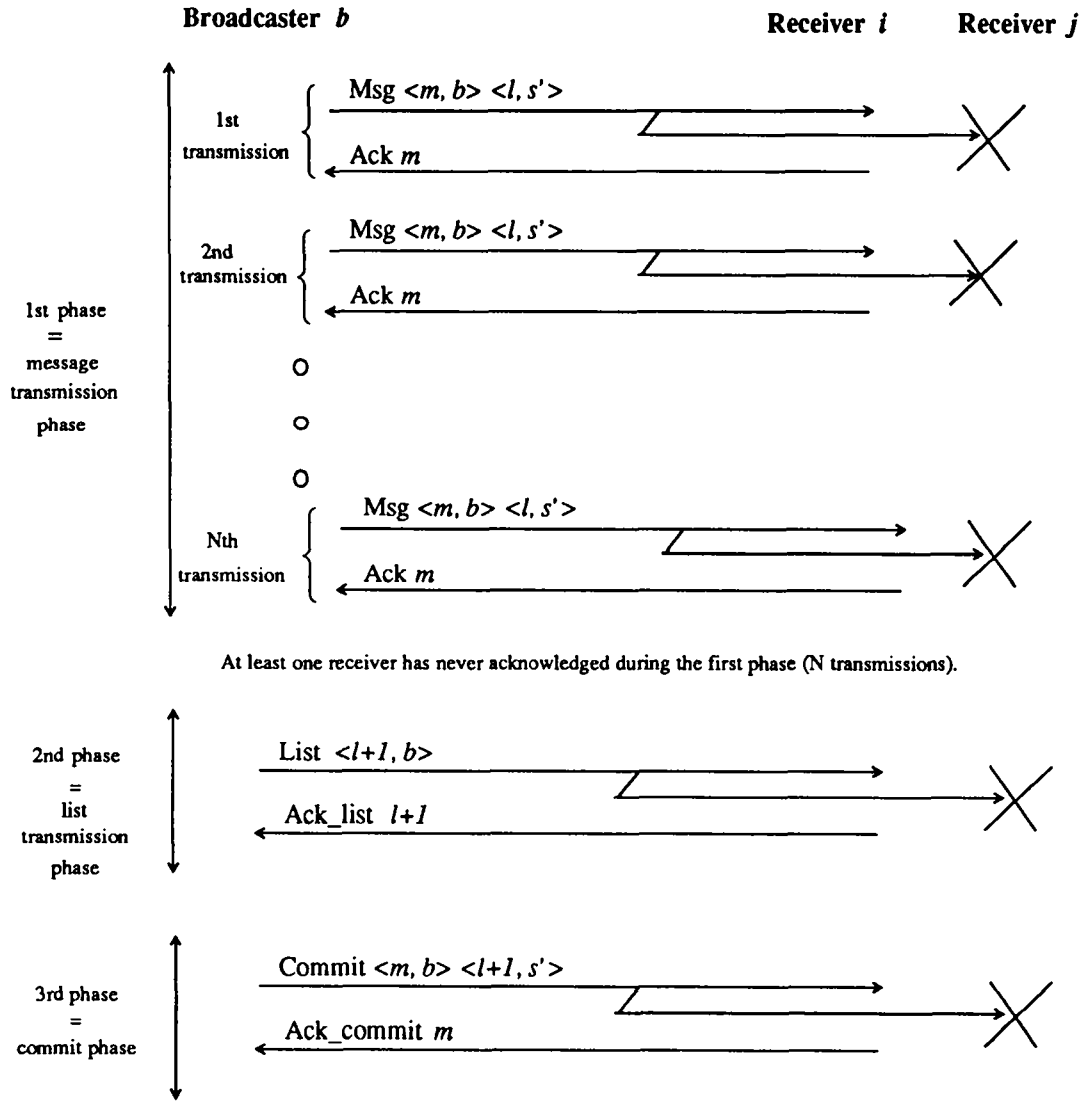

Figure 2 : Receiver failures : broadcast in three phases

A list is considered valid by a NAC with *mlast* = <*m-1*, *s*> and *llast* = <*l*, *s'*> if :

- the broadcaster belongs to the list timestamped by *llast*,

- and :

  - either the list sequence number is equal to *l+1* (normal case)

  - or the list timestamp received is equal to *llast* (it is a retry)

  - or the list sequence number is equal to *l* (crash of the sender of the list timestamped *llast* (see sender failure)).

A NAC that violates the weak fail silence assumption must recognize that it has done so and must halt. A NAC detects such a violation if :

- either it receives an out-of-sequence message with a list timestamp equal to *llast*,

- or it does not belong to the in-sequence received list,

- or it receives a commit for a message it has not acknowledged,

- or it does not meet the Majority Rule.

The correct list broadcast phase is followed by the commit phase, which enables all the NACs belonging to the last correct list to deliver the message to their Host.

3.4. Sender failure

When a broadcaster s fails, the outcome of its broadcast depends on the phase concerned by its failure. Let M be the message broadcast by *s* and timestamped <*m*, *s*> and <*l*, *s'*>, L is the current list timestamped <*l*,*s'*>. Let *n* be the next broadcaster (this one which broadcasts just after *s*) ; *n* takes over the pending broadcast. It is the surrogate of *s*. The broadcaster *s* can fail either during the message transmission, or during the correct list transmission, or during the commit phase. We examine successively these three cases :

- sender failure during the message transmission : some of the correct NACs may have received the message M and the others may not. Two sub-cases are possible :

  - in the first case, if *n* (the surrogate of *s)* has received the message M, then it broadcasts a duplicate of M with the same timestamps <*m*, *s*> and <*l*, *s'*>. It then computes a new list excluding at least *s* with the timestamp <*l+1*, *n*>, and completes its broadcast with the commit phase. M is committed with the list L provided that the Majority Rule is met,

  - in the second case, if *n* (the surrogate of *s)* has not received the message M, then it broadcasts its own message M' with the timestamps <*m*, *n*> and <*l*, *s'*>. When a correct NAC receives two messages with the same sequence number but with different sources, it rejects the first one and takes into account only the last one. Then, *n* computes a new list L+1 excluding at least *s* with the timestamp <*l+1*, *n*>, and completes its broadcast with the commit phase. M is aborted and M' is committed with the list L+1, provided that the Majority Rule is met ;

- sender failure during the correct list transmission : all the correct NACs have received the message broadcast by *s*, but some of them have not received the new list computed by *s* and timestamped <*l+1*, *s*>. Two sub-cases are possible :

- in the first case, if *n* (the surrogate of *s)* has received this new list computed by *s* then it broadcasts a duplicate of M, with the last list L+1 timestamped <*l+1*, *s*> sent by *s*. This enables all the correct NACs to receive this new list. It then computes a new list excluding at least *s* with the timestamp <*l+2*, *n*>, and completes its broadcast with the commit phase. M is committed with the list L+1, provided that the Majority Rule is met,

- in the second case, if *n* (the surrogate of *s)* has not received this new list, it broadcasts a duplicate of M with the list timestamp <*l*, *s'*> (the last list timestamp known by *n*). It then computes a new list L+1 excluding at least *s* with the timestamp <*l+1*, *n*>, and completes its broadcast with the commit phase. When a correct NAC receives two lists with the same sequence number but with different sources, it rejects the first one and takes into account only the last one. M is committed with the list L, provided that the Majority Rule is met ;

- sender failure during the commit phase : some of the correct NACs have seen the commit phase and have delivered the message to their Host ; but the others have not seen it, and according to this protocol, they have not delivered M to their Host. Two sub-cases are possible :

- in the first case, if *n* (the surrogate of *s)* has committed the previous message M with the list L, it can broadcast a new message M+1 timestamped <*m+1*, *n*> and <*l*, *s'*>. During the collect of acknowledgements, it will notice that some of the correct NACs have not seen the commit phase initiated by *s*. Then this broadcaster commits the message M with the list L (the Majority Rule is no more required to repeat the previous commit message) and the message M+1 with the list L+1 (excluding at least *s)* provided that the Majority Rule is met,

- in the second case, if *n* (the surrogate of *s)* has not received the commit message, it sends a duplicate of M timestamped <*m*, *s*> and <*l*, *s'*> (the last list timestamp known by *n*). Two sub-cases are possible :

- in the first case, if one of the NAC has received the commit message of M with the list L, then it signals it in its acknowledgement. In any case the message M is committed with the list L (the Majority Rule is no more required to repeat the previous commit message),

- in the second case, no NAC has received the commit message. The surrogate *n* computes a new list excluding at least *s* with the timestamp <*l+1*, *n*>, and commits M with the list L, provided that the Majority Rule is met.

A broadcaster may commit several messages in the same commit phase in case of previous sender crash.

To tolerate omission faults, a surrogate can broadcast a duplicate up to N times.

A duplicate of M is considered valid by a NAC with *mlast* = <*m-1*, *s*> and *llast* = <*l*, *s'*> if :

- either it is the duplicate of the message timestamped by *mlast* and

- either the list sequence number associated is either *l-1* or *l*,

- or the list sequence number associated is equal to *l+1* and the list L+1 is provided ;

- or it is the duplicate of the message with the sequence number *m* (the sequence number of *mlast*+1) and the list sequence number associated is *l*.

3.5. Maximum drift between correct NACs

Let us consider a correct NAC with a message sequence number equal to *m*. Due to possible sender crashes, three cases are possible :

- either all the correct NACs have the same message sequence number *m* (normal case),

- or some of the correct NACs have the same sequence number *m* and the others have a sequence number equal to *m-1* (the sender fails during the transmission of the message with a sequence number equal to *m)*,

- or some of the correct NACs have the same sequence number *m* and the others have a sequence number equal to *m+1* (the sender fails during the transmission of the message with a sequence number equal to *m+1)*.

Let us consider a correct NAC with a list sequence number equal to *l*. Due to possible sender crashes, three cases are possible :

- either the correct NACs have the same list sequence number *l* (normal case),

- or some of the correct NACs have the same sequence number *l* and the others have a sequence number equal to *l-1* (the sender fails during the transmission of the list with a sequence number equal to *l)*,

- or some of the correct NACs have the same sequence number *l* and the others have a sequence number equal to *l+1* (the sender fails during the transmission of the list with a sequence number equal to *l+1)*.

The number of pending messages (not yet committed) at a correct NAC is at most equal to the number of successive sender crashes. As soon as a sender commits its broadcast, all the correct NACs which have acknowledged the commit deliver the pending messages in the same order to their Host.

3.6. Join handling

Only correct NACs are allowed to handle joins. A join is deferred until the end of the current broadcast.

When a previously failed NAC *j* comes up again, it proceeds as follows :

- *j* transmits a join request to a correct NAC *c* . Several solutions can be considered :

  - either the correct list L is periodically broadcast. The NAC *j* waits for this list to know the correct NACs, and sends its join request to one of them,

  - or the NAC *j* asks each NAC successively until it receives an answer from one of the correct NACs.

- the correct NAC *c* computes a new correct list including *j* and broadcasts it with its timestamp and with the timestamp of the most recently received message,

Only a correct NAC belonging to the current correct list is allowed to update this list. At the end of the procedure this new joined NAC has the correct list and has initialized its variables *llast* and *mlast*.

3.7. Graceful departure

When a correct NAC wants to exclude itself from the correct group, it updates the correct list and broadcasts it. After this broadcast, it halts. This procedure allows a zero-latency detection of a graceful departure.

The procedures for the graceful departure and the join are the only cases where the correct list is updated in the first phase of this protocol.

3.8. Partition handling

If a NAC *f* does not meet the weak fail-silence assumption, it can exceed the omission degree without halting and without knowing about its exclusion. If several NACs behave like *f* during the same broadcast (they have the same timestamps), partitions occur. Physical partitioning may occur too. Partitioning is handled as follows : when a broadcaster *s*, after the broadcast of its message M, detects that the Majority Rule is no more met, it halts. The next broadcaster *n* attempts to recover from *s* failure with the broadcast of a duplicate of M, and like *s*, detects that it belongs to a non-majority partition (the Majority Rule is no met) and so halts.

4. COMPARISON WITH RELATED WORK

4.1. Advantages of this atomic broadcast protocol

In a failure-free environment, this atomic broadcast protocol completes in two phases. In a failure-prone environment, the failures of receivers are detected as soon as a message is broadcast. The number of phases in this atomic broadcast protocol is then three. In case of broadcaster failure, all the correct NACs take the same decision on the outcome of the atomic broadcast. No NAC takes an opposite decision, even the faulty ones.

If *n* processors are correct at the beginning of a broadcast, our protocol tolerates up to *t* failures with *t* the greatest integer smaller than *n*/2. As soon as a processor is unable to communicate with less than the majority of the current correct processors, this processor halts. If more than *t* failures occur, then each correct NAC detects it and halts.

A correct NAC halts with a pending broadcast if and only if the three following conditions are met :

- the broadcaster fails during the commit phase,
- a majority of NACs belonging to the current correct list has failed since the beginning of the uncomplete commit phase,
- all the receivers of the commit message have failed.

A receiver detects that it has exceeded the allowed omission degree upon receipt of either a list excluding it, or an out-of-sequence message with a list timestamp equal to *llast* or a commit for a message it has not acknowledged. It then halts : this enforces the weak fail silence rule.

A broadcaster halts if it does not meet the Majority Rule : after *N* transmissions it has received less acknowledgements than the majority required.
This protocol allows to manage the dynamic changes (departures or joins) in the correct group membership.

4.2. Related work

In this paper we do not consider atomic broadcast dealing with Byzantine failures [3], [4]. Atomic broadcast protocols have deserved considerable attention. Our purpose is not to give an exhaustive list of all these protocols, but rather to compare our protocol with the closest ones.

The AMp (Atomic Multicast protocol) in [5] allows transparent multicasting inside logical groups. Logical groups can be dynamically created and updated. The broadcast is successful only if the last transmission is acknowledged by all group members. The delivery order is then determined by the receive order of the last transmission. This protocol requires that all the non-crashed receivers acknowledge the last transmission of the message, this seems to be a too strong condition.

With the ABCAST protocol described in [1] and [6], messages are delivered according to their timestamp order. The message timestamp is computed by the broadcaster as the maximum timestamp given by the receivers. This protocol allows concurrent broadcasts from different sources. It is very efficient in a failure-free environment, but if the broadcaster and a receiver fail, the order property may not be met, as shown hereafter. If the broadcaster fails during the broadcast of the final timestamp m and if this final timestamp is received only by the receiver which has proposed m, and this receiver fails, then all the other receivers agree on a common timestamp smaller than m. Hence the delivery order of this message differs from the order at the processors failed after having committed.

The Single Value Agreement protocol described in [7] for general omission failures tolerates up to t crash failures with $n>2t$ processors. As soon as a processor receives less than $n-t$ messages, it halts. When the failure number exceeds t, correct processors may take inconsistent decisions. Consider the following scenario. In the first round, the processors do not receive the transmitter message ($n-1$ receive omissions) and during the second round, some processors do not receive the transmitter message (receive omissions) but the others do. Then the first ones abort the broadcast (they have received neither the message nor the echo), although the others deliver the message. The decision of a processor is valid only if the failure number is smaller than t. The question is how does each processor know that the maximum failure number t is not exceeded ?

5. CONCLUSION

The design of an atomic broadcast protocol belongs to a more general task : the design of a dependable distributed computing system [8]. The first step consists in defining the assumptions made concerning the failure of each component. A main problem is to provide a means for a component that violates these assumptions during its operational life to recognize that it did it. That is why the correct list has been introduced in this atomic broadcast protocol : it enforces the weak fail-silent NAC assumption. As soon as a processor is unable to communicate with less than the majority of the current correct processors, it halts. If more than t failures occur, then each correct NAC detects it and halts.
If n processors are correct at the beginning of a broadcast, our protocol tolerates up to t failures with t the greatest integer smaller than $n/2$. Up to t failures, this atomic broadcast protocol is non-blocking.

REFERENCES

[1] Birman K.P., Joseph T.A., "Exploiting virtual synchrony in distributed systems", 11th Symposium on Operating System Principles, p.123-138, November 1987.

[2] The Delta-4 Project Consortium, "DELTA-4 Overall System Specification", edited by Powell, ISBN:2-907801-00-7, printed by LAAS, CNRS,France, December 1988.

[3] Cristian F., Aghili H., Strong R., "Atomic broadcast from simple message diffusion to Byzantine agreement", FTCS 15, Ann Arbor, Michigan,USA, p.200-206, June 1985.

[4] Lamport L., Shostak R., Pease M., "The Byzantine generals problem", ACM Trans. on Programming Languages and Systems, Vol. 4, p.382-401, 1982.

[5] Verissimo P., Rodrigues L., Baptista M., "AMp : a highly parallel atomic multicast protocol", Computer Communication Review, Vol.19, Number 4, p.83-93, September 1989.

[6] Birman K. P., Joseph T. A., "Reliable communication in presence of failures", ACM Trans. on Computer Systems, Vol.5, Number 1, p.47-76, February 1987.

[7] Gopal A., Toueg S., "Reliable broadcast in synchronous and asynchronous environments", 3rd int. Workshop on Distributed Algorithms, Nice, France, September 1989.

[8] Laprie J.C., "Dependable computing and fault-tolerance : concepts and terminology", FTCS15, Ann Arbor, Michigan, p.2-11, June 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399