Reliable Multicasting in
High-speed LANS
INESC Technical Report RT/___-90
P. Veríssimo, Luís Rodrigues
May 1990

Presented at the NATO Advanced Research Workshop on Architecture and Performance issues of high-capacity LANs and MANs, France, June 1990

Reliable Multicasting in High-speed LANs

Paulo Veríssimo, Luís Rodrigues[1]

INESC - Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6° - 1000 Lisboa - Portugal, Tel.+351-1-545150.

Abstract: There is an increasing number of distributed applications, some of them fault-tolerant, and it has been recognized that its construction may benefit from the existence of reliable broadcast protocols. Reliable broadcasting has deserved considerable attention recently. Some systems are clock-driven, exhibiting tight synchrony: they rely on clock synchronization and space redundancy. Others, like the *AMp*, an atomic multicast protocol for local area networks, are clock-less.
In application-independent systems, most of the time domain requirements are not of the *hard real-time* kind – which clock-driven protocols are most suited for – but rather of the *on-line*, or *soft real-time* kind. To encourage utilization of reliable broadcast protocols in such applications, it is mandatory that benefits in quality of service are not considered too costly in performance. Clock-less protocols provide an answer to this requirement, since they can trade synchronism for fast termination.
This paper deals with the performance implications of supporting distributed applications, with clock-less reliable broadcast protocols. The question of the network influence in protocol performance is analyzed. We predict the performance of AMp on two target LANs: the 10Mb/s token-bus, and the 100 Mb/s FDDI. We observe that the figures achieved with a LAN based approach are good enough to match the requirements of high performance distributed kernels. Additionally, we show that, under these conditions, using a high-speed LAN does present an advantage: not only throughput increases, a natural consequence, but also *speed*, measured in duration of single AMp executions, for small messages.

Keywords: Communication, Distributed system, Fault-tolerance, Reliable broadcast.

1 Introduction

There is an increasing number of distributed applications, some of them fault-tolerant, and it has been recognized that its construction may benefit from the existence of reliable broadcast (RB) protocols. These protocols provide a service which has a set of order, agreement and synchronism properties, in the presence of disturbing factors (load and faults). In consequence, the user is alleviated from the task of ensuring them for each application.

[1]This work has been supported in part by the CEC, through Esprit Project 1226 - DELTA-4, and by JNICT, through Project 87634 - ESTIMULO. This paper contains some material previously published in the proceedings of the ACM SIG-COMM'89 Conference, Austin-USA, September 1989.

Reliable broadcasting has deserved considerable attention recently. Some works, exhibiting tight synchrony, are clock-driven: they rely on clock synchronization and space redundancy [2,10]. The *AMp*, an atomic multicast protocol for local area networks, is a loosely-synchronous protocol which does not use clocks. It relies on network properties, to enforce timeliness. It compares with other clock-less approaches [5,7,8], although we have worked in bounding termination times to suitable limits, which make it possible to define the situations for which the communication system is capable of representing real-time interactions, both from the point of view of meeting communication delay bounds and respecting temporal precedence of events.

This paper deals with the performance implications of supporting distributed applications, with reliable broadcast protocols. In application-independent systems, most of the time domain requirements are not of the *hard real-time* kind – which clock-driven protocols are most suited for – but more of the *on-line*, or *soft real-time* kind. That is, rather than requiring an exact meeting of deadlines and a high degree of simultaneity, for actions or messages, applications require responsivity, 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).

The service properties and the operation of *AMp* are detailed in [16]: the protocol offers a service featuring strong unanimous agreement and consistent and causal order properties. From the point of view of fault-tolerance and distributed computing, they are useful to implement distributed synchronization and replication control algorithms. The AMp is envisaged to evolve to a multi-fold service, featuring weaker properties (eg. FIFO order, datagrams), with the corresponding improvement in performance. Given that the quality of service we are analyzing is the highest possible in terms of agreement and order, we believe the results we present give some insight into the expectable performance of the multi-fold service.

Performance-wise, there are three questions in the design of reliable broadcast protocols, which influence the final result: (i) which fault model; (ii) what level in communications stack; (iii) which network.

The fail-silent assumption

Given the cost associated to distributed agreement approaches [13], that consider arbitrary behaviour of components, we assume the communication system fails in a controlled manner, exhibiting what is called the *fail-silent* property [14]: the network adapters where AMp runs are thus confined to always deliver correct messages, tolerate a bounded number of temporary errors – such as transmission errors – and halt after their first failure. This already represents a performance head-start, because it simplifies protocol design. However, the main improvement of the protocol, performance-wise, is to take advantage of LANs.

The low-level approach

The protocol was designed both to run on top of LANs and not to depend on a particular LAN. The architecture is built on standard LANs, in view of taking advantage of the availability of communications hardware and of the possibility of coexistence with standard stations, in the same network. Additionally, LANs have architecture and technology attributes which can be used for improved performance and dependability.

The data link layer seemed to be the adequate level to engineer the AMp. This low-level approach is justified by several reasons: for one, tradeoffs regarding performance are best made, using LAN facilities; for another, it is low enough to allow several options for upper layers: OSI-like multipoint stacks [14], transfer layers (eg. XTP) [9] or high-performance distributed application support environments [3].

So the AMp offers a data link level interface to the user. In order to make the protocol design LAN independent, and thus very portable, it interfaces an "abstract local area network", whose properties are guaranteed by a harmonizing driver built on top of the exposed MAC[2] interface of the particular VLSI LAN controller.

High-speed LANs?

In a sense, the *abstract network* extends the concept of LLC[3], the LAN independent sub-layer of the IEEE, and later ISO, 802 standard: besides offering a message (service data unit) delivery service, it makes visible a set of functional properties which are common to LANs in general. These are used to optimize protocol design.

Now that we have a LAN independent interface, we start by analyzing the suitability of the AMp for LANs in general, by making some predictions about its execution time, on a target LAN: a 10Mb/s ISO 8802/4 token-bus. Secondly, we analyze whether AMp performance will benefit from migrating to a higher throughput LAN, such as the 100 Mb/s ANS X139 FDDI.

We show that not only AMp throughput increases, a natural consequence, but also *speed*, measured in duration of single AMp executions, for small messages. This fact is of outstanding importance, since it has been recognized that communication speed is the dominating requirement for distributed computing [6,9,11]. 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.

Prior to presenting our predictions for AMp, on token-bus and on FDDI, we introduce the protocol operation details which are relevant to our analysis.

---

[2] Medium Access Control sub-layer.

[3] Logical Link Control sub-layer.

2 Reliable Multicasting with the AMp

The **AMp** relies on the properties of an underlying *abstract network*. The set of properties defined are: *Unreliable Broadcast* and *Authentication* — by a frame check sequence — ensuring that although frames may be lost, the destinations that receive a frame, receive the one that was transmitted. *Bounded Omission Degree*[4] specifies the number of successive transmission errors of a correct network to be lower than a known value k. *Bounded Transmission Delay* ensures that any frame queued for transmission is sent within a known delay T_{td}, in absence of faults. *Network Order* guarantees that any two frames received at any two sites, are received in the same order. *Full Duplex* implies that the sender itself is also included in this ordering property as a recipient.

The AMp provides highly parallel reliable group communication, which takes place inside groups of participant entities. The AMp provides a service of reliable group communication which ensures that when a message is delivered, it is delivered to all correct participants (*unanimity*), in the same precedence order to all of them (*order*) and within a known bounded time (*termination*). The message delivered is the one transmitted by the sender and it is always delivered, unless some participant(s) is(are) inaccessible (*validity*). Inaccessibility will not be discussed here. The reader may find details about it in [16]. It is a temporary state whereby a recipient may refuse, in a given execution, to accept a message, causing the protocol to terminate timely with a negative confirmation. We are concerned in this paper with the cases where a message is delivered. The protocol is resilient to omission errors and stopping faults during its execution. A bounded omission degree is tolerated, but any number of nodes may fail, in a single phase of the protocol[5]. In case of sender failure, a termination protocol is ran by a monitor function, to ensure completion of the transmission in course. Node failures and group membership changes are indicated to all participants, consistently ordered relatively to the information messages. Any group member may initiate a transmission and the sender also receives the message transmitted. Multicasting is *transparent*, in the sense that only one message is sent and there is no need for a previous knowledge by the user, of the whereabouts or number of destinations. In fact, the destination address is location independent and related to a unique designation each group possesses in the system.

Groups

Each **gate**, the entity used by a participant to communicate, uses an instantiation of the AMp machine, comprising the $EmitterMachine$ and the $ReceiverMachine$, 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. Error detection is done on a transfer-by-transfer basis, and relies on consistency of the group views

[4]Omission degree (Od) is the number of successive omission errors that a correct component does. A component exceeding its specified Od is considered failed and must shut-down.

[5]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. The organization of AMp in phases is important for the enforcement of timeliness properties.

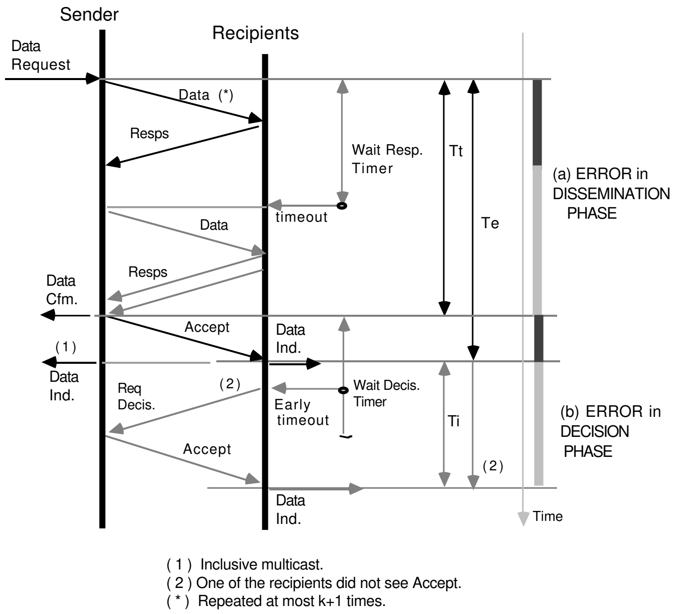

Figure 1: Timing diagram of AMp: In grey, in both phases, a transient omission error: a) Dissemination; b) Decision.

of all members. We proceed by describing some assumptions that support protocol operation; then we present the description of the protocol operation itself.

Assumptions

Assumptions 1.i are the foundation of the correctness of operation of our two-phase-accept protocol implementation.

Assumptions

1.1 there may be several competitive transmissions in course, in the same group.

1.2 there may be several concurrent transmissions in course in the network, from different groups.

1.3 at any time, there may be at most one transmission in course, per group, per node.

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

An EmitterMachine, once started, executes atomically, i.e. it is not preempted by other sending actions in the group, for example, from the ActiveMonitor.

The decision frame is not acknowledged, for *accept*, in the interest of improved performance. Safety of this method is based in a simple algorithm:

- assumption 1.4 above;
- all participants log the sequence number of the last message delivered ($lastDeliv$ for sender) or accepted ($lastAccept$ for recipients);

- omission errors in decision 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.

The protocol

Our atomic multicasting service relies on a **two-phase accept** protocol, sketched in fig.1, which resembles a commit protocol, only in that the *sender* coordinates the operation: 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, delivery is no longer ensured. Should the message be delivered, unanimity is nevertheless fulfilled. The phases are implemented with *transmission- with- response rounds*, and end after reception of all expected responses. In worst case, it may take as many rounds as the allowed omission degree plus one *(k+1)*.

An atomic multicast transmission is initiated by the protocol coordinator, the sender (E), by sending a multicast frame containing the message. The *Dissemination* phase (fig. 1) 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 analyzed 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.

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

- The *reject* frames always require response (assumption 1.4). A station that does not answer within the retry limit, is considered failed.
- The *accept* frame, on the other hand, does not require response. A timeout 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 to the sender[6] (fig. 1).

[6] The reader will notice, in fig. 1, that the first decision request is triggered by an *early timeout*. In fact, the waitDecision timer is a two-shot timer. Please refer to the end of sec. 3.1, for details.

- Note that in case of reject, a sender only starts a new transmission after assuring that all the group members received the reject (assumption 1.4). 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, we chose to make it negatively acknowledged, which optimizes transmission rate, due to the pipelining effect, in absence of faults. It allows to decrease the transmission cycle time, once 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 1.3 and 1.4 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.

Group Monitor

There is a *Group Monitor function*, which 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 joins and leaves from a Group, so that all GroupViews change consistently.

The distributed Group Monitor function relies on information provided by the several 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. At most one monitor is active at a time, in each group, and if it fails, it is replaced by another GroupMonitor who detects the failure. The procedure is recursive.

The monitor executes in two phases (StepOne and StepTwo). The first phase (StepOne) includes the identification of failed stations and search for the presence of pending messages from failed senders. It is an investigation process, responded by the local monitor entities. After this, a second phase (StepTwo) is performed, including the decision to *accept* or *reject* those messages and finally the dissemination of the new group view.

	All cases
T_e	$O_1.(t_{acc} + t_{waitResponse}(n-1))+$ $\overline{O}_3[\ \overline{O}_2.(t_{acc} + t_{xwr}(FR) + t_{prBg}(n))+$ $O_2.(t_{acc} + t_{waitResponse}(n-1) + t_{prBg}(n))\ ]+$ $O_3[\ t_{acc} + t_{xwr}(FR) + \overline{O}_6.t_{earlyDecision} + O_6.t_{waitDecision} + t_{acc}+$ $O_6.O_4.(t_{waitResponse}(1) + t_{acc})+$ $\overline{O}_5.(t_{xwr}(rqDEC) + t_{prRx}) + O_6.O_5.(t_{waitResponse}(1) + t_{AM})\ ]+$ $\overline{O}_5.(t_{acc} + t_{DEC} + t_{prRx})$
	Active Monitor action
t_{AM}	$O_7.(t_{acc} + t_{waitResponse}(n-1)) + t_{acc} + t_{xwr}(Step1) + t_{prBg}(n)+$ $O_9.(t_{acc} + t_{xwr}(Step1) + t_{prBg}(n))+$ $O_8.(t_{acc} + t_{waitResponse}(n-1)) + t_{acc} + t_{xwr}(Step2) + t_{prRx}$
	Variables for error situations
O_1	Number of omission errors in Dissemination.
O_2	Recipient(s) failure: true - O_2=1; false - O_2=0.
O_3	Decision errors: true - O_3=1; false - O_3=0.
O_4	Number of omissions in decision request.
O_5	Sender failure: true - O_5=1; false - O_5=0.
O_6	Omissions in 1st dec. req.: true - O_6=1; false - O_6=0.
O_7	Number of omissions in Step1.
O_8	Number of omissions in Step2.
O_9	Group has changed: true - O_9=1; false - O_9=0.

Table 1: Temporal expression for AMp execution time, T_e, with several error scenarios.

3 Performance issues

Let us discuss now time-related properties of AMp. We will base our analysis of AMp performance in its phases of *transmission with-response* ($TxwResp$) rounds. Absolute duration of an execution depends very much on the LAN used, and the particular protocol implementation. Let us define:

- *Execution Time* (T_e): The time between the send request primitive and the issuing of the last receive indication for that message.

For the time domain analysis which forms this section, we start by deriving some generic LAN variables. Then we define expressions for the execution time, T_e, in several situations. We define equally a scenario which will be the target of our observations.

Bounded execution times are only possible if all the terms in T_e are bounded. Two critical terms here are LAN and user related, respectively: access delay and queue delay. They depend on the individual and global offered load. In LANs, they can only be controlled if all nodes speak the same language, i.e. if they can cooperate in some form to control the offered load. In cyclic hard real-time systems, where the delay of urgent messages must be bounded with accuracy, we believe it is preferable to tune operation in order that queue delay is zero – at least for high priority/urgency services – when running on single LANs. We are looking at systems with mixed cyclic and bursty traffic, so we consider average values, but we will use this design principle, which requires, very simply, that the user cycle is lengthier than the network cycle. The latter is given by access delay, which is influenced by user load, indirectly, through a measure of the average channel utilization, ρ. As for access delay, both LANs tested, token-bus and FDDI, have access control mechanisms based on the timed-token principle [12,1]. Given the minimum token rotation time for these LANs, R_{mn}, the *average access delay*, t_{acc}, is half the average token rotation time, which, in turn, is inversely proportional to channel load, ρ:

$$t_{acc} = R_{av}/2 = \frac{R_{mn}}{2(1-\rho)} \tag{1}$$

Let us assume we have the token-less LAN-independent protocol, as described in [16]. The execution time (T_e) expression, taking the possible errors into account, is given in table 1. Let us define the general scenario, in an industrial environment, for example, a small cell network for real-time manufacturing control:

SCENARIO:

- **configuration:** channel length $C_l = 500m$ and a total number of stations $N_{st} = 32$; network omission degree is $Od \leq 2$; channel propagation delay then comes $t_{PD} = C_l.5ns/m = 2.5\mu\ s$[7];
- **protocol parameters:** message length, l_{IF}, ranges from 80 to 1280 octets; groups of $n = 6$ and $n = 12$ participants are analyzed; protocol frame length is $l_{PF} = 40$ octets (RF- response frame; DEC- decision; rqDEC- request decision);

3.1 The AMp on Token-bus

Let us clarify the meaning of the variables at stake, starting by the AMp on token-bus.

ENVIRONMENT 1:

- **network operation:** channel rate is $C_r = 10Mb/s$, station delay is $t_{SD} = 2\mu s$; we consider a total background offered load of $\rho_{BK} = 10\%$, in the network; we are now able to derive the following time variables:
  - token duration: $t_{TK} = l_{TK}.8/C_r = 18\mu s$
  - minimum token rotation time in token-bus:

$$\begin{aligned} R_{mn} &= N_{st}.(t_{PD} + t_{SD} + t_{TK}) \qquad (2) \\ &= 707\mu s \qquad (3) \end{aligned}$$

[7]Assuming a wave propagation velocity of $0.2m/ns$.

T_e	80 octets		n = 6	
	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 2: Execution time, T_e (ms), for several message lengths and number of group participants, in the presence of several errors (AMp on TB).

- average access time ($\rho_{BK} = 10\%$)(eqs 1 e 2): $t_{acc} = 393\mu s$
- information frame duration: $t_{FR} = l_{IF}.8/C_r(minimum : 64\mu s)$
- protocol frame duration: $t_{RF} = t_{DEC} = t_{rqDEC} = 32\mu s$

Given that we expect to take advantage of the token rotation, to cycle our transmissions with-response, we redefine *token rotation time subsequent to a transmission*, $t_{RT}(i, \rho)$, as the time needed for the token to rotate, carrying i responses, over a background load of ρ:

$$t_{RT}(i, \rho) = \frac{R_{mn}}{1 - \rho} + i.t_{RF} \tag{4}$$

We define *processing time variables*, which account for the CPU time to execute protocol processing steps. We will consider two values: t_{prRx}, for processing of reception of generic protocol frames, and for processing of reception of information frames and generating the response; $t_{prBg}(n)$, for processing of a bag of n responses and preparing the decision. Processing times are estimated by protocol analysis, counting the number of 'C' instructions in the sequential path of the relevant processing steps[8], obtaining: $t_{prRx} = 0.7 \times 250 = 175\mu s$, and for $n = 6$, $t_{prBg} = 0.7 \times 420 = 294\mu s$. These values are estimates for an optimized implementation. They may be validated with adequate protocol measurement tools. The duration of a transmission of frame FR with response, is then:

$$t_{xwr}(FR) = t_{FR} + t_{prRx} + t_{RT}(n - 1, \rho) \tag{5}$$

In order to mutually control activity, we had seen in the last section that both sender and recipient use timers. Their values are dependent on the network operation conditions presented below, and are computed by the communications management entity. The sender starts timer $TwaitResponse$, after receiving confirmation of transmission by the network:

$$t_{waitResponse}(i) = (1 + v_P).t_{prRx} + t_{RT}(i, \rho_{BK} + v_L) \tag{6}$$

[8] We are assuming a protocol processor capable of executing one 'C' instruction per μs, and an optimized compiler with 30% efficiency, whence $1 inst. = 0.7\mu s$.

T_e	80 octets		n = 6	
	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

Table 3: Execution time, T_e (ms), for several message lengths and number of group participants, in the presence of several errors (AMp on FDDI).

The number of responses expected to come from the network is $i = n - 1$ (n: number of group members), whereas variables v_P e v_L account for, respectively, CPU and LAN load variability, which we arbitrate to be $v_P = v_L = 30\%$. It seems to be a reasonable assumption: given $\rho_{BK} = 10\%$, we are accommodating average LAN loads between $10 - 40\%$; for the CPU, we are considering average instruction execution times from $0.7 - 1\mu s$. Since the timer value depends on group dimension, each group has a timer. Each recipient starts timer $TwaitDecision$, to control sender activity. This timer also depends on group dimension, n, so, it also belongs to each group. Besides, its first purpose, in absence of permanent failures, is to detect and recover from omissions in the (non-acknowledged) *decision*; in consequence, it is a two-shot timer, with a first timeout given by $t_{earlyDecision}$:

$$t_{earlyDecision} = 0.5 \times t_{RT}(n, \rho_{BK}) + t_{prBg}(n) + \frac{R_{mn}}{2(1-\rho_{BK})} + t_{DEC}$$

$$t_{waitDecision} = t_{RT}(n-2, \rho_{BK} + v_L) + (1 + v_P).t_{prBg}(n) + \frac{R_{mn}}{2.(1-\rho_{BK} + v_L)} + t_{DEC}$$

The rationale for dimensioning of these timers is that, after confirmation, from the network, of transmission of the response, the recipient starts the timer ($t_{earlyDecision}$), dimensioned for the average case: the recipient is the $\frac{n}{2}^{th}$ to reply in the t_{RT} process; after the token rotates, it allows an optimistic processing time for the bag of responses followed by transmission of the decision, after waiting the access time[9]. The first signal of the timer occurs after this interval, if a decision did not come. It is used to send a $rqDEC$ datagram (see tab. 1), continuing until the final value of $t_{waitDecision}$, calculated by the expression above. Then, if necessary, a transmission-with-response $rqDEC$, is sent, with a timer value of $t_{waitResponse}(1)$. The timer values, for this environment, are: $t_{waitResponse}(5) = 1566\mu s$, $t_{waitResponse}(1) = 1438\mu s$, $t_{earlyDecision} = 1191\mu s$, and $t_{waitDecision} = 2310\ \mu s$.

Expressions for duration of the AMp, in the presence of several errors, are given in table 1. We extracted some example situations, as a function of message length and number of group participants, and present the relevant values for T_e, in table 2.

[9]Optimistic, because no variabilities are taken into account.

T_{dlv} [μs]	$\rho(\%)$	t_{acc}	80 oct.	320 oct.	1280 oct.
TB	10	392	460	650	1420
	40	590	653	845	1610
FDDI	1	11	18	37	114
	4	11	18	37	114

Table 4: Datagram service time (μs), in function of several lengths and loads, for *token-bus* and FDDI, including access time, for loads of 10% e 40% (weighted by 1:10 in FDDI).

3.2 The AMp on FDDI

The use of FDDI, with a $100Mb/s$ rate, seems to be advantageous for complex protocols:

- given the increased available bandwidth, which reduces the impact of protocol frames in channel utilization;
- given the increased speed, because of the shorter rotation and transmission times, for the same load condition.

With the purpose of comparison with the *token-bus* LAN, we redefine environment 1, in the viewpoint of the individual user. We are not observing the performance of a particular LAN, but that of a protocol using LANs. Besides, we are concerned with the analysis, proper of real-time services, of the service provided to the individual user, despite some background load. So, we scale down the relative background load, ρ_{BK}, by a ratio of 10:1, to maintain the same absolute load (in octets), and maintain message lengths, decreasing frame duration.

ENVIRONMENT 2:

The configuration of the network is identical to that of environment 1. Protocol parameters are the same. The channel rate becomes $C_r = 100Mb/s$ and the station delay, $t_{SD} = 0.6\mu s$; the background load becomes, as explained, $\rho_{BK} = 1\%$. Once more, we compute the following network operation variables:

- token duration: $t_{TK} = 0.88\mu s$
- we recompute R_{mn} for FDDI: $R_{mn} = t_{PD} + N_{st}.t_{SD} + t_{TK} = 22.6\mu s$
- average access time ($\rho_{BK} = 1\%$)(eqs 1 above): $t_{acc} = 11\mu s$
- minimum information frame duration: $t_{FR} = 6.4\mu s$
- protocol frame duration: $t_{RF} = t_{DEC} = t_{rqDEC} = 3.2\mu s$

Processing times are AMp code dependent, so they remain unchanged. Variabilities remain at $v_P = v_L = 30\%$. Timers are based in the same expressions, but they do change: $t_{waitResponse}(5) = 281\mu s$, $t_{waitResponse}(1) = 268\mu s$, $t_{earlyDecision} = 330\mu s$, e $t_{waitDecision} = 454\mu s$. We repeat, in table 3, the predictions for T_e in the same situations as done for token-bus, in table 2.

In the graphics we present in the end, the differences become clearer. Firstly, they reveal the main points of dependence of AMp execution time:

- message length, l_{FR};
- group dimension, n;
- LAN load, ρ;
- LAN dimension, R_{mn} (function of C_l and N_{st})

Secondly, they show the impact of running AMp on FDDI, rather than token-bus, for the same conditions.

4 Conclusions

We have analyzed a protocol which provides atomic multicasting at the Data Link layer. The interest of this approach, if compared with other existing reliable broadcast works, is that it takes advantage from the inherent facilities of the underlying Lans, from which we name broadcast capability, multicast addressing, token-based access control, priority scheduling, low-level synchronization. Either based on a software shell on top of the exposed interface of VLSI controllers [16], or based on hardware support [17], the aim of this kind of approach is efficiency at the low cost of a non-replicated standard Lan, for distributed and fault-tolerant applications. The communication primitive provided was proven to be of great help in assisting the implementation of distributed computations. In a report, we describe the programming of the Amaze game [4] as a replicated and distributed state machine. The state machine replicates directly use the AMp, with no further "glue", for their interactions. In the Delta-4[10] system [14], AMp is used as the low-level primitive in an OSI-like stack, which has fault tolerance mechanisms based on replication and voting, implemented at the Session layer; the resulting communication architecture provides support for a dependable computation system, featuring several fault tolerance tools.

As we could see in table 2, the net performance of AMp on a moderate throughput LAN (10Mb/s token-bus) seems to be very satisfying, since execution time is of the order of 3ms for short messages. If compared with other "system-level" services in most systems, eg. point-to-point null RPCs, it is of the same order of magnitude, if not faster, but with a significantly higher quality of service.

Migrating to FDDI yields an impressive improvement. In a first analysis, we can observe a neat performance improvement, not only in the large frames (throughput), but also in the small frames (speed). Some reasons for this are clear in table 4: delivery

[10] 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).

time includes transmission time and access time: both are far more favourable in FDDI. Particularly, the access time is a significant amount of the overall delivery time, in any of them. We saw that t_{acc} depends on token rotation time (eq. 1), which is shorter in the FDDI token-ring, because the token is released right after frame transmission and, of course, tokens are 10 times shorter in FDDI than in token-bus. So, this is one of the reasons for speed improvement, and it remains valid for all qualities of service, down to the datagram. The second, as we saw, has to do with reduced transmission times overall, both for information and protocol frames, and with its consequence of scaling down the relative network load, for the same amount of absolute offered load. The more complex the protocol is, the greater the improvement, and this is particularly true of the described atomic service.

Indeed, this quality of service is often too high for a diversity of requirements. Due to atomicity and active participation of all group elements in communication, the protocol does not scale well with group dimension. This will be solved in a future version, with development of a multi-fold service, accommodating different qualities of service [15]. Then, concerning reduced quality of service, the reader will note, from table 4, that a multicast datagram execution time in FDDI is, in average, $30\mu s$, for short messages. From equation 5, we obtain $230\mu s$ for a transmission-with- response with the same average length. This value is a good approximation for an AMp variant where consistent order is relaxed to FIFO, keeping the other properties, namely unanimity. If we approximate *cycle time*[11] to execution time, this would mean 4000 short reliable broadcasts per second. Token-bus estimates for the same situation, although still very good, are much more modest: $600\mu s$ for a datagram service, and $1.4ms$ for a transmission-with-response, yielding around 700 reliable broadcasts per second.

These predictions make us feel optimistic about the performance of communications intensive applications in high-speed LANs. They also seem to confirm the suitability of FDDI as a front-end network in high performance distributed computing, not only in scientific or graphic applications, but also in other areas, such as continuous and discrete distributed process control, control and monitoring of physical experiments.

[11]I.e., the maximum frequency of protocol executions by a sender.

References

[1] *FDDI Token-Ring Media Access Control (MAC).* ANS X3.139, 1987.

[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] P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, D. Seaton, N. Speirs, and P. Veríssimo. The Delta-4 Extra performance architecture (xpa). In *Digest of Papers, The 20th International Symposium on Fault-Tolerant Computing*, IEEE, Newcastle-UK, June 1990.

[4] E.J. Berglund and D. Cheriton. A distributed multi-player game program using the distributed V-Kernel. *IEEE Software*, (2), May 1985.

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

[6] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[7] 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.

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

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

[10] 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.

[11] Norman C. Hutchinson and Larry P. Peterson. Design of the x-Kernel. In *SIGCOMM'88: Communications Architectures and Protocols*, ACM, Stanford, USA, August 1988.

[12] ISO. *ISO DIS 8802/4-85, Token Passing Bus Access Method.* 1985.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Prog. Lang. and Systems*, 4(3), July 1982.

[14] 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.

[15] 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.

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

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

Figure 2: AMp execution times, as a function of message length, l_{FR}; group dimension, n; LAN load, ρ; LAN dimension, R_{mn}. Each graphic compares Token-bus (TB) with FDDI.