TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Design and implementation aspects of remote procedure calls

Citation for published version (APA):
Coenen, J. A. A., van de Sluis, E., & van der Velden, H. A. A. M. (1990). *Design and implementation aspects of remote procedure calls*. (Computing science notes; Vol. 9018). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1990

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 13. Apr. 2024

Design and Implementation Aspects of Remote Procedure Calls

by

J.Coenen E. van de Sluis E. van der Velden

90/18

November, 1990

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Design and Implementation Aspects of Remote Procedure Calls

J. Coenen
E. van de Sluis
E. van der Velden
Eindhoven University of Technology

November 30, 1989

Abstract

This report is not intended as a survey in the field of remote procedure calls. Neither do we claim it to be complete in some way or another. What we do claim is that we present the reader the major principles of remote procedure call design. We start off by investigating those principles that seem to be most generally applicable. These principles are illustrated by a number of rather different, but equally beautiful examples. One example is the Rajdoot protocol of Shrivistava and Panzieri, that is in some respect complete. Another example is the REX protocol of Otway and Oskiewicz, that perhaps is less complete, but certainly not less elegant. Although these two protocols originated from different starting points, they do share some important characteristics. These common characteristics emerged from the common design principles discussed in the introduction.

1 Introduction

One of the main issues designers of distributed systems are faced with is the choice of appropriate communication primitives. The computers forming a distributed system normally do not share primary memory, so communication via shared memory techniques such as semaphores and monitors is generally not applicable. Instead message passing in one form or another is used.

In distributed systems most communication takes the form of a request message from a client to a server, followed by a reply message from the server back to the client. From the client's point of view, sending a message to a server and then waiting for a reply looks very much like calling a procedure and then waiting for it to finish.

The similarity between this communication model, called the client-server model and the well-known procedure mechanism has led towards the introduction of remote procedure calls (RPC) as a message passing primitive in distributed systems.

In this report the key issues concerning the design for an RPC are discussed, together with a number of RPC implementation examples. Chapter 1 discusses these key design issues. In chapter 2, the RPC mechanism for the Cambridge Ring Local Area Network is considered. As a second example of an RPC implementation, the Rajdoot RPC mechanism is discussed in chapter 3. In chapter 4 the REX protocol, which uses the RPC mechanism, is described. Also a similar protocol, called Cedar, will be briefly discussed. In chapter 5, some concluding remarks are given.

2 Key issues for RPC design

The design issues for any RPC system can be divided into four broad categories (cf. [Tan88]):

1. Interface design

2. Client design

3. Server design

4. Protocol design

In the following sections we will deal with each of these categories seperately.

2.1 Interface design

2.1.1 Transparency

The purpose of the RPC mechanism is to give the caller the illusion that it is making a local call. There are, however, some difficulties involved with obtaining full transparency.

The principal problem occurs with the parameter passing. Passing parameters by value is easy, but problems arise when the language allows parameters to be *passed by reference*. For a local call, a pointer (the address of the parameter) is normally passed to the called procedure. Obviously, this strategy fails completely for a remote procedure call.

One possible solution is to replace the call-by-reference mechanism by a *call-by-copy/restore mechanism*. The call-by-copy/restore mechanism locates the item that is pointed to and passes it to the (remote) procedure. The called procedure puts the item in its local memory and is able to access it in the usual way. Whenever control is returned to the caller, the data item is sent back to the caller, which uses it to overwrite the original reference parameter.

Although this copy/restore mechanism frequently works well, it can fail in certain pathological situations. Consider for instance what may happen if very large datastructures are passed as reference parameters, using the copy/restore mechanism. If these parameters are too large to fit into a single message, then this will at least introduce considerable communication problems.

Many RPC systems get rid of the whole problem by prohibiting the use of reference parameters, pointers and procedure or function parameters in remote calls. Such a design decision makes the implementation easier, but the transparency is lost, because the application rules for local and remote calls are different.

A similar situation occurs with *exception handling*. Using an RPC mechanism, three different types of exceptions can be distinguished:

- exceptions raised during execution of the (remote) procedure call (e.g. division by zero),
- exceptions raised because of a server failure (the server crashed during execution of the call),
- exceptions raised because of communication failures (e.g. messages are corrupted during transportation).

To maintain transparency, the client process should only be concerned with the first type of exceptions, since this type is not different from the local case. The second type has to be dealt with by the RPC mechanism, underlying the call made by the client process (see also section 2.2.1). Finally, the third type of exceptions can be handled by the communication protocol underlying the RPC mechanism. There are well-known mechanisms (based on checksums and sequence numbers, see e.g. [Tan88]) that enable a receiver to treat messages that arrive out of order, corrupted or are copies of previously received messages.

2.1.2 Binding

In the Client/Server model (which is recognised in distributed systems based on the RPC paradigm), binding leads to a relation between client and server, and facilitates the exchange of request/reply messages between the two. Binding requires that the (logical) name of the intended server is known. It follows that a mechanism (a mapping) is required to map names to addresses. This naming problem in distributed systems is well-known. Names are needed to (uniquely) identify

servers but are often required to be location-independent (for transparency reasons). It is therefore that a distributed or centralized *name server* has to be used.

A server can now make its service(s) available to clients by exporting them to the name server. A client wishing to use a service imports it by communicating with the name server, thus establishing a *binding* between the client and the server. Precisely how servers export their services, and how and when rebinding works (e.g. after a server crash) are all issues that have to be considered.

Beside translating from (logical) names to (physical) addresses the name server must also check whether a client is allowed to use a particular service. In this case the name server becomes also an *access control* server.

Note that binding only has to take place when the client makes its first remote call. From this point on the client knows how to locate the server, so binding is not required for subsequent calls (provided no crashes occur). So after the first call the client can keep the server's address in local memory (e.g. local cache, for efficiency) for subsequent calls. Note also that the distribution of names by the name server implies that the name server has to prevent any name inconsistencies (e.g. non-unique or ambigue names) to occur.

As a final remark on binding it can be said that the name server can also be used for encryption-based secure (RPC) communication. The design of such a communication protocol, built as part of an RPC package, is described in [Birr84]. Here the analogon of the name server is the *authentication server*. When a client wishes to communicate securely with a server, they negotiate with the authentication server to obtain a shared, secret, *conversation key*. This conversation key is used to encrypt/decrypt subsequent calls/replies between the client and the server.

Finally, *unbinding* is simply the act of disconnecting (releasing) a binding. However, if the binding process was also used for access control or authentication it must be ensured that all copies of access rights or conversation keys are destroyed. This revocation of capabilities might be very difficult in case of local caching or if the client has distributed its capabilities in an uncontrolled way.

2.2 Client design

2.2.1 Server crashes and time outs

In this section we consider problems caused by server crashes. Consider what will happen if the server crashes after carrying out a request, but before sending a reply. To deal with this situation, at least three possibilities are available to the client.

1. *Wait forever for the reply.*
   This approach is similar to what happens when a program calls a local procedure containing an infinite loop. Manual intervention is required to kill the program. If semantic transparency is a main design issue, this approach can be considered.

2. *Time-out and raise an exception.*
   In this approach the fault-recovery is done by a higher-layer entity. It is analogous to what happens if a called procedure gets a memory protection error or tries to divide by zero.

3. *Time-out and retransmit the request.*
   In this approach the fault-recovery is done by the client itself. The client usually has to rebind (possibly to another server) and try the call again. In this approach it is possible that the operation will be carried out more than once.

Operations that can be repeated over and over again, each time yielding the same results without any side effects are said to be *idempotent*. If all operations could be carried out in an idempotent form, then the third approach is obviously a good choice.

However, if all services are made idempotent, it is rather difficult to provide servers with arbitrary services (e.g. a server cannot easily provide an increment operation).

In view of this kind of problems, a classification for the RPC semantics can be introduced: [Nel81] [Rajdoot]

1. *At least once semantics.*
   A normal termination (i.e. a reply message from the called service is received) implies one or more executions at the called server. This semantics is not suitable for operations that are not idempotent.

2. *Exactly once semantics.*
   A normal termination implies exactly one execution at the called server.

3. *At most once semantics.*
   This is the same as exactly once semantics, but in addition calls that do not terminate normally do not produce any side effects.

Choosing appropriate fault-tolerant capabilities and semantics is one of the most important design decisions to be taken in RPC design. The goal of fully transparent RPC semantics is considerably complicated by the possibility of crashes that affect the server but not the client.

2.2.2 Client crashes and orphans

Now consider what may happen when a client crashes after starting an RPC. If the client resumes execution after recovery by reissuing the call to the same server, it is possible that a server can receive multiple call messages for a single invocation by a client, thereby causing superfluous, undesirable and possibly interfering executions. Such executions are referred to as *orphan* executions, since they have no clients (parents) waiting for their results.

Orphans can cause trouble in a variety of ways. At the very least they waste CPU time and other resources, or may even lock them. In addition, results sent back by orphans may cause confusion, because of the possible interference. This is especially troublesome in case of operations which are not idempotent.

From the above discussion it should be clear that some provision for orphan detection and killing must be available. How this is done, may differ for different implementations of the RPC mechanism.

One method is to have each server periodically check to see if the client that started the current computation is still interested. A variation on this idea is *dead man's handle*. A client is expected to poll a server working for it periodically. If a poll fails to come in on schedule, the server just kills all computations for this client.

A different approach is to program all clients to log all RPCs on stable storage before making them. When a client reboots after a crash, it checks to see if there were any servers working for it, and if so, tells them to stop. This solution is expensive because writing a log for each call about doubles the cost of each RPC.

Another, more sophisticated, approach to orphan detection and killing is described in chapter 4, when the Rajdoot RPC mechanism is discussed.

No matter which method is chosen for detecting and killing orphans, there is always the danger an orphan will be in the middle of a critical section at the time it is killed, or that it holds many locks on resources. In this case, killing the orphan can lead to race conditions and deadlocks.

2.3 Server design

In this section, we briefly discuss two issues concerning server design. The first issue is parallelism, the second is a mechanism for fault-tolerance. Two issues concerning server design will be discussed briefly in this section. The first issue is that of parallelism and the second issue is that of a mechanism for fault-tolerancy.

2.3.1 Parallelism

At one extreme, whenever a request comes in for execution of a server procedure, a new process is created and the procedure is run as part of that process. If other requests come in from other clients before the first one is finished, more processes are created and run independently in parallel. At the other extreme, there exists a single process associated with each server procedure. If a second request comes in before the first one is finished, it is queued and must wait for its turn.

The first approach bears the burden of process creation on each RPC, but allows (quasi) parallel execution of RPCs. The second is simpler and faster if there is only one call at a time but does not allow any parallelism if there are multiple calls.

The choice between these two approaches will heavily depend on the particular function of the server.

2.3.2 Fault Tolerance

In this section a scheme for fault tolerant RPC is outlined as presented in [Yap88].

In the proposed scheme, fault tolerance is provided by having copies of a service reside on multiple (server) nodes. Each copy is known as an incarnation. The incarnations are organized in a linear chain. For the i^{th} incarnation the $(i+1)^{st}$ incarnation forms its backup. A service request is made by the client (caller) to the primary incarnation, which is the first copy in the chain that has not failed. All other (active) incarnations in the chain are called secondary incarnations.

There are four types of messages in the proposed scheme. A *call* message invokes an RPC call. A *result* message contains the return values of an RPC call. A *done* message is sent to inform a secondary incarnation that the call is completed. These three types of messages require their recipients to acknowledge by means of an *ack* message.

We now discuss how the scheme functions, in case when there are no failures and under failure conditions.

Execution without failures On receiving a call message, the primary incarnation propagates this message to its immediate subordinate. In this way, the call message is propagated to all secondary incarnations. Acknowledgements are then propagated back through the chain of incarnations. On receiving an acknowledgement to the call message it has sent, an incarnation performs the requested service. After completing a call, a secondary incarnation waits for a message from its immediate superior. The primary incarnation will send a result message to the caller and waits for an acknowledgement. On receiving this acknowledgement, the primary sends a done message to inform its immediate subordinate that the call is completed. This message is propagated to all other secondary incarnations and acknowledgements are propagated back.

Execution with failures After sending an RPC message to the primary incarnation, the caller waits for an acknowledgement. If before this acknowledgement is received, the caller detects that the primary has failed, the same RPC message is sent to the primary's immediate subordinate. If this backup incarnation has already received a similar call message from the old pimary, it acknowledges the message, but takes no further action.

If the primary crashes after sending the request acknowledgement, the caller takes no action. The immediate subordinate of the crashed primary will take over its role and will eventually send the result to the caller.

If a secondary incarnation fails before it acknowledges a call message, its immediate superior will detect the failure and send the same call to the immediate subordinate of the failed incarnation. Between sending the call message and sending the done message, an incarnation need not be aware of its immediate subordinate's status. If an incarnation fails to acknowledge the done message, this message will be sent to its immediate subordinate.

It is claimed in [Yap88] that with this scheme a fault tolerant RPC mechanism is established, although it is possible that under certain circumstances the client/caller may receive more than one copy of the (correct) result.

2.4 Protocol design

For an RPC mechanism, the main issue with respect to protocol design is achieving correctness and high performance, using some kind of interprocess communications facility. Such a facility can be based on either of the following two major approaches: one which favors the provision of a connection-oriented or "virtual circuit" interface (e.g. that of a X.25 network) or one which favors the provision of a connection-less or "datagram" interface.

The protocol supporting a virtual circuit interface generally implements features such as end-to-end acknowledgements, flow control and detection and recovery from various errors. Moreover, the protocol guarantees that messages are kept in sequence, so they are delivered in the same order as they were sent.

A datagram service, instead, provides its user processes with rather primitive operations for transmitting and receiving individually addressed messages. Each message is encapsulated as a distinct data object, termed a datagram. Each datagram is transmitted and received independently of other datagrams, that is to say, the datagram service does not guarantee sequenced and ordered delivery of distinct datagrams. Flow control and end-to-end acknowledgements are also not implemented and the service does not provide the facility of guaranteed deliverance of datagrams.

Since reliable communication facilities would obviously simplify RPC protocol implementation, it may appear reasonable to base an RPC implementation on a virtual circuit based service. However, the maintenance of state information and the messages needed to implement flow control and end-to-end acknowledgements introduce overhead that may very well result in a considerable reduction of the communication bandwith available on the network. A simple datagram service might be much more efficient. In [ShriPan] it is furthermore noted that from the view of the client the acknowledgement is not the affirmation that its message (the call) has been delivered, but that the requested service has been performed. This is implicit in the reply (result) message from the server. In [ShriPan] it is therefore concluded that especially in case of local area networks (providing a rather high reliability) datagrams certainly provide an attractive alternative to virtual circuits.

3 The Cambridge Ring Local Area Network

This chapter discusses a remote procedure call implementation for the Cambridge Ring Local Area Network. For a more comprehensive discussion of the Cambridge Ring LAN we refer to [ShriPan]. The protocol is also used as a basis for the remote procedure call implementations used in the distributed UNIX system built at Newcastle. The outline of this section is as follows. We start with a summary of the characteristics of the protocol. Then we discuss the protocol, followed by a survey of the clock management needed for the implementation of this protocol.

3.1 Characteristics

The main characteristics of the protocol are:

- exactly once semantics are supported (see section 2.2.1),
- a sequential execution model is used (cf. section 2.3.1),
- the implementation is based on a datagram service.

3.2 The protocol

In absence of node crashes the RPC mechanism behaves as follows. The client sends its call request with a sequence number and waits for the reply. Any exceptions during the send phase are dealt with by timeouts and retransmission with the same sequence number.

The server receives a call request (discarding further requests with the same sequence number), executes the requested service and sends back the result, thereby forgetting that the call ever happened.

In presence of server crashes a complication arises. After a crash a server must be initialized so as to reject pre-crash requests, thus guaranteeing abnormal termination of those requests. This requires that the sequence numbers are monotonically increasing despite the occurrence of crashes. This requirement is met by making use of synchronized clocks for generating the sequence numbers. A sequence number is derived by concatenating the current value of the local clock of the node with the unique node number. This method is known as *time stamping*, a well known method to detect delayed duplicates.

3.3 Clock management

The clocks of the system are kept loosely synchronized, i.e. they roughly represent the same physical time. To achieve this it is necessary for each node to maintain two processes: a broadcaster process that regularly sends its current time to the rest of the nodes and a synchronizer process that receives the times sent by others and advances its clock if it is behind that of a given sender. This implies that the fastest clock determines the local time at each node.

The problem arises that a client with a slow clock can experience difficulty in obtaining services from a server, if the server relies on its own clock for deciding whether to accept or reject a request. Therefore a table (called the 'llsn-array') containing the last largest sequence number (llsn), i.e. the highest sequence number, received for every node is maintained by the server. This array is initialized with the current clock value at startup time.

Since the clock is always advanced 'fast' clock errors will accumulate. Therefor a facility for setting the clocks back is provided. The authority for setting clocks is vested in the broadcaster of one node only. This broadcaster will be referred to as the Time Lord.

For the sake of efficiency a broadcaster only sends its time to nodes that are currently 'up'. To achieve this an 'uplist' is maintained at every node containing the nodes that are currently 'up'. The 'uplist' is shared with the synchronizer process and protected by a semaphore.

3.3.1 The broadcaster

The broadcaster process of each node regularly sends synchronization messages, containing its current time, to all nodes in its 'uplist'. On initialization it sends a 'I am alive' message, i.e. a synchronization message with time zero, to all possible nodes in the system. The responses received are used for the initialization of the 'uplist'.

3.3.2 The time lord

The Time Lord is a special broadcaster. The user at the Time Lord's node has to supply a GOBACK command after which the Time Lord sends a goback message to all 'up' nodes (including its own synchronizer) thereby stopping all broadcasters. When the Time Lord gets the user supplied new time it sends this time to all 'up' nodes using the 'set' message.

3.3.3 The synchronizer

The synchronizer process performs the following actions upon receipt of a message:

- synchronization message: advance its clock if the time contained in the synchronization message is greater than the current clock value and update the 'uplist'.
- goback message: stop the broadcaster.
- set message: set the clock to the time contained in the set message and restart the broadcaster.

After the clocks have been set back the 'llsn-array' should be re-initialized. This is achieved by a reject count maintained by each server. When a server has to reject many requests it is to be expected that the time has been set back, so the llsn array is re-initialized.

4 Rajdoot

Rajdoot is a protocol design by Shrivastava and Panzieri, as discussed in [Rajdoot]. Only the novel orphan handling aspects of Rajdoot are addressed here. First we look at the characteristics of the Rajdoot protocol, next we will discuss the orphan handling aspects, which are divided in three subproblems. This section is concluded with some remarks on the design decisions.

4.1 Characteristics

Rajdoot implements exactly once semantics (cf. section 2.2.1). The execution model used is a one-server-per-client model; at the first call of a client to a node a server is created which only executes calls of this client. The client on the other hand sends all its call intended for that node to the created server. Like the Cambridge ring LAN protocol Rajdoot uses a datagram service for interprocess communication.

Rajdoot is a rather complete protocol design. It allows for orphan detection and killing, without introducing unacceptable many overhead messages. Moreover Rajdoot meets the correctness (serializability) criterium (CR):

> Let C_i denote a call made by a client and W_i the corresponding computation invoked at the called server.
> Let C_i and C_j be such that:
>
> - C_j happens after C_i (C_i *then* C_j)
> - W_i and W_j share some data
>
> then,
> **CR**: C_i *then* $C_j \Rightarrow W_i$ *then* W_j.

In presence of a fixed finite number of communication failures all calls will terminate normally. The call will terminate abnormally if the number of communication failures exceeds this fixed number. If a server crashes, any unfinished call is guaranteed to terminate abnormally. Like REX, Rajdoot allows for arbitrary nesting of remote procedure calls.

4.2 Orphan handling

Before we explain the three orphan handling mechanisms employed by Rajdoot, we discuss the RPC protocol itself.

As mentioned before, the first remote procedure call issued by a client to a node is converted by the RPC mechanism into a request for the manager of that node to create a server. The created server replies by sending the client its address in a 'createserver' message. The client then directs all its remote calls intended for this node to the created server. When no reply is received within a predefined timeout interval, the 'createserver' message is retransmitted. A newly created server starts an idle timeout and waits for a call request. If the timeout expires the server aborts itself, guaranteeing that only the active servers survives. The manager node is stateless: after creation of a server it forgets about the request.

A client sends its calls to the server with a sequence number. Exceptions during transmission of the call are dealt with by retransmitting the message with the same sequence number. If the client does not receive the reply within the timeout interval and the 'retry' value is nonzero, the call message is sent again and the retry value is decremented by one. A server maintains the results of the most recently executed call, so it can cope with retry requests due to lost replies by retransmitting the result.

4.2.1 Abnormal Termination

If a client call terminates abnormally, i.e. no reply message is received from the server, then it is guaranteed that any computations the call may have generated have terminated also.

To meet the above property every call contains a deadline, representing the maximum amount of time available for executing the call by the server. The server sets a timer based on the deadline. If the deadline expires and the server is still executing the call, execution is aborted.

4.2.2 Client node crashes

After the crash of a client node there are two possibilities. Either the node makes a call to some service on some node C after recovery, or it makes no further call to node C.

A call to C is made after recovery In this case it is guaranteed that all orphans the client may have left on node C will be terminated before execution of the call starts.

Every call contains a crashcount value (the local, stable clock value at the time of rebooting). Every node maintains a table of crashcount values of all nodes that have made calls to it, the so-called 'C-list'. A newly created server checks the client supplied crashcount value against the corresponding entry in the 'C-list'; if the former is greater, then this indicates that the caller had a crash since the last call, so there could be orphans on the server node. The newly created server aborts all other servers created by the client before executing the call.

No further calls are made to C If the client is not restarted or after recovery no calls are made to C, it is guaranteed that any orphans on node C will be detected and killed within finite amount of time.

After a server finishes a call it waits for the next call to come. If this call has not arrived within a predefined timeout interval (a few minutes), the server marks itself a potential orphan and resumes waiting. On receipt of a call the server unmarks itself before executing the call. Every node has a terminator process that regularly constructs a list of all potential orphans on its node and calls relevant nodes to see if they are running. The managers of these nodes send their current crashcount values in the reply. If no reply is received after a few retries or the received crashcount value is larger than the one in the table, the terminator process aborts all relevant orphans.

4.3 Concluding remarks

Given the provision of stable clocks at each node, no stable storage facility is required, neither is there any need for keeping clocks synchronized. The amount of state information is minimized; the manager maintains no state information and a server only maintains the last sequence number and the result of the last call.

The most controversial choice is the use of a deadline mechanism. A possible solution to the problem of estimating deadlines could be found in the binding mechanism. A server could add the estimated execution time to the exported information.

5 REX: a remote execution protocol for object-oriented distributed applications

In this section we discuss the REX protocol as described in [REX88]. At the end of the section we shall also look at an ancestor of REX, called the Cedar protocol ([BiNe84]). We start with a description of the environment the REX protocol is designed for.

5.1 The Distributed Applications Support Environment (DASE)

The DASE is a reference model for distributed processing that is currently under development by ECMA/TC32-TG2. It is intended for cooperating open systems and can thus be considered as an extension of the OSI reference model. It comes thus not as a suprise that REX is connection oriented and has a rather large overhead, i.e. it is not suited for real time applications.

The main concept in the DASE is the object. An object is an abstract datatype with a set of operations which are the only means to access and manipulate that datastructure. These operations are specified by defining their external effect on the object. In a way processes are also objects. Objects can interact with each other by invoking operations. Because objects may reside on different sites, there is a need for an RPC mechanism.

There are two ways in which objects can interact (i.e invoke) each others operations. One way is *interrogation*, which means that requests and responses are synchronized. Another way of invocation is *announcement*, which means there are asynchronous requests without responses (results). Objects react to invocations by executing the requested service and possibly sending a response. As already mentioned in section 2.1.2 a binding is created when one object imports an operation exported by another object. The creation of bindings, called *associations* in REX terminology, is provided by the REX protocol.

5.2 The Remote Executions protocol (REX)

The DASE protocol that is responsible for the creation of associations is REX. In the previous section we said that there are two ways of invoking objects, viz. interrogation (synchronous) and announcement (asynchronous). This means that REX should support both synchronous and asynchronous interactions. As a matter of fact REX supports three types of interrogation:

1. synchronous, reliable interactions called *calls* (figure 1);
2. asynchronous, unreliable interactions known as *casts* (figure 1);
3. sessions of nested bidirectional calls and casts.

Moreover, REX is designed in such a way that both client and server applications can use encryption to protect message contents from disclosure, modification, reply and insertion of fake messages (cf. [Birr85]).

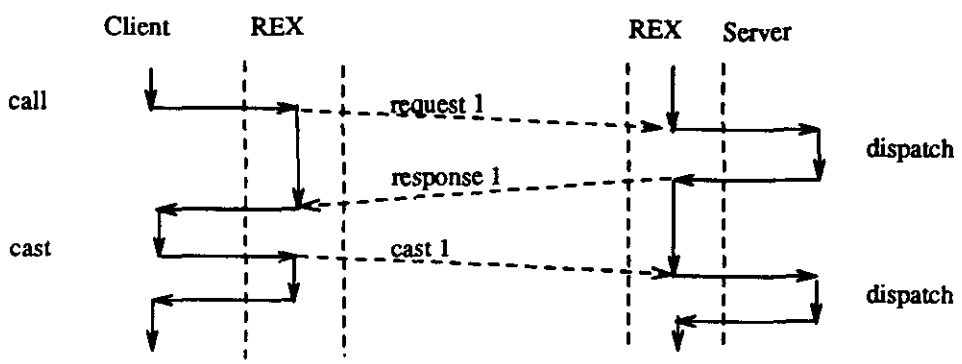

Figure 1

We will look at the implementation of each of the above mentioned types of invocation seperately. But first we will describe the creation of an association in REX.

5.2.1 Creation of associations in REX

Before a REX association can be created, the server must export its service by calling the export function. The export function registrates the instance of the interface and nominates a dispatch

procedure to react to incoming invocations and assigns a locally unique export time stamp. This means that each node should have a locally stable clock. The export time stamp together with the node address of the server uniquely identify the instance of the interface being exported. The time stamp is included in every message to detect whether the server has crashed or rebooted during the lifetime of the association.

The client uses the import function to send a message to the specified server node, requesting for details of a particular instance of the interface. If the interface specification matches one exported by that node, the request is accepted and the export time stamp is returned; otherwise the request is rejected.

5.2.2 Synchronous, reliable interactions

Calls are interactions that resemble remote procedure calls as described in previous sections. They are used to implement interrogations. For synchronization purposes REX uses buffers which are called threads. Threads are not part of the REX protocol itself (they are used only for optimization).

A simple call starts with a request message and ends with a response message. Request and response messages have a sequence number and, to ensure reliable delivery, they are retransmitted at regular intervals until an acknowledgement is received. The receipt of a response implicitly acknowledges the arrival of the corresponding request, and the receipt of the next request implicitly acknowledges the arrival of the previous response.

If both the call duration and the time between calls are less than the retransmission timeout, no explicit acknowledgements are needed and thus only two messages are necessary per call. Unfortunately these assumptions need not to hold, because

1. messages may be lost;
2. the call duration may be longer than the retransmission timeout;
3. servers may acknowledge without responding promptly;
4. the time between calls may be longer than the retransmission timeout.

In case 1. the message will not previously have been seen by the receiver, so provided the original assumptions hold it can process this message without any problems. In cases 2. and 4. the receiver will already have received a copy of the message, allowing it to recognize from the sequence number that this message is a copy. So the receiver sends an acknowledgement, but discards the message. In case 3. the client has received an acknowledgement, but not yet a response. The client will send probe messages to the server to make sure it is still functioning. Probe messages must be acknowledged in the same way as requests (figure 2). In this way it is possible to detect communication failures and node crashes and to preserve the local procedure call semantics for the client.

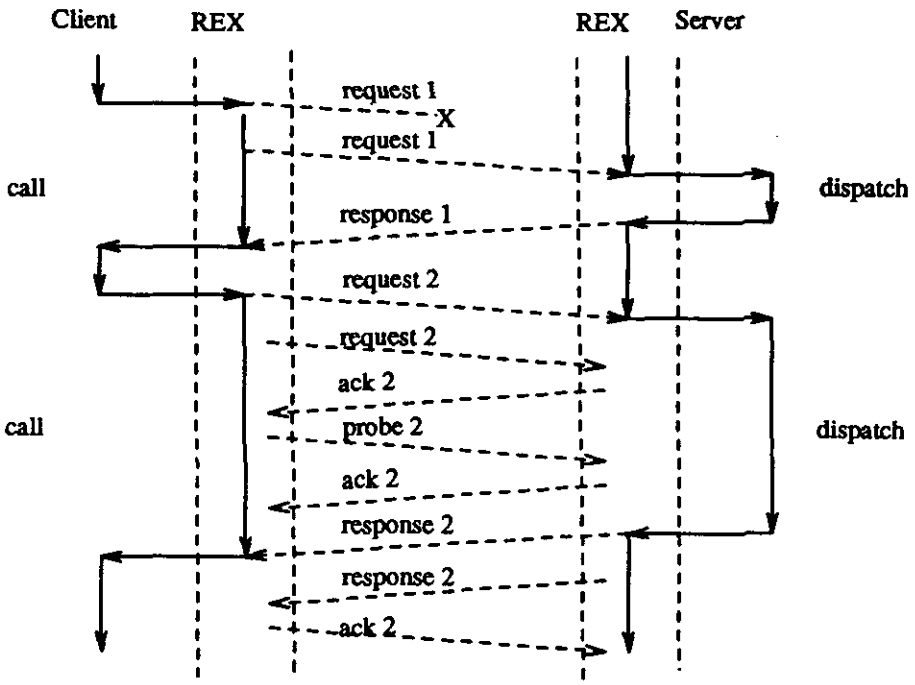

Figure 2

If the server continues to acknowledge the probes, the client will wait forever. To protect clients from cheating servers one could include timeouts. However, these timeouts should be similar with those in local procedure calls so that transparency is preserved.

It can be easily seen that REX calls have exactly-once semantics.

5.2.3 Asynchronous, unreliable interactions

To implement DASE announcements, REX also provides asynchronous unreliable interactions, called casts. In contrast to calls, casts are not acknowledged or retransmitted. The reliability of their delivery is determined by the underlying network. Requests transmitted by casts are dispatched in the same way as those sent by calls, except for the responses generated by the invoked operation. These responses are simply surpressed by the server.

Casts are sequenced in the same way as calls; if a cast arrives out of sequence it will be discarded. This strategy ensures the at-most-once semantics of REX casts.

5.2.4 Nested bidirectional calls and casts

A REX session is initiated and terminated by a server simply by entering and returning from the dispatch procedure. The simplest form of session is constructed from nested back calls (figure 3). Back calls rely on an implicit token that confers to the recipient the right to make a call back instead of sending a reply. In terms of sequencing, this means that the token confers the right to generate the next sequence number. Naturally, nested call backs are strictly two way alternate.

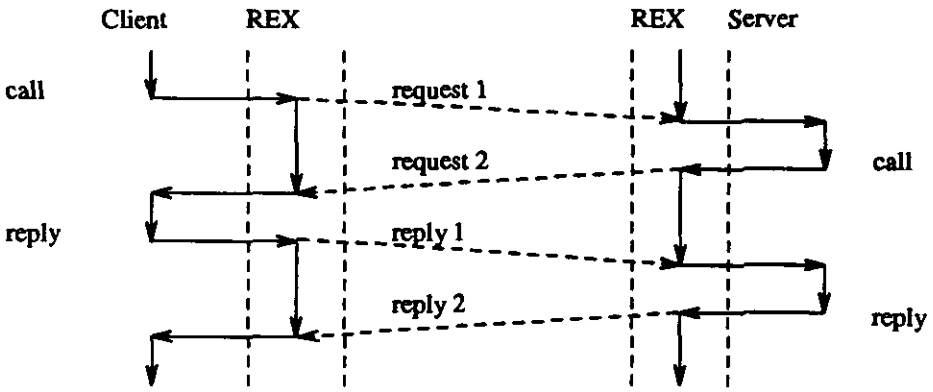

Figure 3

The session facility in REX allows for blast protocols. A forward blast session starts with a call which synchronizes both ends and checks that all the starting conditions have been satisfied. The data can then be transferred quickly, without acknowledgements, by using casts. When all the data has been transmitted, another call resynchronizes both ends and checks that all data has been received correctly (figure 4). Any faulty message can then be retransmitted using calls for reliable transfer. The reverse blast protocol is almost a mirror image of the forward blast protocol, but for a different distribution of calls and casts. In particular, the second message (reply 1 in fig. 4) is a cast in the reverse protocol, because the server needs to retain the token to transmit further messages containing the data to be transferred.

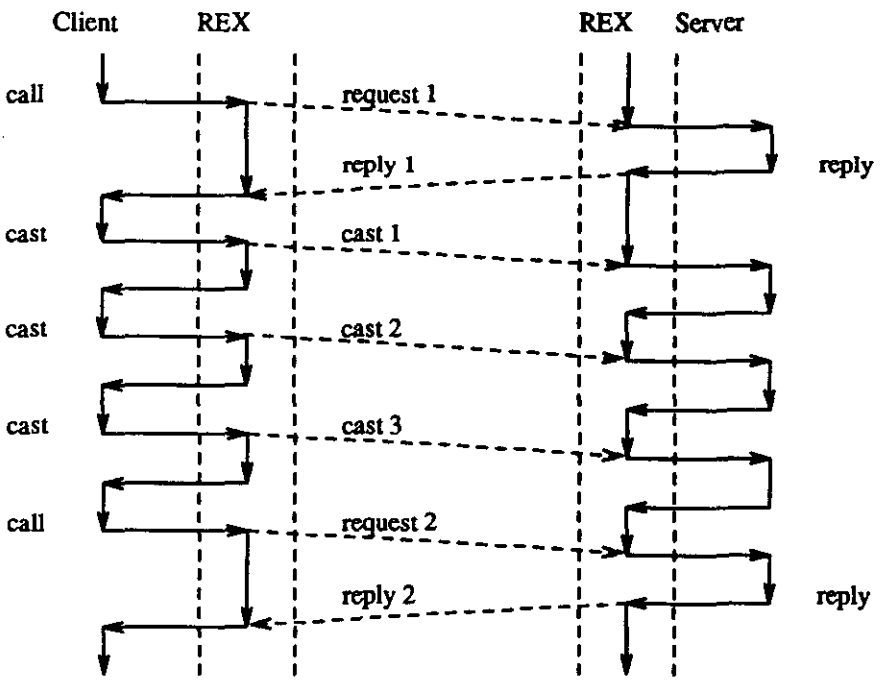

Figure 4

Now consider a blast protocol in wich all casts are lost. In this case both protocols degenerate to a reliable two way alternate sequence, where each request message reliably transfers the token that gives the right to transmit (Casts do not transfer this token and therefore do not change the direction of the transmissions). The token is used to enforce coordination of the application level protocol by only allowing the node which holds the token to transmit messages (requests or casts).

Casts may be lost, but requests will still be delivered reliably, because they will be retransmitted until acknowledged.

To avoid problems REX uses a rate control algorithm for flow control, which seeks to prevent congestion, rather than make futile attempts to cure it. The rate of message transmission is limited to that which is sustainable by the slowest element in the end-to-end communication path. The rate is negotiated at import time, starting with a value suggested by the clients application and asking every element in the data path to agree to support it or revise it downward. The rate is then enforced by REX by temporarily blocking a client that attempts to transmit too quickly; thus applying back pressure directly to the data source.

5.3 The Cedar protocol

The Cedar protocol is in some way an early version of the REX protocol. Simple calls are carried out in the same way as in REX. To make a call, the caller sends a *call packet* containing a call identifier, data specifying the desired procedure and the arguments. The procedure returns a *result packet* containing the same call identifier and the results. The protocol is analogous to the REX protocol. This means that the same problems arise, which can be solved by the same strategies. For example, if the called server has entered an infinite loop while executing a service the client may wait forever, which is similar to local procedure call semantics. Like in REX local stable clocks are needed to guarantee unique call identifiers.

There are however, some differences between REX and Cedar. In the Cedar protocol retransmitted messages need to be acknowledged explicitly, whereas in REX implicit acknowledgements are sufficient. This seems to make REX more appropriate for complicated calls (sessions) than Cedar.

Another important difference is the exception handling facility in Cedar. At this point the semantics of remote procedure calls differ from local procedure call semantics. The callee is permitted to communicate only those exceptions that are defined in the exported interface. In this way simplicity is gained at the expense of transparency.

Like REX Cedar also allows extensions for security mechanisms.

6 Concluding Remarks

In this report we have considered aspects of remote procedure call (RPC) design and illustrated aspects of implementation by examining several examples. Although Rajdoot seems to be the most complete protocol (it supports orphan detection and killing) it is hard to point out a 'best' protocol. For instance, if synchronized clocks already have been implemented one might choose a protocol like the Cambridge Ring protocol, because it is less complicated. In a purely object oriented environment it is nice to have an object oriented protocol like REX. So, the environment is an important factor in deciding what kind of RPC protocol should be implemented in a particular distributed system.

At this point it is not yet clear what is the appropriate protocol for the DEDOS [1] project. We suspect that, because of its numerous facilities (for instance orphan detection and killing), Rajdoot might be a suitable choice. However, it may be advisable to redesign and implement Rajdoot in an object oriented way (for instance by using the C++ programming language).

References

[Birr84] Birrell, A.D., *Secure Communications Using Remote Procedure Calls*, ACM Trans. on Comp. Systems, Vol. 2, No. 1(1984), pp 39-59.

[1] DEDOS is a DEpendable Distributed Operating System, currently being developed at Eindhoven University of Technology.

[Birr85] Birrell, A.D. and B.J. Nelson, *Implementing Remote Procedure Calls*, ACM Trans. on Comp. Systems, Vol. 3, No. 1(1985), pp 1-14.

[Nel81] Nelson, B.J., *Remote procedure call*, Ph.D. dissertation, Comput. Sci., Carnegie-Mellon Univ., Pittburgh, PA, Rep. CMU-CS-81-119, 1981.

[Rajdoot] F. Panzieri and S.K. Shrivastava, *Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing*, IEEE Transactions on Software Engineering, Vol. 14, No. 1(1988), pp 30-37.

[REX88] D. Otway and E. Oskiewicz, *REX: a remote execution protocol for object-oriented distributed applications*, Proceedings 7th Int. Conf. on Distributed Computing Systems (1988), pp. 113-118.

[ShriPan] Shrivastava, S.K. and F. Panzieri, *Reliability Aspects of Remote Procedure Calls*, in: B.K. Bhargava ed. Concurrency Control and Reliability in Distributed Systems Van Nostrand Reinhold 1987.

[Tan88] Tanenbaum, A.S., *Computer Networks (2nd edition)*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[Yap88] Yap, K.S., *Fault Tolerant Remote Procedure Call*, Proceedings 8th Int. Conf. on Distributed Computing Systems (1988), pp. 48-54.

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits.
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes.
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films.
85/04	T. Verhoeff H.M.L.J.Schols	Delay insensitive directed trace structures satisfy the foam the foam rubber wrapper postulate.
86/01	R. Koymans	Specifying message passing and real-time systems.
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specification of information systems.
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures.
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several systems.
86/05	J.L.G. Dietz K.M. van Hee	A framework for the conceptual modeling of discrete dynamic systems.
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP.
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers.
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987).
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language.
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing.
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86).
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes.
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards ~~V~~Verifying object oriented systems (Fund. Informatica

86/14	R. Koymans	Specifying passing systems requires extending temporal logic.
87/01	R. Gerth	On the existence of sound and complete axiomati zations of the monitor concept.
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases.
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems.
87/04	T.Verhoeff	Delay-insensitive codes - An overview.
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns - searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language.
87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts.
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas.
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples.

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces.
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen.
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations.
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG.
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence.
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems.
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples.
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to Handle Structured Information.
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems.
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication.
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way.
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification.
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling.
88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits.
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases.
88/12	A.E. Eiben	Abstract theory of planning.
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees.
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types.

88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations.
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors.
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems.
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a paralell program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.

89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.
90/1	W.P.de Roever-H.Barringer C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, pp. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.

90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.
90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p.