The Client/Server Model in Distributed Computing*

J. van Leeuwen
Department of Computer Science, University of Utrecht
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

A variety of design issues for distributed computer systems is explored in terms of the Client/Server model. An attempt is made to expose the basic ingredients of the Client/Server model, in order to facilitate further theoretical analyses and applications in the design of complex systems. Also, a detailed but standard 2-phase commit protocol is presented and verified in a form that is suitable for applications in distributed systems.

1. Introduction

Increasingly computer networks and distributed systems are being introduced in traditional application environments for computers. Both local and wide-area networks and their supporting software are being produced to meet with customer demands, and a considerable research effort is invested worldwide to solve the new and complex design issues that come with the development of distributed information systems. In this paper we will explore *remote operations architectures*, which allow for flexible client/server type interactions between distributed application processes on top of an existing and reliable transport level. In such architectures sets of primitives are provided for application association and message passing following a request/reply protocol, with facilities for service authorization and recovery. The remote operations architectures usually conform to internationally agreed standards, for the general requirements of interconnection and interchangeability.

In order to deal with the complex issues of communication and control in a distributed system, it is necessary to have a consistent architectural model underlying the design and development of a system. Additional requirements are imposed by the agreed ISO and ECMA standards for remote operations and transaction processing. In many distributed systems the client/server paradigm is used to explain the underlying system view, suggesting the possibility of a formal model and correctness proofs of the design. *In this paper an attempt is made to expose the essential ingredients of the Client/Server model* which should be taken into account, in a form that should facilitate the use

*This work was carried out as part of a research agreement with NCR Systems Engineering BV, P.O. Box 492, 3430 AL Nieuwegein, The Netherlands.

and further study of the model for both practical and theoretical purposes. The *Client/Server model* can be recognized in many distributed system architectures. Without much difficulty all aspects of distributed system design at the network level and beyond can be brought to bear on the model. Yet, despite its ubiquity, no detailed study seems to have appeared that presents a complete review of the Client/Server model. In this paper we will address a large number of issues that are often not well-documented in the current literature nor analysed very well theoretically, and the paper hopefully serves as a stimulus for further research. At the same time we hope to show the context for many concrete research issues in distributed algorithm design as it can be seen today.

A remote operations system comes with two design aspects: an 'applications architecture' describes the system from the perspective of an applications programmer, and a 'communications architecture' describes the available network functions. We adopt the view that an application may consist of several, distributed application processes that cooperate to achieve some goal. An application may exist 'forever', i.e., may have to be supported for an indefinite period of time. An application (i.e., an application process) will be activated by an external cause or by another application. Application processes can activate network services that are generally available. Network services in turn can activate other network support tasks. In addition, autonomous processes may exist that play a role in standard network control and function maintenance. Application processes communicate using the facilities of the communications architecture. To shield application processes from the details of the functions provided by the communications architecture, one or more interface modules can be designed that provide 'higher lever' communications facilities.

In this paper we will describe the Client/Server model as a unifying framework for the high level description and specification of communications and interactions between processes. The model is geared to understanding general Client/Server architectures, but carries the idea of the model much further. The model as we describe it only serves as a structuring tool for a network architecture at the session and presentation layer level. The model is heavily based on the '*abstract object*' approach to distributed system design (see e.g. Watson [33]) and aims at viewing a number of aspects of Client/Server architectures from this perspective. The abstract object approach has successfully been applied in other contexts of networking (see Sloman & Kramer [27] for an extensive summary). *It is the specific aim of this paper not to digress into theoretical analyses and complicated algorithms, but to give an integral view of the many concrete problems that must be faced in the design and implementation of a distributed system.*

The paper is organized as follows. In Section 2 we will briefly introduce the issue of distributed computing, the Client/Server model and the notion of Remote Function Modules. In Section 3 the Client/Server model is presented in more detail. In Section 4 a large number of architectural issues in computer networks are reviewed from the unifying viewpoint of the Client/Server model. In Section 5 we present some ideas for implementing the Client/Server model

in general environments, using the notion of Remote Function Modules. Section 6 discusses the complex issues concerning binding and unbinding. Finally, in Section 7 we consider more involved 'atomic' interactions between clients and servers and the need for atomic commit protocols. An integral development and correctness proof is given for a standard 2-phase commit protocol that seems to be appreciably simpler and more clarifying than similar treatments in the current literature (see e.g. [2,6]).

2. Distributed systems and remote operations

A large number of computer-oriented hardware or software systems are referred to as 'distributed systems'. Generally, the phrase is used to refer to any network of communicating machines or processes designed to achieve some overall goal of a collection of users or an organisation. It can refer to the supporting network hardware and software only (which is a very common use of the term), or to a distribution of application programs and control functions over various sites with an underlying scheme for concurrent operations and communication. In all cases there are complex issues of system operation and control not found in traditional computer applications, and encountered in only a limited form in the modern approaches to operating systems (see e.g. Peterson & Silberschatz [25]).

There are a number of different approaches that can be recognized in the design of distributed systems: (a) the *Hierarchical model*, and (b) the *Client/Server model*. In the Hierarchical model, an executing program can send subtasks to other machines for execution. The model is suitable for applications that have a hierarchical structure and is commonly found, for example, in distributed database systems (see e.g. [6]). The model requires the supporting network to provide a service like 'send subtask X to another machine (or, to machine Y)' and some form of message passing between a program and its subtasks. In some cases it is possible to send the complete code of a subtask to another machine. More often it is possible to 'call' a stored copy of a program at the other machine and pass it the necessary parameters and/or data in some form or another, like in systems that implement remote procedure calls (RPC) as described in Birrell & Nelson [4]. Normally program execution is not resumed until the result of a subtask has been received.

In the Client/Server model, each program may request services from designated service providers (servers) that are shared with other users or application programs and that are usually situated on other machines. Examples of servers are: *name servers, file servers, printer servers* and other special programs. Again the model requires the network to provide a suitable interface for expressing and transporting service requests, and it is common to use the logistics of remote procedure calls or a relaxed, asynchronous version of it. In particular, a server call should not keep the calling program from further execution.

From a systems point of view the Client/Server model subsumes the Hierarchical model, although the requirements of control will be more complex. But in many respects the Client/Server model seems to be the more

fundamental paradigm in distributed computing, as it appears at the basis of the network developments of major vendors (see e.g. [11]).

The Client/Server model is based on the idea that in a distributed system environment certain processes (*clients*) will be requesting functions from other processes (*servers*) and that all distributed activity can be seen from this perspective. The model tends to declare fixed processes (processors) as clients and others as servers, which will be adequate in systems where Client/Server functions are logically and physically separated. This clearly need not always be the case.

An obvious analogy with known notions from programming suggests that Client/Server functions can be viewed as logical entities that could well reside in a single process. More precisely, one can elevate the notion of a Client/Server function to the notion of a *Remote Function Module* (RFM). The RFM notion is primarily a tool for structuring the specification and hence, the coding of Client/Server applications. Remote function modules provide sets of functions that can be naturally grouped together, for local or remote use. Remote function modules can call on the services of other remote function modules.

Remote function modules facilitate abstraction, in the sense that its services can be used without necessarily understanding the way they are coded. When remote function modules are designed, irrelevant details can be suppressed by making use of other (lower level) remote function modules. Thus remote function modules facilitate hierarchies of abstractions, a known and powerful tool in system design.

3. The Client/Server model

The rationale for the Client/Server (or Customer/Server) model was first described by Gentleman [9], who identified the issues that need to be resolved in any system designed from this perspective. The Client/Server paradigm is now standard in distributed system design (see Comer [7]).

The Client/Server model can be viewed as a model for *process interaction*. One process (A) identified as the *client*, requests a service from some other process (B) which acts as the *server*. The server, after performing the requested service, posts a reply to the client. The client-server relationship between A and B exists only for the duration of the interaction. Thus a process that acts as the client in one interaction may become the server in another, and vice versa. We will later allow that servers request services from other processes while processing a request and thus act as server and client simultaneously. The Client/Server model is recognised in distributed systems based on the remote procedure call (RPC) paradigm (see e.g. Lampson [20] and Birrell & Nelson [4]) or the nested transaction paradigm (Moss [23]).

3.1. Requests and replies

The *request* of a client (A) must be passed to the server (B) in a specified way. Ideally it should be immaterial to the model whether B is local to A or remote. The same formats and procedures should apply in either case. In the

Client/Server model, requests and replies are passed as messages. (See e.g. Gentleman [9] for an appraisal of the various semantic issues involved in message passing.) An important issue is whether the message passing primitives are '*blocking*' or not. For example, after a request has been sent out, the client may remain blocked until a reply has been received from the server. (It is sometimes called '*connection-less interaction*', because the association between the client and the server only exists for the duration of the request.) In the non-blocking case, the client typically engages in a sequence of message exchanges with the server but does not need the reply to each individual request before a new request can be sent out. (It is sometimes called '*connection-oriented interaction*', because the client and the server will normally commit a fair amount of resources to maintain the connection.) The Client/Server model requires that a '*session*' is opened and maintained, in order for A and B to interact in this way. This leads to the problems of *connection management*. There are various alternatives for the client to observe replies in the case of non-blocking requests. (Essentially there are blocking and non-blocking variants for it too, see e.g. the excellent discussion in Svobodova et al. [29].)

3.2. Complications because of possible failures

The Client/Server model is deceptively simple when communication failures and other failures are not taken into consideration. There are various types of communication failure that can occur: messages may be lost or 'stuck' (e.g. because of deadlock), messages may be undeliverable (e.g. because one or more communication lines are down), and processes may 'die'. Failures complicate the simple interaction scheme of the Client/Server model, because failures directly affect the request/reply mechanism. In particular, the reply a client receives on a request should either be a valid reply from the server or an indication that no valid reply from the intended server can be obtained because of some (identifiable or non-identifiable) failure. In the Client/Server model, a client must receive a reply (and no more than one) to every request it sends out, and the replies received must correspond exactly to the 'facts' for the associated requests. It follows that replies are not simple and must be 'interpreted'. In the model adopted in this paper, replies strictly follow the Client/Server paradigm.

3.3. The use of timers

Failures affect the Client/Server model in yet another way. In physical communications systems, it is common to guard for failures by the use of *positive acknowledgements* and *timers*. Acknowledgements can be merged in with replies, but timers are different entities. Svobodova et al. [29] discuss the difficulty of 'timeout parameters' for programming at length, but there is no question that timers are often the simplest and most adequate solution to communications control problems in concrete systems. For example, timers are used in the implementation of *reliable remote procedure calls* (see e.g. Lampson [20] or Shrivastava & Panzieri [26]), in *connection management protocols* (see e.g. Fletcher & Watson [8] or Tel [32]), and for *'crash control'* in transaction

management systems (see e.g. McKendry & Herlihy [22]). In typical distributed systems timers are available implicitly (at the client side) or explicitly (at the server side) for request-handling purposes and session maintenance. At the client side, a timeout value can be set with every request to indicate how long one is willing to wait for a reply. A *timeout* is triggered when no reply is received in the prescribed time-interval or when an exception has occurred in the underlying transport layer. At the server side, a timeout value can be set with every session. In this case the timeout value indicates how long the server is willing to wait for 'new' requests from one of its clients viz. from the particular session. (It is common to assume that the corresponding timer is reset to the timeout value each time a new request in the session is received before timeout.) Multiple timers must be implemented explicitly by means of incremental timeout queues (as described in e.g. Tanenbaum [30]). In all cases timeouts are handled as interrupts, and some action must be specified when a timeout occurs. At the client side it typically involves 'retrying' a request that timed out, which leads to the problem of *duplicate detection* by the server. At the server side a session that timed out is to be closed explicitly. In the model described in this paper we will assume that timeout values can be specified for all requests. (Semantically the use of timers by a server is part of the session-management protocol and not of the Client/Server model.)

3.4. Sharing and concurrency

An important aspect of the Client/Server model is the fact that servers can be *shared*. It implies that any number of clients can independently request the services of a same server process B, and (hence) some policy must be set for every server of how to handle 'simultaneous requests'. In case all client-server interaction is session-oriented (as in most Client/Server architectures), the policy must essentially indicate whether a server can support only one session at a time or multiple sessions simultaneously. In the Client/Server architecture this has led to the distinction between '*single-threaded*' and '*replicated*' servers, respectively. The effect of concurrent sessions must be equivalent to the effect of some serial ordering of the sessions, in order to guarantee that semantically each session can still be regarded as the unique association between one client and the server. It follows that a server can be of the 'replicated' type only when its semantics allows for it. When server replicas do not share resources, the semantic constraints are trivially satisfied and the effect is that of many clients having their own copy of the server and (thus) having access to the same services individually. In the Client/Server architecture, a syntactic difference can be made between single-threaded and replicated servers at the level of the allowable primitives.

3.5. More complex atomic actions

Aside from 'many-to-one' interactions, the Client/Server model may allow for 'one-to-many' interactions between processes. In this case, a client process (A) requests the services of various servers and (hence) engages in multiple sessions simultaneously. It is the client's task to synchronize the various sessions, when

necessary. Additional complications arise if the simultaneous requests a client A sends out are not independent and constitute 'one' (intended) transaction. Typically, a request should only be processed if all requests that are part of the transaction can be (by the various servers). This leads to the well-known problems of *transaction management* (see e.g. Gray [10], Ceri & Pelagatti [6], or [12]), viz. transaction commitment and recovery protocols. Atomic actions will be discussed further in Section 7.

4. Client/Server architectures

We now elaborate on a number of architectural issues for the Client/Server model. (Some terminology is taken from [18].)

4.1. Common Client Modules

The environment for a Client/Server architecture is a distributed system of 'service providers', i.e., processors of known functionality connected by a local area network. Application processes may invoke operations that are only available across the network, at a remote site. The invocation of a remote operation triggers an involved mechanism that is (or, should be) largely transparent to the application process, and whose sole purpose is to perform the desired operation reliably and return the result to the invoker. (Note that the 'result' may be a diagnostic e.g. indicating that the desired operation cannot be performed at the present time.) The design of some Client/Server architectures suggests that invocations should be handled by a Common Client Module, or CCM. The CCM provides the same support to all application processes running on a given processor. An application process can invoke a remote operation by passing a suitable 'operation request block' to the CCM. In fact, the CCM may maintain a queue of outstanding operation requests for each (local) application process. An *operation request block* must have a well-known format, including fields for e.g.

(i) the logical name of the requesting application process (the client),
(ii) a unique identifier for the request (e.g. a sequence number),
(iii) the logical name of the desired remote operation,
(iv) class information about the desired remote operation,
(v) parameter values for the desired remote operation,
(vi) a pointer to the 'operation result block',
(vii) reserved fields for use by the CCM,
(viii) the size of the operation request block.

The *operation result block* corresponding to each operation request must have a well-known format as well. (It obviously is sufficient for an application process to pass a pointer to the appropriate operation request block when invoking a remote operation through the CCM.)

4.2. Binding

The CCM must see to it that each operation request is shipped across the network to the appropriate service provider, and that the result of the operation request is written back into the corresponding operation result block. In order

to ship a request across, the CCM must '*bind*' the service provider. For now it is sufficient to view binding as the initiation of a session between the client and a server process (the server). Binding can be implicit or explicit. Clearly explicit binding requires that the (logical) name of the intended server is known. The session between a client and a server process is identified by a unique session-id which exists for the duration of the association.

4.3. Primitives and data structures

It follows that the CCM must be a rather complex support module, when considered in its most general form. The CCM must support e.g. the following primitives:

(i)	*Bind.req*	('bind request'),
(ii)	*Bind.repl*	('bind reply'),
(iii)	*Oper.req*	('operation request'),
(iv)	*Oper.repl*	('operation result'),
(v)	*Cancel.req*	('cancel an outstanding operation request'),
(vi)	*Cancel.repl*	('result of cancelling an outstanding operation request'),
(vii)	*Unbind.req*	('termination request for a session'),
(viii)	*Unbind.repl*	('termination reply').

In addition the CCM may provide several other primitives, e.g. for inspecting the status of a current session with non-blocking operation requests. The CCM maintains a session-table with information on all current (active) sessions, including e.g. the unique session-id of each session and various status indicators. The session-table is implicitly maintained by the bind and unbind primitives. We defer the discussion of binding and unbinding to a later section. With each entry in the session-table the CCM may maintain a queue of outstanding operation requests. The queues are implicitly maintained by the Invoke and Oper.repl primitives. (Interrupts caused by time-outs on outstanding operation requests could also be handled through the *Oper.repl* primitive, or a variant of it.)

4.4. Formats and message passing

There are several further issues involved in the design of a CCM. In order to ship an operation request from a client to the intended server process, the CCM must generate a message (or packet) and some kind of 'interchange' must take place that will guarantee that the message is delivered to the server (and, eventually, that the operation result is delivered back to the client). Gentleman [9] gives an insightful discussion of the design considerations for message formats. In the design of Client/Server architectures one may assume that the problem of message formats is solved at a lower 'level' and that the request/reply formats are 'well-known', relying on a standard transport level (such as *Netbios*) to handle variable-length messages. The message formats will be related to the formats of the operation request and operation result blocks in a simple manner (compare e.g. [11]). In order that a message can be sent to the intended server, it is normally required that the network address of the

service provider and the 'port number' of the intended server are provided with the message (together, of course, with the return address for the reply message). We defer a discussion of naming and addressing to Section 6. Normally, the address information is maintained in a 'connection descriptor' stored with the unique session-id. Thus the necessary information for name-to-address binding must be set up as part of the *Bind.req* and *Bind.repl* primitives. It also follows that the CCM must possess (and maintain) a network directory with the 'well-known' address and attribute information of the available service providers on the network, in order that it can set up associations. Note that at the application (i.e., client) level, operations and servers need only be known by generic names or aliases. In many virtually all distributed systems, a client process must name the intended server when invoking a (remote) operation.

Service providers (servers) may be invoked by remote clients to perform certain operations. In most distributed systems, servers must post their availability to the network as part of a 'start up' procedure at initialization time. Typically this is achieved by 'opening' the well-known socket for the server and authorizing the connection. In general servers 'export' their availability and various other attributes (like the operations they provide) across the network and potential clients 'import' this information into their network directory. Alternatively, it may be assumed that the servers on the network are 'known', and the only status information that need to be maintained is whether particular servers are available and can be connected to. (The status information will be part of a potential client's *Bind.repl* result block.)

4.5. Common Server Modules

In a Client/Server architecture, servers operate on a 'receive any' basis (cf. Gentleman [9]). In other Client/Server architectures it is proposed that all messages (e.g. bind requests and operation requests) received by a server must be handled through a Common Server Module, or CSM. The CSM authorizes bind requests, maintains sessions, 'interpretes' and dispatches operation requests to the server, and sends replies to clients. The CSM is the server-version of the CCM, and combines the functionality of a session handler and a request handler. We will not digress into the design of the CSM, but some further issues related to it will be mentioned in the later discussion of binding and unbinding. A CSM must be designed in a 'server-independent' manner and (thus) needs attribute information of the particular server it supports in a standard format, in order that it can perform the necessary validations of operation requests and replies.

4.6. Maintaining connections

Once a client (A) has succeeded in 'binding' a server process (B), the server is prepared to receive and process any suitable operation request which the client sends. Operation request messages are sent under control of the client's CCM, by executing *Oper.req* commands. A Client/Server architecture should support both blocking ('synchronous') and non-blocking ('asynchronous') requests, and

with suitable precautions both types of requests must be allowed over the duration of a single session. It follows that the type (class) of a request must be specified in every operation request block and operation request message, to ensure the proper handling by the *Oper.req* primitive. It may be necessary to know in a session how many operation requests can be outstanding at a time, whether it is blocking or non-blocking. Note that the operation result block of an outstanding operation request contains the necessary status information which the client can check for this purpose (in the model as described here). A Client/Server architecture may allow that a client revokes a (non-blocking) operation request (cf. the *Cancel.req* primitive in the model described here). In view of the earlier requirements only the unique session-id needs to be supplied as a parameter, as it uniquely identifies the operation request in it that is currently outstanding (if there is one). Of course one needs to specify the effect of a *Cancel.req* at the server's end very precisely, but quite apparently it can only 'intercept' (and cancel) an operation request before its processing has begun and cannot trigger an 'undo' if the processing of the request has started or is completed. Blocking requests cannot be revoked.

In general a server will have to know whether an operation request is blocking or non-blocking at the client's end, for the purpose of connection and resource management. In some architectures this is not necessary, and servers can be oblivious to the class information of an operation. In this case the server primitives can remain of a simple kind, and indeed need to be no more complicated than 'receive' and 'send'. Of course servers can perform non-specific actions (like aborting all sessions) and inform clients, with implicit effects for outstanding operation requests. The handling at the client's end must be part of the *.repl* primitives. In a Client/Server architecture this may all be handled by the CCM and CSM modules, thus providing the client with a request/reply mechanism of similar transparency as the well-known RPC.

4.7. Failure semantics

Again the possibility of failures complicates the Client/Server model and (hence) the design of the common interface modules in the Client/Server architecture as described. For example, it must be carefully specified what performance of remote operation requests is required under different conditions of communication or processor failure. If only communication failures can occur, one typically requires either an 'at-least-once' semantics or an 'exactly-once' semantics for remote operations. (The 'exactly-once' semantics is usually preferred and guaranteed in RPC-based remote operations architectures, cf. Birrell & Nelson [4].) If processor failures can occur as well, several other options have been proposed (cf. the detailed taxonomy in Spector [28]). A discussion of reliability issues in the Client/Server model is deferred to a later section. In most Client/Server architectures, all communication requests are handled by a reliable transport level. In this case communication failures (errors and network problems) reported to a client are all connection-oriented.

4.8. *Further support services*

A Client/Server architecture will usually require that the CCM and CSM provide various additional support services, e.g. various network security functions and resource allocation services. In this paper we will not discuss these services.

5. REMOTE FUNCTION MODULES

The Client/Server model is based on a strict separation between client and server functionality. More precisely, the Client/Server model captures all aspects of distributed systems in which processes communicate on a request/reply basis. It follows that the Client/Server model is not restricted to systems with separate client and server nodes and equally well permits that client and server functions reside on a single processor or in a single process. Indeed a Client/Server architecture should permit client-to-client and server-to-server communication using the client-server interaction primitives, and even server-initiated sessions between servers and clients are possible. Thus it is necessary to distinguish between clients and servers as they exist on the network and clients and servers as they exist at some (logical) level of process interaction.

5.1. *Building blocks*

The apparent lack of uniformity in client and server structures makes it difficult to distinguish common building blocks on which the Client/Server model can be based. At an abstract level all notions related to the theory of concurrent processes (see e.g. Peterson & Silberschatz [25]) can be brought to bear on the Client/Server model, but this clearly does not fully capture all requirements in a concrete network context. In this section we will describe the notion of a *Remote Function Module* (RFM) as a possible basis for the Client/Server model. RFMs are intended primarily for structuring purposes in the design of Client/Server architectures. Thus, designs and specifications should be written in terms of RFMs, but the implementation language need not support the concept as such (as long as its semantics can be realized somehow). Every application, whether client or server (in the network sense), will be composed of a number of local processes and RFMs. At the same time, the necessary support environment for RFMs as we will describe it implies a possible structuring of the CCM and CSM module-concepts.

5.2. *Structure of RFMs*

A RFM essentially is a set of functions that can be used by or use functions of RFMs at other sites. Thus it consists of a set of entry functions (for local or remote use), a list of remote function names, a set of auxiliary procedures (for internal use), an initialization and driver routine (executed and initialized upon creation of the module) and local data (accessible through the functions and procedures of the module only). RFMs resemble tasks in Ada (see e.g. Barnes [1]) and modules in Modula-2 (see e.g. Wirth [34]), although the implementation model will be different. RFMs are entire self-contained, except for the

possible references to remote functions. The use of (logical) function names rather than RFM or server names and addresses gives RFMs a high degree of configuration independence. Also, RFMs can easily be tested in isolation by providing 'stubs' for the remote functions it employs. Remote functions can be called by sending off operation request blocks and waiting or checking for a reply, as explained in the previous section. We will discuss momentarily how this should be specified at the RFM level. Note that RFMs, even when they reside on the same processor, have no shared storage and communicate using a mechanism very similar to RPCs.

5.3. Sharing

As its entry functions can be requested ('called') by other RFMs, all RFMs are inherently 'shared objects' and thus must provide for shared usage. In typical Client/Server architectures clients are not shared but servers are and, in our terminology, sharing is implemented either by scheduling connections one-at-a-time (single-threaded servers) or by providing a separate instance of the RFM for each connection that requests its services (replicated servers). In the latter case, the separate instances may have to cooperate on the (shared) data of the master RFM. In general the sharing of a RFM requires that an 'administrator' or 'monitor' is added to handle the simultaneous access by other RFMs (cf. Gentleman [9]). The administrator of a RFM basically maintains the request/reply queues as shared buffers between the RFM and the requesting parties (as in the producer/consumer problem, cf. [25]), and allows the RFM to handle one request at a time. The administrator can implement any desired policy for selecting requests.

From now on we will assume that the shared usage of RFMs is arranged for by standard methods. For a further discussion of the administrator concept, see Gentleman [9]. (In Svobodova et al. [29] the related notion of a 'guardian' is introduced for the very same purpose.)

5.4. Connectors

A RFM can only communicate with another RFM if an explicit 'connection' (session, association) between the two has been set up. Each connection is to be managed by a new type of (shared) object, a '*connector*'. Basically a connector is an object that governs the message traffic between the two RFMs that it connects and deals with all problems caused by the two RFMs in this respect. To apply the concept in the Client/Server model, it is necessary that we view the 'network' (or the Netbios) as one RFM that is local to every processor and that provides basic transport level services. It is thus no restriction to require that a connector can only connect two RFMs that reside on the same processor. A connector essentially maintains and manages a session between two RFMs in the network sense. The complexity of a connector will depend on the type of RFMs it connects. (The usual connection will be between a RFM and the network, viewed as a RFM.) Yet the main idea of a connector is that it connects local RFMs and thus provides for a high degree of configuration independence and ease of testability. As a RFM, the network

is assumed to provide all remote functions by their name and server address. Without much difficulty a connector can maintain a stream of request/reply messages in the case of asynchronous requests and a restriction to one outstanding request at-a-time per connection is not necessary. (The restriction may still be enforced in case of memory limitations on a supporting processor.)

5.5. Configurations

All connection and disconnection requests of the RFMs on a processor are handled by one general (shared) object on this processor, called a 'configuration manager' (following the CONIC design of Kramer et al., cf. Sloman & Kramer [27]). We allow that connection requests specify a sub-set of the functions to which access is desired on a RFM. The configuration manager authorizes, administers and dispatches connectors and provides dictionary services like the mapping of function to server names and of server names to network addresses. In the terms of the previous section, the configuration manager is designed to handle all bind and unbind requests. Note that if a RFM wants to communicate with a 'client' or a 'server', it has to request a connection to the network (as a RFM) for the desired set of client or server functions. The configuration manager will interact with the administrators of both RFMs before dispatching a connector. (In the case of the network RFM this will initialize a network session with the desired client of server.) In the same way connection requests by the network (as a RFM) are handled. Thus all RFMs interface with the configuration manager in an identical manner. The configuration manager can very well be extended and put in charge of more complex actions like the creation (and eventually, the destruction) of copies of a RFM in case of shared access through the 'replication' policy. It seems better to handle this at the level of the RFM administrator however. Of course the configuration manager will be in charge of the creation (initialization) and destruction (termination) of RFMs in general.

5.6. Use of RFMs

The few abstractions on which the given view of the Client/Server model is based, are simple and yet sufficiently powerful to accommodate an integral presentation of any Client/Server architecture. For example, authorization requests are easily understood in terms of the request/reply interaction between two suitably defined RFMs. The view requires that all clients and servers are explicitly specified as RFMs, i.e., in terms of the entry functions they provide for local and remote use. All additional services like logging and security services can be fit into the framework. Note that such services can be realized by either enhancing the internal specification of the RFMs (without changing their interface to the other RFMs) or by introducing 'filters' between RFMs. Even the functionality of connectors can be enhanced to provide for additional facilities. For example, connectors may be designed for governing one-to-many rather than one-to-one message exchanges and the necessary concurrency control. It should be an interesting research project to develop the complete Client/Server model along the lines suggested. All mechanisms

described can be implemented as a 'common kernel' and thus are the basis of a possible design of the CCM and CSM modules.

6. Binding and unbinding

We continue our exploration of the Client/Server model and discuss the issues dealing with binding and unbinding. (We will use the Client/Server model as understood up to Section 4 and make no reference to the abstractions proposed in Section 5.) In the Client/Server model, binding is the act of connecting a 'logical reference' to a 'physical object'. Binding involves naming and server location, port determination and activation, and connection opening. Binding leads to a connection between client and server, and facilitates the exchange of request/reply messages between the two. Unbinding is the act of disconnecting (releasing) a binding.

6.1. Naming

The *naming problem* in distributed systems is well-known. Names are needed to (uniquely) identify objects and operations but are often required to be location-independent. It follows that a mechanism (a mapping) is required to map names to addresses, in some way or another. The naming policy adopted in a Client/Server model is important, and must be convenient at the application level and efficient when it comes to binding. (Also, the uniqueness or non-ambiguity of a name should be easy to verify during name assignment.) Mullender [24] distinguished four different naming policies that prevail in distributed systems for determining system names for services:

(i) *domain naming*: a system name consists of a host name and an object name on this host. The policy is simple but not very flexible, as it does not allow objects to move through the network without changing their system name.

(ii) *global naming*: a system name is any globally unique name. In this policy names are not bound to the location of an object, but now a (centralized or distributed) name server is needed to complete the binding. The name server should be at a well-known address, and is in charge of maintaining the name-location information. The binding procedure may augment a system name by a port name (both at the client's and the server's end) to further identify a temporary access point for the connection.

(iii) *static port naming*: a system name is a 'well-known' port. In this policy ports are associated with services rather than objects. A client wishing to connect to a service only needs to know the (network-wide) well-known port. The service provider sees to it that an object (a process) is maintained for every port that is accessed.

(iv) *dynamic port naming*: a system name is a port that is created and managed by a 'user'. When a server decides to offer a service to the network, it generates a port for that service, announces the port's name to potential clients, and starts 'listening' to the port (i.e., it is willing to engage in a connection at this port). The action of the server can be triggered by a signal from a particular client, which may have allocated a reply port at its end (but which the server does not have to know).

6.2. Binding

Binding normally refers to the mapping of user-level names to system names. A system name should be sufficient for the transport level to determine host addresses and routes. Distributed name servers and dynamic port naming are current research topics. Port determination policies for the Client/Server model are described in more detail in e.g. Bershad et al. [3]. We will assume that the basic client and server processes are 'pre-activated', and do not go into the details of 'activation through binding' (or 'auto-activation'). In dynamic (extendible) networks the naming problem is more complicated, because new hosts must be assigned new and preferably compact names that are unique on the network. Typical solutions either use centralized name servers or employ a distributed scheme in which existing nodes have sets of names available that can be assigned to new nodes that are attached to it. (The latter leads to 'dynamic domain naming'.)

6.3. Establishing connections

The next, essential part of binding concerns the opening of a connection (session, association) between the client and the server. Naming and port determination are implicit in the actions taken to establish a connection. The common protocols for binding (see e.g. Knowles et al. [16]) are all based on a 2-way handshake, using the following primitives:

(i) CONNECT.*request*
(ii) CONNECT.*indication*
(iii) CONNECT.*response*
(iv) CONNECT.*confirm*

(Only in datagram networks should a 3-way handshake be used, for reliably opening and closing connections. We do not consider this in the present paper.) The primitives are usually prefixed with a letter that identifies the layer of service in which they are implemented (e.g. *T* for 'transport layer'). Thus, at the session level a connection can be opened by performing a T.CONNECT.*request* and waiting until a corresponding T.CONNECT.*confirm* is performed. The ISO standard (see e.g. [14]) specifies in detail what components can be distinguished in the whole process. The T.CONNECT.*request* can be viewed as a primitive but will be implemented using lower level CONNECT.*request* and CONNECT.*confirm* primitives. Using the model from before, the T.CONNECT.*request* (at the client's end) must begin by naming a reply port and setting up a connection record. Now two policies can be followed, depending on the model assumptions:

(i) *connect-to-server*: the client possesses the well-known address of the server. It sends the proper TPDU ('Transport-Protocol-Data-UNIT') to the server and triggers its T.CONNECT.*indication* procedure.
(ii) *connect-to-port*: the client waits until the service it wants announces its port name on the network. When the port is known, the client sends off its T.CONNECT.*request* to the port.

In both cases the T.CONNECT.*indication* primitive at the server will name a request port which the potential client should address when sending messages. In general the T.CONNECT.*request* primitive is used to convey a client's requirements (in terms of required resources), and the other primitives arrange for these requirements and inform the requester of the quality of service that will be provided (if at all).

Once a connection (session, association) has been established, 'data' can be transfered. In most Client/Server architectures, data can be transfered in the standard '2-way simultaneous' mode. No tokens are needed and, indeed, no token management primitives need to be provided. Logging services are required for recovery purposes, and 'minor/major synchronization' must be provided for in the session maintenance protocol.

6.4. Releasing connections

The protocols for unbinding (release, close) are now based on a slightly different handshaking principle, using the following primitives:

(i) DISCONNECT.*request*
(ii) DISCONNECT.*indication*

Either party in a connection can initiate a DISCONNECT.*request*, but normally it will be the connection initiator that does so after its use of the desired services. A DISCONNECT.*request* by the client results in an ACK as soon as the server detects that there are no outstanding messages of the client. The server subsequently issues a DISCONNECT.*request* from its end and a similar procedure repeats. Ultimately both parties close the connection.

6.5. Unbinding

A Client/Server architecture must likewise provide primitives for unbinding, following the ISO standards. Unbinding is achieved by releasing the connection record and removing the port that existed for the connection, by suitable actions at the sending and the receiving end.

7. Managing atomic actions

The Client/Server model assumes that clients and servers interact strictly on a request/reply basis. In applications it may be desirable that a client and a server interact in a more complex manner during a session and engage in an activity (a set of operations or 'an action') that affects the information stored at the client and the server simultaneously and in an indivisible manner. Activities of this kind are called 'transactions' or 'atomic actions'. Atomic actions would pose no particular problem if it weren't for the fact that in all realistic applications 'exceptions' and 'failures' (like link failures or site crashes) can occur during an atomic action. We will use the term 'failure' to refer to any abnormal condition that arises during an atomic action. The possibility of failures requires that the effects of an atomic action must be recoverable at all times throughout its elaboration, until the atomic action can be regarded as 'safely completed' at both ends. The implementation of atomic actions thus requires two basic facilities (see e.g. Gray [10]):

(i) a *recovery mechanism*, i.e., a mechanism for 'undo-ing' the effect of one or more atomic actions. Recovery mechanisms are always based on the use of logs that record information on the processing of atomic actions at each site, and on a method for effectuating a rollback. Logs must be recoverable in case of failures and thus must be kept on 'stable storage'.
(ii) an *atomic commit protocol*, i.e., a protocol for detecting 'safe completion' and committing the effects of an atomic action at both ends. Atomic commit protocols are atomic actions and thus must be recoverable themselves.

The occurrence of a failure at some site does not necessarily imply that the atomic actions in progress must all be aborted and rolled back. It may be possible for a node to recover to a consistent state, based on information in its log. (This is called 'independent recovery'.) After recovery, a site may wish to have an atomic action re-started.

7.1. Implementation of atomic actions

The implementation of atomic actions in the context of possible failures is a well-studied problem. In this section we will only discuss some aspects of the atomic commit problem, as it will appear in enhanced Client/Server architectures. An excellent introduction to concurrency control and recovery in distributed databases was given by Bernstein et al. [2]. Also, a detailed recommendation for the implementation of atomic commit protocols appears in the CCR standard of ISO [13]. It suggests that atomic commit protocols follow some version of the well-known *2-phase commit protocol* due to Gray [10] and Lampson & Sturgis [19], and use the following primitives:

(i) C - BEGIN
(ii) C - PREPARE
(iii) C - READY
(iv) C - REFUSE
(v) C - COMMIT
(vi) C - ROLLBACK
(vii) C - RESTART

Our main goal here will be to give a more refined and complete presentation of the protocol than is usually given (cf. [2,6]) and prove its correctness. As it will require no extra effort, we will describe the protocol for the more general case of a client-initiated atomic action that involves multiple servers. It is assumed that the client remains the 'coordinator' (or 'superior') of the atomic action and thus of the atomic commit protocol. The client will only initiate the commit protocol (with a C-PREPARE primitive) if it has reason to do so, i.e., if the activity of the atomic action at its own site has ended (which implies that the servers have provided their operation results insofar as needed by the client). At all times during the atomic action, the client and the servers must be ready to honor a C-ROLLBACK or C-RESTART request from any party in the atomic action. Note that the servers only communicate with the client, but not with each other (during the atomic actions).

7.2. Atomic commit protocols

Applied to an atomic action, an atomic commit protocol is a distributed algorithm for the client and the servers that should guarantee that they all commit or all abort the atomic action. Following Bernstein et al. [2] the situation for an atomic commit protocol can be rephrased in more precise terms as follows. Each party (client or server) may cast exactly one of two votes: C-READY ('yes') or C-REFUSE ('no'), and can reach exactly one of two decisions: C-COMMIT ('commit') or C-ROLLBACK ('abort'). An atomic commit protocol must satisfy the following requirements:

AC1 : A party cannot change its vote after it has cast a vote.
AC2 : All parties that reach a decision reach the same decision.
AC3 : A party cannot change its decision after it has reached one.
AC4 : A C-COMMIT decision can be reached only if all parties voted and voted C-READY.
AC5 : If there are no failures and all parties voted C-READY, then the decision C-COMMIT will be reached by all parties.
AC6 : Consider any execution of the protocol in the context of permissible failures. At any point in this execution, if all current failures have been repaired and no new failures occur for a sufficiently long period of time, then all parties will eventually reach a decision.

The requirements can be viewed as the minimal correctness criteria for atomic commit protocols. (Except for AC1, the requirements are taken from Bernstein et al. [2].) AC1 through AC5 can usually be satisfied quite easily, but AC6 requires that a suitable recovery procedure is part of the protocol. Note that after voting C-READY, a party (client or server) can not be certain of what the decision will be until it has received sufficient information to decide. Until that moment, we say that the party is 'uncertain'. If a failure occurs that cuts an uncertain party off, then this party is said to be 'blocked'. A blocked party cannot reach a decision until after the connection to the other parties has been restored. Blocking is usually concluded if no messages arrive during a certain timeout interval within the uncertainly period. (Heuristic commit protocols allow a blocked party to make a calculated guess of the decision. In some versions of the protocol a blocked client is allowed to commit.) A different situation arises when a party fails (crashes) during its uncertainty period. In this case a more involved recovery procedure may have to be followed (see Bernstein et al. [2] or below).

7.3. The 2-phase commit protocol

The standard (2-phase) atomic commit protocol is as follows, in terms of the recommended CCR primitives. (Note that the desired steps of the recovery procedure after failure are part of the overall protocol, but these are not included in the basic specification below.)

2-PHASE COMMIT PROTOCOL

Client's Commit
c-0. Vote ready;
Phase 1
c-1. **Write** 'prepare' record **to** the Log;
c-2. **Send** C-PREPARE messages **to** all servers; **activate** timer;
c-3. {Await answer messages (C-READY or C-REFUSE)
from all servers using a timer and act as follows}
Case condition **of**
c-3.1. Timeout or C-REFUSE message received:
begin
Write 'rollback' record **to** the Log;
Send C-ROLLBACK messages **to** all servers;
C-ROLLBACK
end;
c-3.2. All servers answered and answered C-READY:

continue with Phase 2
end;
{End of Phase 1}
Phase 2
c-4. **Write** 'commit' record **to** the Log;
c-5. **Send** C-COMMIT messages **to** all servers; **activate** timer;
c-6. {Await answer messages (ACK) from all servers using a timer and act as follows}
Case condition **of**
c-6.1. Timeout: **Write** 'incomplete' record **to** the Log;
c-6.2. All servers answered (ACK):
Write 'complete' record **to** the Log;
end;
c-7. C-COMMIT;
{End of Phase 2}

Server's Commit

Phase 1

s-1. **Await** a C-PREPARE message **from** the client **using** timer;
{The server will only pass this point if it has indeed
received a C-PREPARE message from the client or timed out
}
s-2. **If** not timed out **then** Vote ready or refuse;
s-3. **Case** condition **of**
s-3.1. ready:
begin
Write 'ready' record **to** the Log;
Send a C-READY message **to** the client;
continue with Phase 2
end;
s-3.2. refuse:
begin
Write 'refuse' record **to** the Log;
Send a C-REFUSE message **to** the client
end
s-3.3. Timeout:
Write 'refuse' record **to** the Log
end;
{End of Phase 1}

Phase 2

s-4. **Await** a decision message (C-COMMIT or C-ROLLBACK) **from** the client **using** timer;
{The server will only pass this point if it has indeed
received a decision message from the client or times
out}
s-5. **Case** condition **of**
s-5.1. a C-COMMIT message with received:
begin
Write 'commit' record **to** the Log;
Send ACK message **to** the client;
C-COMMIT
end;
s-5.2. a C-ROLLBACK message was received:
begin
Write 'rollback' message **to** the Log;
C-ROLLBACK
end;
s-5.3. Timeout:
take whatever action to deal with blocking
end;
{End of Phase 2}

Although it is not explicitly specified, each party can unilaterally decide for a C-ROLLBACK at any time if it has not (yet) voted 'ready'. We assume that the Client's Commit protocol is only initiated by the client after it has voted 'ready'. If a client never votes or votes 'refuse', then it never sends a C-PREPARE message and the servers automatically time out eventually in their protocol. (We could have treated the client as a server in the protocol, but have chosen not to do so in order to save messages.) We require that C-COMMIT messages are ack'ed, but this can be omitted from the protocol without any harm. (We do not require that C-ROLLBACK messages are ack'ed, but could incorporate it easily if desired.) Voting essentially amounts to determining whether the local activities in an atomic action have ended and properly been logged or not, but we only need it as an 'abstract' operation. Where ever a 'Log' is specified in the protocol, the local (client or server) log is meant. A 'Phase 2' is not entered automatically after a 'Phase 1', but only on an explicit 'continue'.

7.4. Correctness proof

It should be an interesting research topic to develop a completely formal 'correctness proof' for the 2-Phase Commit Protocol. We outline a less formal proof here. We refer to line-numbers as c-0, c-1, etcetera.

THEOREM 7.4.1. *If no timeouts and no failures occur, then the 2-Phase Commit Protocol is correct.*

PROOF. The requirements AC1 through AC5 are trivially satisfied. For AC2, note that if any party (client or server) decides spontaneously for C-ROLLBACK when it can, then all parties must follow suit eventually and cannot decide for anything else. AC6 is vacuously true. □

The next step is to consider the possibility of timeouts and a limited type of failures, namely 'loss of protocol messages'. In all cases except one, message loss necessarily leads to a timeout and thus it is sufficient to consider the latter only. The one exceptional case can arise in c-3, when some messages (including a C-REFUSE) have arrived but some have not and the clients acts on the C-REFUSE. In this case the client acts just like it would have in the case of timeout (line c-3.1.). Note that heavily delayed messages are considered 'lost'.

THEOREM 7.4.2. *If timeouts and loss of messages can occur but no server gets blocked, then the 2-Phase Commit Protocol is correct.*

PROOF. The requirements AC1, AC3 and AC4 are trivially satisfied. For AC2, observe the following. If a server times out on s-1, it will never send a message and can only decide C-ROLLBACK (if it ever decides, cf. s-3.3.). The client necessarily executes c-3.1. and decides for C-ROLLBACK too. Other servers either time out on s-1 as well or receive the C-ROLLBACK decision in s-4. By assumption no server times out on s-4 (the blocked case). If the client times

out on c-3.1., then it decides C-ROLLBACK and the servers can only reach the same decision by the very same argument. If the client times out on c-6.1., the only possible decision in the system is C-COMMIT and all servers must be in Phase 2. As we assume no blocking, all servers will eventually decide C-COMMIT. Thus AC2 is satisfied and, by the latter argument, AC5 as well. AC6 is vacuously true. □

In order to get any further we must some how deal with the problem of blocking. A blocked server timed out on s-5.3. and thus knows that it is blocked, but it is uncertain of the decision that may have been reached. In order to keep the 2-Phase Commit Protocol correct, a Server's Commit Termination Protocol must be added. The purpose of the Server's Commit Termination protocol is to enable a blocked server to determine the (apparent) decision reached in the system. Several possible strategies for a successful Server's Commit Termination protocol have been proposed, all based on polling. (For example, if another server can be reached and it appears to have timed out on s-1, then the blocked server can decide C-ROLLBACK.) But no Server's Commit Termination protocol can guarantee that it will remove the possibility of blocking. (For example, if a blocked node is cut off permanently, it will forever remain uncertain.) We assume that any server can eventually reach the client again and thus a simple polling of the client will do as a Server's Commit Termination protocol (cf. requirement AC6). We conclude the following result.

THEOREM 7.4.3. *If timeouts and loss of messages can occur, then the 2-Phase Commit Protocol enhanced with the Server's Commit Termination protocol is correct.*

The final step is to allow timeouts and arbitrary failures, i.e., 'loss of protocol messages' and 'site crashes'. If a site crashes permanently, the 2-Phase Commit Protocol will simply continue in the remaining sites and perform as if the messages of the crashed site are lost from some point onwards. The Server's Commit Termination protocol should be extended in this case and somehow detect permanent crashes of the client. In general there is no guaranteed solution that avoids blocking, if permanent crashes can occur. Thus we assume that each site that crashes eventually recovers and resumes the commit protocol. We also assume that a site can actually recover to the point where it crashed, using the information in its log. The only problem now is to determine how to continue with the commit protocol, knowing that the other sites may have advanced in it after the crash. We present a possible recovery protocol below.

COMMIT RECOVERY PROTOCOL

Client's Commit Recovery

cr-1. **If** the client crashed before c-4 **then**
begin
Write 'rollback' record **to** the Log;
Send C-ROLLBACK messages **to** all servers;
end;
cr-2. **If** the client crashed after c-4 but before c-6.2. **then**
begin
Write 'commit' record **to** the Log;
Send C-COMMIT messages **to** all servers; **activate** timer;
{Await answer messages from all servers using
timer and act as follows}
Case condition **of**
Timeout: **Write** 'incomplete' record **to** the Log;
All servers answered:
Write 'complete' record **to** the Log
end;
C-COMMIT
end;
cr-3. **If** the client crashed after c-6.2. **then**
begin
perform c-7 if necessary
end;

Server's Commit Recovery
sr-1. **If** the server crashed before s-4 and without having executed s-3.1.
then
begin
C-ROLLBACK
end;
sr-2. **If** the server crashed after s-4 while being uncertain
then
begin
use the Server's Commit Termination protocol
to remove blocking
end;
sr-3. **If** the server crashed after s-4 while being certain
then
begin
perform remaining activity if necessary in the
C-COMMIT or C-ROLLBACK
(whatever applies)
end;

We assume that **Write/Send** commands in the 2-Phase Commit Protocol are atomic, and thus no crash occurs 'in between' a **Write** and the subsequent **Send.** (While the assumption is reasonable, it would nevertheless be of interest to analyse the protocol if this assumption is not made.)

THEOREM 7.4.4. *The 2-Phase Commit Protocol enhanced with the Server's Commit Termination protocol and the Commit Recovery Protocol is a correct atomic commit protocol.*

PROOF. If the client crashes before c-4, then each server either has C-ROLLBACK as the only option or is uncertain. Thus cr-1 is a correct recovery action, in the sense of satisfying the requirements. If the client crashes in its Phase 2, then each server has either decided C-COMMIT or is uncertain. Replaying Phase 2 (in so far as it is needed according to the recovery log) is again the right action, assuming that the servers detect duplicate messages. If a server crashed in its Phase 1 and without sending a C-READY message, then the remaining sites will have progressed on the assumption that its messages are C-REFUSE or 'lost'. It means that the remaining sites are on their way to decide C-ROLLBACK, and sr-1 is fully consistent with this. If a server crashed after having sent a C-READY message, then either it had decided (and the decision is recovered) or is uncertain. Thus sr-2 and sr-3 are the actions to take. With the earlier analyses it easily follows that the complete protocol satisfies AC1 through AC6 and hence is correct. □

REFERENCES

1. J.G.P. BARNES (1982). *Programming in Ada*, Addison-Wesley Publ. Comp., London.
2. P.A. BERNSTEIN, V. HADZILACOS, N. GOODMAN (1987). *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publ. Comp., Reading, Mass.
3. B.N. BERSHAD, D.T. CHING, E.D. LAZOWSKA, J. SANISLO, M. SCHWARTZ (1986). *A Remote Procedure Call Facility of Heterogeneous Computer Systems*, Techn. Rep. 86-09-10, Dept. of Computer Science, University of Washington, Seattle, Wa.
4. A.D. BIRRELL, B.J. NELSON (1984). Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 39-59.
5. CCITT (1987). *Remote Operations [part 1]: Model, Notation and Service Definition*, draft version, CCITT-COM VII-116-E.
6. S. CERI, G. PELAGATTI (1984). *Distributed Databases—Principles and Systems*, McGraw-Hill Book Comp., New York, NY.
7. D. COMER (1987). *Operating System Design—Volume II: Internetworking with XINU*, Prentice-Hall Inc., Englewood Cliffs, NJ.
8. J.G. FLETCHER, R.W. WATSON (1978). Mechanisms for a reliable timer-based protocol. *Computer Networks 2*, 271-290.
9. W.M. GENTLEMAN (1981). Message passing between sequential processes: the reply primitive and the administrator concept. *Software—P&E 11*, 435-466.

10. J. GRAY (1978). *Notes on Data Base Operating Systems*, Report RJ2188, IBM Research Lab., San Jose, Ca.
11. IBM (1986). *IBM Programmer's Guide to the Server/Requester Programming Interface for the IBM Personal Computer and the IBM 3270 PC*, first edition, IBM Corp.
12. ISO (1987). *Working Draft Transaction Processing Service Definition*, ISO TC97/SC21 N1715.
13. ISO (1985). *Specification of Protocols for Application Service Elements—Commitment, Concurrency and Recovery*, draft international standard, ISO TC97/DIS9805.2.
14. ISO (1984). *Basic Connection Oriented Session Service Definition*, ISO TC97/SC21 N266.
15. P. JANSON, L. SVOBODOVA, E. MAEHLE (1983). Filing and printing services on a local-area network. *Proc. 8th Data Communications Symposium*, 211-220.
16. T. KNOWLES, J. LARMOUTH, K.G. KNIGHTSON (1987). *Standards for Open Systems Interconnection*, BSP Professional Books, Oxford.
17. W. KOHLER (1981). A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comp. Surv. 13*, 149-183.
18. J.P. KRUYS (1987). *Client/Server Architecture*, 2nd draft, NCR, unpublished.
19. B. LAMPSON, H. STURGIS (1976). *Crash Recovery in a Distributed Data Storage System*, Techn. Rep., Computer Science lab., Xerox - PARC, Palo Alto, Ca.
20. B. LAMPSON (1981). Remote procedure calls. B.W. LAMPSON ET AL. (eds.). *Distributed Systems—Architecture and Implementation*, Lect. Notes in Comput. Sci., Vol. 105, Springer Verlag, Berlin, 365 - 370.
21. B.W. LAMPSON (1981). Atomic transactions. B.W.LAMPSON ET AL. (eds.). *Distributed Systems—Architecture and Implementation*, Lect. Notes in Comput. Sci., Vol., 105, Springer Verlag, Berlin, 246-265.
22. M.S. MCKENDRY, M. HERLIHY (1986). Time-driven orphan elimination. *Proc. 5th IEEE Symp. Reliab. Distrib. Software and Datab. Syst.*, 42-48.
23. J.E.B. MOSS (1985). *Nested Transactions—An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, Mass.
24. S.J. MULLENDER (1985). *Principles of Distributed Operating System Design*, Ph. D. thesis, Free University, Amsterdam.
25. J.L. PETERSON, A. SILBERSCHATZ (1983). *Operating System Concepts*, Addison-Wesley Publ. Comp., Reading, Mass.
26. S.K. SHRIVASTAVA, F. PANZIERI (1982). The design of a reliable remote procedure call mechanism. *IEEE Trans. Comput. C-31*, 692-697.
27. M. SLOMAN, J. KRAMER (1987). *Distributed Systems and Computer Networks*, Prentice-Hall Int. (UK) Ltd., London.
28. A.Z. SPECTOR (1982). Performing remote operations efficiently on a local computer network. *Comm. ACM 25*, 246-260.

29. L. SVOBODOVA, B. LISKOV, D. CLARK (1979). *Distributed Computer Systems: Structure and Semantics*, Techn. Rep. MIT/LCS/TR-215, Lab. for Comput. Sci., MIT, Cambridge, Mass.
30. A.S. TANENBAUM (1988). *Computer Networks*, 2nd Ed., Prentice-Hall Inc., Englewood Cliffs, NJ.
31. A.S. TANENBAUM, R. VAN RENESSE (1986). *Reliability Issues in Distributed Operating Systems*, Rapport IR-120, Dept. of Mathematics and Computer Science, Free University, Amsterdam.
32. G. TEL (1987). *Assertional Verification of a Timer-Based Protocol*, Techn. Rep. RUU-CS-87-15, Dept. of Computer Science, University of Utrecht, Utrecht, The Netherlands.
33. R.W. WATSON (1981). Distributed system architecture model. B.W. LAMPSON ET AL. (eds.). *Distributed Systems—Architecture and Implementation*, Lect. Notes in Comput. Sci., Vol. 105, Springer Verlag, Berlin, 10-43.
34. N. WIRTH (1982). *Programming in Modula-2*, Springer-Verlag, Berlin.