Chapter 10

The Atomic Multicast protocol (AMp)

The utility of reliable group, or *multicast*, communication protocols was recognised in section §6.9. The rationale behind the desired qualities of service was also discussed, i.e., which properties the service implemented by these protocols should offer, to support distributed fault-tolerance. We will have the opportunity to formalize those properties and to present the relevant protocols, in this chapter.

In building a reliable communication service to support distribution in Delta-4, we were faced with three concerns: (i) network design, in order that it would be as portable as possible; (ii) architecture design, so as to harmonize the use of standard components with the procurement of adequate levels of fault tolerance; (iii) protocol design, in order to take advantage of the network architecture, both in terms of performance and reliability.

These concerns were materialized in *AMp*, the Atomic Multicast protocol. The present chapter concentrates on the design of the overall architecture supporting the AMp service, and the description of the protocols. We begin by presenting a model for the kind of problems Delta-4 communications wish to solve, and we formally describe the required properties of AMp. However, since no service can be more reliable than the system supplying it, the dependability model of the communications architecture deserved particular attention: section §10.3 identifies the building blocks of the architecture; determines their individual behaviour and the way they interact with one another; establishes a system fault model, and finally, discusses the measures to ensure system dependability. Two aspects of relevance are the use of *self-checking* components and of a *non-replicated* LAN, and their implications on the fault model, i.e., the fact that a "nice" behaviour can be expected, and that time domain redundancy must be used, to recover from transmission errors. Sections 10.5 and 10.6 deal with the provision of an *abstract network* service by the channel[1], and of a group communications service, by a protocol built on top of the abstract network.

A note about the framework concerning AMp, in the project: protocol design has been discussed in several papers, where variants of the basic multicast protocol were presented: some, specifically for given local area networks, namely the token-ring and the token-bus networks, require hardware modifications [Guérin et al. 1985, Veríssimo et al. 1987]; the other version is a software implementable protocol, which is LAN independent [Veríssimo et al. 1989], and which is currently ported to the various LANs supported in the project (token-ring, token-bus, FDDI). Detail about these various implementations is given in §10.8.

The implementation specifications of both these versions are fairly complex, recommending the use of semi-automated methods for validation. In consequence, a fault injection campaign is in course, with the aim of forecasting faults and assisting in their removal, with the help of a

1 In the remainder of the chapter, the word *network* will be used not to mean layer 3 of OSI, but rather the *networking infrastructure*, that is, the entities concerned with providing the abstract network service: the LAN Medium Access Control (MAC) sub-layer and Physical layer, and the relevant firmware.

specialized tool [Arlat et al. 1990]. The software variant was formally specified, in Estelle, and is being formally validated [Baptista et al. 1990]. Detail about this work is given in chapter 15.

10.1. Notions about Reliable Group Communication

A reliable broadcast protocol is a protocol that fulfils a set of safety and liveness and/or timeliness properties, in the form of agreement, order and synchronism paradigms. If the addressing modes supported by the protocol concern subsets of participants (multicast), rather than *all* the participants (broadcast), then it is called a *reliable group communication protocol*. Delta-4 is concerned with the latter.

10.1.1. Agreement

Distributed *agreement* in the presence of faults has been the subject of a number of publications. Useful paradigms have been identified, among which the well-known Byzantine Agreement [Lamport et al. 1982]. Informally, it seeks to make a participant disseminate a value to all the other participants, so that those who are correct, accept the same value; if the sender is correct, the value accepted has to be the one it sent, otherwise it is some default value. The problem as originally equated, is concerned with a specialized, phased-execution environment in which faulty participants can behave arbitrarily. To ensure that participants behave consistently, forms of agreement such as those found in *atomic* broadcast protocols specify, alternatively, that a message may or may not be delivered, but in affirmative case, it is delivered to all intended recipients. The word "intended" is used to underline that the specification may not include all participants, but only a *group* of them, or refer to relaxed forms of agreement, such as majority, at-least-N, etc.

In general, the conditions for achieving distributed agreement can be equated in terms of *agreement* and *validity* properties [Perry and Toueg 1986]. The strongest form of agreement is unanimity:

Unanimity: Any message delivered to a recipient, is delivered to all correct recipients.

The ancillary *validity* properties specify the conditions in which agreement is, or is not, performed. A normally necessary validity condition called non-triviality, specifies that any message received is a "useful" message, i.e., not forged or spontaneously generated, not a pre-agreed message, etc.:

Non-triviality: Any message delivered, was sent by a correct participant.

In systems where components exhibit fail-silent behaviour (see section 6.2), it is possible to define accurately situations where components may temporarily *refrain* from providing service, without that having to be necessarily considered a failure. We call this concept *inaccessibility* and it is detailed in section 10.3; the resulting validity property is therefore:

Accessibility: Any message delivered, was delivered to a recipient correct and accessible for that message.

Given that a reliable group communication protocol is supposed to actually deliver messages, it is necessary to specify the (hopefully rare) conditions when the message may not be delivered without that constituting a failure:

Delivery: Any message is delivered, unless the sender fails, or some recipient(s) is(are) inaccessible.

A practical example of the use of inaccessibility in Delta-4 is to represent buffer-full conditions at recipients: this situation — under limits — is one valid situation for the protocol not to deliver a message. The other one concerns sender failure: logically, it is rather irrelevant whether a failure occurred right before a sender sent a message, or right after it did it, let alone during the execution of the protocol. What is important is that recipients perceive whatever happens consistently — this is guaranteed if the protocol secures the unanimity property.

The unanimity property has a cost that may be unnecessary in some situations. For instance, queries to a group of replicas need only to reach one of replicas, or a quorum of them, it does not matter exactly which. Depending on the validity condition, i.e., whether the message must always reach N recipients, or whether it may not reach all of them if a failure occurs (e.g., the sender), an *at least N* or a *best effort N* agreement semantics is obtained, respectively. For completeness, the situation where $N = 0$ is the well-known *datagram* semantics.

> **At-least-N**: Any message delivered to a recipient, is delivered to at least N correct recipients.
>
> **Best-effort-N**: Any message delivered to a recipient, is delivered to at least N correct recipients, in absence of faults.

A slightly different but also useful semantics, is the one where the set of recipients is a *named* subset of recipients. Consider the example of passive or semi-active replication schemes, where there is a privileged participant, i.e., the primary or leader replica, or the example of a cooperating activity where there is a coordinator. For correctness reasons, it is mandatory that any message arrives at that privileged participant; for performance reasons, it may be interesting that the message also arrives at the other participants, should they need it, taking advantage of the multicasting facilities[2]. The considerations made about validity conditions are still relevant, and the following properties are then obtained:

> **At-least-To**: Given a set P_{to} of recipients, any message delivered to a recipient, is delivered to all correct recipients in P_{to}.
>
> **Best-effort-To**: Given a set P_{to} of recipients, any message delivered to a recipient, is delivered to all correct recipients in P_{to}, in absence of faults.

10.1.2. Order

10.1.2.1. Partial and Total Orders. In a distributed system, so that participants coordinate their actions in a decentralized way, they must perceive how the system evolves, i.e., the order in which actions and events take place. Each participant will observe system evolution, as a sequence of events, which may not be the same for all participants, due to the relativity of their positions (space-time view). In other words, the best that can be achieved is a *partial ordering* on events [Lamport and Schneider 1985].

The cause-effect relation is the natural partial ordering of events in a system. Consider two events a and b occurring at different sites of a distributed system. The event a and b can be ordered if an information departing from the site where a occurred, arrives at the site of b before b occurs. The event a is said to *precede* event b, $a \rightarrow b$, in those conditions. Given the space-time relation, it may occur that neither of them can cause the other, in which case they are *concurrent*, i.e., $(a \not\rightarrow b \text{ or } b \not\rightarrow a)$, and may be ordered in any way. For practical reasons, protocols that seek to respect the causal order, normally do so by guaranteeing a "precedes" order, i.e., that events are *potentially causally* ordered. So, a potential causal (causal, for simplicity) order in communication is defined as:

> **Causal Order:** If any two messages, delivered to any correct recipients, are not concurrent, they are delivered to each recipient in their "precedes" order. If the messages are concurrent, their order of delivery is undefined.

2 Although, in case of error, it can be forwarded by the privileged participant.

There are essentially two ways of implementing the "precedes" relation. If participants can be made to only exchange information by sending and receiving messages, they can only define causality relations through those messages. Messages ordered in this way are said to be in *logical* order. Lamport proposed such an implementation, using logical timestamps [Lamport 1978], and Birman later gave an implementation using message piggybacking [Birman and Joseph 1987]:

> **Logical Order:** A message m_1 is said to logically precede message m_2 if: m_1 is sent before m_2, by the same participant **or** if m_1 is accepted by the sender of m_2 before it sends m_2 **or** there is a chain of such recipients and senders linking m_1 to m_2.

For clarity, it is to be noted that the order obtained is thus a *logical (potential) causal order*.

There are however circumstances where the logical order does not correctly represent causality. This can occur when interactions have to take place in real-time (see §9.5.1 for details). The solution for these situations consists in implementing potential causality with a technique based on *temporal* order of messages, for example, by locally time-stamping them, from synchronized physical clocks [Lamport 1984].

> **Temporal Order:** A message m_1 is said to temporally precede message m_2 if: m_1 and m_2 are sent by the same or any two participants, respectively at *real times t1* and $t2$, and $t2-t1 > \delta t$.

The variable δt is introduced to achieve an implementation-independent definition of the relationship defined above — which we shall call *δt-precedence*. Given that message transmission speed is not invariant in the systems dealt with, messages do not arrive naturally in order. One solution is to transport precedence (space-time) relations to the time domain, establish a discrete quantifier, δt, to account for the influence of space[3] and order them based in their time differences of module δt. Thus, δt is the granularity with which it is possible or desired to distinguish orderings between messages. This way, the *temporal (potential) causal order* definition obtained is implementation independent: δt is just the minimum real time difference for the order between two messages to be recognizable, by a given communication system. One way to enforce it is the one mentioned above: using approximately synchronized real-time clocks. The clock precision will correspond to δt. A number of methods exist to maintain local clocks approximately synchronized [Cristian 1989, Schneider 1986, Srikanth and Toueg 1987].

Clearly, not all events that are potentially causally related are actually causally related. Distinguishing this implies that the ordering discipline acquires some specific knowledge. In consequence, when serving a particular application semantics, or computational model, or replication technique, chances are that the causal order may be relaxed. These situations are detailed in section §9.5.1, since they form a practical example of how orderings can be relaxed.

The most obvious example is that of a causal order degenerating to *FIFO* (first-in-first-out) order, if senders are all concurrent, for example, if requests from different clients commute, which is a frequent situation in some distributed client-server applications:

> **FIFO Order:** If any two messages, delivered to any correct recipients, were sent by the same participant, they are delivered in the order sent. If the messages were not sent by the same sender, their order of delivery is undefined.

Given the orthogonality of properties, if messages exchanged by a group are all commutative, order may be completely relaxed, while maintaining other useful properties of the

3 A distance over a velocity.

protocol. An example may be a replicated state machine application where all requests commute [Schneider 1990].

Regardless of the way in which participants causally relate themselves, if a participant is actively replicated, there is the requirement (already discussed in section 6.5), that the replicas process the same messages in the same order. One way to do this is to ensure that they receive *all* of them in the *same* order. So, to start with, messages should be **totally** ordered, as opposed to partially:

> **Total order:** Any two messages delivered to any correct recipients, are delivered in the same order to those recipients.

Then, to receive all messages, the replicas also require unanimity, an agreement property. The combination of total order with unanimity, yields what is called an *atomic multicast* protocol. In other words, a message is either delivered to all recipients, or not at all. Any two delivered messages are seen in the same order by all recipients [Cristian et al. 1985].

As said before, in other forms of replication, namely semi-active, there is a privileged replica, the leader, which performs ordering operations. In this case, a simpler protocol can be used, simply providing unanimity, and no order or at most FIFO order. It is called a *reliable multicast* protocol in Delta-4.

10.1.2.2. Incomplete and Complete Orders. Given that maintaining a globally consistent view of system events by participants obliges the properties of the service to be valid system-wide [Chang and Maxemchuck 1984, Cristian et al. 1985], and given that a significant part of those events are not related to one another, it will be advantageous, from a point of view of performance and simplicity, to ensure consistency only between *related* participants instead of *all* participants in the system. Birman has studied the problem in [Birman and Joseph 1987]. The ISIS CBCAST uses a labelling method controlled by the high-level user, which allows the paths of causality inside the system to be traced. In consequence, it only orders the potentially causal relationships that are significant (for the application). This means restricting the universe of observation, to a subset of the messages exchanged by all participants, or to the messages exchanged by a subset of all participants — in short, by means of incomplete orders (as opposed to complete):

> **Incomplete Order:** An order is incomplete, if the set of messages under the ordering discipline is not the set of all messages delivered.

Being complete or incomplete is orthogonal to being total or partial. "Partial" has been used to name both complete and incomplete orderings in previous work: using clabels in the CBCAST protocol or establishing orderings in the conversation groups in [Peterson et al. 1989] are ways of implementing incomplete orderings. Notwithstanding the fact that the user may end up perceiving a single order, either from using a protocol providing a complete partial order, or several protocols supplying incomplete orders, the fundamental difference is that partial ordering results from an *a posteriori* observation on the way the system evolved. This obliges all events in the system (messages in this case) to obey the discipline defined. On the contrary, incomplete orders are obtained after *a priori* precluding relationships between separate flows of information. Groups are a way of structuring a system in order to reason in terms of incomplete orders.

10.1.3. Synchronism

From our viewpoint, a synchronous protocol is one that has bounded and known execution times. This is a mandatory property for real-time operation and when it necessary to establish temporal order. The reader may find a more detailed discussion of synchronism in chapter 5.

10.2. Related Work

Architecturally, existing systems which provide reliable broadcast services, belong to two major groups: (i) problem-oriented, closed solutions, with dedicated hardware and software, normally designed around Byzantine agreement protocols and networks with multiple redundant message-passing links [Babaoglu and Drummond 1985, Cristian et al. 1985]; (ii) high-level network-independent solutions over standard systems (e.g. UNIX, Ethernet), which are open, but whose achievable performance and dependability are limited by network independence [Birman and Joseph 1987, Garcia-Molina et al. 1988].

Few works exist, however, which directly use standard local area networks as a low level solution [Cart et al. 1987, Chang and Maxemchuck 1984]. This is a fundamental aspect of the Delta-4 architecture. Another fundamental aspect is the preference for local area networks. LANs are a standard means of communication, and components are largely available. Additionally, they have architecture and technology attributes that can be used for improved performance and dependability.

Disadvantages of LAN based solutions are the limited scope of the low-level approach, and the scale problem. Generally, LAN dimension is somewhat limited both in distance and number of nodes[4]. Network-independent approaches scale better than LAN based ones. However, they will hardly be able to take advantage of the optimizations achievable by low-level data-link solutions. On the other hand, the data link is sufficiently low-level to allow several options for upper layers: OSI-like multipoint stacks, such as the Delta-4 MCS architecture, described in §8.1; transfer layers [Chesson 1988], architectures compacting the network and transport layers and providing optimized access to and from user space, very suitable for high-performance real-time; or collapsed application support environments, such as the Delta-4 XPA communications architecture, described in §9.6.4. Importantly, high-performance, hard real-time or highly fault-tolerant applications, may take advantage of the optimized and controlled environment yielded by a single LAN used in a closed fashion.

Further to that, our system, unlike the protocols in [Babaoglu and Drummond 1985, Cristian et al. 1985], does not use clocks; it relates more directly with other *clock-less* approaches [Birman and Joseph 1987, Cart et al. 1987, Navaratnam et al. 1988]. However, among other differences, we have studied the capability of addressing real-time applications, by using techniques to enforce known and bounded execution times. This clock-less approach trades the predictability of clock-based protocols, for faster termination in absence of errors.

In [Babaoglu and Drummond 1985], a phased execution Byzantine protocol is presented, using exact clock synchronization and multiple LAN channels. The protocol family of [Cristian et al. 1985] is diffusion-based. It relaxes the clock assumption to approximate synchronization, but requires all processes to participate in the protocol and delays termination to a worst case time Δ. Δ depends on network parameters and clock precision.

Chang describes an asynchronous atomic broadcast protocol that provides a global order; requests pass through a centralized token holder to be ordered. To tolerate failures of the token site, it is rotated; in consequence, a message is only guaranteed to be committed by all recipients, after two token rotations, introducing a significant latency, which is not bounded a priori. The work of Navaratnam is based on the approach taken by Chang.

Two works on reliable *group* communication — i.e., multicast instead of broadcast — use piggy-backing to establish incomplete causal orderings, materialized in [Birman and Joseph 1987] by piggy-backing messages and using *clabels* in the CBCAST protocol, and in [Peterson

4 Although emerging fibre-optic standards like FDDI, feature up to 1000 nodes, in 100-200 Km, added to an improved *bit error rate*, typically 10^{-14}.

et al. 1989] by piggy-backing references in *conversations* in the Psynch primitive. These orders are partial: only subsets of messages are bound to be ordered so they may be satisfied by several orderings. Birman also provides another primitive, the ABCAST, that enforces a total but not causal order. The AMp provides an ordering that is both total and causal. It uses properties of the underlying network, to achieve it at low cost, in comparison to the alternative methods, based on explicit sequencers [Chang and Maxemchuck 1984, Navaratnam et al. 1988], logical clocks [Cart et al. 1987, Lamport 1978], or piggybacking [Birman and Joseph 1987], involving significant context exchange to enforce logical ordering of messages.

Our approach takes advantage of the properties of broadcast LANs and, such as in [Cart et al. 1987, Chang and Maxemchuck 1984], it integrates communication layer error processing in the reliable broadcast layer. Additionally, it integrates participant management with communication. Participant management includes failure detection, which is normally performed by a centralized monitor, elected or selected in some way [Birman and Joseph 1987, Chang and Maxemchuck 1984, Navaratnam et al. 1988]. However, instead of a preexistent monitor [Birman and Joseph 1987, Chang and Maxemchuck 1984, Navaratnam et al. 1988], the monitor is only elected when needed, on a contention basis. The information needed for recovery actions is very little, making the whole process of election, investigation and recovery, reasonably fast. Failures trigger reconfiguration, which can be a complex process, if histories of past and pending transmissions are to be kept [Chang and Maxemchuck 1984]. In our protocol, any node can become monitor, but the history is reduced to the last accepted message, per sender node.

Approaches using space redundancy, like the architectures proposed in [Babaoglu and Drummond 1985] or in [Cristian et al. 1985], assume replication of the message passing layer. Our architecture uses standard LANs, and space redundancy only exists in the physical layer[5], to resist permanent medium failures.

The V-kernel group IPC [Cheriton and Zwaenepoel 1985] is worth mentioning. It differs from AMp because, while very efficient in multicasting, it does so at the cost of other attributes: faulty process behaviour is not handled, agreement semantics is at-least-one and order is not provided. The kind of attributes of reliable group communication just discussed would have to be built on top of V [Navaratnam et al. 1988].

10.3. System Architecture

This section identifies the building blocks of the Delta-4 communication system architecture and explains the relevant fault model. It should be borne in mind that standard LANs are to be used in order to gain in openness and portability.

The characteristics of the Delta-4 communications architecture, i.e., not using a global clock for communication, and not having space redundancy (single logical LAN), raise two interesting problems, related with its real-time capabilities: how to maintain nodes interconnected in the presence of channel faults and how to achieve synchronism in protocol execution. The first is treated in this section, whereas the second is postponed until section §10.7.3.

10.3.1. Fail-Controlled or Fail-Arbitrary

System modelling and relevant assumptions have been discussed in chapter 6. Let us recapitulate some concepts, and introduce others, which will support the explanations of the

5 I.e., the design of the protocol does not take into account the possible replication of the physical media — if redundant *physical* media exist, they are treated as a single *logical* message-passing link.

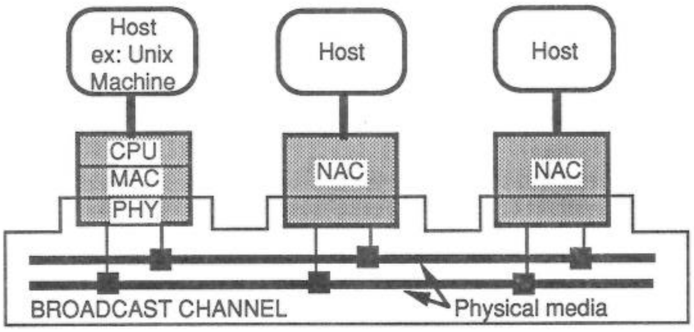

Fig. 1 - Local Computer Broadcast Network

following sections. In the system model used, components interact exclusively through input and output ports, or *service access points*, *delivering* messages to one another. A component produces system errors because it *fails* to follow its service specification. Although each component may be regarded itself as a system, at the level of abstraction of figure 1, where a Delta-4 local computer network is depicted, the system components are: a broadcast channel (possibly composed of several redundant physical media), interconnecting several network attachment controllers (NACs). These serve the computing units (Hosts). Each host-NAC set is a *node*.

The aim of this coarse granularity is trying to decouple errors in the interactions between components, from errors inside components, in order to obtain practical results concerning system design. In fact, the former (*system errors*), are visible outside component boundaries: they are the subject of the system error processing measures that will be discussed throughout the text. The latter are *component* errors: they should propagate to the outside only in accordance with the admissible faulty behaviour specified for the component. The separation between component and system errors yields a well-founded *fail-controlled* system: given the universe of all possible faults, components only display a subset of that universe; the remainder of the universe is supposed to have a negligible probability of occurrence. From now on, when just mentioning *errors*, we mean system errors.

In essence, system errors are caused by component failures. From now on, when just mentioning *failures*, we mean component failures. Component failure modes in the *value* domain are avoided, in our model, in the interest of building an efficient protocol; it has been shown that the comparative cost of coping with them in reliable broadcast protocols is high [Cristian et al. 1985], in relation to time domain errors, the most general of which are timing errors. This work is concerned with *timing* and *omission* errors.

Omission errors in communication may have many origins: mechanical defects in a cable or electromagnetic interference may garble a passing frame; a modem loosing synchrony; a receiver overrun or a transmitter under-run, etc. These component failure modes sometimes occur in bursts. We call *Omission degree, Od*, the number of consecutive omissions produced by a component.

Certain kinds of components may temporarily *refrain* from providing service, without that having to be necessarily considered a failure. That state, that we call *inaccessibility*, is definable, if: (i) it is made known to the client of the component; (ii) inaccessibility limits (duration, rate) are part of the component specification; (iii) violation of those limits implies permanent failure of the component. This attribute will assist the definition of timeliness properties of the service.

10.3.2. NAC Fail-Silence Assumption

The last section discussed the conditions for obtaining a well-founded *fail-controlled* system. Restrictions on the behaviour of the NACs and the Channel will be imposed so that errors do not propagate to the outside of the communication system. Assumptions like *fail-silence* are the most restrictive type (see chapter 6): a component delivers messages correctly (as specified), until it stops functioning (after its first failure). In essence, this means the system always exchanges correct messages on behalf of the hosts. In consequence, systems built to this assumption are: simple, provided the measures taken to justify the fail-silent assumption are also simple; and efficient, because they do not have to take into account processing of errors like timing or value. So, we begin with the working hypothesis that components are *fail-silent*, and that a single component may fail during a *protocol phase*[6].

However, the broadcast channel is equivalent to a single LAN. There is no space redundancy at the message level and consequently, transients in the medium[7] during transmission are unavoidable; they cause omission errors, and have the same impact as if caused by the NAC. So, the model will be weakened in order to take a certain omission degree (Od) into account. The failure-of-a-single-component assumption then means:

Failure of a single component:

- During a protocol phase, any errors result from failures in the same component: it may produce *multiple* errors *($Od = j$, j integer)*, or it may fail permanently, by silencing.

Forgetting for a moment the particular NAC interconnection method, i.e., blaming the NACs for any errors, including those of the network, the overall behaviour of a fail-silent NAC is defined by:

Fail-silent NAC:

- **Fs1:** A NAC may omit to deliver messages to other NACs; if k is the allowed omission degree *(Od)*, then a NAC with $Od>k$ fails permanently (remains silent).
- **Fs2:** All the messages a NAC does deliver, are delivered correctly.

Although having some similarity with the fail-stop processor approach [Schneider 1984], rather than a processor, the NAC is a communication component that provides a reliable communication service to the hosts (processors), which they may use to implement fault-tolerant computing [Powell et al. 1988]. Certain errors are allowed, but when the pre-defined behaviour criteria are violated, there is an abrupt transition to a permanent failure state, by silencing (Fs1). This decoupling yields a simple design of the NAC, using self-checking by duplication and comparison. Additionally, unlike fail-stop processors, it does not require stable storage since all NAC information is volatile. Furthermore, the failure of a NAC is detected not through a local readable failed predicate (which would require the NACs to be fail-operational) but indirectly, through a distributed monitor function.

Now, we proceed by correctly identifying in the model the causes of temporary errors. Remembering that there is a network underlying the NACs, those errors will arise both in the NAC and in the medium. We are going to rework the model to represent these errors in a uniform way, given the broadcast nature of the channel.

6 A **phase** is a well-delimited portion of a protocol execution, which is a containment domain for error detection. A protocol may have several phases.

7 The medium is the passive part (cabling) of the channel.

Observing the internal structure of a real NAC (in the left of figure 1): the operative part — processor, memory, etc. — dubbed CPU, runs the protocols software; the MAC contains the LAN specific medium access protocols — normally cast in communications VLSI — being a frame-level part[8]; the PHY part — containing interface components, like modems, codecs, amplifiers, etc. — is the bit-level, electrical signalling part. PHY is the omission error prone part, so system structure is modified, by moving component boundaries, to include PHY in the channel, as shown in the figure. In consequence, the portion of the physical NAC that must exhibit fail-silence includes the CPU and MAC parts: it executes the communications software. The remainder of the parts, encapsulated by the thin line in the figure, compose the channel. Clearly, if "channel" is substituted for "NAC" in Fs1 and Fs2, we have the behaviour required of the channel.

10.3.3. Channel

The broadcast channel is itself formed by several sub-components recalled here: (active) receiving and transmitting parts (the physical layer (PHY) entities of the NACs) and the (passive) medium. Interaction-wise, the channel offers a pair of ports (input and output) at each node.

The remarks made in the beginning of §10.3.2 should now be apparent: the duplication shown in figure 1, is concerned with providing high availability of the channel. Please remark that the dual cables shown are only symbolic. They may be switch-over buses, or reconfiguring dual rings, depending on the LAN being considered. MAC receives from a single medium at a time (from the part of the channel that is currently selected for reception), so if a transmission error occurs, the frame has to be retransmitted, because it is lost. Then, taking our failure assumptions into account, the faulty behaviour of a channel during a limited number of frame deliveries — a *protocol phase* — will be the following: omission errors only; up to k consecutive errors, in the same outputs, for deliveries coming from any input (receiver or medium failures); up to k successive errors, in any output, for deliveries coming from a single input[9] (transmitter failures). The issue is further discussed in section 10.5, where the behaviour of the channel with competing transmissions from several AMp groups is characterized.

The major consequence of this observation of channel behaviour is property Pn3, in table 3 (Pn3 is discussed in the next section), allowing to establish the foundation of our *bounded omission degree technique:*

- Pn3, in short: if $(k+1)$ series of N transmissions are made, then in at least one series, all N transmissions are indicated in all destinations;
- the protocol is resilient to temporary omission failures, provided that, during each protocol phase, they are produced by one single component, with a bounded omission degree of value k ($Od \leq k$) and k is known during the life-time of the system.

In certain kinds of networks, the channel may become temporarily *inaccessible* (e.g., upon token loss recovery or medium reconfiguration). Although with a very low rate of occurrence, duration may be large, compared to normal frame delivery delay. We specify:

[8] The protocol offers a message-level delivery service to the user. A whole protocol execution is composed of several network-level *frame* deliveries.

[9] Successive does not mean consecutive: consecutive input failures may be interleaved with other transmissions.

Channel inaccessibility limits:

- A channel, during a whole *protocol execution*, has at most one inaccessibility period, whose duration is bounded to a known limit T_{ina}, for all envisaged cases.

In conclusion, this section stated the behaviour required of the different components of the local computer network, in order to meet assumption *Fs*. Next, it must be discussed how to enforce it: the upper part of the NAC will not be addressed here (see chapter 11); channel implementation issues are discussed in the following sections.

10.3.4. The Extended AMp or xAMp

The *Atomic Multicast protocol* (AMp) is the basic communication primitive of Delta-4, on which the various fault tolerance mechanisms rely. The AMp service offers some of the properties identified earlier as being useful in this context. The strongest quality of service (QOS) possible in this architecture, *atomic multicast*, whose properties are highlighted by arrows in table 1 below, formed the basis of the design of AMp. This was the only QOS available during the early phases of the project. The need, discussed earlier in this chapter, for a range of QOSs to obtain the best match between performance and functionality, led to the multi-service *xAMp*. This is an extension of the original AMp, where the atomic QOS is retained, and the additional QOSs derived from the main core of the protocol. Discussions about architecture, dependability and performance are thus made around this core implementing atomic multicast. We will be using AMp and xAMp interchangeably, except where specifically noted. The complete set of properties offered by xAMp is enumerated in table 1.

10.3.5. xAMp Execution Model

A generic communication system should support several applications, possibly disjoint; the entities running them are *fault-tolerant participant groups*. Those groups should operate with parallelism and independence. For example, several independent fault-tolerant groups in the same system, or several groups working in parallel for the same fault-tolerant application, such as groups of replicated clients, accessing the same group of replicated servers. That support should additionally allow for dynamic evolution of groups, if possible in a location independent manner.

This may be established by isolating sets of participants. Let us say that participants group themselves in *universes*. Membership of participants in the different universes may overlap, nest, or be completely disjoint. The word universe was chosen to signal that what happens inside a set of participants is *a priori* independent from the rest of the system, i.e., in other disjoint universes. Each universe is identified by a designation *e*: U_e.

Let us define how participant interactions are supported, and which properties they should observe. Participants interact through the messages they send one another; they are disseminated, universe wide, rather than system wide (i.e., *multicasted*). The attachment of a participant to a universe is thus materialized by an entity used to send and receive messages — a communication *gate*, with the universe designation — and membership, at a given time, is given by the group of gates of that designation that exist in the system. There is a one to one mapping between a node (S_u), a gate ($G_{e,u}$) and a participant ($P_{i,u}$)[10], i.e., there is a single point of access to a universe, in each node, owned by a single participant. A protocol (the AMp) controls all the messages exchanged between participants, ensuring that the sets of received messages — the receive queues at each gate, $M_{e,u}$ — observe certain order,

10 It is recalled that the objective is a low-level primitive. Multiplexing of a single gate by several high-level entities is likely to be performed by a participant that is an upper layer protocol, a service element, etc.; applications or processes access the group through that participant.

Table 1 - AMp Properties
("⇨" designates properties of the *atomic* QOS)

- **Addressing**
    - **Pa1** — *Selective addressing*: The recipients of any message are identified by a pair *(g,sl)*, where *g* is a group identification and *sl* is a selective address (a list of physical addresses).
    - ⇨ **Pa2** — *Logical addressing*: For each group *g* there is a mapping between *g* and an address *Ag*, such that *Ag* allows all correct members of *g* to be addressed without the knowledge by the sender of their number or physical identification.
- **Agreement**
    - ⇨ **Pa3** — *Unanimity:* Any message delivered to a participant, is delivered to all correct participants.
    - **Pa4** — *At-least-N*: Any message delivered to a participant, is delivered to at least *N* participants.
    - **Pa4.1** — *At-least-to*: Given a subset *Pto* of the participants, any message delivered to a participant, is delivered to all correct participants in *Pto*.
    - **Pa5** — *Best-effort-N:* Any message delivered to a participant, in absence of sender failure, is delivered to at least *N* participants.
    - **Pa5.1** — *Best-effort-to*: Given a subset *Pto* of the participants, in absence of sender failure, any message delivered to a participant, is delivered to all correct participants in *Pto*.
- **Validity**
    - ⇨ **Pa6** — *Non-triviality*: Any message delivered, was sent by a correct participant.
    - ⇨ **Pa7** — *Accessibility*: Any message delivered, was delivered to a participant correct and accessible for that message.
    - ⇨ **Pa8** — *Delivery*: Any message is delivered, unless the sender fails, or some participant(s) is(are) inaccessible.
- **Order**
    - ⇨ **Pa9** — *Total order*: Any two messages delivered to any correct recipients, are delivered in the same order to those recipients.
    - **Pa10** — *Causal order*: If any two messages, delivered to any correct recipients, have the same *clabel*, they are delivered by their logical order of precedence.
    - **Pa11** — *FIFO order*: If any two messages from the same participant, with the same *clabel*, are delivered to a correct participant, they are delivered in the order that they were sent.
- **Synchronism**
    - ⇨ **Pa12** — The time (T_e) between any AMp service invocation by a correct participant and the subsequent indication at any correct and accessible recipient is known and bounded.
- **Consistent Group View**
    - ⇨ **Pa13** — Given any receive ordering observed by the participants of a group, each change to group membership is indicated, in a total order, to all correct group participants.

agreement and synchronism properties. Once this is ensured, the consistency rules for each universe or group of participants can be defined. Location transparency is achieved by mapping the universe designation onto a logical address, which is used as a location independent message destination address. This feature is obtained efficiently by using hardware level LAN multicast addresses.

A *correct* participant in a group is a pair $(P_{i,u}, G_{e,u})$, operating according to its specification. So, not only must the participant itself be operating, i.e., timely servicing its input queue $M_{e,u}$, but also the gate must exist. Gate closure implies participant failure, and vice-versa.

The execution view of the system is the one shown in figure 2, where the concepts presented in §6.9.7 appear and one can see how the relevant services map onto the various components. Bottom-up: the *network* service, provided by the channel and all MAC parts, implements the interconnection between protocol entities. In fact, all happens as if there were several virtual broadcast channels (one per group) over a single physical broadcast channel, as shown in the figure. Each group communicates on a virtual channel, through instantiations of the *xAMp* in every node where a gate of the group exists. The xAMp runs on the CPU part, and provides a service to *participants* $P_{i,u}$, residing at nodes S_u. These participants may use more than one gate (e.g., $P_{1,2}$ is in groups *e* and *f*).

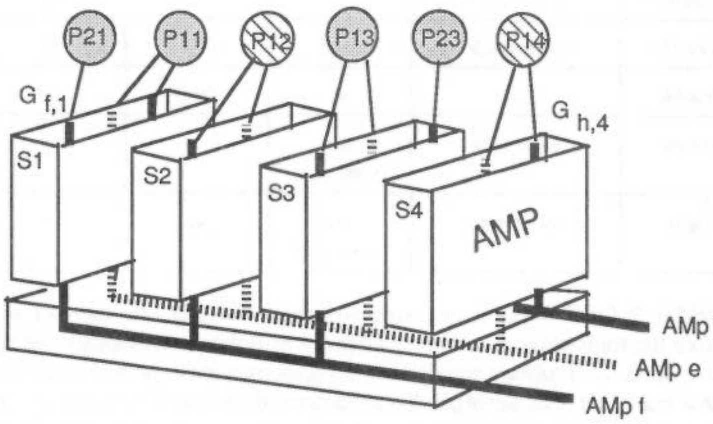

Fig. 2 - Execution View of the System

10.4. Summary of xAMp Services

The properties of the various xAMp qualities of service are summarised in table 2. These qualities of service are available through a multi-fold *amp.request* primitive. The relevant primitives have the following format:

```
res = amp.request ( QOS, groupId, destList, frame,
                             prio, {QOS dependent params} );
amp.indication ( groupId, frame, priority, nSlots );
getSlots.request ( groupId, nSlots );
```

As far as group management is concerned, in addition to the *gateOpen* and *gateClose* primitives, offered by the basic AMp protocol, two new primitives are included: *AttachGate* and *DetachGate*, that allow non-group members to communicate with a group.

In the *amp.request* primitive, *QOS* stands for quality of service, which can be one of the following: *bestEffortN*, *bestEffortTo*, *atLeastN*, *atLeastTo*, *reliable*, *atomic*, and *tight*. They derive from a common AMp core, and each of them offers an incremental quality of service at the price of an incremental loss of performance. All primitives share the same three parameters: *groupId* (the gate identifier), *destList* (the selective list of the destination stations), *frame* (the frame to be sent), and *priority*. All the primitives are then *selective* since only a subset of all the group members needs to be addressed.

Table 2 - Summary of xAMp Service Properties

Quality of service	Agreement	Total order	Causal order per clabel	Rx queue re-ordering
bestEffortN	*best effort to N*	*no*	*FIFO*	*no*
bestEffortTo	*best effort to list*	*no*	*FIFO*	*no*
atLeastN	*assured to N*	*no*	*FIFO*	*no*
atLeastTo	*assured to list*	*no*	*FIFO*	*no*
Reliable	*all*	*no*	*FIFO*	*no*
Atomic	*all or none*	*yes (same gate)*	*yes*	*no*
Tight	*all or none*	*yes (same gate)*	*yes*	*yes*

The *bestEffortN* QOS requires an extra parameter, *nResps*, the number of responses necessary before the transmission is to be considered as finished. Obviously, *nResps* must be less than size of *destList*. If *nResps* equals zero, no responses are expected which is equivalent to an *unreliable multicast*. The *bestEffortTo* QOS accepts instead a response list, *respList*, of stations whose response is necessary to consider the transmission finished. With these two qualities of service, agreement is relaxed. The frame is retransmitted in order to obtain a positive acknowledge from at least *n*/ *respList* of the addressed stations. Accessibility constraints are not tested and no assurance of delivery is provided when the sender fails. The "at-least" qualities of service enforce stronger agreement, assuring that a given sub-set or number of the participants will receive the frame, even if the sender fails. As with best effort QOSs, both *atLeastN* and *atLeastTo* primitives are available. The *reliable* quality of service is a particular case of *atLeastTo*, where delivery is assured to all the addressed participants.

The frames sent through *bestEffortN*, *bestEffortTo*, *atLeastN*, *atLeastTo* or *reliable* QOS are delivered to the user as soon as they are received. No effort is made to assure that the frame is totally ordered in relation with other frames sent through this or through any other QOS. Also, no effort is made to avoid the violations of potential relations of causal order. Only FIFO order is guaranteed, between frames with the same *clabel*.

The *atomic* QOS is the basic AMp quality of service, with "slotted" messages and providing incomplete orderings through the use of *clabels*. As in the basic AMp, frames are, upon reception, always inserted at the end of the receive queue. The *tight* QOS provides the same service but allows the negotiation of the final position of the frame in the receive queue. The frame can be inserted between two pending frames or even between the remaining slots of a frame being consumed. In addition, the *tight* QOS offers a reduced *inconsistency* time.

With these last two QOSs, accessibility is tested in all destinations. The frame is only delivered if all destinations are accessible to received it. If a frame is delivered to at least one participant, it is delivered to the participants in all the addressed stations. Frames sent through *atomic* or *tight* QOS are totally ordered in relation to other frames sent in the same group. Potential relations of causal order are preserved at the communication level between all frames carrying the same clabel.

Frames sent through *atomic* or *tight* QOS can be associated with several "slots". When *nSlots* > 1 a descriptor of the frame is kept at the head of the receive queue. No other frame with ordering constraints is delivered until *nSlots* - 1 slots are removed using the *getSlots.request* primitive. Since high priority frames can negotiate their place within the remaining slots, a new frame can be received as the result of getSlots operation.

10.5. The Abstract Network

In the design of AMp, it was tried to take advantage of LANs, but the network interface, although LAN oriented, has a general set of properties, being in essence, LAN independent. That independence was limited to guaranteeing those properties to be fulfilled by a set of existing LANs, namely: 8802-4 token-bus [ISO 8802-4], 8802-5 token-ring [ISO 8802-5] and FDDI [ISO 9314]. In the criteria of choice, relevant factors were: standardization, existence as industrial products, with availability of (possibly second-sourced) VLSI, possibility of implementing media redundancy, inherent real-time and reliability attributes.

In consequence, the *abstract network* service interface formulates, in a way usable by the AMp, a set of helpful properties (table 3) that are typical of LANs, along with the reliability attributes of the channel, discussed in the last section. At each node, there is a pair of service access points to the network, one for transmitting frames (source), the other for receiving them (destination). These access points will be used by the xAMp machines. The properties are defined in terms of a network delivering each frame, transmitted at a source, to all destinations. Pn1, Pn2, Pn3, Pn6 satisfy the channel model developed in §10.3. Pn4 and Pn5 are the foundation of the ordering attributes of the protocol. Using LAN terminology, the abstract network implementation will comprise functions of the PHY and MAC layer communication and management entities, complemented with the necessary hardware and/or software.

Pn1 and Pn2 impose detection of value domain errors, in a broadcast. This derives directly from the CRC protection mechanism used in LANs, and has its coverage. Pn4 and Pn5 can also be provided by LANs.

Behaviour in the time domain is defined by Pn3 and Pn6. Let us define a protocol phase to be composed of up to t series of up to N broadcasts from N different stations. In consequence, Pn3, the *bounded omission degree* property, guarantees that at least one fault-less series of N broadcasts is obtained in any protocol phase, provided that $t \geq k+1$, where k is the allowed omission degree. This holds even if omission errors are not consecutive in the network[11]; the property also holds for any virtual channel used by the AMp groups, replacing N by the group dimension. It is easy to verify that a single group of the maximum dimension, N, yields the worst case scenario; smaller groups in competition for the channel have a more favourable situation, because they "share" channel errors.

[11] In that case, they must, by assumption, originate in the same component. Reliability of individual components has thus to be such that the single failure assumption holds during the worst case duration of a phase: $(k+1)N$ transmissions.

Table 3 - Network Properties

- **Pn1** — *Broadcast*: Destinations receiving an uncorrupted frame transmission, receive the same frame.
- **Pn2** — *Error detection*: Destinations detect any corruption by the network in a locally received frame.
- **Pn3** — *Bounded omission degree*: In a network with N nodes, in a known interval, corresponding to $(k+1)$ series of unordered transmissions, such that each of the N access points transmits one frame per series, all transmissions are indicated in all destination access points, in at least one series.
- **Pn4** — *Full duplex*: Indication, at a destination access point, of frame reception, during transmission by the local source access point, may be provided, on request.
- **Pn5** — *Network order*: Any two frames indicated in two different destination access points, are indicated in the same order.
- **Pn6** — *Bounded transmission delay*: Every frame queued at a source access point, is transmitted by the network within a bounded delay $T_{ina}+T_{td}$.

Acceptable coverage of the bounded omission degree assumption can be enforced through redundancy in the physical and "medium" layers. An example of an implementation for a dual cable token-bus LAN, allowing real-time switch-over between media, is described in detail in [Veríssimo 1988]. On the other hand, when the channel deviates from the behaviour postulated, it fails permanently. A distributed failure detection and fault treatment mechanism for a redundant LAN channel, which acts upon each node's opinion of the channel state, is also given in [Veríssimo 1988].

Pn6 depends on the particular network, its sizing, parameterizing and loading conditions, which must be known, in order to calculate T_{td}. The value of T_{ina} depends on the network alone and can be predicted for a set of known local area networks. According to the definition of inaccessibility, exceeding T_{ina} implies permanent failure.

To end with, it must be underlined that, in a sense, the *abstract network* extends the concept of LLC — Logical Link Control sub-layer — the LAN independent sub-layer of the IEEE, and later ISO, 802 standard. Historically, the LLC was intended to hide the differences between LANs, multiplex the broadcast channel access, and provide some additional reliability to the MAC datagram service. The abstract network concept is innovative in that, besides offering a message (service data unit) delivery service, it makes visible a set of functional properties that are common to LANs in general. It is believed that the principle can be used to optimize design of protocols intended to work on LANs.

10.6. Two-Phase Accept Protocol

Without restriction on the kind of faults allowed, the cost in traffic and time to achieve agreement is rather high. In consequence, the Delta-4 approach relies on fail-silent property of the attachment controllers, to centralize protocol execution. The protocol, forming the core of xAMp, that implements the atomic multicast quality of service, is a *two-phase accept* protocol. Its operation resembles that of a commit protocol, in that the *sender* coordinates the protocol: it sends a message, implicitly *querying* about the possibility of its acceptance, to which recipients reply (dissemination phase). In the second phase (decision phase), the sender checks whether *responses* are all affirmative, in which case it issues an *accept* — or *reject*, if otherwise. In the event of sender failure, protocol execution is carried on by a termination protocol. However, in this case, although Pa3 is always respected, delivery is no longer ensured (see Pa8) for that execution. This core protocol is formally described in annexe J; together with the formal presentation, the outline of a correctness demonstration is also given.

Two protocol variants have been derived from that description, differing in whether mutual exclusion is ensured between executions, or not:

- The *token-based* protocol assumes that there is *at most one message transmission in course* in a group, at any time, in the whole network. For an efficient implementation, such a token should be managed by hardware, otherwise execution times increase and the forced serialization becomes a performance problem. The token may be the LAN token itself, in token-based LANs. Such a token implementation is discussed in §10.8.1. In fact, these hardware based variants collapse the AMp and the abstract network in a single modified MAC layer, since they are based on changing the standard 802.x machinery.
- Avoiding hardware modifications, the *token-less* variant is discussed in §10.8.2, allowing *several concurrent* message transmissions, for different groups, and *several competitive* message transmissions, for the same group. That variant relies on an abstract network implementation as discussed in the last section.

An informal explanation of the protocol implementing the atomic multicast quality of service is now given. Since some functionality is variant dependent, we base ourselves on the token-less variant in what follows, in order to cover all the relevant details.

10.6.1. Protocol Structure

Each *gate*, the entity used by a participant to communicate, uses an instantiation of the AMp machinery. This comprises a local *GroupMonitor* agent, which participates in error recovery and fault treatment procedures, and two context structures, the *GroupView* and the *ReceiveQueue*, containing, respectively, the group composition and the frames received for that group. A station may belong to any number of gate groups and their number in the LAN is only bounded by implementation limits. Users (mapped on gates) join and leave a group at any station, through local gate opening or closing operations. Due to the nature of the communication architecture, joins and leaves are not truly independent of communication, such as found in [Cristian et al. 1986]. Instead, those actions are performed in a privileged state, which temporarily obliges all participants to synchronize. This ensures that the group views in all gate group members are updated consistently, in relation to the ordered flow of information (§10.6.5). However, the operation is perceived as being dynamic in most situations. The group join/leave protocols in [Birman and Joseph 1987] work similarly.

Error detection is done on a transfer-by-transfer basis, and relies on consistency of the group view by each member. The minimum information needed is a *concise GroupView*, which contains only the number of group members. The concise view is used to detect station failures or undelivered frames (omission errors). For instance, if an emitter requests acknowledgment to a frame, it can compare the number of responses received with its view to detect the presence of an error. However, to allow fast identification of failed stations, the permanent maintenance of a complete list of member identifiers would be desirable: a so-called *extended GroupView* was implemented[12].

[12] Given that the approach is very expensive in terms of storage, a compression technique was used.

10.6.2. Assumptions

We proceed by describing some assumptions that support protocol operation, followed by the description of operation itself. The set of assumptions that guarantee correctness of operation of the token-less protocol implementation is presented below.

Assumptions:

A1 There are at most k message transmission pending from each node, at any time.

A2 There may be several concurrent transmissions in course in the network.

A3 There may be several competitive transmissions in course, in the same group.

A4 From each node, at any time, there may be only one transmission in course associated with a given *clabel.*

A5 The sender positively confirms that all correct participants receive a decision, if it is *reject.*

A6 A transmission, once started, executes atomically, i.e., it is not preempted by other emitting actions, for example, from the GroupMonitor.

Assumption A2 is the source of external parallelism in AMp: unlike other approaches providing global order [Chang and Maxemchuck 1984, Cristian et al. 1985], AMp enforces *incomplete* orderings and in consequence several concurrent executions run simultaneously. On the other hand, assumption A3 allows internal parallelism in a simple way. Group members just run transmissions competitively, in a fully decentralized fashion. There is no coordinator to achieve order, unlike the work in [Chang and Maxemchuck 1984]; several transmissions may be initiated simultaneously.

Order and agreement are achieved by the protocol, based on the network properties, and the error detection and recovery mechanisms provided, which rely on Assumption A1. In fact, this assumption reflects a subtle restriction to parallelism, which maintains protocol simplicity, namely, in obtaining order properties. Together with Assumption A5, it also allows safe use of an error recovery algorithm detailed later in the text, which uses no context about previous transmissions. In consequence, maintenance of histories or lists of significant dimension often found in other approaches is avoided; this greatly simplifies recovery and makes the monitor operate very efficiently.

10.6.3. Atomic Multicast Transmission

A multicast transmission is performed by a protocol entity called the *Emitter Machine*. Assumption A1 limits the number of simultaneous transmission from each node, so there is a limited number of Emitter Machines available at each node. Emitter Machines are locally identified by an integer, s, in the range *1* to k. Thus, every Emitter Machine in the network can be identified by a pair (n,s), where n is the node id. In the following text, an Emitter Machine will be simply designated as "*the sender*".

An atomic multicast transmission is initiated by the protocol coordinator, the sender (E), by sending a multicast frame containing the message. The *Dissemination* phase (figure 3) then proceeds as follows:

- After transmission, E will expect a number of responses indicated by its group view, within a predefined response time (TwaitResponse). When all responses arrive or TwaitResponse has elapsed, they are analysed and if some recipient cannot accept the frame, decision = *reject.*

- Normally, responses are of "can accept" type, meaning recipients are accessible; then, if all recipients responded, according to the sender GroupView, decision = *accept*. If there are responses missing, the data frame is retransmitted.
- If some station does not answer within the retry mechanism, it is considered failed. However, the execution proceeds, allowing timely termination: an *accept* decision may be sent if all the remaining stations can accept the message. Stations considered as having failed are removed from the group view, by the Group Monitor.

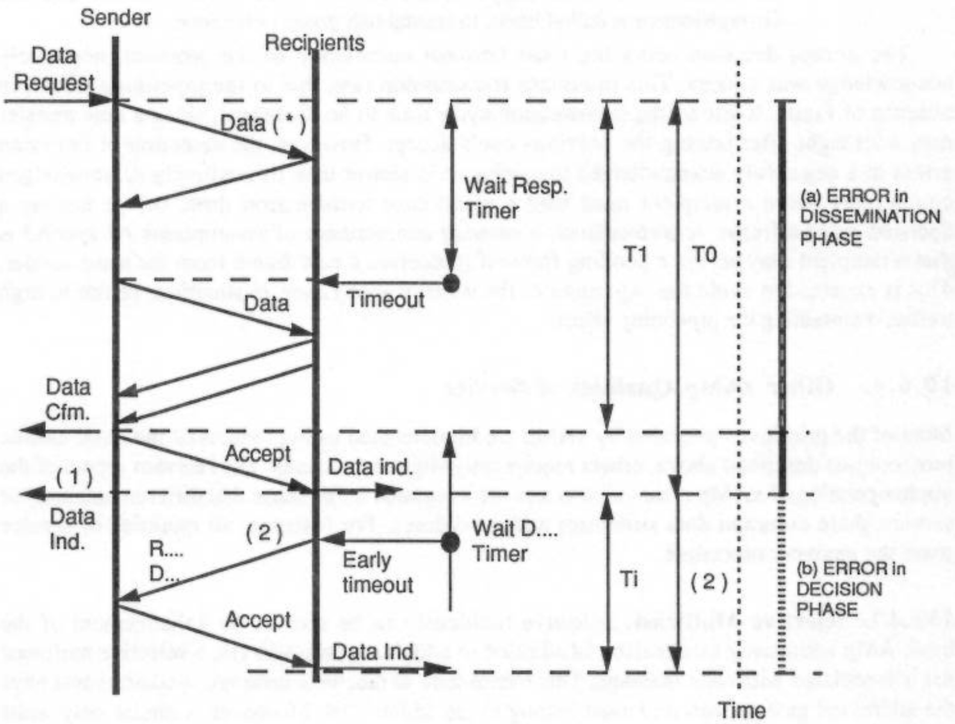

Fig. 3 - Protocol Timing

The *decision* phase is implemented in the following way:

- The *reject* frames always require response (assumption A5). A station that does not answer within the retry limit, is considered failed.
- The *accept* frame, on the other hand, does not require response, in the interest of improved performance. A time-out mechanism, at the recipients, covers omission errors in the transmission of a decision: after receiving an information frame and responding, a timer is started with a predefined TwaitDecision time. If no decision is received within this time, a recipient requests the decision from the sender (figure 3). Safety of this method is based on a simple algorithm, relying on assumptions A1 and A5:
  - all participants log the sequence number of the last message delivered (*lastDeliv* for sender) or accepted (*lastAccept* for recipients);

- omitted decisions can then be recovered very simply: a recipient requests a missing decision, and the sender may respond *accept* if *lastDeliv* has a higher sequence number; else, it is not yet finished with processing it.
- Note that in case of reject, a sender only starts a new transmission after ensuring that all the group members received the reject (assumption A5). So, when a sender receives such a decision request it can answer with an accept without any knowledge of the past, or proceed, if it was still processing that frame. The recipients will retransmit the decision request, until the retry limit is exceeded. When that happens, the sender is considered failed and the GroupMonitor is called upon, to reestablish group coherence.

The *accept* decision being the most frequent completion of the protocol, negatively acknowledge was chosen. This optimizes transmission rate, due to the pipelining effect, in absence of faults. It allows the transmission cycle time to be decreased, since a new transfer may start right after issuing the previous one's accept. However, the detection of omission errors in a negatively acknowledged transmission is slower than its positively acknowledged counterpart, since a recipient must wait a worst case transmission time, before issuing a decision request frame. A performance improving consequence of assumptions A1 and A5 is that a recipient may accept a pending frame if it receives a new frame from the same sender. This is expected to avoid the expiration of the waitDecisionTimer, in situations of fair to high traffic, maintaining the pipelining effect.

10.6.4. Other xAMp Qualities of Service

Most of the primitives provided by xAMp are implemented as byproducts of the basic atomic protocol just described above, others require only slight expansions. The relevant aspect of the implementation of xAMp is that all services are integrated in the sense that different qualities of service share common data structures and procedures. For instance, all qualities of service share the recovery procedure.

10.6.4.1. Selective Multicast. Selective multicast can be seen as an enhancement of the basic AMp addressing mechanism. In addition to a logical group address, a selective multicast list is associated with each message. This means that, to receive a message, a station must have the addressed gate opened *and* must belong to the address list. Moreover, a sender only waits for responses from stations in the selective address list. This excludes non addressed stations from the multicast transfer. The *abstract network* recognises selective address lists in order to perform the address recognition task in the lower layers of the architecture and, whenever possible, by the underlying hardware.

10.6.4.2. Exploiting AMp Message Delivery Procedures. The frame delivery procedure of the basic AMp, TxwResp, is sketched out in annexe J. Its properties, that were illustrated in Lemma *1, can* be stated informally in the following way: when a sender executes TxwResp, frame delivery is assured for all the intended recipients if and only if the sender remains correct during the execution of the primitive.

This procedure is exploited in xAMp to obtain the *bestEffortN*, *bestEffortTo*, *atLeastN*, *atLeastTo*, and *reliable* qualities of service. The *bestEffortN* and *bestEffortTo* QOSs are direct sub-products of the *TxwResp* procedure. The "at-least" qualities of service require extra work since delivery must be assured even when the sender fails.

In the basic AMp primitive a frame is only accepted after being received by all group members. If the sender fails before unanimity is reached the frame is rejected. This means that, in case of failure, the recovery procedures do not need to re-transmit the frame, only to disseminate a *Reject* decision. Now we want to develop a primitive where a frame is accepted as soon as it is received, before the unanimity had been assured, avoiding the use of the two-phase accept protocol. However, since the sender may fail before termination of the TxwResp procedure, only those correct stations who did receive the message can assure protocol termination. So, to offer this quality of service, every receiver must be ready to act in the role of the sender, retransmitting the frame until the protocol terminates. The *"at-least"* qualities of service are then implemented as follows:

The sender transmits the frame and executes the TxwResp procedure. When a recipient receives a message, an Emitter Machine is activated at that node, also executing TxwResp, to control the termination of the protocol. Emitter Machines, when activated at the recipient side, omit the first step of the TxwResp procedure and start immediately collecting acknowledgments, avoiding unnecessary retransmission of the data message.

10.6.4.3. Exploiting Two-Phase Accept. The two-phase accept protocol can be enhanced to allow the sender, which is acting as a coordinator, to obtain the state of the recipients receive queues. If the access points to the receive queues are locked until the frame is accepted, the sender may choose an appropriate insertion point for high priority messages, such that the total delivery order is not violated. The Accept would then disseminate the position of the high priority message in the queue (possible within the "slots" of a message being consumed). This mechanism is the basis for the negotiation protocol available in the *tight* quality of service.

10.6.4.4. Using Responses to Exchange Group Views. The algorithms described until now rely on the knowledge of the group membership to assure protocol properties. This conflicts with an xAMp goal: the ability the allow non-group members to send messages to a given group. For xAMp a solution is envisaged which consists of inserting the group view in the response frames, using a similar mechanism to the one currently used in the AMp *GetView* procedure. This will allow the sender to obtain the group view *during* message transfer, without significant loss in performance.

10.6.5. Distributed Group Monitor Function

The *Group Monitor function* executes, under a privileged state, critical activities relevant to correct operation of the protocol. Namely, it maintains consistency of the GroupView, recovering from station failures. Additionally, it runs the termination protocol in case of sender failure. It also controls group membership: joins and leaves from an MGateGroup require activation of the GroupMonitor so that all GroupViews change consistently.

The distributed Group Monitor function relies on information provided by the various local GroupMonitors of a group. It may be invoked by several groups simultaneously, executing with total independence from the monitors of other groups. The local GroupMonitors are normally inactive. So to speak, a GroupMonitor only exists when needed. At that time, an election is performed, if there is more than one contender, information is gathered from the local entities if necessary and a decision is made and disseminated to group members. If an Active Monitor fails, it is replaced by another GM who detects the failure. The procedure is recursive.

10.6.6. Active Monitor Election

The need for group monitor intervention can be detected simultaneously at two or more group members so it is to be expected that several GroupMonitors (GM) will compete for the activity.

To ensure that only one monitor becomes active, the first frame sent by a candidate carries the Suspension attribute, which suspends multicast traffic on the MGateGroup. Other candidate GMs will find the traffic suspended and return to the Standby State.

However, this simple solution must be improved to avoid deadlock and contention. Deadlock occurs when an Active Monitor fails, leaving the traffic suspended. Contention can occur in the presence of an omission fault during the competition for the activity if two different monitors lock different subsets of the recipients.

These two problems are solved with a mechanism based on the association of a value, the suspension level, to the suspension attribute. When a receiver is suspended it stores the suspension level associated with the suspending frame; this value is the Current Suspension level. If another frame, with the Suspension attribute, is received during the suspended state, this frame is rejected unless it carries a suspension level higher than the current one. In this case, its sender will become the new Active Monitor, preempting the old one (and the current suspension level is updated). Deadlock and contention avoidance with the suspension level mechanism are now explained.

To solve the deadlock problem a timer is started when the traffic is suspended. Whenever the Active Monitor (AM) sends a new frame the timer is restarted. If the AM remains silent for a long time, the timer will expire and the failure of the AM is assumed. The GroupMonitor who first detects the failure becomes active, incrementing the suspension level. The contention problem is also easily overcome: when one monitor detects contention (receiving some responses reporting traffic suspension and others acknowledging its frame) it retransmits the frame incrementing the suspension level. In both situations, the new frame with a higher suspension level than the current one will be accepted by all the suspended participants, establishing consensus on who is the new GroupMonitor.

10.6.7. Handling of Failed Stations

Whenever a station fails, the GroupMonitor function is invoked, to reestablish group coherence. The GroupMonitor winning the activity must, if needed, finish the transmission interrupted by the failure and disseminate a new group view. To accomplish this objective, the monitor executes in two phases (StepOne and StepTwo).

These phases include the identification of failed stations, search for the presence of pending messages from failed emitters, decision to *accept* or *reject* those messages and finally the dissemination of the new group view.

The decision process for the frames pending from failed emitters is the most difficult step of all monitor actions. The Monitor must ensure that the pending transmissions are finished correctly. This means that the Monitor must investigate if the message had been accepted by any member of the multicast group and if so, all the other members must also accept it. If none of the group members had accepted the message, it can be rejected.

This action can only succeed if the recipients are able to provide some knowledge that can map onto past transmissions, since many other messages may have been received (from other emitters) prior to detection of the failure. Since no context is kept about previous transmissions (see §10.6.2), an indirect information will help solve the problem. Each station keeps a table with the Multicast Data Number (MDN) of the last message accepted from each sender in the LAN[13] (StationMdnTable[14]). The active monitor reads its contents concerning the failed

[13] From assumption A1 there are at most k senders at each node, thus only *k*N*sizeof (MDN) bytes* need to be reserved for monitor action (where N is the number of nodes in the system).

[14] Note that a station may belong to many gate groups and more than one group may keep information about the same station. Thus, to save both execution time and storage, a structure containing information about all the stations with AMp capability is created in every station: the *StationMdnTable*.

station(s), on all group members, chooses the highest MDN value for each sender and disseminates pairs *senderId/MDN*, during the next phase. These search for the presence of a message in the receive queue, sent by the failed senders. If found, should its MDN be lower or equal to the received MDN, it is accepted, else it is rejected.

The first phase (StepOne) covers the identification of the failed stations, the search for pending messages and, finally, the investigation of the *StationMdnTable*. The investigation frame carries the identification of the failed stations. The responses will carry a list of triplets containing the id of the failed station, the content of the StationMdnTable for that station and a Boolean stating if there is any pending message in the receive queue sent by that station. After this, a second phase (StepTwo) is performed, including the dissemination of the decision for the pending frames and group view. An exceptionDecision frame simply contains pairs *senderId/MDN* where the MDN is the highest MDN (for that sender) received in the first phase. The recipients will use this value and the decision algorithm presented above, to finish pending transmissions. The first frame sent in the monitor action, always carries the suspend attribute, while the last one carries the resume attribute.

10.6.8. Joining and Leaving the Group

There are two control frames sent to change the group membership, inserting a participant in or removing it from a gate group. These frames, called respectively *OpenGate* and *CloseGate*, are sent in response to a local request by a participant, to join or leave a given MGateGroup. Since these two frames change the GroupView, they may interfere with other transmissions in course. So the traffic is temporarily flushed and suspended, prior to the processing of a Gate frame. In a normal situation, this gap is hardly perceived by group users.

The *CloseGate* action is very simple. A frame with the identification of the participant to be removed is sent with the Flush and Suspend attributes. If the traffic was not already suspended by another participant, the end of the current message transmission is awaited for, before a *close accept* frame resuming the traffic is sent. When the accept frame is received, each participant changes its GroupView and group activity is restarted.

The *OpenGate* action needs an extra step since the participant desiring to enter a multicast group must, prior to sending the Gate frame, obtain a view of the group. The OpenGate action is then started with an GetView frame, which is sent with the Suspend and Flush attributes as above: this ensures correctness of the View obtained. If no response is received to the GetView investigation, the frame is retransmitted. If silence persists after $k+1$ tries — allowed omission degree plus one — the requesting participant assumes it is the first member of the MGateGroup and initializes his GroupView. If the GetView investigation obtains a response, it is followed by the *OpenGate* frame, which is retransmitted if needed. When it is acknowledged by all the participants, an *accept* frame is sent. At this moment, the GroupViews are changed, inserting the new participant in the group. Traffic is resumed.

10.7. Performance and Real-Time

Definitions about real-time communications protocols have been given in chapter 5. Requirements for XPA high-performance real-time communications, have been stated in chapter 9. This section addresses the derivation of the execution time expression, calculation of execution times under several scenarios and, finally, the demonstration of the existence of a known upper bound for the execution time, necessary for synchronous operation. The framework covered by the present section is mainly that of high-performance, fault-tolerant, real-time systems. These issues are addressed in chapter 9. The relevant scope in communications is that of a service with timeliness guarantees — among other attributes — specially important for XPA (see section 9.6.4).

10.7.1. Atomic QOS Execution Time

To quantify the duration of an execution of AMp, one has first to observe that it depends very much on the LAN used, and the particular protocol implementation. The following observations will be made on the token-less protocol implemented in firmware, as described in [Veríssimo et al. 1989]. The atomic QOS, being the most complex and covering many aspects of weaker qualities of service, was chosen for this example. The execution time (T_e) expression[15] for the atomic service, taking the possible errors into account, is given in table 4. Observe that the protocol is structured in transmission-with-response series of the several frame types, $t_{xwr}(FR)$, where FR is: message *msg*; decision *DEC*; request for decision *reqDEC*; monitor action *Step1* and *Step2*. These alternate with several processing or waiting steps, accounted for by CPU times (t_{pr}) and timers ($t_{waitResponse}$, $t_{waitDecision}$ and $t_{earlyDecision}$). The transmission-with-response is itself composed of datagram transmissions and processing. Its duration highly depends on the target network and NAC performance parameters.

The sender uses timer $t_{waitResponse}$, after receiving confirmation of transmission by the network, to control recipients activity. Each recipient uses his timer $t_{waitDecision}$, to control sender activity. In fact, it is a two-shot timer, to optimize recovery in case of omissions, with a first time-out given by $t_{earlyDecision}$.

The variables O_i take values according to the allowed error scenarios, and the execution time expression changes accordingly. For example, O_3 accounts for the existence of errors in transmission of the decision frame. If errors occur, the term on $\overline{O}_3$ in the expression is replaced by the one on O_3.

10.7.2. Performance

This section deals with the performance implications of supporting distributed applications, with reliable broadcast protocols. In general-purpose computing systems, most of the time domain requirements are of the *on-line*, or *soft real-time* kind. That is, applications require responsiveness, fastest possible reaction and a probabilistic treatment of worst-case response times. To encourage utilization of reliable broadcast protocols in such applications, it is mandatory that the above-mentioned benefits in quality of service are not considered too costly in performance, by the user(s).

From the performance viewpoint, there are three questions in the design of reliable broadcast protocols, which influence the final result: (i) which fault model; (ii) what level in the communication stack; (iii) which network? It is assumed that the communication system is *fail-silent*. The protocol was designed both to run on top of LANs, i.e., at the data link level, and not to depend on a particular LAN.

There is clearly a difference between LANs with moderate date rates, from 4-16 Mb/s, and LANs with high data rates, of the order of 100 Mb/s. So, to predict performance of AMp, we will concentrate in one example of the lower class, namely a 10 Mb/s 8802-4 token-bus. Second, it is analysed whether AMp performance will benefit from migrating to a higher throughput LAN, such as the 100 Mb/s FDDI ring.

The scenario defined is a small cell network for real-time manufacturing control, in an industrial environment: a 500m network with 32 stations. The performance assessment will be

15 We recall that the execution time is the time between the send request primitive and the issuing of the last receive indication for that message.

Table 4 - Temporal Expression for AMp Execution Time (T_e), with Several Error Scenarios *(token-less variant*

	All situations
T_e	$(O_1+1).t_{xwResp}(msg)+$ $\overline{\mathrm{O}}_5.\{t_{pr}+(O_2+1).t_{xwResp}(DEC)+t_{td}\}+$ $O_5.(-t_{td}+t_{AM})+O_6.t_{ina}$
	Active Monitor action — after sender failure
t_{AM}	$(O_3+1).t_{xwResp}(INV)+t_{pr}+$ $(O_4+1).t_{xwResp}(DEC)+t_{td}$
	Variables for error situations
O_1	Number of Dissemination errors (≤ k)
O_2	Number of Decision errors (≤ k)
$O_{3,4}$	Number of Monitor action errors (≤ k)
O_5	Sender Failure after Dissemination: true — O_5=1; false — O_5=0
O_6	Inaccessible Network: true — O_6=1; false — O_6=0

based on an evaluation of the execution time, T_e. Values will be extracted from the expression in table 4. The various parameters will be quantified taking into account the specific LAN, the scenario, and assuming a well-engineered implementation on a high performance NAC. This supposes a very efficient local executive, very powerful CPU and fast data paths. The objective of this "optimistic" approach is to show the possibilities of the architecture and protocol, rather than those of a given implementation, and mainly, to appreciate the comparative behaviour between different LAN ports.

Real implementations will hardly combine all of these favourable attributes. A normal AMp implementation will run at about four times the optimum-case execution times quoted.

Some example situations were extracted, as a function of message length, number of group participants, and several typical error situations. The relevant values are presented for T_e of AMp on a token-bus LAN, in table 5.

With the purpose of comparison with the token-bus LAN, predictions for an FDDI network with the same dimensions are then made. The channel rate, which was 10Mb/s, becomes 100Mb/s. The predictions for T_e in the same situations as done for token-bus are repeated in table 6.

These results are detailed in [Veríssimo and Rodrigues 1990]. The use of FDDI, with a 100Mb/s rate, seems to be advantageous: it is shown that for small messages not only AMp throughput increases, a natural consequence, but also *speed*, measured in duration of single AMp executions[16]. This fact is of importance, since it has been recognised that communication

[16] In essence this is due to the following facts: (i) the increased available bandwidth, which reduces the impact of protocol frames in channel utilization; (ii) the increased speed, because of the shorter rotation and transmission times, for the same load condition; (iii) the assumed low values for processing times, which are achievable for a low-level implementation.

speed is the dominating requirement for distributed computing. On the other hand, it shows a way of using technology to improve performance without compromising portability. While keeping a neat, independent interface in the LAN world, something can be done to increase performance, by merely changing LAN.

Table 5 - Execution Time Predictions, *Te* (ms) (AMp on TB)

	80 octets		n = 6	
T_e	n = 6	n = 12	320 oct.	1280 oct.
no faults	2.7	3.1	2.8	3.6
1 om. f. diss.	4.5	5.1	4.7	5.5
1 om. f. dec.	5.5	6.2	5.7	6.5
k om. f. diss.	6.4	7.2	6.6	7.4
sender fail..	13	14	13	14

Table 6 - Execution Time Predictions, *Te* (ms) (AMp on FDDI)

	80 octets		n = 6	
T_e	n = 6	n = 12	320 oct.	1280 oct.
no faults	0.72	0.94	0.73	0.81
1 om. f. diss.	1.0	1.3	1.0	1.1
1 om. f. dec.	1.15	1.4	1.2	1.3
k om. f. diss.	1.3	1.6	1.3	1.4
sender fail..	2.4	3.0	2.4	2.5

10.7.3. Synchronism

One of the questions raised in the beginning of this chapter was how to achieve synchronism with a clock-less protocol, given that classical approaches to synchronous protocols are clock-based [Babaoglu and Drummond 1985, Cristian et al. 1985]. Synchronism is taken in the sense of the existence of a known time bound for all executions.

A fundamental issue about synchronism is the way timing errors are treated, i.e., if there can be, for example, delivery of messages outside the bound, and if so, how is this event treated. Timing errors are avoided by imposing a performance specification on the NAC. However, load variability imposes a significant amount of head-room in that specification, since worst case delay situations are far from normal ones. Since the execution proceeds in transmission-with-response series, late deliveries are detected as not belonging to the present series and can be rejected. This is equivalent to transforming a timing error into an omission error, reflected in the omission degree: a NAC doing more than *k* successive timing errors is failed. This way, sporadic timing errors are allowed, and thus the performance specification can be tightened.

The main issues about the clock-less AMp structure concerning synchronous operation are the following:

- achieve and determine upper bounds on frame delivery delays by the abstract network, in the presence of overload and faults (T_{td} and T_{ina}, discussed in §10.5);
- impose a performance specification on the NAC hardware and software (CPU, kernel, etc.) in order that processing times of the protocol actions be bounded and known for the specified worst case traffic scenario;
- structure the protocol in phases, so that an execution predictably has a bounded number of phases; clearly delimit phases, in what concerns error detection and recovery (omission and timing), and permanent failure detection;
- structure each phase as a series of timed-out transmissions-with-response, so that it can be decomposed in time, in a sequence of frame deliveries and protocol actions as specified above, thus having a known duration bound.

With these measures, the AMp execution time expression of table 4 is bounded to a known value (and in consequence, T_i, once $T_i \leq T_e$). Given the assumptions made in §10.3, namely the single failure assumption, only some error combinations are allowed. To determine $T_e = T_{emax} - T_{emin}$, those error scenarios have to be exercised in the expression to compute T_{emax} and T_{emin}. The existence of the bound ultimately means the AMp is *synchronous* in the sense of property Pa12 in table 1.

10.8. Implementation Issues

Two implementation approaches are possible, as already seen. One consists of embedding the atomic multicast mechanisms in the Medium Access Control (MAC) sub-layer, by intervention in the MAC state machines. The second approach consists of implementing atomic multicast on top of the exposed MAC interface of a LAN VLSI, while still presenting the same extended interface to the user (figure 4).

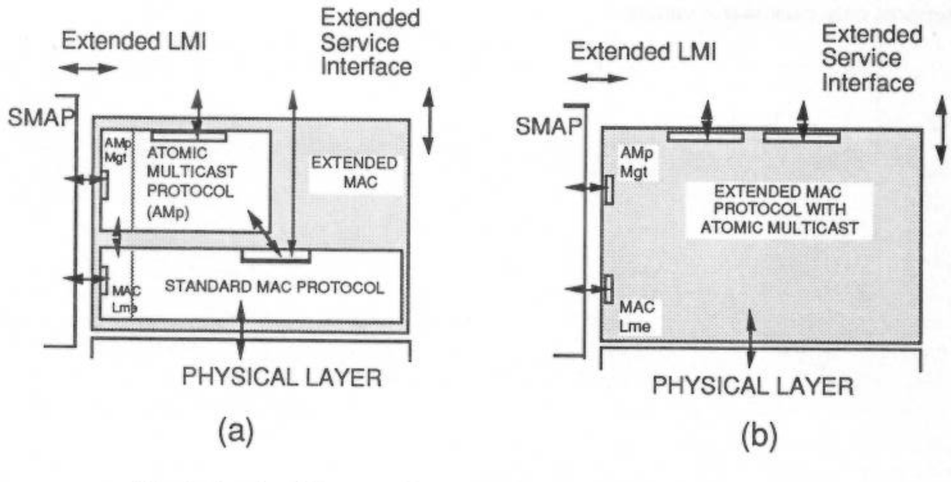

Fig. 4 - Functional Decomposition of the Alternative AMp Implementations

10.8.1. The Token-Based Protocols

Hardware support was considered mandatory to achieve an efficient solution for the token-based protocols in Delta-4. The approach followed consisted of designing a superset of the MAC sub-layer, in order to include AMp services. The neat interface between the *abstract network* and the *AMp* itself disappears, since they are compacted in a "super"-MAC. Consequently, the protocol becomes LAN dependent.

A theoretical study of the approach for token-bus was made, whereas a prototype for token-ring is actually being built in Delta-4: it is dubbed "Turbo-AMp" and the protocol was formally specified and validated. It is a superset of the standard, with the existing services plus the AMp functionalities (figure 4b). Turbo-AMp only implements the atomic multicast service. Although implying modified VLSI (for example an ASIC), it is an interesting approach for providing a standard LAN with atomic multicast capability. Intervention in the state machines allows establishing a clear protocol execution containment domain, through *token hold control.* This hardware token[17] yields an efficient implementation of a variant of the two-phase accept protocol of figure 1, annexe J. A high degree of *synchrony* and low *error latency* are given by a transmission with multiple acknowledgment mechanism, based on on-the-fly operations on the passing frame.

10.8.2. The Token-Less Protocol

The token-less variant implementation, shown in figure 4a, is less efficient, yet readily implementable in software or firmware. The protocol implementation is detailed in [Veríssimo et al. 1989]. The main points are: higher parallelism, portability (LAN independence by means of the abstract network); evolvability (being software-based). The protocol particularities, with regard to the core protocol in annexe J, are the non acknowledged decision and the group monitor actions. The protocol was described informally in section 10.6.

In consequence, the token-less variant is a *generic* protocol, LAN independent, and ported to the various LANs in Delta-4, just requiring an implementation of the abstract network on each target LAN. It was also the basis for the evolution that led to xAMp, whose extended services only exist in this variant.

---

[17] Which is the same token as the LAN itself.