# A distributed kernel for reliable group communication

*Samuel T. Chanson & K. Ravindran*

February 1986
Revised August 1986
Technical Report 86-4*

ABSTRACT

Multicasting provides a convenient and efficient way to perform one-to-many process communication. This paper presents a kernel model which supports reliable group communication in a distributed computing environment. We introduce new semantic tools which capture the nondeterminism of the underlying low level events concisely and describe a process alias-based structuring technique for the kernel to handle the reliability problems that may arise during group communication. The scheme works by maintaining a close association between group messages and their corresponding reply messages. We also introduce a dynamic binding scheme which maps group id's to multicast addresses. The scheme allows the detection and subsequent recovery from inconsistencies in the binding information. Sample programs illustrating how the semantic tools may be used are also included.

* Also appears in the Proceedings of the IEEE Computer Society Real-Time Systems Symposium, New Orleans, Dec. 1986.

# A distributed kernel model for reliable group communication

Samuel T. Chanson & K. Ravindran

Dept. of Computer Science,
University of British Columbia.
Vancouver, B.C., Canada V6T 1W5

ABSTRACT

One-to-many (group) interprocess communication is useful in many real-time distributed applications. It may be conveniently and efficiently realised using the multicast feature available in contemporary local area networks. This paper presents a kernel model which supports reliable group communication in a distributed computing environment. We introduce new semantic tools which capture the nondeterminism of the underlying low level events concisely and describe a process alias-based structuring technique for the kernel to handle the reliability problems that may arise during group communication. The scheme works by maintaining a close association between group messages and their corresponding reply messages. Sample programs illustrating how the semantic tools may be used are also included.

## 1. Introduction

In distributed system architectures, one-to-many (group) interprocess communication (IPC) is a useful paradigm for structuring intra-program communication in distributed programs[7]. It refers to an IPC activity by which a single message may be transferred from one part of a computation to many other parts which may be in the same or different machines in the network. Examples are searching a file in the network and propagating an update to replicated databases.

The need for group communication is inherent in many real-time distributed applications. An example is an embedded system such as an on-board control system for aircrafts in which the functional components are replicated and/or distributed, for overall system reliability, across a number of processors interacting with one another over a shared bus[11]. One-to-many IPC is useful in such systems to co-ordinate among the various components. Another example is a set of machines interconnected by a local area network (LAN) and interfaced to an industrial plant (e.g., a radar control system[12]) in which the monitoring and the control functions are performed by these machines. One-to-many IPC is the tool of choice in this type of systems to propagate the state value from a sensor to multiple monitoring/control stations and to transmit a processed output to multiple actuators in the plant. Besides these application-driven requirements, the availability of multicast feature in many of the current LAN technologies[4] is providing an impetus to programs, particularly those with real-time constraints, to use one-to-many IPC because multicasting supports one-to-many IPC more efficiently[3].

In order that distributed programs may systematically employ one-to-many IPC for intra-program communication, adequate operating system tools are required. In general, such tools should handle the associated semantic issues which are different and more complex than those of one-to-one IPC. Since different applications may require different degrees of reliability in the communication mechanism, the tools should provide a powerful and flexible semantics to one-to-many IPC as well as a uniform structural base for using this form of IPC. The requirement for such type of tools is stronger in real-time programs because of the time constraints under which they operate. The lack of adequate program-level tools hitherto has forced the realisation of an equivalent communication by repeated one-to-one IPC with the attendant penalty on efficiency and semantic consistency. This could have adverse implications on the structure and the real-time behaviour of distributed programs.

In this paper, we describe the mechanisms provided by the kernel to support a flexible form of reliable one-to-many IPC. Applications may build the required recovery procedures on top of these kernel mechanisms in the client-server i/o interface.

## 2. Process groups and group communication

Different types of operating system objects such as multicast sockets[6] and process groups[3] have been introduced to encapsulate the notion of one-to-many IPC. Because they can be more easily extended to a distributed environment, we have adopted process groups based on the message-passing abstraction as the basis for our discussion.

A process group is a set of processes that share one or more abstract objects and interact among themselves to provide a unified interface to the external world. An example is a distributed file server structured as a group of processes and running on a set of workstations on a LAN. The members in a group may be structured in a linear, hierarchical or other relationships based on the nature of information flow among them[9]. For simplicity, we shall consider only the linear group structure in our discussion.

Group communication is a kernel mechanism by which a process may interact with process groups. It refers to a communication activity whereby a sender sends information to be simultaneously delivered to one or more recipients with the latter named as a group rather than individually. The reliability semantics associated with the group communication, i.e., the notion of success or failure of the group communication, must capture the failures that may occur *during* communication with members of the group. We describe these issues in sections 3 and 4.

## 3. Semantics of group communication

To understand the underlying semantic issues better, we first compare this mechanism with one-to-one IPC. One-to-one IPC refers to a message-passing activity between one sender and one receiver with the communication partners known to each other and often explicitly named. The semantics of such an activity is well-defined because with a single receiver, the sender knows from which process to expect a reply if the activity is successful, and which process has not received the message if the activity fails for whatever reasons. On the other hand, during a group communication activity, each of the group members as well as the interaction with that member may fail independently. For example, in a group consisting of N members, K of them may fail independently; of the remaining (N-K) members, perhaps only R $[R \leq (N\text{-}K)]$ may receive the group message due to the independent failure of the communication activities. Is a group communication activity successful when some members of the group did not receive the message (either due to communication failures or because the processes have failed)? Also how should the sender deal with the reply messages, the number and sources of which are unknown before they arrive? How does the sender know if all members in the group have received the message since acknowledgements or replies may have been lost? These inherently non-deterministic issues make the reliability semantics of a group IPC more complicated than that of a one-to-one IPC.

### 3.1 Event-based analysis of group communication

The semantics of any inter-machine activity is derived from the low level events $\{E_i\}$ that occurred during the activity. We shall denote this association by

$$\{s_j\}_{j=1,2,..} \xrightarrow{sem} \{E_i\}_{i=1,2,..}$$

where $\{s_j\}$ is the abstract characterisation of $\{E_i\}$ exposed by the kernel to the applications. Consider, for example, a synchronous IPC between processes C and S residing on sites $M_C$ and $M_S$ respectively. The outcomes specified by the kernel are $s_1$ = SUCCESS, $s_2$ = NON_EXISTENT_PROCESS, $s_3$ = SITE_FAILED, $s_4$ = NETWORK_FAILED and $s_5$ = SITE_UNREACHABLE[2]. Each of these outcomes is an abstract representation of the state of the participating components in the IPC, namely the penta-tuple <C, $M_C$, network, $M_S$, S>. Since each of the components may be operational (UP) or failed (DOWN) denoted by the states U and D respectively, the semantics of each of the outcomes as viewed from the client process may be given by:

$$\text{SUCCESS} \xrightarrow{sem} (U \wedge U \wedge U \wedge U \wedge U),$$

$$\text{NON\_EXISTENT\_PROCESS} \xrightarrow{sem} (U \wedge U \wedge U \wedge U \wedge D),$$

$$\text{SITE\_FAILED} \xrightarrow{sem} (U \wedge U \wedge U \wedge D \wedge D),$$

$$\begin{aligned}\text{NETWORK\_FAILED} \xrightarrow{sem} &(U \wedge U \wedge D \wedge U \wedge U) \vee \\ &(U \wedge U \wedge D \wedge D \wedge D) \vee \\ &(U \wedge U \wedge D \wedge U \wedge D),\end{aligned}$$

$$\begin{aligned}\text{SITE\_UNREACHABLE} \xrightarrow{sem} &(U \wedge U \wedge U \wedge D \wedge D) \vee \\ &(U \wedge U \wedge D \wedge U \wedge U) \vee \\ &(U \wedge U \wedge D \wedge D \wedge D) \vee \\ &(U \wedge U \wedge D \wedge U \wedge D).\end{aligned}$$

Thus a quantitative analysis of the events that comprise the activity enables us to provide an adequate and concise abstraction of the underlying events.

Basically, group communication supported by a distributed kernel may be of two types -- *stateless* and *blocking* (analogous to their one-to-one counterparts). We consider them separately:

*3.1.1 Stateless group communication.* A stateless group communication is a single event activity that does not maintain any state information as to its success or failure. It typically uses a network level multicast datagram with no guarantee on reliable or confirmed delivery. The stateless nature of such a multicast datagram transmission at the network level can be easily mapped onto a high level stateless group communication operation.

We may abstract multicast datagram as a series of unicast datagrams triggered at the source at infinitesimally spaced intervals. Each of such datagrams may arrive successfully at the destination (ARRIVAL event) or lost/damaged (NON_ARRIVAL event). Thus a multicast datagram generates a set of ARRIVAL and NON_ARRIVAL events, one at each of the recipient sites, given by

$$E_{dgrm} \subset \{\text{ARRIVAL, NON\_ARRIVAL}\} \times \{M_1, M_2,.., M_N\}$$

where the $M_i$'s (i=1,2,..,N) are the component datagrams of the multicast message and '$\times$' denotes the cartesian product of the two sets. Note that (ARRIVAL, $M_i$) and (NON_ARRIVAL, $M_i$) are mutually exclusive events.

Since each member of the group may independently exist or fail, represented by the states EXISTENT and NON_EXISTENT, the set of events generated by a stateless group communication operation built on top of multicast datagram is given by

$$E_{stls} \subset \{\text{EXISTENT, NON\_EXISTENT}\} \times \{G_1, G_2,.., G_N\} \times E_{dgrm}$$

where the $G_i$'s (i=1,2,..,N) are the individual members of the group. Note that (EXISTENT, $G_i$) and (NON_EXISTENT, $G_i$) are mutually exclusive elements (i=1,2,..,N).

*3.1.2 Blocking group communication.* In a blocking type primitive, the sender is blocked until a certain number of replies from the group members are received. The multicast datagram transmission triggers, say R arrival events ($R \leq N$) at the receiving sites given by

$$E_{msg} = \{X : X = (\text{ARRIVAL}, M_i) \wedge X \in E_{dgrm}, (i=1,2,..,N)\}.$$

The $R'$ ($R' \leq R$) reply events generated at the receiving sites is given by

$$E_{rep} \subset \{\text{EXISTENT, NON\_EXISTENT}\} \times \{G_1, G_2,.., G_N\} \times E_{msg}.$$

These $R'$ events cause $R''$ events ($R'' \leq R'$) pertaining to the arrival of these replies at the sending site are given by

$$E_{blk} \subset \{\text{ARRIVAL, NON\_ARRIVAL}\} \times E_{rep}.$$

The above analysis shows the large number of low level events that may occur during a group communication activity. Applications should be designed to handle the associated reliability semantics, namely, $s_{stls} \xrightarrow{sem} E_{stls}$ and $s_{blk} \xrightarrow{sem} E_{blk}$, and be structured accordingly ($s_{stls}$ and $s_{blk}$ are the representations used to specify the outcomes of the activities concerned).

## *3.2 Semantic tools*

Despite the large combinations of low level events that may occur as illustrated by the event-based analysis, the semantics can be concisely abstracted. This is because, unlike the one-to-one IPC, each individual low level event does not by itself significantly influence the outcome of the group communication activity, thus not all low level events need to be considered. Neverthless, existing semantic tools used for one-to-one IPC were not designed to handle the non-determinism associated with group communication thereby necessitating new tools. Such tools must concisely, adequately and uniformly convey the semantics $s_{stls} \xrightarrow{sem} E_{stls}$ and $s_{blk} \xrightarrow{sem} E_{blk}$ to applications.

Two basic notations are introduced to characterize the success/failure of group communication:

*3.2.1 Degree of delivery.* It is defined as the fraction of the members in the group that has received the message. Thus, for example, if R members in a group of N actually receive a message sent, then the degree of delivery *r* of the message is given by

$$r = \frac{R}{N}, (0.0 \leq r \leq 1.0).$$

This notion allows a sender initiating a group message to specify the desired delivery requirement independently of the size (cardinality) of the group. It also implicitly specifies the desired degree of synchronisation with the group members, a useful paradigm for real-time applications. It may be used by the IPC layer supporting group communication to quantify the degree of success or failure. The notion of confirmed delivery is based on the number of replies the sending site has received. Thus if R members of the group have received the message and replied ($R \leq N$), and $R'$ replies arrive at the sending site ($R' \leq R$), then

$$\text{Degree of delivery} = \frac{R}{N}, \text{ and}$$

$$\text{Degree of confirmed delivery} = \frac{R'}{N}.$$

These notions capture the underlying characteristics that the success/failure of a group communication usually does not depend on that of any of the individual members. For example, even if a member fails during a group communication, the operation may succeed by the arrival of replies from other members. In other words, the semantics of the operation depends more on the macroscopic events pertaining to the group behaviour than those of individual members.

For applications requiring a specific number of replies and/or where the group size is not known, we provide another form of specifying the degrees of delivery and confirmed delivery where the desired number of replies is specified as an integer.

A group send primitive incorporating the above notion takes the form status = *group_send (msg, group_id, r, time_out).*

The message <msg> is sent to the members contained in the group <group_id>. <status> is a data structure returned by the kernel to indicate the outcome of the primitive; <r> is a data structure used to define the scope of the communication operation requested (i.e., it specifies the desired degree of confirmed delivery):

**Case 1.** r = <FRACTION, 0.0>.

The operation degenerates into a datagram send operation. The sender is unblocked immediately after the message is sent. At the receiving sites, reply messages to this operation are inhibited by the kernel. This may be useful in certain real-time applications such as sending periodic sampled data values from a sensor to multiple monitoring stations in a plant; such updates may not require confirmed delivery since errors due to lost samples will usually be corrected by a subsequent sample.

**Case 2.** r = <FRACTION, val>, where $0.0 < val \leq 1.0$.

The operation blocks the sender. As part of the message transport mechanism, the kernel acquires the cardinality N of the group from the hosts[*] containing the members of the group. After receipt of at least r*N replies or timeout, whichever occurs earlier, the kernel unblocks the sender.

**Case 3.** r = <INTEGER, num>, where num > 0.

The operation blocks the sender until the specified number of replies are received or timeout occurs, whichever is earlier.

The sender may either belong to the same group as the recipients or be external to the group.

*3.2.2 Completion variables.* When the kernel terminates a group communication, it returns two values to the sender through the variables **ATLEAST** and **ATMOST**. These values are interpreted as follows:

**ATLEAST**$(s_1)$

At least a fraction $s_1$ of the total number of members in the group has received the message.

**ATMOST**$(s_2)$

At most a fraction $s_2$ of the total number of members in the group has received the message.

Note that $(0.0 \leq s_1 \leq s_2 \leq 1.0)$. These completion variables characterise the degree of success/failure of the group communication, and they are returned in the <status> data structure (refer to the primitive in section 3.2.1).

[*] A host is an abstract object associated with a particular instantiation of a machine.

The **ATMOST**$(s_2)$ and **ATLEAST**$(s_1)$ tools bring out two characteristic aspects of group communication, namely, ***partial completion*** and ***partial success*** respectively. Partial completion refers to a situation where communication with all N members of the group is not yet completed. If communication is known to have been completed with $R_1$ members $(R_1<N)$, and of these, only $R_2$ is known to be successful $(R_2 \leq R_1)$, then the semantics of the operation is $\mathbf{ATLEAST}(\frac{R_2}{N}) \wedge \mathbf{ATMOST}(\frac{R_1}{N})$. Partial success refers to a situation where communication with *all* N members of the group is completed but only $R_2$ of these is known to be successful $(R_2<N)$. This is captured by the expression $\mathbf{ATLEAST}(\frac{R_2}{N}) \wedge \mathbf{ATMOST}(1.0)$. In terms of these semantic tools, complete success and complete failure are just two special cases, namely **ATLEAST(1.0)** and **ATMOST(0.0)** respectively.

For a stateless group communication, since no information is maintained about the success/failure of the activity, the kernel may only return **ATLEAST(0.0)** $\wedge$ **ATMOST(1.0)**. A simple form of blocking group communication is provided in the V-System[3] where $r = \frac{1}{N}$, i.e., the sender blocks until at least one member of the group has received the message and replied, or timeout occurs, whichever is earlier. In this form, the sender need not possess knowledge of the size and/or membership of the group. In terms of our semantic tools, the return code for the V-System primitive is $\mathbf{ATLEAST}(\frac{1}{N}) \wedge \mathbf{ATMOST}(1.0)$ when the primitive is unblocked by a successful reply; when unblocked by a timeout, the return code is **ATLEAST(0.0)** $\wedge$ **ATMOST(1.0)**.

## *3.3 Partial communication*

It is important that applications are aware of the possibility of partial communication because of its implications. Consider the group communication primitive introduced earlier in section 3.2.1. Partial communication is possible in certain communication architectures where all members of the group cannot be reached by a single message-passing event. Examples are group communication implemented on top of one-to-one IPC and group communication spanning interconnected LANs.

Consider the situation in which an application requires a confirmed delivery to R members. We consider the following scenarios:

(i) $R \ll N$.

The communication activity may be declared successful after completion of communication with $R'$ members $(R \leq R' < N)$. Thus the IPC layer has not *attempted* to communicate with all members in the group. In this sense, the communication activity is incomplete though successful.

(ii) Inadequate timeout intervals.

If the timeout interval specified for the group communication is not sufficient to contact enough members in the group, as is likely in applications with certain real-time deadlines, premature termination of the communication is likely. Note that this is different from the case where *all* members have been contacted but the timeout interval is not sufficient to receive enough replies.

(iii) Network partitioning in the midst of the message-passing activities.

A network failure may occur during the sequence of message transmissions which may result in partitioning of the group. With some network hardware, the kernel can detect this failure enabling appropriate recovery.

(iv) Incorrect binding of group id's to the members in the group.

Any inconsistency in the binding that arises due to changes in the group constituency may lead to undetected incomplete communication in certain situations.

Our proposed semantic tools may be used to characterise such anamolies in group communication.

*3.3.1 Group communication on top of one-to-one IPC.* The IPC layer that implements the group communication primitives maintains a list L(G) of the members of the group G, and sends a one-to-one message to each member at the corresponding unicast address; each of these one-to-one communication is supported by a point-to-point protocol such as the symmetric polling protocol[2] and the paired message protocol[10]. The cardinality N(G) of the group is derived from L(G).

The semantics of such a group communication primitive $S_{grp/1\text{-}1}$ is to be built in terms of that of the individual one-to-one IPC primitives $S_{1\text{-}1}$ and the low level events that may occur during such a sequence. Let $S_{1\text{-}1}$ be {SUCCESS, NON_EXISTENT_PROCESS, SITE_UNREACHABLE} describing the various failure conditions[2], and let $L(G) = \{g_1, g_2,.., g_N\}$. Suppose after K one-to-one IPC (with success or failure, $K < N(G)$), the group communication activity terminates for any of the reasons discussed before. Then the output events that constitute the group communication form a set given by

$$E_{1\text{-}1} \subset S_{1\text{-}1} \times \{g_1,g_2,..,g_K\},$$

where (SUCCESS,$g_i$), (NON_EXISTENT_PROCESS,$g_i$) and (SITE_UNREACHABLE,$g_i$) are mutually exclusive events for i = 1, 2,.., K. Thus the possible degree of delivery (r) and the confirmed degree of delivery (r′) are given by

$$r = \frac{K}{N}, \quad \text{and} \quad r' = \frac{K'}{N}, \text{ where}$$

$$K' = \text{card}\,[\ \{\ X: X = (\text{SUCCESS}, g_i) \wedge X \in E_{1\text{-}1},\ (i=1,2,..,K)\ \}\ ].$$

'card()' is a function that returns the cardinality of a set. Thus the semantics of the primitive is

$$S_{grp/1\text{-}1} = \textbf{ATLEAST}(r') \wedge \textbf{ATMOST}(r) \overset{sem}{\rightarrow} E_{1\text{-}1}$$

*3.3.2 Group communication on top of network multicast.* Suppose the IPC layer implements the group communication primitives on top of network multicast. If the group is located on a single LAN, the transmission of the message to the individual members is a single event. The multicast address A(G) associated with the group G is essential in such an approach. The IPC layer multicasts the message at address A(G) and collates the replies from the individual members over the timeout interval. The semantics of the primitive, then, is to be built in terms of the multicast transmission event, the reply and other related events. A timeout or network partitioning event may place the sending site in one of two situations, namely the message has been multicast or not multicast. In the former case, all members in the group may have received the message implying an **ATMOST(1.0)** semantics while the later case indicates a failure resulting in an **ATMOST(0.0)** semantics. Intermediate values of the degree of delivery is not possible[†].

However, if a process group spans across interconnected LANs, a single group communication activity causes a sequence of multicast events to be triggered at the intervening gateways. Each multicast event atomically delivers the message to the subset of the group on the associated local network. Since gateways may fail, partial completion may result; but the granularity of delivery on a particular LAN is the entire subset of the group that resides on it. For example, let $s_{G_1}$ and $s_{G_2}$ be the subsets of the group G that reside on $LAN_1$ and $LAN_2$ respectively, and let a sender on $LAN_1$ initiate a group message. If the gateway failed before the multicast message is relayed onto $LAN_2$, the semantics is given by

$$S_{grp/M} = \textbf{ATLEAST}(r_1) \wedge \textbf{ATMOST}(r_2) \overset{sem}{\rightarrow} E_{grp}$$

where $r_1 = \frac{R_1}{N}$, $r_2 = \frac{\text{card}(s_{G_1})}{N}$, ($R_1$ is the number of replies received by the sender from $s_{G_1}$), and $E_{grp}$ is the set of events comprising the communication activity.

*3.3.3 Implications of partial communication.* Depending

[†] Thus group communication implemented by network multicast is semantically different from that by repeated one-to-one IPC.

on the applications, partial communication might require specific high level protocols to counter it. For example, in the CIRCUS system[5] which implements group communication using one-to-one IPC, partial completion may result if the group constituency changes without properly updating the membership list at the sender's site. Since CIRCUS requires confirmed execution of the operation specified in a group message by *all* members of the group, the designers circumvented this problem by using an unconventional protocol whereby the group id changes whenever the group constituency changes. This forces the sending site to rebind to the changed id, and in the process, correct its membership list.

Also, for group communication using one-to-one IPC's and those across interconnected LANs, there is an implicit ordering in which the message is communicated to the group members. This implies a priority assignment among the group members which violates the semantics of process groups. Typically, members lower down in the ordered list are less likely to be communicated during the group communication activity. Though some form of dynamic priority reassignment techniques might be used to enhance the chance of fairness in communication, such techniques do not eliminate the issue. Note that the high level issues arising from these anamolies in communication might be different depending on the underlying cause.

Examples illustrating how the above semantic tools may be used by application programs are given in the appendix.

## 4. Structuring tools for group communication

We now look into the structural elements needed by the distributed kernel to handle the reliability issues that may arise during group communication. Basically, this requires monitoring of events on the machines participating in a group communication as well as the flow of state information among them. We have used process aliases* as the primary structuring tools in kernel designs to perform such functions and hence to provide appropriate semantics to one-to-one IPC against failure events and process relocations[1,2]. Such tools may, in principle, be used in designing kernels to support process groups as well. However, the functionality of such aliases depend on the type of semantics required.

The kernel at the sending site may create invisible aliases for the sending process at the member sites; these aliases may monitor the progress of events during the group communication and dispatch state information back to the sending site. However, unlike the one-to-one case, the number of microscopic events that may occur per communication activity is large, creating excessive message traffic and making event monitoring at the low level expensive. Secondly, as noted earlier, failure events with respect to individual members do not individually affect the outcome of the communication activity. Therefore, it is necessary only to monitor essential macroscopic events.

### 4.1 The kernel requirements

The kernel should provide a consistent message-passing semantics against failures of the group members during a group communication. Furthermore, unlike the case of a blocking one-to-one IPC, the sender and the recipients in a group communication are not tightly synchronised. For example, a sender may be unblocked during a group communication and send further messages to the group many times before some recipients get around to replying to the first message. This may arise, for example, because a member site is heavily loaded or is much slower than other member sites. This may lead to the following undesirable phenomena at the kernel level:-

- The queueing up of group messages at some member sites pertaining to already-completed group communication activities. Such messages are meaningless particularly if messages have expiry deadlines as in real-time applications.
- The flow of unnecessary reply messages for already-completed communication activities.
- The undetected pairing of reply message and group message belonging to different communication activities. For example, a reply from a member for the j-th group message may be construed by the sender as a reply to its on-going k-th group communication ($k > j$).

The first two phenomena consume system resources unnecessarily and should be eliminated or minimised. The third phenomenon is logically incorrect, and should be eliminated. The solution to the problems lies in maintaining the association between group messages and reply messages.

### 4.2 Process alias based structure

In this structure, the association between group messages and reply messages is maintained partly in the member's process descriptor and partly in a remote alias residing at the member site created on behalf of the sender (see Figure 1). When a sender C sends a message to the group consisting of N members, the kernel at the sending site sets up an invisible alias $A_{cl}$ at the local site and an invisible alias $A_{cr_i}$ at the i-th member site ($i=1,2,..,N$). These aliases

* A process alias is a light-weight executing entity created by a process or kernel (known as invisible alias in the latter case) to perform a well-defined and simple function. It may reside on a different address space (including the kernel space) on the same or a different machine from that of its creator. It does not have independent existence. It can not set up strong bindings in the system and satisfies other properties so that it may be created and destroyed inexpensively.

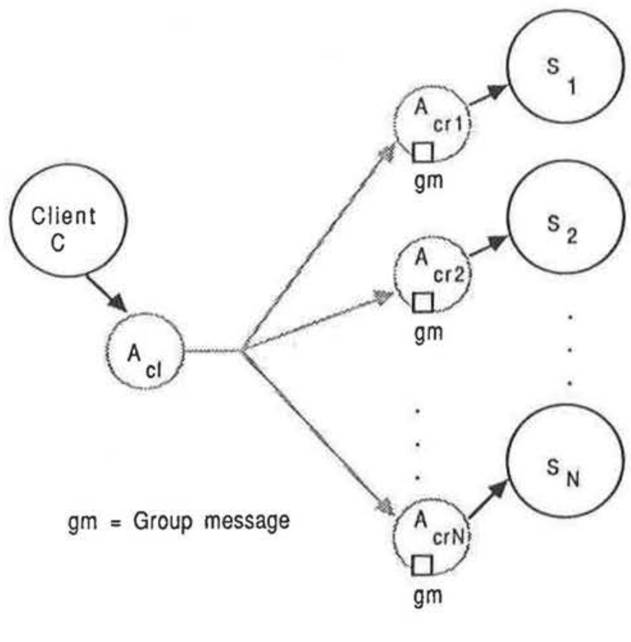

Figure 1. A process alias based structure to implement group communication.

are set up by the initial message that is multicast[**], and are destroyed upon completion of the communication activity or the failure of the sender. In addition, when the i-th member replies or fails, the alias $A_{cr_i}$ associated with it is destroyed.

The remote alias $A_{cr_i}$ serves two functions: i) maintains an image of the state of the sender at the i-th member site, and ii) acts as a remote repository of the group message M(snd_seq) where <snd_seq> is the sequence number of the on-going group communication activity at the sender's site. $A_{cr_i}$ may be construed as maintaining the state variable <snd_seq> that is updated by the initiation and completion events occurring at the sending site for the group communication activity. Another state variable <rcv_seq>, which is the sequence number of the group message currently being processed by the member, is maintained in the member's process descriptor. There is one <rcv_seq> for each concurrent communication stream in which the process is engaged in. If the member is not currently processing any message from the sender, the corresponding <rcv_seq> is assigned a null value.

*4.2.1 The multicast protocol.* $A_{cl}$ engages in an asymmetric protocol with the $A_{cr_i}$'s. $A_{cl}$ periodically multicasts an I_AM_HERE(<snd_seq>) message to the $A_{cr_i}$'s indicating that the sender C and the communication activity is still alive. If the communication activity completes, $A_{cl}$ destroys itself after multicasting an ACTIVITY_COMPLETE(<snd_seq>) message. If C fails, $A_{cl}$ destroys itself after multicasting an ACTIVITY_ABORT message. On receiving either one of the messages, each $A_{cr_i}$ destroys itself. If the message is lost or if the machine $M_c$ fails or if the network partitions isolating one or more members from $M_c$, the $A_{cr_i}$'s affected eventually recover by the protracted absence of the I_AM_HERE message from $A_{cl}$, and destroy themselves. When a group member (say the j-th member) issues a reply operation to a group message, the kernel may dispatch the reply only if the concerned alias $A_{cr_j}$ exists; otherwise the kernel fails the reply operation. $A_{cr_j}$ is destroyed when the reply message is sent.

*4.2.2 Recovery procedure at the receiver site.* Consider the following events occurring at the i-th member site.

**Event 1.** The member issues a *grp_receive* operation.

If the associated <rcv_seq> is not null, the kernel fails the attempt by the member to receive a message before the message M(rcv_seq) currently being processed has been replied. If <rcv_seq> has a null value, the kernel checks for the existence of $A_{cr_i}$. If $A_{cr_i}$ exists, the group message M(snd_seq) is delivered to the member and <rcv_seq> is updated to the value contained in <snd_seq>. Otherwise the kernel blocks the receiver to await a new message from the sender, and resorts to an asynchronous protocol to ascertain the existence of the sender.

**Event 2.** The member issues a stateless *grp_reply* operation.

If the corresponding <rcv_seq> has a null value, the kernel fails the attempt by the member to reply to a non-received message. If <rcv_seq> is not null, the kernel checks for the existence of $A_{cr_i}$. There are two cases to consider:

(i) $A_{cr_i}$ exists, indicating that there is an on-going group communication activity with the sequence number <snd_seq>.

If <snd_seq> is equal to <rcv_seq>, the kernel dispatches the reply message with a sequence number=<snd_seq> and destroys $A_{cr_i}$. Otherwise the kernel fails the operation with an error code NON_EXISTENT_COMMN indicating that the communication activity has already been completed. In either case, the kernel sets <rcv_seq> to null. Inconsistency in the semantics of the reply operation is possible if the sender has either terminated the communication activity or failed but $A_{cr_i}$ has not yet noticed it; in this case the reply operation fails at the sending site whereas it succeeded at the member site. The inconsistency arises due to the stateless nature of the *grp_reply* operation. Another primitive *grp_reply_with_ack* is

[**] We assume a realisation of group communication on top of network multicast.

introduced whereby the success or failure of the operation at the sender's site is reported back to the member site in the form of an acknowledgement. Such an operation may be used by applications which require a confirmed delivery of the reply message.

(ii) $A_{cn}$ does not exist, indicating that currently there is no on-going communication activity from the sender.

The kernel fails the reply operation. For non-critical operations, the member may issue the stateless *grp_reply* primitive which returns an error code REPLY_FAILURE indicating that the error may be due to either the failure of the sender or the completion of the concerned communication activity. For critical operations, the member may issue the *grp_reply_with_ack* primitve whereby the kernel may ascertain the existence of the sender by a simple failure detection protocol using an AYT (synonyom for Are You There?) probe message and return either the NON_EXISTENT_COMMN or the NON_EXISTENT_SENDER error code, the latter indicating that the sender has failed. These well-defined error code enable applications to be structured accordingly.

### *4.3 Concurrent access to the sender's address space*

Access to the sender's address space by recipients is sometimes useful. When the sender is blocked on a group communication, one or more recipients may move large chunks of data to/from the sender's address space. In the one-to-one IPC model, the blocking semantics of the IPC operations guarantees mutual exclusion between the sender and the recipient in accessing the sender's addess space. However, group communication operations violate this exclusion mechanism. Since any recipient of the message in the group may potentially access the sender's address space, it raises a non-deterministic mutual exclusion problem among the recipients. Furthermore, when the message sender is unblocked after the desired degree of delivery is achieved, there may still be recipients in the group which have not completed the "receive-reply" cycle; such recipients may try to access the sender's address space independently later on.

The issue of concurrent access to the sender's address space may be controlled by enforcing a policy for such access when the sender is blocked on a group communication. In the simplest form, a receiver may be prohibited from accessing the sender's space after receiving a group message. Or, the kernel may implement a lock-based access policy to provide increased functionality. In this policy, a group member must acquire a lock before accessing the sender's address space. The lock will be granted by the kernel if the sender is blocked on the group to which the member belongs and the lock on the address space has not been allocated to another process. If the lock is already given to another member of the group, then the lock request is queued up by the kernel until the current holder of the lock releases it.

If a member that has acquired a lock on the sender's address space fails, the lock must be recovered so that the sender may become ready to execute and other members which intend to access the sender's space are given a chance. Since the state of the lock on the sender's address space is maintained at a single point (the kernel at the sending site), a mechanism for one-to-one notification of failures suffices to protect the lock. It may be realised using process aliases inside the kernel. When the kernel concedes the lock to a group member, it creates an invisible alias at the sending site and one at the member site. These aliases engage in a symmetric failure detection protocol by exchanging periodic probe packets[2]. On detecting a failure, the kernel may recover the lock.

## 5. Conclusions

We have examined the reliability issues associated with group communication in a distributed environment. A kernel model with new semantic tools concisely capturing the non-determinism associated with group communication has been presented. Sample programs illustrating how the semantic tools may be used are included in the appendix. We have also described a process alias-based structuring technique for the kernel to handle the reliability problems that may arise during group communication. The scheme works by maintaining a close association between group messages and their corresponding reply messages.

We feel that the characterization of group communication, as described in this paper, will be useful in the design of language(s) for real-time applications as well as lending insight into the implementation of group communication in a system.

## References

1. K.Ravindran and S.T.Chanson, *Process alias-based structuring techniques for distributed computing systems*, Proc. of the 6th IEEE-CS International Conference on Distributed Computing Systems, Cambridge, Mass., May '86, pp.355-363.

2. K.Ravindran and S.T.Chanson, *State inconsistency issues in local area network based distributed kernels*, Proc. of the 5th IEEE-CS Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, Jan.86, pp.188-195.

3. D.R.Cheriton and W.Zwaenepoel, *Distributed process groups in the V Kernel*, ACM Trans. on Computer Systems, Vol.3, No.2, May '85, pp.77-107.

4. W.J. Neilson, and U.M. Maydell, *A survey of current LAN technology and performance,* CINFOR Vol.23, no.3, Aug.'85, pp.215-247.

5. E.C.Cooper, *Replicated distributed programs,* Proc. of the 10th ACM Symposium on Operating System Principles, Vol.19, No.5, Dec.'85, pp.63-78.

6. M.Ahamad and A.J.Bernstein, *Multicast communication in UNIX 4.2 BSD,* Proc. of the 5th IEEE-CS Symposium on Distributed Computing Systems, May '85, pp.80-87.

7. G.Rossi and G.Garavaglia, *A proposal for an improved network layer of an LAN,* Computer Communication Review (ACM SIGCOMM), Vol.16, No.1, Jan.-Feb.'86, pp.13-17.

8. K.Ramamirtham and J.A.Stankovic, *Dynamic task scheduling in hard real-time systems,* IEEE Software, Vol.1, No.4, July '84, pp.65-75.

9. A.J.Frank, et al., *Group communication on net computers,* Proc. of the 4-th IEEE-CS Conf. on Distributed Computing Systems, May '84, pp.326-335.

10. K.White, *An implementation of Remote Procedure Call Protocol in the Berkeley UNIX Kernel,* Technical Report No. UCB/CSD 85/248, June '85.

11. C.J.Walter, et al., *MAFT: A multicomputer architecture for fault-tolerance in real-time control systems,* Proc. of the IEEE-CS Symposium on Real Time Systems, Dec.'85, pp.133-140.

12. E.T.Fathi and N.R.Fines, *Real-time data acquisition, processing and distribution for Radar applications,* Proc. of the IEEE-CS Symposium on Real Time Systems, Dec.'84, pp.95-101.

## Appendix

We present examples to illustrate how applications may be structured by using the semantic tools described in section 3 (we use a `C'-like syntax for the code skeletons).

### A.1 Group synchronisation

In hard real-time systems [8], external events such as a plant variable exceeding a threshold (indicating an emergency) usually requires corrective action to be initiated on multiple sites simultaneously. We present a code skeleton for this type of applications using our model of group communication.

```
function event_notifier()
  { forever
      {
          receive(event_msg, event_monitor_id);
          reply(ack_msg, event_monitor_id);
          status = group_send(event_msg,
                    event_handler_grp, 1.0, time_out);
          if (status.atleast < 1.0)
            {
              if ((status.atleast ≤ LOW_LIMIT) ||
                  (status.atmost ≤ INSUFFICIENT))
                    <Raise alarm or reissue group_send>;
            }
          else ; /* All members have synchronised */ }
  }  /* End of function */

function event_handler()
  { /* A member of the event handler group */
    forever {
          status = grp_receive(grp_msg);
          grp_reply(event_notifier_id, ack_msg);
          <handle event>; } }
```

### A.2 Replicated data bases

We describe how update operations may be requested on a tuple in a replicated data base.

```
function dbase_update()
  { struct msg rep_msg, grp_msg;
    grp_msg.rqst_code = WRITE_TUPLE;
    grp_msg.attributes = <tuple attributes>;
    grp_id = DATA_BASE_GRP;
    deg_del = 1.0;
    repeat {
          status = group_send(grp_id, grp_msg,
                         deg_del, time_out);
          for (; dequeue_reply(rep_msg)
                   != NO_MORE_REPLY ;)
           if (rep_msg.err_code == UPDATE_FAIL)
             <Note down failure>;
           else
             <Note down success>; }
    until ((status.atleast == deg_del) ||
         (++i == MAX_TRIES))
    if (i == MAX_RETRIES)
      <report failure>;
    else
      if (<update failures noted>)
          <recover>;
  } /* End of dbase_update */
```

### A.3 File location

The following is the code skeleton of a client that wishes to locate a file maintained by a file server group.

```
function file_query()
  { struct msg rep_msg, grp_msg;
    grp_msg.rqst_code = QUERY_FILE;
    grp_msg.attributes = <file attributes>;
    grp_id = FILE_SERVER_GRP;
    deg_del = 1.0;  /* Specify the largest value */
    file_found = FALSE;
    repeat {
        status = group_send(grp_id, grp_msg,
                       deg_del, time_out);
        for (j=0; dequeue_reply(rep_msg)
               !=NO_MORE_REPLY ;) {
           file_srvr_pid[j++] = rep_msg[0].process_id;
           file_found = TRUE; } }
    until ( (file_found) || (++i == MAX_TRIES) )
    if (i == MAX_RETRIES)
      <File not available>;
    else
      <Establish connection to the file server>;
  } /* End of file_query */
```