The SOS object-oriented communication service

Mesaac Makpangou, Marc Shapiro

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 801

THE SOS OBJECT - ORIENTED COMMUNICATION SERVICE

Mesaac MAKPANGOU
Marc SHAPIRO

MARS 1988

The SOS object-oriented Communication Service (Le Service de Communication à objets de SOS)

Mesaac Makpangou
Marc Shapiro
INRIA, B.P. 105, 78153 Le Chesnay Cédex, France
tel.: +33 (1) 39-63-54-26
telex: 697 033 F e-mail: mak@corto.inria.fr

Abstract

SOS is a general-purpose, object-oriented distributed operating system, based on the Proxy Principle.

The SOS Communication Service provides flexible communication mechanisms for executing distributed applications with conflicting requirements (e.g multimedia document access, real-time voice transfer, moving image, and reliable office activities), especially with respect to the speed/reliability trade-off. These mechanisms must be efficient enough to encourage development of distributed applications. Futhermore, each application should only pay the price of those mechanisms it really uses.

The SOS Communication Service provides both reliable and unreliable communication; unreliable communication incurs no reliability-related overhead. We provide both unicast and multicast communication. All of these are multiplexed on a single host-to-host transport channel.

Our object-oriented design allows progressive construction of the invocation-level protocols, with extensive re-use of code and design.

Résumé

SOS est un système d'exploitation réparti à objets, basé sur le concept de mandataire. SOS doit permettre l'exécution d'applications réparties de natures différentes, telles le transport de la parole, le transport d'images animées, ou les applications bureautiques. Chacune de ces applications a ses besoins spécifiques, souvent incompatibles (certaines applications nécessitent des communications fiables, alors que d'autres, comme les applications temps réel, préfèrent se passer de la fiabilité).

Le service de communication de SOS offre des mécanismes permettant de supporter efficacement ces différents cas. Ces mécanismes reposent sur un unique canal de transport inter-sites et un protocole de transport très flexible. Celui-ci assure aussi bien les communications fiables que non fiables. Les messages non fiables sont transmis sans coût supplémentaire dû au fait que le système supporte des transferts fiables. Le protocole de transport de SOS supporte aussi la diffusion fiable et non fiable.

L'approche objet permet une construction progressive des protocoles de niveau invocation nécessaires aux applications. Le mécanisme d'héritage offert par l'approche objet permet la réutilisation du code.

1 Introduction

The SOS[1], or SOMIW Operating System, is a general-purpose distributed object-oriented operating system. It is described in [Sha87,Sha86a]. One of its objectives is to provide *flexible* mechanisms for executing distributed applications. An application is a distributed set of entities, co-operating via some application-specific protocol. This protocol may be characterized by its synchronization model, and also by the lower-level communication functionalities that it needs. These functionalities may be provided, either by the communication system, or by the application-specific protocol itself. Placing functionality in the communication system or in the application is a difficult design decision. It depends on the application type and also on the characteristics of the network environment (e.g [Sal81] gives an example of a careful file transfer where reliability is best provided by the application end-points). Another design decision is what assumptions the communication system may make about the applications, e.g concerning their synchronization model.

The SOS network environment is made of local area networks, interconnected by wide area networks. We assume that most office-automation applications are localized in a particular office. In this paper, we restrict the SOS domain to a set of workstations interconnected by a local area network. We assume that this medium provides at least an unreliable datagram transfer, and an garble-free packet transfer. These characteristics are common to most local networks.

Different applications need different lower level communication functionalities, sometimes incompatible. For instance a banking application needs an extremely reliable protocol, whereas a real-time voice transfert will trade off reliability in favor of speed and regularity of delay. Operating system designs in the literature are based on one of the two following categories of communication system:

Dedicated communication systems. A *dedicated communication system* is designed to take full advantage of the specific synchronization model, while optimizing support for the desired functions, by implementing a specific communication protocol. Examples are the Remote Procedure Call of [Bir84], and the VMTP protocol of the V-System [Che86].

The disadvantage of this solution is that the communication system depends on a particular synchronization model (e.g request/reply).

General-purpose communication systems. A *general-purpose communication system* makes pessimistic assumptions concerning the applications. Such a system tries to satisfy any application type. However, this is difficult. To cope with this difficulty, most systems provide very minimal communication mechanisms. For example, the Chorus-v2 System [Zim84] provides asynchronous message transfer. Higher level functionalities are to be implemented by the application above the basic mechanisms.

The disadvantage is that each application must program its own mechanisms (e.g timeouts, acknowledgements, communication errors detection and recovery); hence a greater complexity and a lower amount of re-use. Applications do not benefit from the distribution without a large amount of programmer work.

The SOS Communication Service is a general-purpose communication system. It is designed taking in account the following constraints:

[1]SOS is a subtask of Esprit Project 367, "Secure Open Multimedia Integrated Workstation" (SOMIW). The goal of this project is to construct an office workstation for manipulating, transporting, and using multimedia documents.

1. **Efficient support of various application types.**
   SOMIW requires efficient support for various distributed applications (teleconference, distributed execution, bulk transfer, moving image, voice transport, etc.). Each of them has specific needs.

2. **Transparency.**
   The programmer must deal transparently with distribution support mechanisms.

3. **An application should pay only for what it really uses.**
   The price reliability trade-off is not the same for all distributed activities. For instance, an application which dœsn't need the atomic transactions should not incur any transaction-related overhead.

Therefore the SOS Communication System provides a set of replaceable *protocol objects* built on top of a flexible transport protocol. The object-oriented approach facilitates the progressive construction of this set, while avoiding duplication of code or functionalities.

This paper presents the design and implementation of the SOS Communication Service. We focus our presentation on the SOS Transport Protocol. The presentation is as follows. Section 2 presents an overview of the main concepts of SOS: the *object-oriented approach* and the *proxy principle.* Then we describe in section 3 the invocation-level protocols management. The SOS Transport Protocol is described in Section 4. The final section concludes with the presentation of future research.

2 SOS concepts.

The SOS is built upon a minimal kernel, completed with system services. The kernel provides low-level functions such as virtual memory management, interrupt handling, or context switching. High-level functionalities are provided by system services.

The two novel aspects of the SOS are its *object-oriented approach*, and the use of the *proxy principle* as the basic concept for structuring distributed systems.

2.1 Objects in SOS.

SOS is based on the object-oriented approach (see for instance [Str87,Boo86]). All operating system functionalities are encapsulated within system objects. Every visible entity (file, protocol, multimedia document, window, etc.) is an object. An object has internal data which can only be used or set by its own code. It can be used only by calling one of the procedures of its public interface. An object is instantiated within a localized context (virtual address space). Co-operating objects, across contexts, are organized in distributed *groups.* Communication is free within the same context. A group is a collection of objects which can communicate with one another. A group delimits a communication domain: Cross-context communication is allowed only to objects in a same group.

An object is identified by its *concrete* OID (Object IDentifier). A group is identified by an OID also. The concrete OID is a global unique identifier of an object. An object is addressed using its concrete OID, or one of its group OIDs.

2.2 The proxy principle.

This principle [Sha86b] states that: "In order to use some service, a potential client must first import a proxy of the service (an object representing it) in its virtual address space. The proxy is the only visible interface to the service. The object(s) represented by a proxy, is (are) called its principal(s)".

The SOS is built around this concept. Only a proxy has the right to communicate with a principal. The proxy hides the communication protocol. The choice of the protocol is a private concern of the proxy and its principal(s). A proxy looks like a programmable stub [Nel81].

How the proxy is generated is out of the scope of this paper, as well as how the system finds and installs it. These problems are treated elsewhere [Sha87,Sha88]. We are concerned here with communication problems:

- What communication protocols are available and what is the degree of integration between them?
- How do a proxy and its principal(s) select the protocol they want?
- How dœs the system implementate the needed communication functionalities?

2.3 Communication Service definition.

Let us summarize here, the features of communication in the SOS.

1. Communication is free within a context, by object invocation (procedure calls).
2. All communication, other than within a context, is considered unsafe. It is allowed only within a group.
3. Cross-context communication always takes place between a proxy and its principal(s).
4. Between a remote proxy and its principal, the system uses by default the inter-site extension of cross invocation, the SOS Remote Procedure Call (SOS-RPC) protocol.
5. However the proxy and its principal(s) are free to select any other available communication protocol that they want. This choice takes place at the binding time.
6. Only the communication service has access to the network hardware. There is in each site, one communication server, in its own context.

2.4 Definitions.

We designate by *caller* an object which initiates an invocation. Generally this object is a proxy. We designate by *callee*, the target of the cross invocation (the principal), i.e. the object referenced by the caller's `trapReference`[2]. The pair *(caller, callee)* defines an *application connection.*

An ***invocation-protocol*** is a protocol whose functionalities are like that of the OSI session or presentation levels [Zim80]. Such a protocol is above the SOS Transport Protocol. A *protoObject* is an object encapsulating such a protocol. An *invocation-level connection* or simply a *connection*, is a pair of protocol objects across the network.

[2] This is a capability to the callee. It is filled at binding time.

3 Invocation-level protocol management.

The goal of the SOS Communication Service is to provide a set of efficient communication protocols, such that each application may find the adequate communication protocol it wants. We construct this set progressively. The SOS Communication Service dœs this as follows: first, we define a host-to-host transport protocol, which provides lower-level functionalities any invocation-level protocol might need. Second, we define a general scheme of remote invocation. Any invocation-level protocol should respect this scheme. Common functionalities required by this scheme, are provided by a base object. These functionalities include connection management, since in SOS, communication between a caller and its callee are connection-oriented. Third, any protocol object inherits the common functionalities provided by the base object. Our design allows greater of code, while avoiding functionalities redundancy. Examples of invocation-level protocols include the remote procedure call protocol, the transaction-oriented protocols, the synchronous message passing, the asynchronous message passing, and multicast-oriented protocols.

Hereafter, we discuss one of the common functions provided by the base object: connection management. We first state what an invocation-level protocol looks like.

3.1 What an invocation-level protocol looks like.

An invocation-level protocol is defined by two objects: the `protoObject`, and the `protocolManager` object. The protocol object encapsulates functionalities specific for this protocol. It inherits common functionalities from the basic protocol object. The protocol manager object has three functions:

1. **Establishing connections.**
   When asked, the protocol manager associates an `invocation-level connection` to a given application connection. The following section describes how this is carried out.

2. **Managing protocol resources.**
   Each connection has a descriptor in a connection table, made of the caller's and calleés OIDs, plus the address of the local protocol object for this connection.

3. **Dispatching invocation messages concerning its protoObjects.**
   When the transport level receives an invocation message, it passes it to the concerned protocol manager. It is the role of the protocol manager to dispatch this invocation message to the appropriate protocol object.

There is one protocol manager per station, per protocol type. Protocol manager and protocol objects are instantiated in the Communication Service context.

3.2 Connection management

The cost of the connection management is mainly due to the connection establishment and release protocols. Our implementation needs no extra message exchange in order to establish or release a connection.

3.2.1 Installation of a connection

In SOS, The caller and its potential callee are permanently connected by the trapReference binding. However the connection is virtual, until the caller attempts to call its partner. At this time, the Communication Service sets up its connection information, using the protocol chosen at the binding time.

Connection installation is carried out in two steps. When a caller invokes its remote partner, the kernel detects that the callee is located within a remote station (because the trapReference dœsn't designate an object instantiated in a context of the caller's machine). The kernel requests the selected protocol manager, to establish the connection. The protocol manager dœs the following actions: First, it allocates a protoObject for this connection. It makes the protoObject's trapReference point to the caller, and conversly. Then it registers this connection's descriptor in the connection table. This completes the first step.

The allocated protocol object is now ready to relay the caller's call, but it still has no remote partner. At this point, the kernel retries the call; the protoObject allocated at the first step is cross invoked. Since it dœsn't know its remote partner, it asks its protocol manager to relay this call. This request starts the second step of the connection installation.

The caller's protocol manager invokes its counterpart, in the calleés station. The invocation message contains an establishment request containing the caller's and calleés references, plus the call arguments.

The receiving protocol manager performs the following actions:

- First, it checks if this connection is not already allocated. If not, it allocates a protoObject for this connection. It sets this protoObject's trapReference to refer to the callee, then the calleés trapReference is made to refer to the allocated protoObject.
- Second, it registers this connection descriptor in the connection table.
- Third, it connects[3] this protoObject to its remote partner. This operation changes the state of a protoObject, registers the application connection i.e caller's and calleés OIDs.
- Finally, it passes the call to the new protoObject. At the completion of the call, it sends a *connection response* to the requesting protocol manager. If this call has returned a result, this is piggybacked with the connection response message.

At the reception of this connection response, the requesting protocol manager connects the caller's protoObject to its remote partner. Then returns any result, to the caller. This completes the second step of the connection establishment.

Figure 1 shows the relation between caller, callee, and the pair of protoObjects used to connect them.

3.2.2 Releasing a connection.

An invocation-level connection is installed for an unlimited period. As long as the system has available resources, the connection is maintained. If a protocol manger needs resources, it may

[3] A protoObject passes all calls coming from the caller, to its protocol manager as long as this operation is not executed.

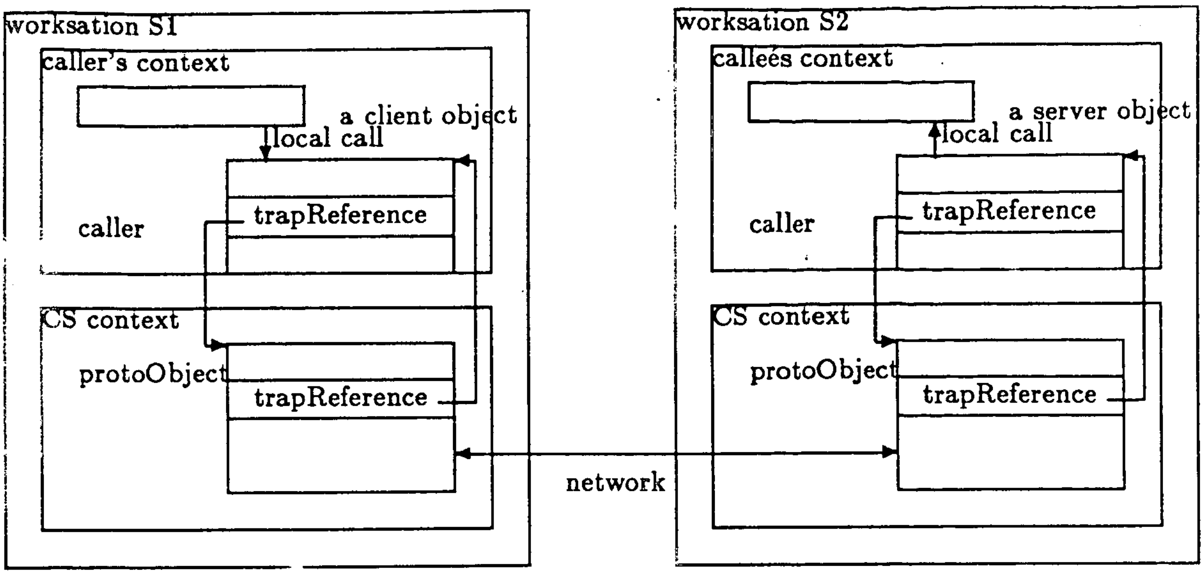

Figure 1: caller/callee connection scheme.

release any unused connection. It is not necessary to inform the other end point. It needs only to re-establish the original binding information.

3.2.3 Conclusion.

To summarize, a connection is installed transparently at no cost, when it is really needed. It may be freed at any time if it is unused. The two extremities of the connection are installed separately and may be released separately too. A connection might be reinstalled later if the caller needs it again.

4 The SOS transport protocol

Reliability may be supported either in the transport level, or by each reliable invocation-level protocol. We think that providing reliability at the transport level simplifies the design of reliable invocation-level protocols. However, one must care that, invocation-level protocols that do not need reliability support must not pay for it.

The SOS Transport Protocol behaves either as reliable, or an unreliable transport protocol. A reliable transport protocol must support at least: the detection and recovery of lost messages, the detection and suppression of duplicate messages, and the detection and recovery of crashes and network failures. In order to provide these functions, reliable transport protocols manage acknowledgements. The SOS Transport Protocol uses the following acknowledgement strategy:

- A receiver is not concerned with lost packet detection. It acknowledges what it receives and makes no assumption on how the sender behaves.
- The sender of a reliable message is responsible for its possible retransmission.

- A receiver does not retain the table of all previous messages it has received. It uses very little information to deal with duplicate detection.

- A compact acknowledgement concerning a sequence of messages is piggybacked on normal messages.

Our solution has the advantage that a sender maintains a single outgoing flow of data for both reliable and unreliable transport messages. Unreliable transport messages are transferred, without paying the price of the reliability support, and without exceptional treatment by the receiver. Our solution also allows an implementation of a reliable multicast, with acknowledgements piggybacked on normal message flow; unreliable multicast messages may be transferred in the same data flow as the reliable ones, without paying for the reliability support.

The following subsections present this protocol in depth. We first recall the SOS transport requirements, then we describe and justify our design decisions. Subsection 4.3 presents the basic mechanisms and principles. Subsection 4.4 describes the multicast transport management. Subsection 4.5 deals with failure detection and recovery.

4.1 Requirements

The SOS Transport Protocol meets the following requirements:

1. Flexibility, to allow efficient and easy implementation of different invocation-level protocol objects.

2. Support for both reliable and unreliable transfer. Providing reliability at the transport level simplifies the design of certain invocation-level protocols; however, those that do not need this support must not pay for it.

3. Support for both point-to-point and multipoint transfer.

4. Optomization of the most frequently-used protocol, SOS-RPC, while preserving the independence of the SOS Transport Protocol.

4.2 Design decisions

4.2.1 Datagram oriented.

The SOS Transport Protocol multiplexes all higher-level messages in a single outgoing stream of data to every other host. This stream is managed by a single *communicator* object in the Communication Service context of each machine. Each chunk of data is carried on the single flow as a separate datagram.

4.2.2 Preallocated fixed message size

Fixed-size preallocated buffers management is cheaper than variable-size. RPC call and reply messages are small, and generally fit in a single network packet. Therefore the maximum size of a transport message, exchanged between two communicators is given by the size of the network packet.

4.3 Basic mechanisms.

We now describe the mechanisms provided by the SOS transport protocol, for reliable and unreliable transfer.

4.3.1 Message identifier

Every transport message has an identifier allocated by its sender. We assume that, at any time, all messages sent by a specific communicator, which have same identifiers, are duplicates: a communicator generates only one flow of messages. These are numbered sequentially, but not all of them are necessarily for the same destination. The numbering space is large enough to guarantee that two different messages sent by a same communicator can't have the same identifier, during a certain amount of time, sufficient to acknowledge and to confirm the oldest one.

A transport message identifier (coded on 32 bits, numbered 31 to 0, high-order to low-order) is made of three components:

- The *ackRequested* boolean (bit 31). If this is set, then an immediate acknowledgement is requested for this message.
- An *activity number* (30–24), is incremented after any recovery from a failure (see §4.5).
- An *ordinal* (23–0). This is the sequence number of this message within this activity. After each recovery, the ordinal is restarted at one.

4.3.2 Acknowledgement management

A communicator acknowledges all the messages it receives. However the receiver makes no assumption concerning how the flow of messages are numbered. The only thing it needs to know at any time, is what is the smallest valid message identifier it can accept from its partner.

A receiver piggybacks acknowledgements on normal messages. We acknowledge a sequence of messages at a time. A sequence is defined by its lower and upper bounds. Ack's are themselves acknowledged. Let us look at an example: suppose that the communicator C2 receives successively from Cl messages numbered 11, 12, 30, 14, 24, 16, 13, 20, 21, 22, 23, and 25. Suppose futhermore that no acknowledgement has been sent yet. C2 piggybacks on the next message to Cl, the *oldest sequence*, the pair (11–14) i.e. the first contiguous sequence of yet unacknowledged messages. C2 receives no new message from Cl. Within the following message destined for Cl, C2 will piggyback (16–16), and if it still receives nothing from Cl, the next message to Cl will contain (20–25) etc.

This example shows that, under normal conditions, this strategy costs no extra message. It also shows that C2 may acknowledge message 16, even if it has not received message 15, which might be lost, or else addressed to another host. This example also shows that we have two types of aknowledegements: ones that are already sent, but themselves unacknowledged, and others, which are waiting to be sent.

Each communicator maintains two lists for acknowledgement management. The *outGoingAcks list* contains descriptors of acknowledgements to be sent. When a communicator receives a *valid* transport message, it allocates an acknowlegement descriptor[4] for this message in this list, before

[4]This contains the message identifier and the sender identifier.

passing the message to its higher level destination. When this communicator sends an acknowledgement, it moves all concerned descriptors from this list to the *ackAwaitingAck list*. The *ackAwaitingAck list* contains descriptors of acknowledgements which were sent, but not yet confirmed. An acknowledgement descriptor remains in one of these two lists until the sender confirms its reception.

The confirmation mechanism is simple. Let us continue the above example. Suppose that C1 has sent message 15 to a communicator C3, messages 17 and 18 to C2, and that it has received the first acknowledgement sent by C1. Now C1 sends message 26 to C2. This message will contain, an acknowledgement request for message 16. 16 is the *oldest unacknowledged message* sent to C2. When C2 receives message 26, with 16 as the *oldest unacknowledged message*, it considers that acknowlegements concerning message older than 16 are confirmed and can be removed.

To summarize, under normal conditions, reliability support costs no extra message exchange. We will now see what happens when a message is lost, or there is no message available for piggybacking.

4.3.3 Lost messages

The sending communicator maintains a queue of outgoing messages not yet acknowledged. Only messages to be sent reliably are in this queue. An entry contains: the message identifier, the acknowledgement mask (used for multicast, see §4.4), a pointer to the transport message, and (possibly) an event provided by the higher-level sender (see below).

When a sender receives an acknowledgement, it deletes all concerned messages from the outgoing message queue. For each, if the higher-level sender has provided an event, it sets this event; then it frees the descriptor.

A communicator maintains one time-out per partner. When the oldest unacknowledged message, sent to a particular partner changes, its corresponding time-out is set. If a time-out expires, the oldest unacknowledged message destined for the concerning communicator is re-sent. While resending a message, the sender requests an immediate acknowledgement. Thus, if the receiver has no available normal message, it may send a specific message containing only acknowledgements. This is the only case where an extra message is used.

By queueing only messages to be sent reliably, unreliable messages incur no overhead. Since messages are numbered sequentially, acknowledgements of unreliable messages will be confirmed even if the sender dœsn't receive them. If these acknowledgements are received, they are ignored.

4.3.4 Detection and supression of duplicate messages

In addition to the `outGoingAcks` and `ackAwaitingAck` lists, a communicator maintains an array of oldest valid messages, one for each remote host. When a communicator receives a message, the following cases are possible:

1. Its identifier is less than the oldest valid message from the same host.

   In this case, this message is a duplicate of an already received, acknowledged, and confirmed message. This message is discarded.

2. There is an acknowledgement for this message in the `outGoingAck` list.

In this case, this is a duplicate of an already-received message for which the acknowledgement was not yet sent. The control information is retrieved, then the message is discarded, and an immediate acknowlewedgement is sent.

3. There is an acknowledgement for this message, in the `ackAwaitingAck` list.

   In this case the previous acknowledgement was probably lost. The communicator retrieves control information, discards the message, and sends an immediate acknowlegement to the sender.

4. The identifier is greater than the oldest valid, and there is no acknowledgement, either in `outGoingAcks`, or in the `ackAwaitingAck` lists.

   This message is not a duplicate message. The communicator allocates an acknowledgement descriptor for it in the `outGoingAcks` list. Then, it passes this message to its higher-level destinator.

4.4 Multicast.

The Communication Service features a lower-level multicast between communicators. We assume that the sender of a reliable multicast knows the set of receivers.

From the sender point of view, the only difference between a reliable point-to-point communication, and a reliable multicast, is that the message descriptor contains a significant acknowledgement mask. If there are N SOS hosts in the network, the acknowledgement mask is an array of N bits. Each host has its corresponding bit. A mask's bit is set to 1, if the corresponding host is concerned by the outgoing multicast.

A receiver makes no difference between a unicast and a multicast message. The receiver still behaves strictly as described in §4.3.

When the sender receives the acknowledgement of a reliable multicast message, it sets to 0 the bit corresponding to this receiver in this messagés acknowledgement mask.

When the acknowledgement mask becomes zero, it deletes this message from the outgoing message queue, and dœs similar treatment as for a unicast reliable message.

This implementation of the reliable multicast, costs no extra message exchange between partners.

When a communicator sends an unreliable muticast, it dœsn't allocate any descriptor for this multicast. Hence, there is no difference between the treatement of an unreliable unicast and an unreliable multicast for both sender and receiver.

4.5 Failure detection and recovery.

The transport protocol is concerned with two types of failure: a broken network, and the crash of a station (this paper dœs not consider network partitions). We assume that a communicator loses its state if its host crashes. In the absence of stable storage, one way to reconstruct the state is to obtain the information from the other communicators. We first describe the communicator initialization protocol. Then we describe how to detect a crash. The final part describes the recovery procedure.

4.5.1 Initializing communicator state.

When a communicator first comes up or is restarted, it requests its previous state from the other (already-running) communicators. This is done by a reliable multicast to the group of communicators. Any communicator which receives a startup message replies immediatly with:

1. The presumed state of the caller. This includes especially its last ***activity number.***

2. The calleés state, especially the initial value of the ***oldest message*** the caller might receive from this communicator.

Non-responding stations are assumed down. The state of the communicator is constructed using the replies it has received. In particular, the new activity number must be greater than all values provided by the running partners. The table of oldest valid messages is initialized with the values provided by the partners.

This initialization might seem expensive. However, this cost is acceptable since this is done only once for each host initialization or reinitialization.

4.5.2 Crash detection.

A crash can be detected, either during message transfer, or by a probe mechanism.

1. **During transfer.**
   Section 4.3.3 describes how to deal with lost messages. After a certain number of unsuccesseful retries, the destination is diagnosed as down.

2. **The probe mechanism.**
   The SOS needs to know if a remote station is still running[5]. A communicator periodically sends a probe message to all other machines. If a communicator dœsn't receive a probe from some partner during a period of certain time, it diagnoses that this partner is down.

When a crash is diagnosed, the communicator dœs the following actions:

- It deletes, from the outgoing message queue, all messages destined for the failed communicator, and returns an exception to their higher-level senders.

- It informs all protocol managers that the failed station is no longer accessible.

- It stops sending any message to the failed communicator.

- It saves the activity number currently in used by the failed station.

4.5.3 Failure recovery

A failure diagnosis may in fact be correct or incorrect, due to transient conditions or to an error of the diagnoser. Consider the two cases:

1. **The remote communicator failed.**
   In this case, when it restarts, it executes the reinitialization protocol (see §4.5.1).

[5]The most important reason is the existence of dependencies between remote objects. Some objects need to be informed when related objects crash.

2. **The communicator did not fail.**
   Applications on the diagnoser's side have already received signal of the failure. To ensure consistency, we force the remote communicator into a failed-like state.

   When a message from a presumedly failed communicator is received, we reply with a restart request (containing the same information as the reply to a startup message).

   In response, the presumedly failed communicator deletes from the outgoing messages queue all messages sent to us, and generates an exception to each sender. It also increments the *activity number* and resets the *ordinal.* Finaly, it updates the value of the oldest message it might expect from us.

   Incrementing the *activity number* dœsn't affect communication with other communicators.

5 Conclusion

The SOS.V2 prototype (second version of the SOS system) is currently distributed to the SOMIW partners. This prototype integrates all the features described in this paper. We haven't yet perfomance measures. However, we are optimistic, mainly because our design and implementation try to minimize the cost of connection management and reliability support. Some design decisions we have made (e.g fixed-size preallocated transport buffers, the generalization of piggybacking strategy, and the implicit flow control) are commonly cited [Wat87,Bir84,Zwa85] as succeptible to improve performance of protocol which did such choices.

Unlike most communication systems, the SOS Communication Service provides neither a specific communication protocol using the knowledge of applications, nor a communication protocol on top of an existent "standarised transport protocol". It may seem that these decisions preclude the performance optimizations available to specific transport protocol [Che86,Bir84]. This is not the case. Most optimizations provided by a specific transport protocol concern connection management, acknowledgement management and flow control [Fle78,Bir84,Che86,Wat87]. In general these optimizations assume the request/reply synchronization model of communication between the application entities [Bir84,Che86]. We argue that, in the case of request/reply, the cost of the reliability support in SOS is, comparable to that of most specific communication protocols [Che86,Bir84]. In addition, the SOS Communication Service supports unreliable transfer without paying for the reliability support. It also supports non-request/reply communication protocols.

The future work includes the specification of the network interconnection, and the exploration of some application facilities, necessary for various application-specific protocols.

Acknowledgements

Our acknowledgements to our colleagues in the SOMIW project, especially V. Abrossimov, P. Gautron, Y. Gourhant, S. Habert, J.P. Lenarzul, L. Mosseri, V. Prevelakis, and C. Valot, and to Azzedine Mzouri for their suggestion during the specification of the SOS Communication Service.

References

[Bir84] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.

[Boo86] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[Che86] David R. Cheriton. A transport protocol for the next generation of communication systems. In *Proc. of the SIGCOMM'86 Symposium on Communications Architectures and Protocols*, pages 406–415, New York NY (USA), August 1986.

[Fle78] John G. Fletcher and Watson Richard W. *Mechanisms for a reliable timer-based protocol*, pages 271–290. North-Holland, Amsterdam, The Netherlands, 1978.

[Nel81] Bruce J. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University, May 1981.

[Sal81] J. H. Saltzer. End-to-end arguments in system design. In *Proc. 2nd. Int. Conf. on Distributed Computing Syst.*, pages 509–512, Versailles (France), April 1981.

[Sha86a] Marc Shapiro. SOS: a distributed object-oriented operating system. In *2nd ACM SIGOPS European Workshop, on "Making Distributed Systems Work"*, Amsterdam (the Netherlands), September 1986. (Position paper).

[Sha86b] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, IEEE, Cambridge, Mass. (USA), May 1986.

[Sha87] Marc Shapiro, Vadim Abrossimov, Philippe Gautron, Sabine Habert, and Mesaac Mounchili Makpangou. *Un recueil de papiers sur le système d'exploitation réparti à objets SOS.* Rapport Technique 84, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), May 1987.

[Sha88] Marc Shapiro and Sabine Habert. Programmer's manual for sos prototype version 2/3. 1988. To appear.

[Str87] Bjarne Stroustrup. What is "object-oriented programming"? In G. Goos and J. Hartmanis, editors, *Proc. European Conf. on Object-Oriented Programming*, Springer-Verlag, Paris (France), June 1987. Lecture Note in Computer Science no. 276.

[Wat87] Richard W. Watson and Sandy A. Mamrak. Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

[Zim80] H. Zimmerman. OSI reference model – the ISO model of architecture for open systems interconnection. *IEEE Trans. Commun.*, COM-28(4):425–432, April 1980.

[Zim84] Hubert Zimmermann, Marc Guillemont, Gérard Morisset, and Jean-Serge Banino. *Chorus: a Communication and Processing Architecture for Distributed Systems.* Rapport de Recherche 328, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), September 1984.

[Zwa85] Willy Zwaenepoel. Protocols for large data transfers over local networks. In *Proc. 9th. Data Communications Symposium*, September 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique