A Low-Level Processor Group Membership Protocol for LANS*

L. Rodrigues
ler@inesc.pt

P. Veríssimo
paulov@inesc.pt

J. Rufino
ruf@inesc.pt

Technical University of Lisboa
IST - INESC †

Abstract

This paper presents a processor group membership protocol designed to run on top of a local area network. The protocol maintains information about a selected group of stations that explicitly join the protocol by keeping a replica of a global membership table at every member. Additionally, the protocol guarantees that a given station always occupies the same entry in the table. As a result, table indexes do uniquely and universally identify a station and can thus be used as short identifiers. The interest of a processor group membership is twofold: it is a powerful auxiliary for *process* group membership management and it provides support for efficient message addressing.

Keywords: *Distributed Systems, Distributed Algorithms, Fault-Tolerance, Communication Protocols, Real-Time.*

1 Introduction

Distributed systems may take advantage of the local availability of up to date information about the nodes in the system. This information is not static: during the lifetime of the system, stations will join, leave and, possibly, fail. A protocol able to dynamically maintain information about the state of a given set of distributed entities is usually called a group membership protocol.

This paper presents a group membership protocol designed for small scale fault-tolerant distributed systems built on top of a local area network. For efficiency, the protocol is implemented in the lower layers of the communication system. More precisely, the protocol is implemented on top of an exposed MAC interface of standard VLSI LAN controllers. The advantages of this approach are twofold: the properties of the architecture are exploited to achieve improved performance and the services provided by the protocol can be used by upper layers.

A particularly useful application of a low-level group membership protocol is the possibility to dynamically assign short-addresses to each node of the system. A short-address is a compact station identifier, built on the assumption that networks always have a number of nodes much lower than that allowed by the address space of standard 48-bit long LAN addresses. This can be exploited to provide an optimized support for message addressing in the distributed system. We are specially concerned with fault-tolerant architectures, implementing replication techniques that are highly demanding on interchange of information in multicast mode. A multicast message is addressed to a set of components, usually resident on different nodes of the system. Thus, a multicast address can be translated into a set of station identifiers. However, for efficiency, at the LAN level multicast addresses should be highly compressed and, if possible, recognized by VLSI controllers. We analyze several alternatives to implement multicast addresses at the LAN level and we conclude that the association of a short-address to each station participating in the multicast traffic can improve the efficiency of the communication system.

Our group membership protocol has then two major goals:

- it keeps a complete, and updated, list of a selected group of stations, participating in the multicast

*A version of this paper has been published in the Proceedings of the 13th International Conference on Distributed Computing System, Pittsburgh, Pennsylvania, May 25-28, 1993, 0-8186-3770-6/93 © 1993 IEEE

†Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6º - 1000 Lisboa - Portugal, Tel.+351-1-3100281. This work has been supported in part by the CEC, through Esprit Project 1226 - DELTA-4, and JNICT, through Programa Ciência.

traffic (target systems typically include up to 64 nodes). This group is called the **M**ulticast **G**roup of **S**tations or simply MGS. The MGS protocol assures that the membership view is updated consistently in the presence of joins, leaves and failures. Changes in MGS membership are indicated to the protocol users.

- it implements a mapping function that translates unique node identifiers into short-addresses. To enable run-time reconfiguration, the mapping is not statically pre-defined and new stations are able to, at any time, obtain a short-address. This mapping is *universal* and *stable*, i.e., in all stations, the same short-address corresponds to the same station and that correspondence remains unchanged during the lifetime of the system[1].

The paper is organized as follows. Section 2 introduces our design goals. In Section 3 we describe the MGS group membership protocol. Some case studies and numerical results appear in Section 4. Comparison with related work appears in section 5 and final concluding remarks are provided in section 6.

2 Design Goals

2.1 Target architecture

Our work was developed in the scope of the DELTA-4 [8] architecture. The DELTA-4 architecture aims at providing a solution to problems requiring varying degrees of distribution, fault-tolerance and real-time [1]. It does so by resorting to techniques based on "macroscopic" replication of components (processes, files, objects, ...) and its distribution over a network. These services are implemented using a versatile multi-primitive reliable multicast service, the xAMp [10], designed to support distributed applications with different dependability, functionality, and performance requirements. A major feature of xAMp is that all communication primitives offer *multicast* addressing.

Multicast communication is extensively used in DELTA-4. Thus, the efficiency of multicast addressing is of major significance for the overall performance of the architecture. The MGS group membership protocol was designed to efficiently support xAMp's addressing scheme. In the communication stack, the MGS is integrated in the network infrastructure, immediately on top of the MAC interface of the local area network, running in parallel with the xAMp layer.

2.2 Efficient Multicast Addressing

Existing LAN VLSI controllers usually provide hardware support for the use of multicast addresses. However, the number of addresses that can be processed by the VLSI is usually small and far less than what is normally required in group-oriented systems. Additionally, these multicast addresses must be configured in all concerned stations before any multicast transmission may take place. In many applications the selection of the destination nodes is highly dynamic and can only be made at send-time.

A common solution to overcome these hardware limitations is to manipulate multicast addresses exclusively by software, using a single multicast (or, more often, broadcast) address of the LAN controller chipset to physically disseminate the messages. In this approach, a multicast address is usually represented by a list of node identifiers exchanged as a message field. However, such a representation does not allow an efficient manipulation. For instance, to check if a given identifier belongs to the multicast address the list must be parsed and each entry read until a match is found. Moreover, in a local area network, a node identifier can be up to 48 bits, increasing comparison times and the multicast address field size.

Another solution is to map each node identifier into a *short-address*. Let the unique node identification be represented as u_i and the short identifier as s_i, an integer in the range $[1 \ldots \mathit{MaxStations}\,]$, such that $u_i \leftrightarrow s_i$. If such a mapping is possible, a multicast address can be represented as a fixed-size array of bits, where the s_i^{th} bit is asserted when station u_i is addressed and negated otherwise. This representation is very compact and allows the operations on multicast addresses to be implemented as binary *logical and/or* operations. This represents a significant performance improvement and when the maximum number of multicast stations is small, it allows the recognition of selective addresses to be implemented in hardware, by the chipset of the underlying network[2].

2.3 Protocol service

Our group membership protocol provides the mapping function referred above by maintaining a ta-

[1] The user can always explicitly request to unmap a given short-address.

[2] For instance, the MC68824[12] token-bus controller has a *group address mask* which can be set to filter messages in function of a bit value.

ble with information about all stations participating in the multicast traffic. For efficiency and fault-tolerance, the table is replicated at every group member. The table includes an array of state entries, each entry storing information about a given member of the group: an entry contains, at least, the node unique identifier and a boolean stating if the node is alive. Additionally, the entry may store user related data. The short-address associated with each MGS member is stored implicitly: it corresponds to the index of the associated entry in the table.

A station may be connected to the network without participating in the group membership protocol. In order to join the MGS group it must execute a *MgsJoin* operation. The join operation requires exchange of messages with the other MGS members to acquire the state table, insert itself and obtain a short-address. Upon an insertion in the MGS, a station is informed of any change in the MGS membership by an *MgsChange* indication. A station may leave the MGS by executing an *MgsLeave* operation. The MGS membership is checked at every execution of a Join or Leave or when a specific *MgsCheck* operation is explicitly invoked. The *MgsCheck* can be called periodically or upon the detection of an event that raises suspicion about the failure of a MGS member.

When a station joins the MGS, it acquires a short-address which will remain associated to that station. Even if the station fails or leaves the MGS group, the short-address remains assigned to the station address so that the remaining stations can refer to it by the associated short-address. If the station recovers and executes a new join, it obtains its old short address. A dedicated operation, *MgsDelete* is used to remove a station from the MGS table and to release the associated short-address. Since there is a local copy of the MGS table available at every station, translation between unique identifiers and short-address is a purely local operation.

3 The MGS Protocol

3.1 Assumptions

The MGS was designed as a low-level membership protocol running on top of local area networks. Thus, it exploits the architecture and technology attributes of LANs that can be used to achieve improved performance and dependability. The MGS benefits of this low-level approach without compromising openness by defining an *abstract network interface* with the properties presented in table 1. The interface, discussed in detail in [13], abstracts the useful communication properties that are common to most existing LANs ensuring MGS portability.

Protocol design assumes that communication components have a fail-silent behavior. That is, a processor fails by stopping producing outputs but never produces an erroneous output. Tests performed in the DELTA-4 project have shown that coverage of this assumption for off-the-shelf hardware is largely acceptable for applications requiring up to a moderate level of fault-tolerance. When high coverage is required, the use of self-checking components must substantiate this assumption. In addition to processor faults, the communication medium may have omission faults as described in property Pn3.

The DELTA-4 communication infrastructure was designed in order to meet high expectancies with regard to fault-tolerance and real-time. The highly reliable and timely environment yielded by a single LAN used in a closed fashion had also to do with the LAN-based approach taken. We carefully devised a dependability model and established its correctness in [13], for such an environment. The MGS protocol, although clock-less (it is not based on synchronized clocks), is synchronous, in the sense that known and bounded execution times are enforced, using the techniques described in [14]. Here we briefly enumerate the major requirements to achieve synchronism of a clock-less protocol: upper bounds on message delivery delays (Pn6), in the presence of overload and faults; performance specification on the hardware/software in order for processing times to be bounded and known; structure the protocol in a predictably bounded number of clearly delimited phases; structure each phase as a bounded series of timed-out transmissions-with-reply, having thus with a known duration bound.

3.2 A reliable communication primitive

The abstract network service, upon which MGS relies, offers an *unreliable multicast* service, presenting a set of properties which are most useful to implement reliable multicast primitives. In absence of faults, the *broadcast* (Pn1) and *full duplex* (Pn2) properties provide message delivered to any processor connected to the network. However, although errors can be considered rare in LANs, the occasional loss of messages – or omissions – cannot be prevented. Thus, the membership protocol must be able to recover from such errors. In the MGS, omission errors are detected and recovered using a transmission-with-response procedure: it uses acknowledgments to confirm the reception of the message and detects omission errors

Table 1: Summary of Network Properties.

- **Pn1** - *Broadcast*: Destinations receiving an uncorrupted frame transmission, receive the same frame.
- **Pn2** - *Error detection*: Destinations detect any corruption by the network in a locally received frame.
- **Pn3** - *Bounded omission degree* : The number of network omission faults (k) is bounded.
- **Pn4** - *Full duplex* : On request, frame indication can be provided at the sender.
- **Pn5** - *Network order* : Any two frames indicated in two different destination access points, are indicated in the same order.
- **Pn6** - *Bounded transmission delay* : Network delays are bounded.

Figure 1: **tr-w-resp** $(\langle m \rangle, ord, first, \mathcal{M}_r, \mathcal{P}_r)$

// $\langle m \rangle$ is a message to be sent.$\mathcal{D}_{\langle m \rangle}$ is the set of recipients.
// "ord" is a boolean specifying if network order is relevant.
// "first" allows the first transmission to be skipped.
// $\mathcal{M}_r$ is a bag of responses.
// a response is expected from each $p \in \mathcal{P}_r$
// (usually $\mathcal{P}_r = \mathcal{D}_{\langle m \rangle}$).

retries := 0;
do *// while*
    if $(retries = 0 \vee ord)$ **then** $\mathcal{P}_w := \mathcal{P}_r$; $\mathcal{M}_r := 0$; **fi**
    if $(retries > 0 \vee first)$ **then** *send* $\langle m \rangle$; **fi**
    $retries := retries + 1$; *timeout := 0; start a timer;*
    while $(\mathcal{P}_w \neq \emptyset \wedge \neg timeout)$ **do**
        when *message* $\langle r_m \rangle$ *received from* $p \wedge p \in \mathcal{P}_w$ **do**
            add $\langle r_m \rangle$ *to* $\mathcal{M}_r$; *remove* p *from* $\mathcal{P}_w$; **od**
        when *timer expires* **do**
            timeout := 1; **od**
    od
while $(retries < MAX \wedge \mathcal{P}_w \neq \emptyset)$

based on the *bounded omission degree* property of the abstract network[3].

The *tr-w-resp* procedure[4] is depicted in figure 1. It consists of a loop, where the data message is sent over the network and responses are awaited for. The procedure waits during a pre-defined time interval for the responses, which are inserted in a response bag, and exits when the desired number of responses is collected. Note that assuming bounded execution and transmission delays (Pn6) a response from all correct processors must be received within a bounded time unless there are network omissions (Pn3). If some responses are missing, the response bag is re-initialized and the message re-transmitted. The main loop finishes when all the intended responses are received or when a pre-defined retry value is reached. Since the number of retries is bounded, there is a well-known worst-case execution time for the complete execution of the *tr-w-resp* procedure by a correct processor (see Sec. 4 for more details).

To preserve *network order*, the procedure retransmits the message until it is acknowledged by all recipients in a same transmission. Note that by abstract network property Pn4, the sender also receives its own frames. When order is not required, the procedure can be optimized by keeping responses in the bag from one re-transmission to the other (response messages are inserted only once in the response bag) . For some omission patterns, this would allow the bag to be filled faster. To activate this mode, the flag *"ord"* must be set to false. Finally, the boolean variable *"first"* can be set to false to prevent the message from being sent over the network on the first cycle of the procedure. This parameter is useful to allow other processors to collect responses – and execute the procedure – on behalf of the sender without re-transmitting the message. The protocol description will give examples of how "first" and "ord" parameters can be used.

Several transmissions with response can be executing simultaneously, on the same or different machines. We assume that messages can be uniquely identified[5]. Different re-transmissions of the same message can also be distinguished. It is thus possible to relate any response with the appropriate *tr-w-resp* instantiation (also called an *emitter-machine*). To make a protocol tolerant to sender crashes, several emitter-machines may be activated concurrently, at recipient sites, for a same message transmission (in this case, responses must be also broadcasted). The next section shows how these features are used by the MGS.

3.3 Protocol Execution

The Multicast Group of Stations protocol maintains, at every member node, a replica of a global *MGS state table* where the relevant information about group membership is stored. The protocol keeps all copies of the state table consistent by ensuring every update is an atomic operation. This is achieved by a distributed lock mechanism: a node must acquire the lock before making any change to the state table. The protocol

[3] The detailed technique, as well as its advantages over other approaches such as diffusion based masking is discussed in detail in [14].

[4] It is a modified version of the procedure given in [13].

[5] The unique message identification is disseminated with the message within an MGS protocol header common to all protocol frames.

assures that, at any given instant, only one node can hold the lock and that only the lock holder is able to change the MGS state.

The protocol is fully decentralized, that is, there is no special node in charge of performing operations on the MGS state table. Instead, any node can try to perform an update at any instant. This has several advantages: the same protocol is used at the initialization, during normal functioning and for failure recovery; upon detection of a failure, any node can immediately execute the reconfiguration of the MGS membership; upon any change, the new MGS state is quickly disseminated to all correct members, despite the occurrence of failures. Different versions of the MGS state table are timestamped with a unique version number.

For clarity, the protocol description is split into three cooperating activities: *Guardian*, *Changer* and *Accepter*. Each of these activities has a different goal:

- Guardian activities are responsible for reading and updating of the local copies of the MGS state table. There is a Guardian running at every MGS member.
- Changer activities perform the computation of the global MGS state table by exchanging information with all active Guardians. A Changer is only activated when a Join, Leave, Delete or Check operation needs to be executed. Several Changers, in different nodes, may compete to obtain the state lock, although only one can hold the lock at a given instant. In a single node only one Changer can be active at a given moment.
- Accepters assure proper protocol termination in case of failure of a Changer. Accepters are created whenever a local copy of the MGS table is updated. Several Accepters can be active at the same time in the same and/or in different nodes.

The sequences of actions performed by each activity are sketched in figure 2 and illustrated in figure 3. The protocol description already includes some optimizations described in the following section. For clarity, in the figures, the changers are drawn separately from the guardians. The reader should remember that there is always a local guardian residing in the same node of the active changer. Note also that local shared variables are not protected with mutual exclusion mechanisms since all activities can be easily executed by the same task/process (we can neglect the processing overhead compared to the delays associated to message transit). The following paragraphs give an informal description of the interaction between these different entities. Some sequences are described in *italic*. These correspond to optimizations that are not mandatory for understanding the basic protocol; the reader may wish to skip them until next section.

1. The Changer starts the protocol by sending a $\langle$*GetState*$\rangle$ message to the Guardians of all MGS members. With this message, the Changer requests the Guardians to provide him with the most recent value of the MGS state table and, implicitly, it tries to acquire the lock of the table. It then waits a response from every MGS member.

2. The Guardians, upon reception of the $\langle$*GetState*$\rangle$ message, send a response message, $\langle$*MyState*$\rangle$, containing the value of the local copy of the table, $\mathcal{T}_{mgs}$. A boolean lock is set to avoid further changes until a new state table is disseminated by the Changer that just acquired the lock. For that purpose, the identification of the lock holder is also stored. Meanwhile, if a $\langle$*GetState*$\rangle$ message is received from a different Changer, a negative acknowledgment, $\langle$*ChState*$\rangle$ is returned. The current changer must disseminate a new MGS state (and implicitly release the lock) within bounded time. A timer, the *lockTimer*, is started to check the activity of the current changer[6].

Since the underlying abstract network is unreliable, it is possible for a single $\langle$GetState$\rangle$ *message to lock just a subset of the Guardians. Thus, two competing Changers can mutually refrain each other of acquiring the lock. This scenario, called a* contention, *is discussed in detail in sec. 3.4.*

3. The Changer collects all acknowledgments.

- If a negative acknowledgment, $\langle$*ChState*$\rangle$, is received from every member, this means that the lock is currently held by another Changer. The local changer waits for the termination of the lock holder to resume activity from step 1. If the active changer fails, the local changer will also resume activity after a pre-defined delay.
- If all responses are of $\langle$*MyState*, $\mathcal{T}\rangle$ type but some responses are missing the $\langle$*GetState*$\rangle$ is retransmitted. The Changer always expects an acknowledgment from every member marked active in the MGS state table. A joining member initializes the set of expected acknowledgments when it receives the first $\langle$*MyState*, $\mathcal{T}\rangle$ message.
- *If both* $\langle$ChState$\rangle$ *and* $\langle$MyState, $\mathcal{T}\rangle$ *messages are received this indicates a contention scenario. The*

[6] The value of the *lockTimer* is a function of the worst-case execution time of the *tr-w-resp* procedure.

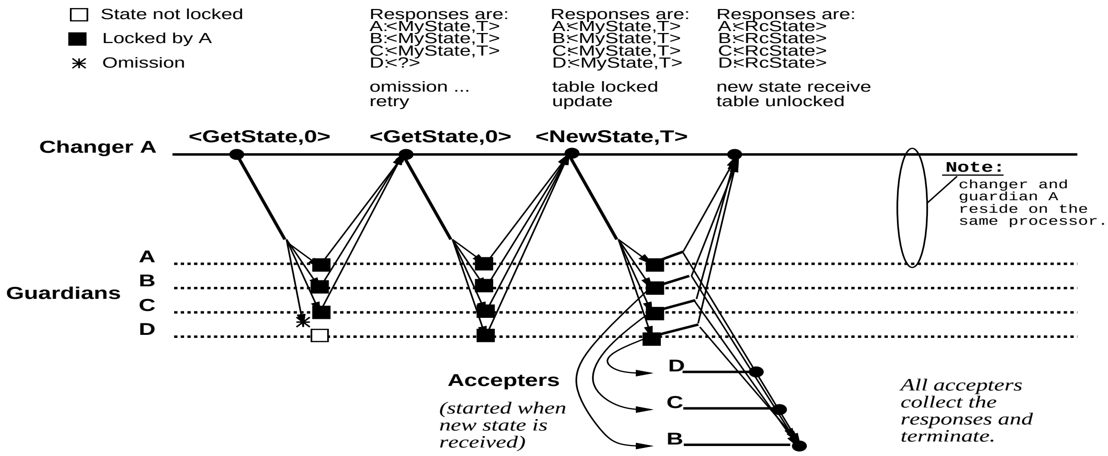

Figure 3: Execution with omissions.

locking level is incremented and ⟨GetState⟩ *is retransmitted. Note that the parameter "ord" of the "tr-w-resp" procedure is activated. This means that a complete new set of responses is expected after each retransmission.*

- If all responses are of ⟨*MyState*, $\mathcal{T}$⟩ type and no responses are missing, the Changer copies the state returned by the Guardians to a temporary variable. The temporary variable is changed accordingly to the operation being executed: new members can be added, marked as failed or entries can be deleted. The temporary state version number is incremented and the state is disseminated in a ⟨*NewState*, $\mathcal{T}$⟩ message. This message propagates a new MGS state table and, implicitly, releases the table lock. The Changer then waits for a confirmation of reception from every Guardian.

  If the state information returned by several ⟨MyState, $\mathcal{T}$⟩ *responses differ, this indicates an unstable state. The* ⟨GetState⟩ *message is retransmitted until the state stabilizes.*

4. When the Guardian receives the ⟨*NewState*, $\mathcal{T}$⟩ message it updates the local copy of the MGS state table and releases the lock. It immediately starts an Accepter to mask the possible failure of the changer. The Accepter will run the same steps as the Changer, from step 5. An acknowledgment ⟨*RcState*⟩ is multicasted to the network. The flag *c_end* is set to indicate to the waiting local changer (if any) that the MGS state is again unlocked.

The accepter invokes the procedure "tr-with-resp" with the parameter "first" inactive. This means that the accepter collects the acknowledgments and only retransmits the new state if some of them are missing.

5. The Changer (and the Accepters) collect ⟨*RcState*⟩ frames. If some acknowledgments are missing the Changer (and the Accepters) retransmit the ⟨*NewState*, $\mathcal{T}$⟩ until an acknowledgment is received from every multicast station in the network. The bounded omission degree property assures that the ⟨*NewState*, $\mathcal{T}$⟩ will be received by all correct processors in an finite time interval.

The ⟨NewState, $\mathcal{T}$⟩ *is disseminated with the "ord" parameter of the "tr-w-resp" procedure inactive. Thus, a positive acknowledgment must be received, at least once, from each guardian.*

3.4 Optimizations

In the protocol described above, operations on the MGS state table are serialized by the locking mechanism. This scheme presented a poor performance when many nodes tried to change the MGS state table simultaneously. During system startup this disadvantage was evident: there all processors try to execute a Join operation at approximately the same time. To overcome this limitation we have enhanced our protocol with a mechanism that allows cooperation between Changers, as described below:

A *changer operation list* is associated to each Changer. At activation time, the operation list only contains the operations to be performed by the local Changer. Whenever a Changer tries to acquire the state lock its operation list is disseminated in the ⟨*GetState*, *oplist*⟩ message. Guardians, upon reception of a ⟨*GetState*, *oplist*⟩, read the operation list field (*oplist*) and add those operations to the operation list of the local Changer. Through this mechanism, active Changers exchange their operation list while trying to obtain the state lock. The winner of the lock is then

Figure 2: MGS Protocol

shared variables

101 *c_end // boolean flag, used when a active changer finishes*
102 *c_opl // used to exchange information between changers*
103 $\mathcal{U}$ *is a set with all the stations in the network*

guardian activity

201 // $\mathcal{T}_{mgs}$ *is the MGS table;*
202 // $\mathcal{V}(\mathcal{T})$ *gives the table version number*
203 // u_{lock} *is the identity of table lock*
204
205 *forever* **do**
206 **when** $\langle$*GetState, opl*$\rangle$ *received from* p **do**
207 *c_opl := c_opl + opl;*
208 **if** $u_{lock} \neq nil$ **then** *send* $\langle$*ChState*$\rangle$;
209 **else** $u_{lock} := p$;
210 *start lockTimer; send* $\langle$*MyState,* $\mathcal{T}_{mgs}\rangle$; **fi**; **od**
211 **when** $\langle$*NewState,*$t\rangle$ *received from* p *and* $p = u_{lock}$ **do**
212 $\mathcal{T}_{mgs} := t$; *send* $\langle$*RcState*$\rangle$;
213 *start Accepter for* $\langle$*NewState,*$t\rangle$;
214 *c_end :=true;* $u_{lock} := nil$; *stop lockTimer;* **od**
215 **when** *lockTimer expired* **do**
216 $u_{lock} := nil$; *start a Changer;* **od** *// lock holder failed*
217 **when** $\langle$*NewState,*$t\rangle$ *received from* p *and* $p \neq u_{lock}$ **do**
218 **if** $\mathcal{V}(t) > \mathcal{V}(\mathcal{T}_{mgs})$ **then** $\mathcal{T}_{mgs} := t$ **fi**;
219 *send* $\langle$*RcState*$\rangle$; **od** *// delayed retransmission*
220 **od** *// forever*

changer activity

301 // $\mathcal{T}_{temp}$ *is a temporary copy of the MGS state*
302 // $\mathcal{I}n(\mathcal{T})$ *gives the set of active members*
303
304 **when** *operation needs to be performed* **do**
305 *// operation is join, leave, check or delete*
306 *c_opl := operation; c_end:=false;*
307 *// get state sequence (implicitly obtains the lock)*
308 *// locking levels not depicted*
309 *tr-w-resp* ($\langle$ *GetState, c_opl*$\rangle, 1, 1, \mathcal{M}_r, \mathcal{U}$);
310 **while** $\neg(\forall i,j \in \mathcal{M}_r, r$ *is of type* $\langle$ *MyState,* t_i $\rangle$
311 $\wedge t_i = t_j)$ **do** *// while state is unstable*
312 *start retryTimer; // wait my turn*
313 *while (* $\neg$ *retryTimer expired* $\wedge \neg$*c_end);*
314 **if** *operation performed by other changer* **then**
315 *return* **fi**;
316 *c_end=false; stop retryTimer;*
317 *tr-w-resp* ($\langle$ *GetState, c_opl*$\rangle, 1, 1, \mathcal{M}_r, \mathcal{U}$); *// retry*
318 **od**
319 *get* $\mathcal{T}_{temp}$ *from responses;*
320 *changeState (* $\mathcal{T}_{temp}$*, c_opl);*
321 *// propagate state sequence (implicitly releases the lock)*
322 *tr-w-resp* ($\langle NewState, \mathcal{T}_{temp}\rangle, 0, 1, \mathcal{M}_r, \mathcal{I}n(\mathcal{T}_{temp})$);
323 *if a failure is detected then the changer is re-activated;*
324 **od**

accepter activity

401 // $\langle NewState, \mathcal{T}_{acc}\rangle$ *is the message to be disseminated*
402
403 **when** *accepter is activated* **do**
404 *tr-w-resp* ($\langle NewState, \mathcal{T}_{acc}\rangle, 0, 0, \mathcal{M}_r, \mathcal{I}n(\mathcal{T}_{acc})$);
405 *if a failure is detected then the changer is activated;*

able to perform all the operations collected during the first phase of the protocol execution. The Changers that loose the competition for the lock wait for the new state table to confirm whether their operations were already executed or not.

Another aspect where the need for optimization was stressed by practical experiments, was the mechanism to recover from *contentions*. A contention occurs when two competing Changers both refrain from acquiring the state lock. This scenario may happen in presence of network omissions: a Changer sends a first $\langle$*GetState*$\rangle$ message that, due to omissions, is only received by a sub-set of the Guardians; another Changer also sends a $\langle$*GetState*$\rangle$ message and locks the remaining Guardians; in this scenario none of the Changers acquires the lock.

Our first version of the protocol did not provide any special mechanism to provide a fast recovery from contentions. When a contention occurs, both Changers set their *retryTimers* (line 312 of figure 2) and Guardians remained locked until expiration of the *lockTimer* (line 215 of figure 2). Recovery was then depended of timers which needed to be configured with long worst case values. To overcome this limitation we have introduced a mechanism that was already used with success in the xAMp based on *locking levels*. Locking levels work as follows. When a Changer request the state lock by sending a $\langle$*GetState*$\rangle$ message, it associates a numeric value to that request. This value is called the locking level. Guardians also store the locking level associated with the lock holder and are allowed to attribute the lock to a Changer other than the current lock holder if (and only if) they receive a $\langle$*GetState*$\rangle$ request with a locking level *higher* than the current level. All Changers start with the lowest locking level and *only* increment the locking level when they are sure that a contention has occurred.

The algorithm becomes more clear with an example (see figure 4): a first Changer, A sends a $\langle$*GetState*$\rangle$ message that, due to an omission, is not received by Guardian D. The lowest locking level, say 0, is associated with that request. Another Changer, B also sends a $\langle$*GetState*$\rangle$ message – also using locking level 0 – and locks D. Changer A detects that some responses are missing and retransmits the $\langle$*GetState*$\rangle$ message keeping its locking level at 0. Changer B receives $\langle$*ChState*$\rangle$ from Guardians A, B and C and receives $\langle$*MyState*$\rangle$ from D: this identifies a contention scenario. Thus, Changer B increments his locking level before retransmitting the $\langle$*GetState*$\rangle$ message. If there are no omissions during the dissemination of B's message, B wins the lock; otherwise the locking lev-

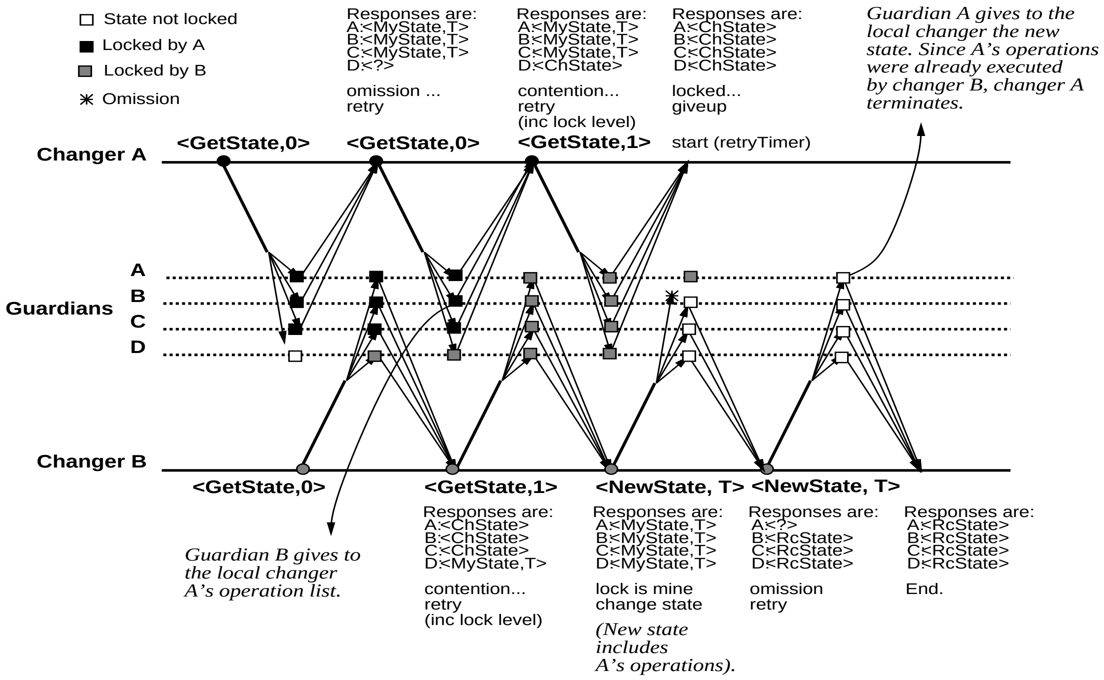

Figure 4: Competitive changers.

el is incremented again (note that as a result of its first retransmission, changer A also detects the contention). The bounded omission degree property assures the convergence of the protocol. The figure also illustrates the cooperation between changers: B's local guardian picks the operation list associated with the $\langle GetState\rangle$ message received from changer A and gives it to changer B before a new state is generated; when the guardian at processor A receives the new state informs the local changer (which then checks if all operations were performed).

Contention can also occur during the release of the lock. This scenario occurs in presence of omissions during the dissemination of the $\langle NewState\rangle$ message or when the lock holder fails. In order to assure proper termination of state propagation the lock winner must always wait for the stability of the MGS state. In order words, the lock winner must retransmit the $\langle GetState\rangle$ message until the *same* state table is returned by *all* positive acknowledgments.

3.5 Handling of failed stations

Failures of a group member are only detected when a Changer is activated to perform an MGS operation. A special MgsCheck operation is provided to activate a Changer only for membership verification purposes. If the Changer itself fails, the protocol also assures that all copies of the state table are left in a consistent state.

The most simple case of failure detection occurs during the execution of the *"tr-w-resp"* procedure. If the absence of an acknowledgment from a member persists after the network omission degree has been masked, then the failure of that member is assumed. If the failure is detected by a changer before the propagation of the new state, this information will be disseminated with the new table. Otherwise, if detected by a Changer or by an Accepter during the propagate sequence, the changer must be activated again.

A slightly more elaborated scenario occurs when a Changer fails. Here recovery depends of the Changer state in the failure instant. When the Changer holds the state lock and fails before propagating the new state, the failure is detected by the Guardians activities when the *lockTimer* expires. When the Changer fails during the dissemination of the $\langle NewState\rangle$ message, the protocol termination is assured by Accepter activities. In the latter case, the failure of the node will be detected by the Accepter during the execution of the *"tr-w-resp"* procedure.

MGS does not have any intrinsic mechanism to detect the failure of a member when all changers are idle. This option was excluded during the design phase based on the observation that many target systems already incorporate some form of "i'm alive" mechanism (as token-rotation on token based LANs). When this is the case, including such a mechanism in the MGS

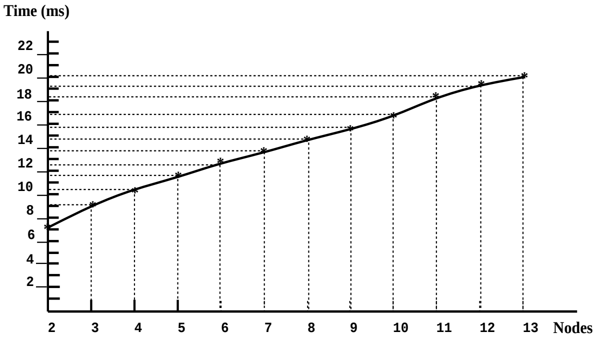

Figure 5: Performance.

would be redundant. However, the MGS does provide the MgsCheck primitive that allows the membership to be checked when there is a suspicion of failure. An example of such a use, is the token-ring based DELTA-4 system: status information concerning the detection of token losses is provided by the token-ring chipset, allowing an MgsCheck investigation action to be triggered. When the target architecture does not provide any support for failure detection, MgsCheck can then be periodically invoked. Any of the alternatives ensures bounded-time failure detection mandatory for real-time operation.

4 Performance

The performance of MGS depends upon the number of nodes involved, the number and type of faults and the number of active changers. The results presented here were obtained using a version implementing the optimizations of sec. 3.4.

Naturally, best-case occurs when only one changer is active and no faults occur. Here, the protocol requires exchange of $2(1+n)$ messages. Let $T_{twr}(n)$ be the time required to execute a *"tr-w-resp"* series, where n is the number of nodes involved, and let T_d be the transit delay for a single message. The best case termination time, defined as the time to execute a MGS operation, is given by $T_{twr}(n) + T_d$. If several changers are competing for the lock, and again in absence of faults, termination can be delayed by the duration of the propagate state sequence. This means that the worst-case termination time in absence of failures is given by $T_{twr}(n) + 2T_d$.

When f ($f < k$) omission failures occur, *"tr-w-resp"* series can generate up to $(f+1)(1+n)$ messages. The MGS involves then the exchange of $(f+2)(1+n)$ messages and exhibits a worst case termination time of $(f+1)T_{twr}(n) + 2T_d$ (omission faults only).

Worst case values are obtained in the presence of crash faults. Worst results for the number of messages exchanged are obtained when stations other than the Changer fail. The Changer needs to execute *"tr-w-resp"* up to the network omission degree, which is $k+1$. Thus, there is a $(k+1)(n+1)$ message overhead per failure. Worst case for termination times occurs when a Changer fails immediately after acquiring the state lock. Loosers only try to obtain the lock again when the *retryTimer* expires. There is an extra delay equal to the value of this timer per Changer failure. The worst case termination time is then strongly dependent on timer configuration.

Numerical results strongly depend on the actual architecture used to support the MGS implementation. We have made our measurements on a cluster of SUN Sparc-Stations with MGS implemented as an Unix[7] device driver. The execution time of an MGS operation, as a function of the membership size, is presented in figure 5. It ranges from 7.3 ms with 2 stations to 20.3 ms with 13 stations, which roughly gives an increment of 1ms per extra member.

5 Related work

To our knowledge, there are not many examples of low level processor group membership protocols in the literature. A group membership that, as ours, keeps information about active nodes in the system is presented in [5]. However, it assumes tightly-synchronous behavior of the system and makes extremely restrictive fault assumptions: for instance, a scenario where a message sent is lost by two receivers but received by the remaining nodes is excluded from the fault hypothesis. This assumption is valid in the highly closed environment of MARS [6] but is clearly unacceptable in most of our target LAN based systems [15]. A set of group membership protocols for synchronous systems based on point-to-point networks is presented in [3]. This work is hardly comparable to ours since it assumes the availability of a clock-driven atomic broadcast primitive [2], thus obviating most of the problems MGS is concerned with. Furthermore, none of the two previous examples provide support for dynamic attribution of short-addresses to system nodes. The protocol in [7] eliminates in the assumptions the ordering problem we are concerned with. In this approach protocol synchrony is based on the 'cycles' of an underlying time domain multiplexing technique where the order in which processes access the network is static

[7] Unix is a trademark of USL.

and known in advance. Approaches like that of [4] are also not comparable to ours since they do not provide consistent views of membership changes. The protocol in [9] is closer to ours and also resembles a two-phase commit protocol. However, it does not exploit the use of broadcast networks and it is designed for asynchronous systems. Thus, it cannot guarantee bounded execution times. On the contrary, our approach relies on bounded execution and transmission delays to enforce the timely behaviour of the protocol. Additionally, our assumptions simplify the recovery algorithm which does not require the use of three-phase protocols. The use of short-addresses to improve the efficiency of a local area network is, for example, exploited in Autonet [11], although oriented for point-to-point communication.

6 Conclusions

The design and implementation of the MGS protocol and of the xAMp multi-primitive service has provided us with a better insight of how to split group support functionality into different architecture levels. We have found that the availability of a low-level *processor* group membership strongly simplifies the implementation of higher-level *process* group membership and communication protocols. Thus, we believe to have made a correct use of the end-to-end argument by implementing such service near the abstract network layer, where technology can be exploited to improve its performance.

When designing the MGS group membership protocol, we tried to follow a kind of "small is beautiful" approach. Actual experiments with prototypes on different networks have shown that over-simplification incurred some performance penalties. We have then implemented some optimizations which, although slightly increasing protocol complexity, provide satisfying performance under highly demanding scenarios such as system startup.

Since MGS is based on the *"tr-w-resp"* technique, thus requiring the transmission of an acknowledgment per node, its performance decreases linearly with the number of nodes involved. Groups of multicast stations up to 64 nodes are in fact those for which our protocol is best suited. The results presented in section 4 show that its use is effective when the number of stations is small and address recognition can be hardware implemented. Currently, MGS is being used to support xAMp's multicast address mechanism.

Acknowledgments

Most of this work has been developed in the scope of the DELTA-4 architecture. We wish to thank H. Fonseca for his contribution to the evolution and engineering of MGS. The authors are indebted to R. Ribot and M. Chereque for the many criticisms and suggestions made during the design and implementation of MGS. In particular, the need for cooperation between changers was identified and suggested by R. Ribot after several tests with token-ring based DELTA-4 prototypes.

References

[1] 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*, Newcastle-UK, June 1990. IEEE. also as INESC AR/21-90.

[2] 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*, Ann Arbor-USA, June 1985. IEEE.

[3] Flaviu Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Digest of Papers, The 18th International Symposium on Fault-Tolerant Computing*, Tokyo - Japan, June 1988. IEEE.

[4] Richard A. Golding. Group membership in the epidemic style. Technical Report UCSC-CRL-91-32, Univ. of California, Santa Cruz, CA, December 1991.

[5] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In *Proceedings of the IFIP WG10.4 Int'l Working Conference on Dependable Computing for Critical Applications*, Sta Barbara - USA, August 1989.

[6] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, pages 25–41, February 1989.

[7] Rogerio Lemos and Paul D. Ezhilchelvan. Agreement on the group membership in synchronous

distributed systems. In *Proceedings of the 4th Intl Workshop on Dist. Algorithms*, pages 1–20, Bari, Italy, September 1990.

[8] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing.* ESPRIT Research Reports. Springer Verlag, November 1991.

[9] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environemnts. Technical Report TR 91-1188, Cornell University, Department of Computer Science, 1991.

[10] L. Rodrigues and P. Veríssimo. xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, Texas, October 1992. INESC AR/66-92.

[11] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring, local area network using point-to-point links. Technical Report 59, Digital,Systems Research Center, Palo Alto, California, April 1990.

[12] TBC. *Token-Bus Controller MC68824 Data Sheet.* Motorola, 1986.

[13] P. Veríssimo and José A. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, Huntsville, Alabama-USA, October 1990. IEEE. Also as INESC AR/24-90.

[14] P. Veríssimo, J. Rufino, and L. Rodrigues. Enforcing real-time behaviour of LAN-based protocols. In *Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems*, Semmering, Austria, September 1991. IFAC.

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