# Reliable Broadcast for Fault-Tolerance on Local Computer Networks

Paulo Veríssimo, José Alves Marques
Technical University of Lisboa
INESC*
e-mail:...paulov@inesc.pt or paulov@inesc.uucp

**Abstract**

The importance of fault-tolerance mechanisms in application-independent systems, has led to the increased use of techniques based in "macroscopic" replication of components and software oriented error processing. In distributed environments, management of the replication of components throughout different sites, may benefit from the availability of *reliable broadcast* or *multicast* protocols.

This paper discusses the definition and design of a generic reliable communication architecture, on a widely used host independent platform, such as a local area network. Two aspects of relevance are the use of *non-replicated* LANs and of *self-checking* components.

The protocol itself is innovative, in the sense that, although clock-less, and running on a non-replicated network, it displays bounded execution times. The architecture is in consequence capable of reliably addressing real-time.

## 1 Introduction

The proliferation of fault tolerance mechanisms in distributed systems has lead to the increased use of techniques based in "macroscopic" replication of components[1], and software oriented error processing and fault treatment. In distributed environments, management of the replication of components throughout different sites, may benefit from the availability of high-quality communication services, such as the ones materialized by protocols generally designated as *reliable broadcast* or *multicast* [3,5,7].

If openness and versatility are desired, host computers may have heterogeneous behaviors, whereas the envisaged applications may have diverse fault tolerance requirements. On these grounds, it may be interesting to consider a generic reliable communication service, on a widely used host independent platform, for example, a local area network. In building such a service, one is faced with three concerns: (i) network design, in order that

*Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6º - 1000 Lisboa - Portugal, Tel.+351-1-545150. This work has been supported in part by the CEC, through Esprit Project 1226 - DELTA-4, and by JNICT, through Project 87634 - ESTIMULO.

[1] Processes, files, objects, etc.

it is as portable as possible; (ii) architecture design, so as to harmonize the use of standard components with the obtainment 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.

Protocol design has been discussed in earlier papers, where we have presented two variants of a basic atomic multicast protocol, dubbed **AMp**, that we describe here: the first, specifically for a token-bus network, requires hardware modifications [21]; later, we gave a software implementable protocol, which is LAN independent [18]. The implementation specification of either of them is fairly complex, recommending the use of semi-automated methods for validation. In consequence, in the Delta-4 architecture[2], where target implementations of the protocol were made, a fault injection campaign is in course, with the aim of forecasting faults and assisting in its removal, with the help of a specialized tool [1]. We specified the software variant formally, in Estelle, and it is being formally validated [13]. Both validation works are being performed by other partners in the Delta-4 project.

The present paper concentrates in the design of the overall architecture, including the network, and in justifying the correctness of some assumptions and protocol design options. The paper is organized in the following way: in Section 2, we model a communication service for the kind of problems we wish to solve. We elect the state machine [9] as our basic paradigm for fault tolerance, and describe the properties required to support it[3]. However, since no service can be more reliable than the system supplying it, in Section 3 we identify the building blocks of the architecture: the *network attachment controllers*, or NACs, and, interconnecting them, the broadcast *channel*. We specify their individual behavior with regard to failures, the way they interact with one another and finally, we discuss 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 system correctness, respectively: the fact that a "nice" behavior can be expected, and that time domain redundancy must be used, to recover from transmission errors. Sections 4 and 5 deal with the provision of an *abstract network* service by the channel, and of an *atomic multicast* service, by a protocol built on top of the abstract network. The implementation issues discussed, reflect the experience of the authors in building a prototype of the architecture, for a token-bus network, in the Delta-4 system.

Finally, the characteristics of our architecture, i.e. not having space redundancy (single LAN) and not using a global clock for communication, raise a concern about the suitability of our system for fault tolerance in real-time. Two problems, concerning its behavior in the time domain, need solution: how to maintain nodes interconnected, in the presence of channel faults; how to achieve synchrony in protocol execution. The first is treated in Section 3, whereas the second is postponed until Section 5.1.

### Related work

We could place our work between the two main approaches known for implementation of broadcast protocols. The protocols in [2,7] are based in the existence of a global clock: we say they are *clock-driven* and they are inherently synchronous; on the other hand, *clock-*

[2]Delta-4 is a CEC Esprit II consortium, formed by Ferranti-CSL (GB), Bull (F), Credit Agricole (F), IEI (I), IITB (D), INESC (P), LAAS (F), LGI (F), MARI (GB), NCSR (GB), Renault (F), SEMA (F), Un. of Newcastle (GB), designing an open, dependable, distributed architecture.

[3]A revised version of those presented in [12].

*less* approaches, as the name implies, do not use clocks, and the protocols known, such as [3,4,11,10], are asynchronous, meaning they do not ensure bounded execution times.

Our protocol, albeit clock-less, uses properties of the underlying network (a broadcast LAN), to enforce known and bounded execution times. We explain the technique in Section 5.1. The *synchronous clock-less* approach trades the predictably of clock-driven protocols, for faster termination most of the times, still having the capability of addressing real-time applications.

Similarly to [4,5], our approach integrates communication layer error processing in the reliable broadcast layer, by means of frame acknowledgments and retries. 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 [5,3,11]; failures trigger reconfiguration, which can be a complex process, if histories of past and pending transmissions are to be kept [5]. So, 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 [2], or in [7], assume replication of the message passing layer. Our architecture uses standard single LANs, and space redundancy only exists in the physical layer, to resist permanent medium failures. Achieving correct protocol operation under these circumstances has lead to some design subtleties, discussed in Section 3.1.

A fundamental difference between our approach and works that simply rely on realistic assumptions of the halt-on-failure kind, such as [5,3,11,4] is that our architecture has explicitly dealt, both at the model and design levels, with the issue of ensuring coverage for the assumptions made, by combining fault avoidance with fault tolerance. Whether it is, on the other hand, capable of attaining high levels of dependability, depends on the level of such coverage, achievable by real-life component implementations. That is an issue currently under study in the Delta-4 project.

# 2 Execution Model

We discuss in this section, the kind of problems we propose to solve, and how to build a reliable group communication service, for that purpose.

If we wish a generic communication system, then it should support several applications, possibly disjoint; we identify the entities running them, as 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. This may be established by isolating sets of participants. Let us say that participants group themselves in **universes**; 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. A universe is identified by a designation $e$.

### Participant interactions

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 which exist in the system. There is a one to one mapping between a node ($S_u$), a gate ($G_{e,u}$) and a participant ($P_{i,u}$)[4], 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, agreement and synchronism properties.

Once we control all interactions, we can define the consistency rules for each universe – group of participants. Once we can instantiate several independent protocol machines – $AMp_{e,u}$ is the instantiation of $AMp$ in each node $S_u$, where a gate with designation $(e)$ exists – we achieve independence and parallelism of operation of those groups. 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.

### AMp properties

Having made clear the existence of several communication flows in the system, inside as many universes, from now on, all discussions about AMp concentrate on what happens inside a universe, and are valid for every single one.

Let us decide about the highest quality of service AMp must provide. Several techniques exist, to achieve distributed fault tolerance. One such technique, called the State Machine (SM), described in [9,14], is generic enough to accommodate several known paradigms for distribution and fault tolerance. We take its communication needs as a reference for the highest quality a versatile communication service should provide. So, in the primitive we propose, agreement is unanimous, and order of delivery is consistent and obeys causality[5], as depicted in table 1. The properties are valid for each universe of participants, to fulfill our objective of parallelism and independence. The consistent group view property is a fundamental tool for support of high-level process group management. The remainder of the properties will be discussed along the text. Synchronism, namely, is an ingredient of real-time systems: we discuss it in Section 5.1.

## 3 System Architecture

Architecturally, existing systems which provide reliable broadcast services, belong to two major groups: (i) problem-oriented, closed solutions, with dedicated hardware and soft-

[4] We recall that our objective is a low-level primitive. Multiplexing of a same gate by several high-level entities is likely to be performed by a participant which is an upper layer protocol, a service element, etc.; applications or processes access the group through that participant.

[5] In the sense of a message causing the delivery of another message, in the same or different nodes.

Table 1: AMp Properties.

---

- **Agreement**
  - **Pa1 -** Unanimity: Any message delivered to a participant, is delivered to all correct participants.
- **Validity**
  - **Pa2 -** Non-triviality: Any message delivered, was sent by a correct participant.
  - **Pa3 -** Accessibility: Any message delivered, was delivered to a participant correct and accessible for that message.
  - **Pa4 -** Delivery: Any message is delivered, unless the sender fails, or some participant(s) is(are) inaccessible.
- **Order**
  - **Pa5 -** Consistent causal order: Any two messages delivered to a participant, are delivered to all correct participants, in the same order, which obeys causality.
- **Synchronism**
  - **Pa6 -** Loose synchronism: (i) The time between any service invocation and the subsequent indication at any recipient ($T_e$), as well as the time between any two such indications ($T_i$), are known and bounded; (ii) $\Delta T_e$ and $\Delta T_i$ are not negligible, in relation to $max\ T_e$.
- **Consistent Group View**
  - **Pa7 -** Given any receive ordering observed by the participants of a group, each change to group membership is indicated, in a consistent order, to all correct group participants.

---

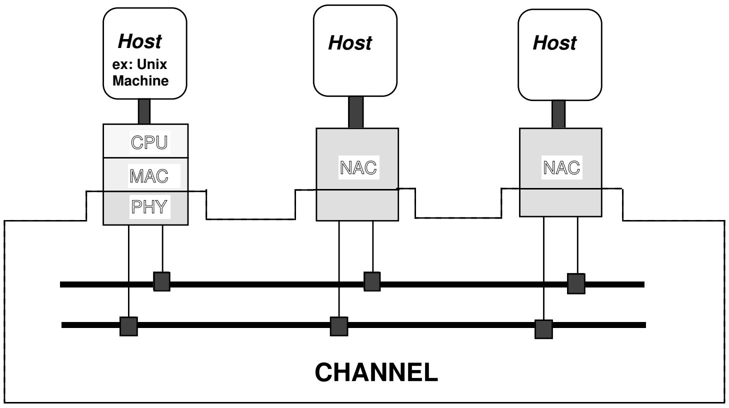


Figure 1: Local computer broadcast network

ware, normally designed around byzantine protocols and space-redundant networks [2,7]; (ii) high-level network-independent solutions over standard systems (eg. Unix, Ethernet), which are open, but whose achievable performance and dependability are limited by network independence [3],[8]. Few works exist, however, which directly use standard local area networks as a low level solution[5,4]. We will be using standard LANs, as a means for combining the openness and portability of the latter approach with the dependability and performance of the former. In this section, we identify the building blocks of our architecture and develop the relevant fault model.

### Fail-controlled or fail-arbitrary

The granularity of our system model is coarse. We are interested in viewing it as a set of components interacting exclusively through input and output ports, or *service access points*. In Fig. 1, where our local computer network is depicted, the system components are: a broadcast channel, which interconnects several network attachment controllers (NACs)[6]. These serve the computing units (Hosts)[7]. Each component is then providing a service to other components or to the outside user, as specified in some form: in essence, delivering messages. A **correct** message delivery by a component, is one which is in accordance with that component's specification, both in the value and time domain.

When a component fails to follow its service specification (either temporarily or permanently), while interacting with another component, an *error* occurs at system level. Given the set of all possible failure modes, if components only fail according to a specified subset, they are *fail-controlled*, as opposed to fail-arbitrary, where any kind of failures is expected. From now on, when just mentioning **failures**, we mean component failures.

Let us call those errors which are visible outside component boundaries, *system errors*. Then, with properly designed fail-controlled components[8], all other errors which may occur – inside them – become irrelevant to us. In fact, these *component* errors only propagate to the outside of the component, in accordance with the behavior specified for it: the allowed set of system errors, which are the subject of the system fault tolerance measures we will be discussing throughout the text. From now on, when just mentioning **errors**, we mean

[6]Let us forget for now the internal composition of NACs, shown in the leftmost NAC.

[7]We will use *node* to designate a host-NAC set.

[8]Measures to process errors at the component level, or to design components in order that those errors do not occur, pertain to fault avoidance.

system errors.

We saw system errors are caused by component failures. Component failure modes yield a combination of system errors in the **time** and/or **value** domains. The latter 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 [7], in relation to time domain errors, the most general of which are timing errors. A component producing **timing** errors may deliver messages in an untimely manner. All the messages it delivers, including these ones, are otherwise (as specified) correct. Similarly, a component producing **omission** errors may omit to deliver messages. All the messages it does deliver are correct (as specified).

Omission errors in communication may have many origins: mechanical defects in a cable or EMI may garble a passing frame; a modem loosing synchrony, a receiver overrun or a transmitter underrun, etc. These errors sometimes occur in bursts. We call *omission degree (Od)* to the number of consecutive omissions produced by a component[9].

Certain kinds of components may temporarily *refrain* from providing service, without that having to be necessarily considered a failure. We say 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. Later in the text, we will find two examples, in communication components: in the LAN, due to token loss or medium failure, and subsequent recovery delay; in the NAC, due to full buffers. This attribute will assist us in the definition of timeliness properties of our service.

## 3.1 NAC Fail-silence Assumption

We saw that combined use of fault tolerance and fault avoidance yields a well-founded *fail-controlled* system. We will impose restrictions on the behavior of the NACs and the Channel, in order that errors do not propagate to the outside of the communication system. Assumptions like "halt-on-failure" or **fail-silence** [12,16], are the most restrictive type: 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 and efficient. So, let us make the working hypothesis that components are *fail-silent*, and that a single component may fail during a *protocol phase*[10].

However, our broadcast channel is equivalent to a single LAN, as detailed ahead. There is, in each node, a single entity which understands what a message is, i.e. no space redundancy at the message level. In consequence: transients in the medium[11] during transmission, whose probability of occurrence is not negligible, are unavoidable; they cause omission errors, and have the same impact as if caused by the NAC; temporary omission errors may occur in bursts, especially in the medium. So, our model will be weakened, in order to take a certain number of consecutive omission errors, or 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

[9]In the same way, we could measure the persistence in time of a timing error, by an *untimeliness degree*, or how late (early) an interaction occurs.

[10]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.

[11]The medium is the passive part (cabling) of the channel.

silencing.

Forgetting for a moment the particular NAC interconnection method, i.e. blaming the NACs for any errors, the overall behavior 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 [15], 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 [12]. Certain errors are allowed, but when the pre-defined behavior criteria are violated, there is an abrupt transition to a permanent failure state, by silencing (Fs1). This decoupling yields a simple conception of the NAC, designed with self-checking duplicated parts. However, it does not require stable storage and the failure of a NAC is detected not through a local readable failed predicate, but indirectly, through a distributed monitor function.

Now, given that we have a network underlying the NACs, the first question, when bringing it into account, is: who to blame when errors occur? The difficulty of the question, which we treat below, lies in succeeding to treat NAC and medium temporary errors in a uniform way, given the broadcasting nature of the channel.

Let us look at the internal structure of a real NAC (in the left of Fig. 1). The CPU runs the software part of the protocols; the MAC contains the LAN specific medium access protocols, it is a frame-level part[12]; the PHY part is the bit-level, electrical signalling part. PHY is the omission error prone part, so we modify the system structure, by moving component boundaries, in order to include PHY in the channel, as shown in the figure. First result: we reduce the NAC to an operative part – the CPU plus the MAC – where protocol machines reside; this part is required to be truly fail-silent[13]. Second result: the remainder of the parts, encapsulated by the thin line, compose the channel; clearly, if we substitute 'channel' for 'NAC' in Fs1 and Fs2, we have the behavior required for the channel. How is it done in our broadcast channel?

### Channel

The broadcast channel is itself formed by several sub-components that we recall: (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 we made in the beginning of sec. 3.1 should now be apparent: the duplication shown in Fig. 1 (dual cables), is concerned with providing high availability of the channel. MAC receives from a single medium at a time, 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 behavior of a channel during a limited number of frame deliveries – a *protocol phase* – will be the following:

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

[13]I.e. to fail nicely by stopping, not even doing omission failures.

- 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 same input[14] (transmitter failures).

The major consequence of this observation of the channel behavior is property Pn3, in table 1, which is the foundation of our technique. In fact, 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. Pn3 is discussed in the next section.

In certain kinds of networks, the channel may become temporarily *inaccessible* (eg. 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:

**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, we stated the behavior we require from the different components of our local computer network, in order that assumption Fs is met. Next, we must discuss how to enforce it: the upper part of the NAC will not be addressed here, it is implemented with duplicated self-checking parts using well-known techniques[22]; we discuss channel implementation in the next section.

The execution view of the system is the one shown in Fig. 2, where the concepts presented in Section 2 appear and one can see how the relevant services map onto the several 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 **AMp** in every node where a gate of the group exists. The AMp 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 (eg. $P_{1,2}$ is in groups $e$ and $f$).

The implementation of these services is described in the following two sections. Detailed comments will focus, whenever needed, on the example of the prototype of AMp we have implemented, over a token-bus ISO 8802/4 LAN, for the DELTA-4 architecture.

# 4 The Abstract Network

The second question about the underlying network is: what functionality can the protocol machine expect from it?

We are going to make our case for the use of local area networks. LANs are a standard means of communication, and components are largely available. Additionally, they have architecture and technology attributes which 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

[14]Successive does mean consecutive: consecutive input failures may be interleaved with other transmissions.

Table 2: Network Properties.

---

**Pn1 - Broadcast**: Destinations receiving an uncorrupted frame transmission, receive the same frame.

**Pn2 - Authentication** : 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 $N$ successive transmissions from $N$ different access points, initiated by any of the $N$ nodes, all transmissions are indicated in all destination access points, in at least one series.

**Pn4 - Full duplex** : Indication, at a destination access point, of reception of the frames transmitted 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 $T_{ina} + T_{td}$.

---

and number of nodes[15]. 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 [12], transfer layers [6] or collapsed application support environments [17]. And certainly, 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.

## Network properties

In the design of AMp, we tried to take advantage of LANs, but the network interface, although LAN oriented, has a general set of properties, being in essence, LAN independent. However, we have limited ourselves to guarantee that those properties can be fulfilled by a set of the standardized LANs, namely 8802/4 Token-bus, 8802/5 Token-ring, ANSI X139 FDDI. In consequence, the *abstract network* service interface formulates, in a way usable by the AMp, a set of helpful properties (table 2), 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)[16]. The properties are defined in terms of a network delivering each frame, transmitted at a source, to all destinations. Pn1, Pn2, Pn3 and Pn6 satisfy the channel model developed in Section 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

[15] Although emerging fiber-optic standards like FDDI, feature up to 1000 nodes, in 100-200 Km, added to an improved *bit error rate* - BER typ $10^{-14}$.

[16] These access points will be used by the AMp machines.

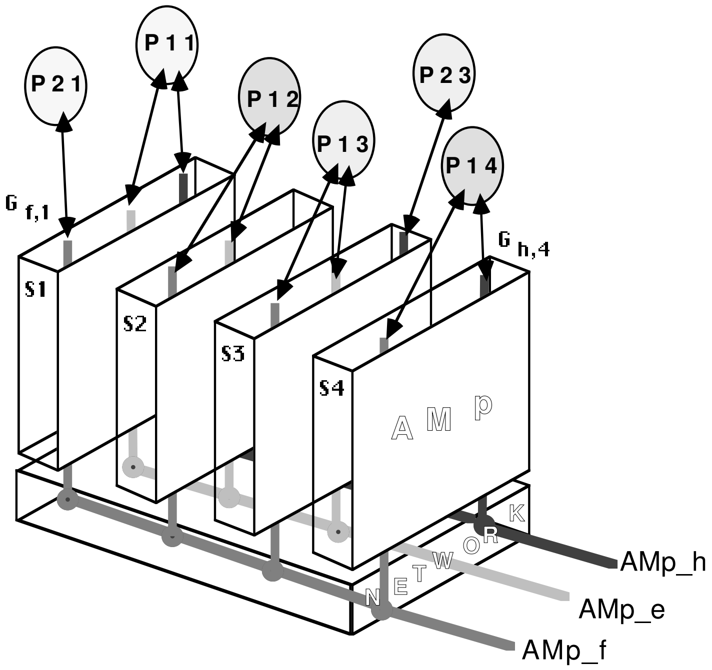


Figure 2: Execution view of the system

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.

Behavior in the time domain is defined by Pn3 and Pn6. $T_{td}$ is the maximum transmission delay in absence of faults, and $T_{ina}$ is the maximum duration of inaccessibility. 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[17]; 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 favorable situation, because they "share" channel errors. Enforcement of the bounded omission degree with acceptable coverage is obtained through redundancy in the physical and "medium" layers. Such an implementation, allowing real-time switch-over between media of a dual cable token-bus LAN, is described in detail in [19].

Pn6 depends on the particular network, its sizing, parameterizing and loading condition: these must be known, in order to calculate $T_{td}$, while 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.

[17] 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.

# 5 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, our approach relies on fail-silent property of the attachment controllers, to centralize protocol execution. The operation of our **two-phase accept** protocol 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. Protocol execution is carried on, in the event of sender failure, by a termination protocol. However, in this case, although Pa1 is always respected, delivery is no longer ensured (see Pa4) for that execution.

The basic protocol is presented in Fig.3; the state machines are locally instantiated for each group, to process relevant transmissions or receptions. The protocol variants we have developed, basically differ in whether mutual exclusion is ensured between executions, or not. The *token-based* protocol, given in [21], requires hardware support. Avoiding hardware modifications, we present, in [18], the *token-less* variant, allowing *several concurrent* message transmissions, for different groups, and *several competitive* message transmissions, for the same group. In any case, a fundamental assumption is always required:

- **A1 - In each group, there is at most one message transmission pending from each node, at any time.**

The protocol execution is followed below, along with discussions on design options and correctness.

### Error recovery or error masking

Protocol progress is straightforward in absence of faults. When faults occur, one may argue about the best way to tolerate them. Reliability of pure diffusion protocols is based on two fundamental assumptions: synchrony and error masking. Thus, network delivery delay must be bounded – this is guaranteed by our network (Pn6) – and there must be redundancy. The n-redundant broadcast channels of [2] or the redundant point-to-point graphs of [7] are an example of space redundancy. Given that we have standard, non-replicated networks, we must use time redundancy. In this case, further to bounding the number of failing components, one should bound the extent (number or duration) of the system errors they produce. These assumptions were made in Section 3.1 and lead to properties Pn3 and Pn6.

Given that the anticipated channel omission error rate is low, instead of masking errors, by systematic $(k+1)$ – $k$ being the allowed omission degree – transmissions, which causes a fixed overhead, we detect them through an acknowledgment mechanism. $TxwResp(Gdest, Psour, data, id; nrResp, nrTries)$, is a transmission- with-response function, sketched in fig. 3b, used by an AMp sender identified by its individual physical address $Psour$, to transmit a frame containing information $data$ with an identifier $id$, to a logical destination identified by group address $Gdest$ (line 52); after transmission, a number of responses $nrResp$ is awaited for, from group members (l. 53); responses are put in a bag $Resp$, and a number of tries $nrTries$ are made, until a full bag is obtained – the bag is emptied before each try – despite the occurrence of errors ($nrTries = k+1$) (l. 51). The bag is returned to the invoker. A simpler transmission function is the $ptpTx(Pdest, Psour, data, id)$, a datagram from sender $Psour$ to a recipient $Pdest$, both individual addresses (eg. l. 25). $RxwResp$, at a recipient (eg. l. 21), is the indication to AMp of an incoming $TxwResp$ reception. Lemma 1 illustrates the properties of $TxwResp$:

*Lemma 1 - If $TxwResp(e, E, m, i; n, k+1)$ is invoked by E, there is at least one try, where a reply to reception of frame m,i is provided by all those among the n intended recipients of logical destination e, which remain correct, if E and the network remain correct.*

The proof, based in Pn3, is left to the reader.

### Group Monitor

Given the clock-less structure of the protocol, its actions rely on a consistent *view* of each group, i.e. the stations where gates of the group are open. For this reason, all actions that are bound to modify this view are performed through a specialized *GroupMonitor* function. Namely, normal joins and leaves, are triggered by calls from the user interface; the sender may raise a suspicion about the state of recipients (*probableRfailure*); a recipient may

---

**TWO-PHASE ACCEPT Protocol**

```
/* Sender E execution: */

status(e) := Ok; i := 0; phase(e) := halted;
...
do phase(e) = halted ∧ status(e)= Ok ∧ send(e,m) → /*dissemination*/
        phase(e) := dissemination;   i := i+1;
        Resp(e) := TxwResp(e,E,m,i; n,k+1)
        if result(Resp(e)) ≠ full → status(e) := probableRfailure fi
        phase(e) := decision
 □ phase(e) = decision → /*decision*/
        if  ∀ r ∈ Resp(e): r = Ok       → d := accept
          □ ∃ r ∈ Resp(e): r = notOk   → d := reject fi
        Resp(e) := TxwResp(e,E,d,i; n,k+1)
        if result(Resp(e)) ≠ full → status(e) := probableRfailure fi
        phase(e) := halted
 □ phase(e) = halted ∧ status(e)= probableRfailure → /*monitor*/
        status(e) := GroupMonitor(status(e);e;Resp(e))
od
 |||
/* Any Recipient R execution: */
status(E) := Ok; state(E) := null
...
do state(E) = null ∧ status(E)= Ok ∧ RxwResp(e,E,m,i_t) → /*reception*/
        state(E) := i;   start(timerWaitDecision(E,i))
        if canAccept(e,m) = true → r(E,i) := Ok
          □ canAccept(e,m) = false → r(E,i) := notOk fi
        ptpTx(E,R,r(E,i),i_t)
 □ state(E) = i ∧ status(E) = Ok ∧ RxwResp (e,E,m,i_t) → /*repetition*/
        start(timerWaitDecision(E,i));   discardAnypending(E,i_{t-1});
        ptpTx(E,R,r(E,i),i_t)
 □ state(E) = i ∧ status(E)= Ok ∧ RxwResp(e,E,d,i_t) → /*decision*/
        stop(timerWaitDecision(E,i));   ptpTx(E,R,ack,i_t);
        if d= accept → deliver m(E,i) to user(e)
          □ d= reject → discardAnypending(E,i_t) fi
        state(E) := null;   status(E) := Ok
 □ state(E) = i ∧ status(E) = probableEfailure → /*monitor*/
    state(E) := null;   status(E) := GroupMonitor(status(E);e,E;i)
 □ anyFrameReceived(E) ∧ status(E) ≠ Ok → /*discard*/
```

```
37                  discardJustreceived
38 od
 |||
/* Timer Actions: */
40 do expired(timerWaitDecision(E,i)) → /*timeout*/
41          status(E) := probableEfailure od
```

Figure 3: 2-phase accept protocol- (a) Main body.

raise that suspicion about a sender ($probableEfailure$) and, in this latter case, it may be necessary to terminate the execution started by a failed sender. There is no permanent monitor activity however – so to speak, a GroupMonitor only exists when needed.

The information required to perform a monitor action on a group is supplied by the recipient or sender machines of that group. Again because the protocol is clockless, the GroupMonitor acts by consensus, established atomically. The invoker of $GroupMonitor(reason; target; information)$ starts by stating the reason, identifying the target of the action (a group or an individual station), and supplying the initial information, when required (eg. identification of failed station, or pending message). Once activated, the action of the monitor is structured recursively in two-phased transmissions, like the normal atomic multicast. This solves the problem of monitor failure recovery, introduced by centralizing monitoring functions: if an active monitor fails, it is replaced by another GroupMonitor, invoked, in the same way as for a normal transmission, by a recipient that detects the failure. The cooperating process between local machines then starts with an investigation phase, where information about the local contexts of group members is gathered, and ends with a decision, disseminated to those members. The decision contains the new group view, after insertion of new members, or elimination of members leaving or having failed. The function then returns $Ok$ to the invoker.

### Message Delivery

The sender, recipient and timer machines of the protocol are sketched in Fig. 3a. Declarations and some obvious initializations are omitted, for space reasons. We consider that group $e$ is formed and has $n$ members; the allowed omission degree is $k$. Delivery of message $m$ to group $e$ is requested (l. 1) by a primitive $send(e, m)$, which triggers execution of the protocol at the node with individual address E. The (/*dissemination*/) action is thus started: sequence number $i$ is appended to $m$, and they are sent to $e$, by a $TxwResp$ (l. 3).

At the recipient, there is initially no context related with sender $E$ ($state(E) = null$). Upon receiving the indication of reception (l. 21), it asserts $state(E)$ to the pending message identification, $i$, and places its response $r$ in a point-to-point response frame addressed to $E$ (l. 25). Prior to that, it has invoked function $canAccept$, which verifies the accessibility of the recipient for the message (ll. 23,24); the response depends on that. Note also that the pair $E, i$ uniquely identifies all recipient variables concerned with a given protocol execution; $i$ is indexed by the sender with the try number (l. 52), so that frames are individually related to a given try. In case of repetitions (l. 26), the response process is the same, but this allows the recipients to suppress duplicates, by eliminating the older frames – $discardAnypending(E, i_{t-1})$ gets rid of all frames of execution $(E, i)$ with indexes up to and including $t-1$ (l. 27).

If no permanent failures occurred, all responses come. The $result$ function, applied by the sender, on $Resp$, decides whether it is $full$ (l. 4). The sender then proceeds to the

**Outline of TxwResp(Gdest,Psour,data,id;nrResp,nrTries)**

```
tries := 0;    Resp := empty
do tries < nrTries ∧ Resp ≠ full →
          Resp := empty;    Tx(Gdest,Psour,data,id_tries);
waitResponsesPutIn(Resp,nrResp);    tries := tries +1   od
return Resp
```

Figure 3: 2-phase accept prot. (b) Outline of *TxwResp*.

(/*decision*/) action, analyzing the content of $Resp$ (lines 7,8) and putting its decision $d$ in a $TxwResp$ frame which it sends to the group (l. 9). The decision is *accept*, if all responded $Ok$. At this time, the atomic transmission is over for the sender (l. 11). The recipients, upon reception of the decision (l. 29), finally deliver the message to the user, or discard any frames related to it (ll. 31,32), according to the decision sent. The reception is then considered ended (l. 33).

Using transmission-with-response has the additional advantages of embedded permanent failure detection, which we discuss below.

**Permanent failures and recovery**

We saw how to process temporary omission errors. How to deal then, with permanent recipient failures? How to deal with sender failure, during protocol execution?

If the omission degree $k$ is exceeded, we have no longer a temporary failure: if the bag $Resp$ is not full after $TxwResp$, there is a probable permanent failure of one or more recipients. This is detected, in both phases, by the *result* function – bag not full (ll. 4,10). AMp accomplishes the service for the remaining correct participants and after the protocol is ended, the (/*monitor*/) action is executed (l. 12). The sender executes the (/*monitor*/) action invoking the GroupMonitor function, for reason= $probableRfailure$, target=$e$, information supplied: the bag. The GroupMonitor restores the coherence of the group view accordingly, and returns $Ok$ (l. 13).

Sender failure, on the other hand, is detected by recipients. Let us go back to the (/*reception*/) action, line 22: a $timerWaitDecision$ is started for this $E, i$ execution, restarted by (/*repetition*/) actions (l. 27), and obviously stopped by the (/*decision*/) action (l. 30). These timers, set to an expected maximum delay to get a decision, perform surveillance of the sender. If the sender fails, the timers expire (/*timeout*/, l. 40), raising a suspicion about the sender's state, by asserting $state(E)$ to $probableEfailure$ (l. 41), prompting for a (/*monitor*/) action that starts as soon as a previous action is over. The recipient terminates the reception and invokes the GroupMonitor (l. 35), supplying the sender identity and the message reference. The GroupMonitor will restore the coherence of the group view, and terminate the pending transmission on behalf of the sender.

## 5.1 Synchronism

One of the questions raised in the beginning of this paper was how to achieve synchronism with a clock-less protocol, given that classical approaches to synchronous protocols are clock-driven [7,2]. Let us explain our approach:

- achieve and determine upper bounds on frame delivery delays by the abstract network, in the presence of load and faults ($T_{td}$ and $T_{ina}$, discussed in sec. 4);
- impose a performance specification on the NAC hardware/software (CPU, kernel, etc.) in order that processing times of the protocol actions are bounded, and known for the worst case traffic pattern specified;
- structure the protocol in phases, so that an execution predictably has a bounded number of phases; clearly delimit phases, in what concerns error detection/recovery (omission and timing), and permanent failure detection (exceeding assumed bounds);
- 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, having thus with a known duration bound.

With these measures, AMp execution can be represented by a chain in time, which is bounded to a known value[18], thus the AMp is **synchronous** in the sense of property Pa6 in table 1.

### AMp Timeliness

In order to quantify the duration of an execution, we have first to observe that it depends very much on the LAN used, and the particular protocol implementation. Let us assume we have the token-based protocol implemented in hardware for the token-bus, as described in [21]. The execution time ($T_e$) expression, taking the possible errors into account, is given in table 3. Error scenarios yield combinations of the $O_i$ variables, thus generating different time chains. Observe that the protocol is in fact structured in transmission -with-response series of the several frame types, $t_{xwResp}(FR)$ – where $FR$ is message $msg$, or decision $DEC$, or investigation $INV$ – alternating with processing steps, $t_{pr}$. Note also that the allowed number of errors, and thus of $t_{xwResp}$ repetitions, is at most equal to the omission degree. The transmission -with-response is itself composed of datagram transmissions and processing steps. Its duration largely depends on LAN type, namely on datagram transmission time, $t_{td}$ (whose upper bound is $T_{td}$). A derivation of $t_{xwResp}$ expressions for real LANs can be found in [20]. One important objective is to determine the best and worst case bounds of $T_e$, for a given implementation. $T_{e_{min}}$ is given by the minimum load operation condition in absence of faults. To determine $\Delta T_e = T_{e_{max}} - T_{e_{min}}$, we then have to exercise each possible error scenario, in order to compute $T_{e_{max}}$[19].

# 6 Conclusions

A reliable broadcast architecture to implement distributed fault tolerance in open systems was proposed. The architecture takes advantage of standard components. Despite this fact, we have shown that with the adequate complements – self-checking components and physical medium redundancy – it is adequate for demanding applications, even with real-time requirements. These are fulfilled because the reliable broadcast protocol, although

[18] And in consequence, $T_i$, once $T_i \leq T_e$.

[19] Note that, given the assumptions we made in Section 3, namely the single failure assumption, only some error combinations are allowed.

Table 3: Temporal expression for AMp execution time ($T_e$), with several error scenarios.

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

clock-less, has known execution time bounds. On the other hand, by weakening the fail-silence assumption with the temporary errors expectable in off-the-shelf components, we are simplifying component implementation. The AMp has been implemented, by several partners of the Delta-4 consortium, in Token-bus and Token-ring. An FDDI implementation is under way. Current work is leading to a multi-fold primitive, with quality of service ranging from atomic to datagram [17]. Support of high performance real-time applications with this architecture is a worthwhile topic, being pursued in the present phase of project DELTA-4. The considerations we made about synchronism properties of clock-less protocols are being applied in this context.

**Acknowledgments**

A large part of this work has been developed in the scope of the Delta-4 architecture, an invaluable vehicle for putting these ideas into testing. The authors wish to thank Dave Powell, who coined the fail-silence expression, by the many criticisms and suggestions made. Luís Rodrigues provided the basic ideas for the token-less protocol and leadered the token-bus AMp prototype work, for whose accomplishment we wish to thank all implementors at INESC. We are indebted to a number of people in the Delta-4 project, particularly to G. Bonn, M. Cherèque, D. Drackley, F. Grandoni, C. Guérin, D. Seaton, J. Voiron. Discussions with Ken Birman have clarified our ideas on rigidity of failure assumptions of participants, in the time domain, and the scale problem of AMp for many senders - few recipients. Flaviu Cristian made a number of useful comments. Comments from the reviewers greatly helped improving the readability of the paper.

# References

[1] J. Arlat, M. Aguera, Y. Crouzet, J. Fabre, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, Special Issue of Experimental C.Sc., February 1990.

[2] Ozalp Babaoğlu and Rogério Drummond. Streets of byzantium: network architectures for fast reliable broadcasts. *IEEE Transactions on Software Engineering*, SE-11(6), June 1985.

[3] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM, Transactions on Computer Systems*, 5(1), February 1987.

[4] Michèle Cart, Jean Ferrie, and Sukrisno Mardyanto. Atomic broadcast protocol, preserving concurrency for an unreliable broadcast network. In J. Cabanel, G. Pujole, and A. Danthine, editors, *Local communication systems: LAN and PBX*, North-Holland, IFIP, 1987.

[5] J. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, 2(3), August 1984.

[6] Greg Chesson. XTP/PE overview. In *13th Local Computer Network Conference*, Minneapolis-USA, October 1988.

[7] F. Cristian, Aghili. H., R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine Agreement. In *Digest of Papers, The 15th International Symposium on Fault-Tolerant Computing*, IEEE, Ann Arbor-USA, June 1985.

[8] Hector Garcia-Molina, Boris Kogan, and Nancy Lynch. Reliable broadcast in networks with nonprogrammable servers. In *8th International Conference on Distributed Computing Systems*, pages 428–437, IEEE, San Jose - USA, June 1988.

[9] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 7(21), July 1978.

[10] P. Melliar -Smith and L. Moser. Fault-tolerant distributed systems based on broadcast communication. In *9th International Conference on Distributed Computing Systems*, pages 129–134, IEEE, Newport Beach - USA, June 1989.

[11] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable group communication in distributed systems. In *8th International Conference on Distributed Computing Systems*, pages 428–437, IEEE, San Jose - USA, June 1988.

[12] D. Powell, D. Seaton, G. Bonn, P. Veríssimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Digest of Papers, The 18th International Symposium on Fault-Tolerant Computing*, IEEE, Tokyo - Japan, June 1988.

[13] J.L. Richier and al. *XESAR: a Tool for Protocol Validation - User Manual.* Laboratoire de Genie Informatique, Grenoble, 1.2 edition, September 1987.

[14] F. B. Schneider. *The state machine approach: a tutorial.* Technical Report 86-800, Cornell U., Ithaca, New York, December 1986.

[15] Fred B. Schneider. Byzantine Generals in action: implementing fail-stop processors. *ACM, Transactions on Computer Systems*, 2(2), 1984.

[16] Santosh Shrivastava, P. Ezhilchelvan, N. Speirs, and C. Nicholls. *Fail-Controlled Processor Architectures for Distributed Systems.* Technical Report, University of Newcastle-Upon-Tyne, Newcastle-Upon-Tyne, England, U.K., December 1989.

[17] P. Veríssimo, editor. *XPA: The Extra Performance Architecture of Delta-4. Design Guide*, Esprit Project Delta-4 G89.129/I1/R, also INESC Technical Rep. RT/54-89, November 1989.

[18] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: a highly parallel atomic multicast protocol. In *SIGCOM'89 Symposium*, ACM, Austin-USA, September 1989.

[19] Paulo Veríssimo. Redundant media mechanisms for dependable communication in token-bus lans. In *13th Local Computer Network Conference*, IEEE, Minneapolis-USA, October 1988.

[20] Paulo Veríssimo and Luís Rodrigues. Reliable multicasting in high-speed lans. In *NATO Advanced Research Workshop on Architecture and Performance issues of high-capacity LANs and MANs*, INRIA, Sophia Antipolis, France, June 1990.

[21] Paulo Veríssimo, Luís Rodrigues, and José Marques. Atomic Multicast Extensions for 802.4 Token-Bus. In *FOC/LAN 87 Conference*, Anaheim-USA, October 1987.

[22] J. Wakerley. *Error Detecting Codes Self-Checking Circuits and Applications.* North-Holland, 1978.

## Appendix: Discussion of protocol correctness

In this section, we outline the demonstration of how the two-phase accept protocol fulfills AMp properties, taking into account that AMp validation is complemented by the methods discussed in Section 1.

*Unanimity*: Given A1, $\exists^1$ $m$ pending from any E; given lemma 1 (L1) it is the same and is everywhere. If there has been a decision, sent by a $TxwResp$, then all have decided on the same message.

*Non-triviality*: Given Fs and L1, any $m$,$E,i$ pending to be delivered in any recipient, is $m,i$ sent by E.

*Accessibility*: If a participant is not correct, its gate is closed, so no message is delivered. We go by contradiction now. If a message is delivered by a recipient $R_j$ to a participant, inaccessible for it, $R_j$ has to have remained correct up to the end of that protocol execution. Given Fs, it must have responded $r_j = notOk$ to the message $TxwResp$, in (/*reception*/) action. Given L1, $r_i \in Resp$, $\forall R_i$. Given Fs, the sender has to make a correct decision in (/*decision*/) action, which is reject, if there is any response *notOk* in *Resp*. Given L1, if $R_j$ remained correct, it must have received that reject decision, sent by a $TxwResp$, and never delivered the message, which is a contradiction with our original assumption.

*Delivery*: We analyze the causes for a message not to be delivered, in the protocol. Observe that both in (/*dissemination*/) and (/*decision*/) actions of the sender, recipients may be considered permanently failed, if $result \neq full$, however, the protocol proceeds. If the sender does not fail, and $canAccept \equiv accessible$, then a message is only undelivered in case of inaccessibility of at least one participant. If the sender fails during a transmission of $m$, the AMp entity which becomes group monitor acquires the state of $m$ in all recipients, using $TxwResp$: if any participant accepted, then the decision to all (using $TxwResp$) is *accept*, otherwise, it is $reject$, regardless of whether all recipients are accessible.

*Order*: We just discuss order consistency, the interested reader may find the proof of causality in [18]. Messages to be delivered, are delivered in the order they arrive from the abstract network, in the sense of Pn5. Given L1 and namely, $TxwResp$ and the recipient (/*repetition*/) action in Fig. 3, transmissions where omission errors occur are discarded, so a message that remains in a queue, exists in all queues, and comes from a single transmission. Given this fact and Pn5, any messages from competing senders in the network have to exist in the same order in all participants' queues. Given Pa1, any two messages delivered anywhere, are delivered to all, and in consequence, in the same order.

*Synchronism* is discussed in the appropriate section.

*Group View*: Since the receive order is consistent, the $GroupMonitor$ ($GM$) controls every group reconfiguration, and the GM recursively uses atomic transmissions to disseminate changes, then they will be ordered consistently with the remainder of the information flow.