# ABP : an atomic broadcast protocol

Emmanuelle Anceaume, Pascale Minet

▶ **To cite this version:**

Emmanuelle Anceaume, Pascale Minet. ABP : an atomic broadcast protocol. [Research Report] RR-1473, INRIA. 1991. inria-00075089

**HAL Id: inria-00075089**
**https://inria.hal.science/inria-00075089v1**

Submitted on 24 May 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

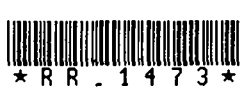
N° 1473

*Programme 1*
*Architectures parallèles, Bases de données,*
*Réseaux et Systèmes distribués*

# ABP : AN ATOMIC BROADCAST PROTOCOL

**Pascale MINET**
**Emmanuelle ANCEAUME**

**Juin 1991**



Rapports de Recherche

Programme 1
Réseaux et Systèmes Répartis

# ABP : AN ATOMIC BROADCAST PROTOCOL

Pascale Minet
Emmanuelle Anceaume

Juin 1991

# ABP : an Atomic Broadcast Protocol

# ABP : un Protocole de Diffusion Atomique

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

**ABSTRACT**

Our atomic broadcast protocol ABP deals with spontaneous concurrent broadcasts in an environment where processors are subject to crash or omission failures and the channel is subject to omission failures. Upper-bounded transmission and processing delays are assumed. ABP guarantees the delivery of the same ordered sequence of messages by all the correct processors provided that the *Progress Condition* is met. A processor is said correct if it is able to communicate with a majority of processors (*Correctness Rule*). Moreover ABP meets the uniform agreement property. Unlike other atomic broadcast protocols, ABP has been designed to avoid the wrong exclusion from the broadcast group : only incorrect processors are excluded. In case of broadcaster's crash, the message abort probability is minimum. If a broadcaster violates the Correctness Rule while running, it detects it and halts without taking wrong decisions. A comparative complexity evaluation is made with closely related protocols.

**RESUME**

Notre protocole de diffusion atomique ABP gère les diffusions concurrentes spontanées dans un environement où d'une part les processeurs sont sujets à des défaillances par arrêt ou par omission et d'autre part le canal de communication est sujet à des défaillances par omission. Les délais de transmission et de traitement sont supposés bornés supérieurement. Sous réserve de la *Condition de Progression*, ABP garantit que tous les processeurs corrects délivrent la même séquence ordonnée de messages. Un processeur est dit correct s'il est capable de communiquer avec une majorité de processeurs (*Règle de comportement Correct*). En outre ABP satisfait la propriété d'accord uniforme. Contrairement à d'autres protocoles de diffusion atomique, ABP, de par sa conception, évite toute exclusion abusive du groupe de diffusion : seuls les processeurs incorrects sont exclus. En cas de défaillance d'un émetteur, la probabilité d'annuler son message est minimum. Si en cours de fonctionnement, un émetteur ne respecte pas la Règle de comportement Correct, il le détecte et s'arrête sans prendre de décision erronée. La complexité de ce protocole est évaluée et comparée avec celle des protocoles les plus proches.

User requirements for dependable computing systems [LAP85] are increasing. For instance the Advanced Automation System [CDD90] in charge of the US air traffic control must guarantee an unavailability of critical services less than three seconds per year. Such dependability requirements can be achieved only by fault-tolerance. In fault-tolerant systems based on active redundancy ([CDD90], [DEL88]) all entities work in parallel : they receive inputs, process them and produce outputs. To ensure that all the entities provide the same outputs, it is necessary to provide them with the same ordered sequence of inputs. Such systems must guarantee that messages broadcast by different sources are received in the same order by all the destination entities. That is why an atomic broadcast protocol ([LG90], [RAY90]...) is required.

Section 1 defines the study framework : what problem is addressed and what assumptions are made. Section 2 points out what distinguishes our atomic broadcast protocol ABP from other similar ones. Section 3 is devoted to ABP description. In section 4, a comparative evaluation of message complexity is done.

# 1. STUDY FRAMEWORK

A broadcast is atomic if it meets the following three properties :

- **Unanimity** : if a correct processor broadcasts a message, then either all the correct processors or none of them deliver the message to their host ; this property is defined in [CAS85] (see section 1.1.2 for a definition of correct processor).

- **Total order** : all delivered messages from all correct senders are delivered in the same order at all correct processors.

- **Termination** : *each correct processor knows the outcome of a broadcast within some known time bound.* This property is defined in [CAS85].

Moreover, some atomic broadcast protocols guarantee uniformity.

- **Uniformity** : if any processor (correct or incorrect) has delivered a message, then each correct processor delivers this message. This property is defined in [GT89]. Notice that uniformity implies unanimity.

## 1.1. Assumptions

### 1.1.1. Model

In our model, processors are assumed to be fully connected by means of a Local Area Network. Each processor is associated with a host supporting the application. The broadcast protocol supported by the processors uses Medium Access Control (MAC) services such as those offered by FDDI (ISO 9314), IEEE 802.4, IEEE 802.5, a deterministic variant of IEEE 802.3 [BFR87] ...

Because of the impossibility result of Fischer, Lynch, Paterson [FLP85], our atomic broadcast protocol ABP assumes **upper-bounded transmission and processing delays**. Each processor accesses a local clock. The drift between any two clocks during a phase of the broadcast protocol is bounded.

### 1.1.2. Failure modes

The processors are subject to crash, or omission failures. However processors are not assumed to be weak fail-silent [DEL88] : there is no omission upper bound beyond which the processors are assumed to be crashed. The channel is subject to omission failures. Physical partition can occur. Let $N$ be the maximum transmission number of a message. $N$ is computed to tolerate up to $c$ channel omissions and up to $p$ send or receive omissions per processor, $N=c+2p+1$. If after $N$ transmissions a broadcaster has received a majority of acknowledgements but not all, the correctness of silent processors must be checked. A processor is considered **correct** as long as it meets the Correctness Rule. If it violates this rule, it is incorrect. It must halt.
**Correctness Rule** : a processor must be able to communicate with a majority of processors.

With the Correctness Rule, only a partition with a majority of correct processors is allowed to work. If the number of processors is $n$, then ABP tolerates up to $q$ (respectively $q-1$) incorrect processsors, with $n=2q+1$ (respectively $n=2q$).

### 1.2. Required properties

The atomic broadcast protocol, we are interested in, must meet the following properties :

- **no synchronized rounds** : ABP does not assume synchronized rounds : any processor is allowed to broadcast a message at any time without waiting for a broadcast initiation time.

- **concurrent broadcasts** : when a processor broadcasts a message with sequence number *m*, it does not know whether another processor has also decided to broadcast a message with sequence number *m*, or whether it has omitted to receive such a message. Concurrent broadcasts are then possible, they must be taken into account by the protocol.

- **no system contamination** : we distinguish two types of system contamination : the first one is caused by an incorrect processor and the second one by a violation of the bounds assumed by the protocol.

The first type can be avoided if :

- a message broadcast by an incorrect processor is rejected by all the correct processors and consequently no processor (correct or incorrect) will commit this message,

- and all messages delivered by an incorrect processor are delivered by all the correct processors (Uniform Agreement property) and in the same order.

The second type of system contamination can be avoided if the system, while running, checks the bound assumptions. If a violation is detected no wrong decision is taken. For instance the system is able to detect that more than the majority of processors have crashed, and halts.

- **no wrong exclusion of the broadcast group** : in a system where omission failures are assumed, an important point to consider is the rule used to exclude a member of the broadcast group. With ABP, a silent broadcaster cannot wrongly exclude correct members.

- **minimum number of aborts** : ABP does its best to complete the broadcast successfully (=delivery of the broadcast message), unlike some atomic broadcast protocols which favour the broadcast termination versus the broadcast success (e.g. upon crash detection of a broadcaster, its message is aborted if it has not been received by all correct processors).

Atomic broadcast protocols closely related to ABP are now presented.

## 2. RELATED WORK

### 2.1. ABCAST

The ABCAST protocol is described in [BJ87a]. Processors are subject to crash failures. Messages can be lost, duplicated or delivered out-of-order. This protocol does not assume synchronized rounds and deals with concurrent broadcasts. It proceeds in two phases : in the first phase, the sender broadcasts its message, each receiver provides a timestamp (the concatenation of a local counter value and the receiver identifier). In the second phase, the sender broadcasts the message commit with the maximum timestamp it has collected for this message. The message is delivered according to this timestamp order. If the broadcaster fails, one member of the broadcast group that has received the message but not the commit takes over and completes the broadcast as the new coordinator. This protocol is simple and attractive but it does not distinguish the following two situations :

- a process delivers a message and then crashes,

- a process crashes before delivering a message.

For instance, if both the broadcaster *b* and the receiver *r* having given the maximum value timestamp fail, the timestamp computed by the new coordinator differs from the one given by the initial broadcaster : the initial broadcaster *b* and the receiver *r* have delivered the message in an order different from the one used by the other processes. This a violation of the total order property.

The ABCAST protocol uses the services provided by a process monitoring mechanism (for process crash detection) and by a site view management. The site view contains all the operational sites. A site *s1* which does not receive in time the periodic message "I am alive" from another site *s2* can exclude *s2* from the site view, *s2* is then forced to commit suicide. This is a wrong exclusion of the broadcast group.

## 2.2. AMp

The AMp protocol is described in [VRB89]. Processors are assumed to be weak fail-silent. The channel is subject to omissions, but the channel replication degree is such that partitioning never occurs. This protocol does not assumed synchronized rounds and deals with concurrent broadcasts. It proceeds in two phases : in the first phase the sender broadcasts its message, which is acknowledged by the receivers. The message is retransmitted until

- either the sender has received the acknowledgements of **the same transmission from all the receivers** (this is a strong condition), the sender enters then the second phase by committing its message. The commit is negatively acknowledged,

- or the maximum transmission number is reached, the sender commits its message in the second phase and triggers the view management protocol to exclude the processors whose acknowledgement has not been received for the last transmission of the message. Wrong exclusion of the broadcast group is possible.

The delivery order is given by the receive order of the last transmission of the message. A sender is not allowed to broadcast a new message while its previous messsage is not committed. If the broadcaster fails during its broadcast, a group manager is elected among the receivers having detected the broadcaster failure : it completes the pending broadcast and excludes the failed broadcaster from the group view. The message is aborted if it has not been received by all the correct processors.

## 2.3. Atomic broadcast for redundant broadcast channels

An atomic broadcast protocol for redundant broadcast channels is described in [CR189]. This protocol tolerates up to $f$ processor crash failures, channel adapter performance failures and channel omission failures. Each processor is connected via $f+1$ independant channel adapters to $f+1$ independant channels. The replication degree is such that a partition never occurs. The protocol assumes synchronized clocks and real-time systems but it does not assume synchronized rounds. It is based on a lazy forwarding technique. A message broadcast at time $T$ on broadcaster's clock is either delivered by all correct processors at time $T+(k+1)(t+d)$ on their clock or by none of them, where $f=2k+1$, $d$ is the maximum drift between two clocks and $t$ is the transmission delay of the message.The sender broadcasts its message on the channels $1, 2, ..., f+1$ in this increasing order. When a processor $p$ (not the sender) receives such a message at time $U$ (on its clock), it accepts this message as timely if $U<T+h(t+d)$ where $h$ is the hop count. Let $c$ be the highest channel on which $p$ accepts this message, if $c<f+1-h$ and $h<=k$, then $p$ must forward this message on channels $c+1, ..., f+1-h$. When no failure occurs the message complexity is equal to $f+1$, it does not depend on the processors number. In presence of failures the message complexity is equal to $n(f-1)+2$ in the worst case. However this protocol seems to be very sensitive to the violation of time bounds.

# 3. OUR ATOMIC BROADCAST PROTOCOL : ABP

## 3.1. Main features

In a failure-free environment, two phases are required to perform an atomic broadcast. In the first phase (message phase) the broadcaster transmits its message and collects the acknowledgements from all the correct processors. In the second phase and provided that the Correctness Rule is met, the broadcaster commits its message. For a broadcaster the Correctness Rule means that after $N$ transmissions it must have received the acknowledgements from a majority of correct processors. Otherwise, it halts.

For performance reasons on the one hand, a broadcaster can combine together the commit of its previous message with the broadcast of its new message. On the other hand, processors are allowed to broadcast concurrently and each broadcaster is allowed to broadcast up to $k$ messages without waiting for their commit (where $k$ denotes the window size). At each correct processor, concurrent messages are received and committed in any order. However, this protocol guarantees that each correct processor will deliver the same messages in the same order. The delivery order is given by the sequence number of the messages. Two messages with the same sequence number are ordered according to their broadcaster's identifier.

In a silent environment send and receive omission faults are tolerated by up to $(N-1)$ retransmissions. If a broadcaster does not receive an expected acknowledgement from another processor, it commits its message and calls the

surrogate.The **surrogate** is the processor with the smallest identifier belonging to the current view. The current view contains all the correct processors belonging to the broadcast group. The surrogate is in charge of checking the correctness of a silent processor and excluding this processor if incorrect, from the group view. If some of the silent processors were broadcasters the surrogate will complete their broadcasts. Even in that case, each correct processor will deliver the same messages in the same order.

The correctness Rule is introduced to avoid the exclusion of a correct processor by silent ones. When a processor *b1* detects that it is unable to communicate with another processor *b2*, a majority of correct processors must agree with *b1* before deciding on *b2*'s exclusion. The majority required by this protocol is self-adaptive. It evolves dynamically according to the changes occurring in the system (failures, departures, and joins).

## 3.2. Concurrent broadcasts

This section describes how ABP deals with concurrent messsages. Each broadcast message is timestamped by a message sequence number and by the source's identifier. Each view is timestamped by a view sequence number and by the source's identifier.

**Broadcaster Rule** : a correct processor $b$ is allowed to broadcast messages with the timestamps $<m+1,b>$ ... $<m+k,b>$ if all the broadcasts with sequence number less or equal to $m$ have already been committed or aborted. Moreover the broadcaster $b$ cannot timestamp its message with a sequence number equal to a message sequence number it has already acknowledged.

**First phase :**

According to the Broadcaster Rule a correct processor $b$ broadcasts its message with the timestamp $<m,b>$, and the current view timestamp. When a correct processor receives such a message, it checks that the view timestamp, provided in the message, is the current one. If yes the processor's response depends on the following conditions :

- either it has no message to broadcast, it sends its acknowledgement with the current view timestamp,
- or it has already broadcast a message with sequence number $m$, it sends its acknowledgement with the current view timestamp and includes the indication that it is a concurrent broadcaster,
- or it broadcasts its message with sequence number $m$ including its acknowledgement for $<m,b>$ and the current view timestamp.

Otherwise, if the received view timestamp is less than the current one, this processor provides its current view in its negative response . The broadcaster must redo the first phase with the current view timestamp only if it belongs to this current view. Otherwise it halts. If the received view timestamp is greater than its timestamp view, it asks for the current view, and acknowledges $<m,b>$.

When the broadcaster $b$, at the end of its first phase, has received a positive response from all the correct processors belonging to the view referred in the message, it knows all the broadcasters which are involved in concurrent broadcasts. It computes then, the delivery order *DOC(m)* of these concurrent messages, which is given by the broadcaster's identifiers. The broadcaster $b$ enters the second phase.

**Second phase :**

The broadcaster $b$ initiates the commit phase by broadcasting the commit of its message. This commit contains the message timestamp and the **commit view** timestamp (view timestamp referred in the message), and *DOC(m)*. The delivery order *DOC(m)* enables all the processors belonging to the commit view to know and to apply the same delivery order for the broadcasts with sequence number $m$. Like the first phase, all the correct processors must acknowledge the commit message which can be transmitted up to $N$ times. When a processor receives the commit of message $<m,b>$, it is allowed to deliver it to its host, if it meets the Delivery Rule.

**Delivery Rule** : a message with sequence number $m$ can be delivered by a processor $r$ only if the following four conditions are met :

- this message is committed,

- and all the previous messages have been delivered by *r*,

- and *r* has not been excluded from the commit view of a previous message,

- and *r* belongs to the commit view of this message or of a previous message.

Consequently, when a processor belongs to the commit view *Vi* of a message, it delivers this message and all the following committed ones until a new commit view *Vj (j>i)* excludes it. When a processor does not belong to the commit view of a message, it halts. It has been excluded.

**Lemma 1** : in a failure-free environment, an atomic broadcast completes in two phases.

## 3.3. Reliable broadcasts

This section describes how an error is detected by the Normal Processing module.

**Lemma 2** : if a processor detects the unexpected silence of a potential broadcaster, it calls the surrogate.

### 3.3.1. Processor silence detected by a broadcaster during its first phase

The silence of a processor *bs* is detected by a broadcaster *bj* which after *N* transmissions of its message <*m*, *bj*> has not received *bs*'s acknowledgement. Provided that the Correctness Rule is met, *bj* is allowed to commit its message with the commit view timestamp and with $QDOC_{bj}(m)$. $QDOC_{bj}(m)$ is the message delivery order computed by *bj* : it takes into account all the broadcasters whose message (with sequence number *m*) has been received by *bj*, and marks with a question mark the silent processors, like *bs* (they could have broadcast a message with sequence number *m* not received by *bj*). After a delay corresponding to *N* transmissions counted from the end of *bj*'s first phase, the following three cases are possible :

- either *bj* has no news about *bs* then it **calls the surrogate with *bs*'s potential missing message indication.**

- or *bj* knows that *bs* is a concurrent broadcaster (receipt of *DOC(m)* including *bs*) then it **calls the surrogate with *bs*'s missing message indication,**

- or *bj* knows that *bs* is not a concurrent broadcaster (receipt of *DOC(m)* not including *bs*) then it does not call the surrogate.

### 3.3.2. Broadcaster silence detected by a receiver

A receiver *r* detects the silence of a broadcaster *b* in the following cases :

- either *r* knows that *b* is a broadcaster, but *r* has not received *b*'s message after a delay corresponding to *N* transmissions, then *r* **calls the surrogate with *b*'s missing message indication,**

- or *r* has received the message, but after a delay corresponding to *2N* transmissions *r* has not received the commit, then *r* **calls the surrogate with *b*'s missing commit indication,**

- or *r* knows that *b* is a potential broadcaster (*b* is marked with a question mark in *QDOC(m)*) but *r* has not received *b*'s message after a delay corresponding to *N* transmissions, then *r* **calls the surrogate with *b*'s potential missing message indication.**

## 3.4. Surrogate algorithm

Upon a processor call, the surrogate runs the surrogate algorithm. The first goal of the surrogate algorithm is to cope with potential missing messages, missing messages or missing commits. The surrogate is the only processor allowed to act on behalf of the silent broadcaster : for instance, it can take the decision to commit a pending message, provided that it meets the Correctness Rule. The second goal of the surrogate algorithm is to decide on silent processor exclusion. The surrogate asks all the processors to determine whether the silent processor violates the Correctness Rule. The surrogate excludes a silent processor *r* only if *r* does not meet the Correctness Rule and the surrogate does.

A broadcaster *r* violates the Correctness Rule with regard to a message <*m*,*r*> (respectively commit <*m*,*r*>) if :

- either $\Sigma$ silent processors + $\Sigma$ processors having received this message (resp. commit) < majority of processors,

- or $\Sigma$ processors having not received this message (resp. commit) >= majority of processors.

A receiver *r* violates the Correctness Rule if :

- either $\Sigma$ silent processors + $\Sigma$ processors having received a response from *r* < majority of processors,

- or $\Sigma$ processors having not received a response from *r* >= majority of processors.

Upon detection of the Correctness Rule violation, the surrogate updates consistently the view of the broadcast group.

For simplicity's sake, in the surrogate algorithm, commit <*m,bj*> stands for the commit of message <*m,bj*>, *DOC(m)* and the commit view timestamp.

### 3.4.1. Surrogate algorithm for a missing commit <*m, bj*>

If the surrogate is called for a missing commit <*m,bj*>, it enters the first phase P1 and proceeds as follows :

**P1:** the surrogate brodcasts the commit of the message if it knows it, otherwise it broadcasts the message itself. A processor responds according to the last information received from *bj* and related to <*m,bj*>, that is nothing received, message received or commit received. Upon receipt of the responses from all the processors seen correct by the surrogate, either the surrogate's algorithm ends or is followed by a second phase P2. The former case hands if both conditions are satisfied : first the surrogate has broadcast the commit message in P1 and second, *bj* has not violated the Correctness Rule with regard to the commit.

**P2** : the surrogate broadcasts the commit of the message and/or a new view excluding *bj* (it has detected that *bj* has violated the Correctness Rule with regard to the commit). A third phase P3 is required only if the surrogate has broadcast a new view in P2.

**P3** : the surrogate broadcasts the commit of this new view which becomes the current one.

### 3.4.2. Surrogate algorithm for a missing message <*m,bj*>

If the surrogate is called for a missing message <*m,bj*> and if it has not received the commit of this message, it enters the first phase P1 :

**P1** : the surrogate broadcasts the message if it knows it, otherwise it broadcasts a request for it. A processor responds according to the last information received from *bj* and related to <*m,bj*>, that is nothing received, message received or whole message in response to a request for message, or commit received. Upon receipt of the responses from all the processors seen correct by the surrogate, the surrogate enters phase P2 :

**P2** : if the surrogate has broadcast the message in P1, and if *bj* violates the Correctness Rule with regard to the message, the surrogate excludes *bj* from the current view. The surrogate broadcasts the commit <*m,bj*> and the new view (if any). Else if the surrogate has broadcast a request for <*m,bj*> in P1 then :

- either at least one processor has received the message <*m,bj*> from *bj* : the surrogate broadcasts the message <*m,bj*> (with a new view excluding *bj* if *bj* violates the Correctness Rule with regard to this message).

- or no processor has received the message <*m,bj*> from *bj* : the surrogate broadcasts "Abort <*m,bj*>" with a new view excluding *bj*.

The surrogate enters phase P3 only if it has broadcast in P2 the message <*m,bj*> or a new view.

**P3** : the surrogate broadcasts the associated commit.

If the surrogate has already received the message <*m,bj*> and its commit, then it just repeats the message <*m,bj*> and its commit.

### 3.4.3. Surrogate algorithm for potential missing message <*m,bj*>

If the surrogate is called for a potential missing message <*m,bj*>, either it has received the message <*m,bj*> then it proceeds as for missing message <*m,bj*>, or it has not received this message and then must determine whether *bj* has really broadcast a message. It runs the algorithm hereafter :

**P1** : the surrogate broadcasts a request for the potential missing message <*m,bj*>. A processor *r* responds as in phase P1 of missing message algorithm if *r* knows that *bj* is a broadcaster. Otherwise, *r* must try to communicate with *bj* and then transmits the result of its try to the surrogate. Upon receipt of the responses from all the processors seen correct by the surrogate, the surrogate enters phase P2.

**P2** : the surrogate broadcasts either the message <*m,bj*> and proceeds as in phase P2 of missing message <*m,bj*>, or aborts <*m,bj*> if no correct processor has received it. It updates the view excluding *bj* if *bj* violates the Correctness Rule. The surrogate enters phase P3 only if it has broadcast in P2 the message <*m,bj*> or a new view.

**P3** : the surrogate commits the message <*m,bj*> or the new view.

### 3.4.4. Surrogate algorithm properties

**Lemma 3** : the surrogate excludes only incorrect processors.
When the surrogate begins its inquiry about a silent processor *bs*, three cases must be considered :

- either *bs* has (normally or abnormally) completed its message broadcast (resp. commit broadcast) with sequence number *m*. That case occurs when the surrogate is called for a missing message (resp. missing commit) after a delay corresponding to *N* transmissions (resp. *2N* transmissions). If *bs* is correct, the surrogate detects at the end of its inquiry that *bs* does not violate CR wrt. message (resp. commit) and then *bs* will not be excluded by the surrogate,

- or *bs* is broadcasting its message while the surrogate is inquirying. The processors having not yet received *bs*'s message will test *bs*. In response to the test request, *bs* broadcasts its message. If *bs* is correct, a majority of correct processors will receive *bs*'s message and then *bs* will not be excluded by the surrogate,

- or *bs* has not yet broadcast a message with sequence number *m*. If *bs* is correct, it indicates in the response of the test request that it has never broadcast a message with sequence number *m* and it will never do so. A majority of correct processors will receive *bs*'s indication and then *bs* will not be excluded by the surrogate.

**Lemma 4** : the surrogate algorithm completes in at most three phases, if no other failure occurs.

**Lemma 5** : the surrogate cannot take an opposite decision to the one taken by the initial broadcaster.
Let us assume that the initial broadcaster has committed its message. This is possible only if a majority of processors has acknowledged that message. Consequently at least one processor belonging to the surrogate's majority has received that message and gives it to the surrogate which will commit it. A broadcaster belonging to the current view is not allowed to abort its message even if it does not meet CR : it just halts and the surrogate will do its best to commit the message. A message sent by a broadcaster excluded from the current view is rejected by the correct processors.

**Lemma 6** : any committed message with sequence number *m* belongs to *DOC(m)* computed by the surrogate.
The demonstration is the same as for lemma 5.

### 3.4.5. Processor silence during surrogate algorithm

The surrogate algorithm given before is slightly modified if the surrogate detects a silent processor at the end of a phase Pi (i=1, 2 or 3). The surrogate must check if the silent processor violates the Correctness Rule (CR). If yes, it establishes a relay mechanism that enables all the messages sent by the surrogate (respectively by this processor) to reach this processor (respectively the surrogate) in two hops (because of CR, any two correct processors communicate with at least one same correct processor). The key idea is to use reachable processors as relays for the unreachable correct ones. This is the purpose of phase P'i+1 and P'i+2.
At the end of phase Pi, the surrogate knows all the reachable processors (it has received their acknowledgement) and

all the unreachable ones (the silent processors). It must determine the minimal weight spanning tree giving the minimal number of relays necessary to cover all the correct processors. The surrogate must run phase P'i+1, P'i+2 and if necessary P'i+3.
In P'i+1, the surrogate asks all the reachable processors to test the unreachable ones. Upon receipt of test responses, the surrogate determines among the unreachable processors the incorrect ones (they do not meet CR) and the correct ones. It excludes the first ones from the current view and establishes the minimal weight spanning tree : among all the reachable processors, the surrogate chooses the processor communicating with the maximum number of correct unreacheable processors as a relay. It repeats this step with the remaining correct unreachable processors until all have been assigned to a relay (boolean function reduction [LAV74]).
In the next phase P'i+2, Pi is redone : the surrogate repeats the information broadcast in Pi and includes the new view (if any), the relays list and for each relay, the relayed processors list. If a new view has been broadcast, a new phase P'i+3 is needed : in P'i+3 the new view is committed as well as the information that would have been broadcast in Pi+1 (if this phase had existed). The relays propagate that information to their relayed processors.Then the surrogate proceeds normally.

**Lemma 7** : a processor's silence detected in a phase Pi of the surrogate's algorithm implies at most three additional phases.
In most cases two additional phases are needed : test of the silent processor and broadcast of the relays list. The only case where a third phase is required is when the silent processor detected by the surrogate at the end of Pi differs from the processor under test in Pi (if any), Pi is a commit phase and the silent processor does not meet CR.

**Progress Condition** : the number of modifications (processor's crash, surrogate crash or communication failure between the surrogate and a correct processor) in the surrogate's communication graph is not higher than $q$ (respectively $q-1$) with $n=2q+1$ (respectively $n=2q$).

If the Progress Condition (PC) is met and the surrogate does not crash then the surrogate's algorithm allows each correct processor to complete successfully the pending broadcast. The maximum phase number of the surrogate algorithm is $2(q+1)$, (respectively $2q$) with $n=2q+1$ (respectively $n=2q$).

### 3.4.6. Surrogate silence

The crash of the surrogate *s1* is detected by a processor which does not get any answer from it. This processor notifies *s1*'s silence to the next processor *s2* in the current view ordered according to processor identifiers. *s2* is not allowed to behave as the surrogate as long as *s1* does not violate the Correctness Rule. If *s1* does not violate it, *s1* is always the surrogate. If *s1* violates CR (result of processors test), *s2* updates the current view in two phases excluding *s1* and broadcasts it : *s2* is then the surrogate. It takes over the surrogate algorithm : inquiry about the pending message, broadcast of the pending message and commit.

**Lemma 8** : the surrogate's crash detected in a phase Pi of its algorithm implies at most five additional phases.
In most cases four additional phases are needed : test of the surrogate, new view broadcast and state of the pending message at the correct processors, pending message broadcast. The only case where a fifth additional phase is required is when the surrogate crashes while broadcasting the pending message commit.

**Lemma 9** : at any time there is at most one active surrogate.
Let us assume that *s1* violates CR. *s2* updates and broadcasts a new view excluding *s1*. If *s1* does not receive this view, *s1* ignores that it is no longer correct and always behaves as the surrogate. When it broadcasts a surrogate's message, two cases are possible :

- either *s1* does not meet CR, it commits suicide,
- or *s1* communicates with a majority of processors. Then at least one processor belonging to both *s1*'s majority and *s2*'s majority has received the view excluding *s1*. This processor rejects *s1*'s message : *s1* commits suicide upon receipt of this reject.

Thanks to PC, the number of changes in the surrogate communication graph is bounded. Consequently the termination property is met. From lemma 1 to lemma 9, ABP meets the atomic broadcast properties provided that PC is met.

## 4. COMPARATIVE EVALUATION

### 4.1. Failure-free environment

Let $n$ be the correct processors number. Let $Cm$ denote **the message complexity** ; $Cm$ is the number of messages needed to perform an atomic broadcast. Let us focus on **response times** : $Thds$ is the delay between the broadcast time and the delivery time at the sender ; $Thdr$ is the delay between the broadcast time and the delivery time at a correct receiver (the fastest one) ; $T2r$ is the maximum drift between two correct receivers for the delivery of the same message. $Tmsg$ denotes the transmission time required to transmit a data/commit message, and $Tmsg+n.ack$ denotes the transmission time required to transmit a data/commit message and to collect $n$ acknowledgements. Let $Cd$ denote the **concurrency degree** : $Cd$ is the maximum number of concurrent broadcasts. $Cd$ is expressed as a function of $k$ the window size if any. The table hereafter gives the message and time complexity for different atomic broadcast protocols in a failure-free environment. The message complexity $Cm$ is evaluated assuming a broadcast channel.

| Atomic Protocols | $Cd$ | $Cm$ for $k$ successive messages | $Thdr = Thds$ |
|---|---|---|---|
| [CRI89] | $k*n$ | $k(f+1)$ with $f+1$ =number of channels | $(f/2+1)(Tmsg+clockdrift)$ |
| [VRB89] | $n$ | $k(2msg+(n-1)ack)$ | $Tmsg+n.ack+Tmsg$ |
| [BJ87a] | $k*n$ | $k(2msg+(n-1)ack)$ | $Tmsg+n.ack+Tmsg$ |
| ABP | $k*n$ | $(k+1)(1msg+(n-1)ack)$ | $Tmsg+n.ack+Tmsg$ |

Table 1 : Concurrency degree, message and time complexity for a failure-free broadcast

### 4.2. In case of $t$ receive omission faults

Let us assume that the $t$ receive omission faults occur in the same phase and one per receiver. In [CRI89], performances of the algorithm are not affected. In [VRB89], the worst case corresponds to $Cm=(t+2)msg+(t+1)(n-2).ack$, $Thdr=Thds=(t+1)(Tmsg+n.ack)+Tmsg$. In ABP, $Cm$ is $3(msg+(n-1)ack$. Notice that $Cm$ does not depend on $t$. The time complexity depends on the fault occurence phase : during the first one $T2r = 0$ and $Thds = Thdr = Tmsg+n.ack+Tmsg+(n-t).ack+Tmsg$ ; during the second one $T2r=Tmsg+n.ack$, $Thds$ and $Thdr$ are the same as in the fault-free environment.

### 4.3. In case of $t$ channel omission faults

Let us assume that the $t$ channel omissions occur while the sender is broadcasting its message. With [CRI89], only $Cm$ is affected, it is equal to $(n(t-1)+1)msg$. With [VRB89] or ABP, the results are identical to the case of $t$ receive omission faults in [VRB89].

### 4.4. In case of receiver crash

Let us assume that the receiver's crash occurs during the first phase. In [VRB89], $Cm$ is equal to $(N+4)msg+(N+2)(n-2)ack$. In ABP, $Cm=(2N+3)msg+3(N+1)(n-2)ack$ : it takes into account two phases for the broadcaster and three phases for the surrogate ; the surrogate decides on the receiver's correctness in P1 (this decision results from the receiver's test by the correct processors), then updates the current view (exclusion of the incorrect receiver) in P2 and commits it in P3. This overhead is due to the fact that only incorrect members are excluded from the broadcast group.

### 4.5. In case of broadcaster crash

In ABP, if the crash occurs during the first phase, $Cm=4(msg+(n-1)ack)$ (one phase for the initial broadcaster and three for the surrogate : it completes the pending broadcast and excludes the failed broadcaster from the broadcast group). If the crash occurs during the second phase, $Cm$ is at worst equal to $5(msg+(n-1)ack)$ (two phases for the

initial broadcaster and three phases for the surrogate). In [VRB89], all the receivers having received the message ask the broadcaster to get the commit. After $N$ unsuccessful attempts, these processors compete to become the Group Manager (GM). The elected GM then updates the view and aborts the message if at least one processor has not received it.

## 5. CONCLUSION

Atomic broadcast protocols are a main component of fault-tolerant distributed systems ([BJ87b], [CDD90], [DEL88], [LEL90]). They guarantee the delivery of the same ordered sequence of messages by all the correct processors. ABP has been designed to deal with spontaneous concurrent broadcasts in an environment where processors are subject to crash or omission failures and the channel is subject to omission failures. It assumes the Progress Condition and upper-bounded transmission and processing delays. Each correct processor is allowed to broadcast spontaneously up to $k$ messages without blocking. Uniform agreement property is met. In case of broadcaster's crash, ABP does its best to complete the broadcast successfully : it minimizes the message abort probability. ABP and other atomic broadcast protocols mainly differ in the handling of processors' exclusion. With ABP, wrong exclusion from the broadcast group is avoided : only incorrect processors can be excluded. With other protocols, a violation of the omission bounds (by the channel or a broadcaster) assumed by the maximum transmission number can lead to a wrong exclusion of a correct processor. While running a broadcaster is able to detect that it violates the Correctness Rule, and if so it halts without taking wrong decisions. In the same way while running the system is able to detect that the Progress Condition is no longer met. An ABP simulation will be done with the event-driven simulator SAMSON.

## REFERENCES


[BFR87] Boudenant J., Feydel B., Rolin P., "LYNX : an IEEE 802.3 compatible deterministic protocol", IEEE INFOCOM 87, San Francisco, California, March 1987.

[BJ87a] Birman K., Joseph T., "Reliable communication in presence of failures", ACM Transaction on Computer Systems, Vol.5, N1, p.47-76, February 1987.

[BJ87b] Birman K., Joseph T., "Exploiting virtual synchrony in distributed systems", 11th Symposium on Operating System Principles, pp.123-138, November 1987.

[CAS85] Cristian F., Aghili H., Strong R., "Atomic broadcast from simple message to Byzantine agreement", FTCS 15, Ann Arbor, Michigan, USA, pp.200-206, June 1985.

[CDD90] Cristian F., Dancey B., Dehn J., "Fault-tolerance in the advanced automation system", FTCS 20, Newcastle, UK, pp6-20, June 1990.

[CRI89] Cristian F., "Synchronous atomic broadcast for redundant channels", The Journal of Real-Time Systems, Vol.2, pp195-212, Kluwer Academic Publishers, The Netherlands 1990.

[DEL88] 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.

[GT89] Gopal A., Toueg S., "Reliable broadcast in synchronous and asynchronous environments", 3rd International Workshop on Distributed Algorithms, Nice, France, September 1989.

[LAP85] Laprie J.C., "Dependable computing and fault-tolerance : concepts and terminology", FTCS15, Ann Arbor, Michigan, USA, pp.2-11, June 1985.

[LAV74] Lavallée I.,"Recouvrement d'un ensemble par une sous-famille de ses parties", Thèse de docteur-ingénieur, Paris VI, France, Juin 1974.

[LEL90] Le Lann G., "Critical issues in distributed system", 2nd IEEE Workshop on future trends of distributed systems, Cairo, Egypt, pp.420-426, September 1990.

[LG90] Luan S., Gligor V., "Fault-tolerant protocol for atomic broadcast", IEEE Transaction on parallel and distributed systems, Vol. 1, N3, July 1990.

[RAY90] Raynal M., "Order notion and atomic multicast in distributed systems : a short survey", 2nd IEEE Workshop on future trends of distributed systems, Cairo, Egypt, pp.420-426, September 1990.

[VRB89] Verissimo P., Rodrigues L., Baptista M., "AMp : a highly parallel atomic multicast protocol", Computer Communication Review, Vol.19, N4, pp.83-93, September 1989.

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

ISSN 0249 - 6399