A Survey of Issues in Remote Procedure Calls

Yusheng Liu

Technical Report 422
October 1991

Basser Department of Computer Science

The University of Sydney

N.S.W. Australia 2006
A Survey of Issues in Remote Procedure Calls

Yusheng Liu

Basser Department of Computer Science
The University of Sydney, N. S. W. 2006 Australia

ABSTRACT

Remote procedure call (RPC) is a useful paradigm for building distributed systems. Many RPC systems have been developed around the world. They address various design and implementation issues of RPC and demonstrate many similarities and differences in handling them. In this report, such issues are discussed and how they are handled by different RPC systems is examined in some detail. Comparisons of various approaches to these issues are also presented.

1. Introduction

Remote procedure call (RPC) [Nelson 81], [Lampson 81] has proved to be a useful paradigm for building distributed systems across a network. During the past decade, this technique has been extensively employed to construct various distributed systems ranging from special purpose applications to generally useful systems [Carpenter 81], [Bai 87], [Sandberg 86], [Mullender 90]. As a result of meeting different requirements in various environments, many kinds of RPC systems have been developed while constructing distributed systems. They demonstrate a lot of differences in the aspects of binding, semantics, synchronization, parallelism, reliability, exception handling, server management, transport protocols, exchange protocols, interface language, data representation, and security. In this report, we examine how these issues are handled by a set of selected RPC systems. Our aim is to gain insight into these issues so that we can address them appropriately when designing, implementing and standardizing RPC systems. Therefore our selection of RPC systems to be examined is based on the issues discussed, as we are only interested in how an RPC system handles a particular issue and each RPC system has placed its emphasis on some issues. We make no attempt either to make a comprehensive survey of existing RPC systems or to be complete in each system. A survey of some existing RPC systems can be found in [Tay 90].

Before discussing various issues of RPC, let us briefly review the concept of RPC here. Remote procedure call is a mechanism for providing communications between a client process and a server process, usually located on different machines. An RPC involves six components: on the client side, a client procedure, a client stub and a transport entity and on the server side, a server procedure, a server stub and a transport entity. Simply speaking, when a client procedure calls a server procedure, it actually calls a client stub in its own address space. The client stub packs the parameters into a message and invokes the transport entity to send the message to the server machine. The transport entity on the server machine passes the message to an appropriate server stub. The server stub unpacks the parameters from the message and calls the server procedure. After the server procedure is finished, it returns to the server stub. The server stub packs the result into a message and invokes the transport entity to send it back to the client machine. When the reply message is passed to the client stub by the transport entity on the client machine, the client stub unpacks the result and finally returns it to the client procedure.

Although the concept of RPC is simple, the design and implementation of RPC systems involves some complicated issues that are not encountered or can be easily handled in local procedure calls, due to the separate address spaces and uncertainties caused by imperfect communication and partial failures in distributed systems. In the following sections, we will examine such issues in detail and see how various RPC systems deal with them.

2. Binding

Before a calling program can call a called procedure, it is necessary to establish the linkage between the calling program and the called procedure. This is commonly referred to as binding. In the case of local calls, this is accomplished statically by a link editor at compile time. On the other hand, in the case of remote calls, this becomes more complicated because the calling procedure and the called procedure usually do not
reside in the same machine. Thus, the binding in the remote case involves locating the machine in which the expected server procedure resides and the particular server which implements the called procedure. The machine address and the server identification will be used later by the client to send call request messages to the right server.

There are many approaches to binding. The static one is to embed the network address of the server into the client’s code. This approach requires recompilation of the client’s code when the server’s network address changes. The most dynamic approach is to use a replicated centralized database. This way, the client contacts the database to dynamically get the binding information of a server, without caring details such as its identity, location, etc. and there is no need to recompile the client’s code if the server’s address changes. Between these two extremes are some other methods such as looking up local files or databases for binding information.

Now we examine some binding examples. Xerox Cedar RPC developed by Birrell and Nelson [Birrell 84] uses the Grapevine database [Birrell 82] for its RPC binding. Servers willing to provide service to clients register themselves with Grapevine. The name of the interface provided by the server and the server’s network address are stored in Grapevine. Detailed binding information such as interface name, unique identification for a particular registration and pointer to the procedure handling incoming calls (called dispatcher) is maintained locally in a table by the communication package (called RPCRuntime). Clients requesting service first contact Grapevine for server’s address and then the server for binding details, which will be included in subsequent call messages sent to the server machine directly. The RPCRuntime on the server machine uses the binding information to verify legality of the call and invoke the appropriate dispatcher which in turn calls the required procedure. This binding scheme is very reliable and secure because Grapevine is a reliable distributed database with access control and at least three copies of its database entries and because of use of unique identifier for each export. This scheme allows the client to specify only the network address of the interface instance, the name of the interface instance or the type of the interface, providing the flexibility of selection in a spectrum from static binding to dynamic binding. Perhaps a shortcoming of this scheme might be that it requires two calls (one to Grapevine and one to the server) before the first call to a requested procedure can be made.

Apollo RPC [Apollo 87], [Apollo 89], [Zahn 90] uses a scheme similar to that of Cedar RPC, but it requires at most one call to obtain the binding information. An RPC server registers itself with the Local Location Broker (LLB) and the Global Location Broker (GLB) via the Location Broker Client Agent on the local host. Both LLB and GLB themselves are RPC servers that maintain information about objects and interfaces located on the local host and throughout the network or internet respectively. GLB may be replicated at different sites to make binding information highly available. A client sends a lookup call to GLB via the Location Broker Client Agent to locate a server, and uses the information returned by GLB to send RPC calls directly to the located server. If a client knows a server’s host, but not its port number, it can send its first RPC call to the LLB on the server’s host, which will forward the call to the appropriate server. The server sends the client directly the response from which the client can learn the binding information for subsequent calls. This results in an efficient way of binding by reducing the communications between clients and servers. One of the flexible features of Apollo RPC is that its runtime library supports RPC calls with different binding states (unbound, bound-to-host and bound-to-server) by broadcasting the message on the local network, sending it to the specified host and to the specified server respectively.

The port mapper plays a similar role in binding for Sun RPC [Sun 86], [Sun 88] as LLB does for Apollo RPC. It maps local RPC programs and version numbers to port numbers. Sun RPC does not use a central database like Grapevine or GLB, so the client must specify the host name to locate a server. However the port mapper supports broadcast RPC calls to the specified remote program on the same machine. It receives a broadcast call from the client on its well-known port and passes it to the specified procedure. If the procedure is successfully executed, the port mapper sends the client a response containing the result of the procedure and the remote program’s port number; otherwise it sends no response to the client. The client can use the port number to send subsequent calls to the remote program. One of the differences between LLB and the port mapper is that the response from the procedure is sent back to the client by the port mapper while with LLB it is sent to the client directly by the remote program.
3. Call Semantics

One of the fundamental differences between local procedure calls and remote procedure calls is their call semantics, which concern how many times a called procedure is executed. With local calls, this is a trivial issue. If a called procedure returns, the caller knows the procedure is executed exactly once, and if the machine crashes during the execution of a procedure, it happens to both the caller and callee simultaneously and nothing can be done for this by either the caller or callee. Consequently, there is no need for the caller to worry about the number of executions of a procedure. However, with remote calls, this is not the case. Unexpected things such as loss of a request or result message and client or server crashing may happen during a call, resulting in uncertainty of how many times a procedure is executed. If a procedure does not return within a certain time after the client makes a call to it, it may execute once, not execute at all, or even execute partially, due to the loss of the response, loss of the request, or server crash while executing the procedure respectively; and if the client retries, it is possible for the procedure to be executed more than once.

There are three different forms of remote procedure call semantics. A weak one of these is at least once semantics, which mean that the remote procedure is executed one or more times. The client cannot be sure exactly how many times the remote procedure was executed due to the possibility of retrying and the uncertainty of failure types. These semantics can be easily achieved by allowing the client to retransmit the request until the response returns. However, to be useful, the remote procedure must be designed such that the effect of multiple executions of the procedure is the same as that of a single execution. Such procedures are known as idempotent.

A form of RPC semantics employed by many RPC systems is at most once. In this case, the remote procedure is executed at most once. If the procedure returns normally, it was executed exactly once. However if an error is returned, the client is only sure that the procedure was executed at most once, but has no idea of whether it was executed once, not at all or even partially executed.

A strong form of RPC semantics is exactly once. As the name suggests, the remote procedure is executed exactly once in spite of failure. This is difficult to achieve due to the possibility of server crashes, which may require complicated recovery mechanisms such as stable storage, timestamp, etc. to distinguish pre-call crashes and post-call crashes and make the expected data consistent. Very few RPC systems implement this form of semantics.

Most of the RPC systems implement at most once semantics. For example, Codar RPC guarantees that the remote procedure is executed exactly once if the call returns to the caller. Otherwise an exception is reported and the procedure is executed either once or not at all. Both client and server machines use retransmission mechanism to handle lost packets. Usually the result of a call acknowledges the call packet while a new call from the same client acknowledges the result of the previous call. When no acknowledgement is seen after a predetermined time, a retransmission or an explicit acknowledgement will be needed. The RPC runtime package on the server machine filters the duplicated calls using the call identifier to guarantee at most once semantics.

Some RPC systems provide both at most once and at least once semantics. Apollo RPC is such an example. By default, the Apollo RPC runtime software provides at most once semantics. To achieve this, the server saves and retransmits the response until its receipt is acknowledged either explicitly or implicitly by a new request. On receiving a retransmitted request, the server sends the saved response, instead of reinvoking the procedure. The server can detect duplicated requests by keeping track of the sequence number of the previous request for each serviced client. If the server finds no information for a request (e.g., due to discard of the saved sequence number or a first request), the server will call back the client to request the client’s sequence number and use it to validate the request. The callback mechanism itself is an RPC server implemented on the client machine. Apollo RPC implements at least once semantics for idempotent procedures. If a procedure is specified as idempotent, the server does not save the result of the procedure, nor does the client acknowledge its receipt. If the response is lost, the client will retransmit the request, and the server will reexecute the procedure and resend the response. In addition, for those procedures whose result is not important to or expected by the caller, Apollo RPC implements maybe semantics, meaning that the procedure may or may not be called. The RPC runtime system does not guarantee delivery of the call.
One of the major differences between Sun RPC and other RPCs is its call semantics. Because Sun RPC layer does not employ any measures such as timeout, retransmission, and duplicate detection against communication failure, its semantics depend on the type of the underlying transport protocols. This makes the underlying transport protocols untransparent to the application. If the transport is a reliable one such as TCP [Postel 81b], the semantics are at most once. However, if the transport is an unreliable one such as UDP [Postel 80], the semantics can be at least once or at most once, depending on the client and server. If the client simply keeps retransmitting the call message until it gets a response, the procedure will have been executed at least once. If the client resends the call message using the original transaction ID of that message and the server uses the transaction ID to detect duplicate calls, at most once semantics can be achieved.

Unlike other RPC systems, Cambridge Mayflower RPC [Bacon 88], [Hamilton 84] does not attempt to hide remoteness from the programmer. Instead, it makes the fact of remoteness explicit. New syntax is added to the language to facilitate this. The programmer is allowed to make a choice of semantics: MAYBE or EXACTLY ONCE when writing call statements. MAYBE is the default semantics of a remote procedure while EXACTLY ONCE can be chosen by appending the keyword "zealously" after the procedure arguments in the call statement. However, the MAYBE semantics here are somewhat different from the maybe semantics of Apollo RPC. Although both mean that no retry will be made. The former does not mean the results of the procedure are not expected and indeed it applies to all kinds of remote procedures, including those with returned results. If an error occurs during a call, an exception will be signaled by the MAYBE call mechanism. The latter applies only to those procedures whose results are not expected. Also, the meaning of EXACTLY ONCE of Mayflower RPC is different from that of the exactly once semantics commonly defined. EXACTLY ONCE here is achieved only in the absence of node crashes, but this is in fact the at most once semantics provided by other RPC systems. The programmer can also use the keyword "timeout" to specify the interval between retries (for EXACTLY ONCE) or the time after which the call to be abandoned (for MAYBE).

4. Synchronization

There are two modes in which a remote procedure can be performed. A remote procedure can execute synchronously or asynchronously with respect to the caller. With synchronous (or blocking) RPC mode, the caller is blocked until the called procedure returns while with asynchronous (nonblocking) RPC mode, both the caller and the called procedure may proceed in parallel. Although the semantics of synchronous executions are relatively simple and easy to implement, synchronous executions may result in low efficiency if the execution of the remote procedure lasts long. Since the caller and the called procedure are running in different processes, it is possible for the caller to perform some other work that has no dependence on the result of the called procedure during the execution of the called procedure. This may lead to high efficiency. The semantics of asynchronous executions are intended to facilitate this.

Due to the simplicity of the semantics of synchronous executions and their similarity to those of local procedure calls, all RPC systems implement the semantics of synchronous executions although some of them also provide the semantics of asynchronous executions or a weak form of these semantics. A typical example of the RPC systems that provide the semantics of synchronous executions only is Cedar RPC, whose design principle is to make the semantics of remote procedure calls as close to those of local procedure calls as possible. Included in this category are many other RPC systems such as Xerox Courier RPC [Xerox 81a], Mayflower RPC and Modula-V RPC [Almes 86].

MIT Athena RPC [Souza 86], Apollo RPC and Sun RPC are examples of the RPC systems that provide blocking remote procedure calls and some degree of nonblocking remote procedure calls. In addition to blocking RPC, Athena RPC provides nonblocking RPC for applications where neither a result nor any status of the call is expected, e.g., terminal output. Control is returned to the client and the client can proceed immediately after the nonblocking call has been sent. There is no guarantee that the call request will be delivered or the result or the status will be returned. Reliability is left to the end-to-end mechanism. Similarly, Apollo RPC uses the blocking mode as its default execution mode. If a remote procedure is specified as "maybe" in the interface, then nonblocking mode will be applied to this procedure. Procedures specified as "maybe" cannot have any output parameters. Normally, the execution mode of a remote call in Sun RPC is blocking mode. However, Sun RPC also provides a weak form of nonblocking RPC known as batching RPC. In this case, the client first sends a sequence of batch call message to the server without
waiting for responses for these calls, and then a usual blocking RPC call to terminate the sequence of batch calls and flush the pipeline. Batch mode RPC can be used to reduce the overhead and increase the throughput of requests when a stream of related requests can be bundled and sent in one batch to the server, that is, these requests do not expect any response.

5. Parallelism

One of the criticisms of the RPC model is its lack of parallelism [Tanenbaum 88], which may result in inefficiency in some situations. This is due to the attempts to make RPC semantics close to those of local procedure calls that are inherently not parallel. However, if we do not insist on this, we can obtain parallelism. For example, the nonblocking RPC mechanism can be used to achieve parallelism between the client and server, as seen above. In this section, we will examine another mechanism, parallel RPC, used to obtain parallelism among different servers.

A parallel RPC mechanism allows a procedure to be executed at different address spaces in parallel. Normally, the caller is blocked while parallel procedures execute although there is no reason to prohibit using nonblocking RPC mode for parallel RPC calls. Handling of responses from multiple servers varies, depending on applications. Parallel RPC systems should provide mechanisms to facilitate processing of responses and abortion of parallel RPC calls.

There exist a few parallel RPC systems. PARPC [Martin 87] is a parallel RPC system that provides a high level execution model for programs making parallel procedure calls. It is built on top of the TCP/IP [Postel 81a], but does not depend on the multicast or broadcast protocol. A parallel procedure is invoked by a normal call statement followed by a result statement. The first parameter of the call specifies the set of n different address spaces where the procedure is to be executed, and the rest (if any) pass data to or return results from the procedure. The result statement is a language construct specifying how to process replies from servers. Both blocking and nonblocking execution modes are allowed. If a result is expected, the caller is suspended while the n procedures execute. When a reply arrives, the caller is unblocked to execute the result statement to handle the result and reblocks to await the next reply. Depending on the result statement, this process may continue until all the replies are processed or terminate prematurely, abandoning all unprocessed replies. On the other hand, if no result is expected, the caller can proceed in parallel with the parallel procedures. PARPC has been used to implement the Gemini replicated file system [Burdick 87].

As an extension to a non-parallel RPC mechanism called RPC2 [Satyanarayan 86], MultiRPC [Satyanarayan 90] allows a client to invoke multiple remote servers. The motivation of MultiRPC was to provide a solution to the problem of updating a potentially large number of cached files on workstations encountered in the distributed computing environment Andrew [Morris 86]. This problem could not be solved satisfactorily by other mechanisms such as iterative RPC, multicast or broadcast and multiple threads of control. MultiRPC is supported by three routines: runtime support routine Multirpc and language support routines MakeMulti and UnpackMulti. The syntax of a MultiRPC call is different from that of a local call. However, MultiRPC retains similar semantics of many other RPC systems. A client invokes a remote procedure via MakeMulti, which in turn calls the runtime system routine Multirpc with input being a request packet marshalled by MakeMulti, a list of connections and a client handler routine passed from the client. The routine transmits the request packet to a list of servers and waits until all replies have arrived or until the call is terminated by the client handler. The client handler is functionally equivalent to the result statement in PARPC. Unlike PARPC, MultiRPC runs on the UDP/IP protocol. The internal routine SendPacketReliably is responsible for retransmission, failure detection, and result gathering. The routine UnpackMulti unpacks parameters in request and reply packets. Like PARPC, MultiRPC does not depend on multicast or broadcast support. Although MultiRPC was evolved from RPC2, there is no interference between them so that no changes to RPC2 clients or servers are needed. In fact, MultiRPC and RPC2 calls can appear in the same client in any order and the server cannot even distinguish between them.

Circus is a replicated procedure call mechanism for constructing fault-tolerant distributed programs [Cooper 84], [Cooper 85], [Cooper 86]. It extended a conventional RPC system [White 85] to support many-to-many communication, consequently, achieving parallelism at both the client and server sides. Replicas of a client or server module form a client or server troupe that constitutes replicated distributed programs. When a client troupe makes a replicated call to a server troupe, each module of the client troupe
sends a call request to each module of the server troupe, and each server module executes the call only once and returns the result to each client module. By default, each client or server module does not proceed until all return or call messages arrive, but routines called collaborators, similar to the client handler in MultiRPC, can be specified to change this "unanimous" approach to the "majority" or "first-come" approach. Like MultiRPC, Circus uses UDP/IP as its underlying protocol and requires no multicast or broadcast support.

Some RPC systems depend on broadcast protocols to provide parallelism. Two examples of such systems are Sun RPC and Apollo RPC. Both of them support broadcast RPC calls. A Sun RPC client invokes a broadcast call and receives responses for the call via the port mapper. Only successful responses are returned to the client through the port mapper and like MultiRPC, they are processed by a result handler routine. An Apollo RPC client sends a broadcast call directly to all the servers on the local network, and waits for the first reply only, with subsequent replies discarded.

6. Reliability

The reliability of RPC systems is one of the most difficult problems facing RPC designers, which has attracted many researchers' attention [Cooper 84], [Aschmann 88], [Yap 88], [Panzieri 82], [Panzieri 88], [Pappalardo 88]. Although distributed systems intend to provide users with highly available systems as one of their goals, at the same time they introduce more potential failures in the systems not usually encountered in a centralized system. These include communication related failures such as messages being lost, corrupted, out of order and duplicated as well as independent node crashes. To achieve the goal of high availability, a distributed system must cope with these problems properly. Communication related failures are relatively easy to handle and there are existing techniques available for handling them, for example, sequence number, checksum and timeout. In fact, almost all RPC systems either depend on reliable transport protocols or use some kind of these techniques on top unreliable protocols to obtain reliable communications. The partial failure problems due to independent node crashes are more difficult to solve and they are inherent in distributed systems. In this section, we will mainly examine some approaches to these problems adopted by researchers.

Replication of a software module on a set of nodes provides a measure to prevent distributed programs from failing due to individual node crashes. Basically, there are two replication based approaches to fault tolerance against node crashes in distributed systems, the modular redundancy scheme and the primary-standby scheme. In the modular redundancy approach, replica of a module are running on a set of nodes in parallel without distinction between them. The correct result can be determined based on voting. In the primary-standby approach, one of the replicas is designated as the primary and performs the required task while others are designated as passive backups. The status of the backups are updated through checkpoints. If the primary fails, one of the backups will take over as the primary and continue the execution starting from last checkpoint.

As noticed from the previous section, Circus uses the modular redundancy approach to achieve high reliability. The high reliability of a distributed program is guaranteed by means of replication of both the client and server modules on a set of separate nodes. The set of replicas of a module forms a troupe. A distributed program constructed from troupes will continue to function as long as at least one member of each troupe is still running. The system preserves replication transparency to the programmer and the troupe members. Inter-troupe interaction is carried out via replicated remote procedure calls. Each replicated procedure call actually results in a set of normal remote procedure calls between the client and server troupe members, but these details are invisible to the programmer. Members within a troupe are not aware of one another's existence and do not communicate among themselves. Thus the replication is transparent to individual troupe members. This property makes Circus different from other fault-tolerant software systems. Replication transparency is guaranteed by troupe consistency, which means that all the members of a troupe behave in the same way. A noticeable property of the Circus replicated procedure call facility is its replication flexibility. The replication is flexible in that the granularity of replication is a module rather than a whole program so that the degree of replication of a module can be chosen based on the importance of the module. Furthermore, the degree of replication of individual modules can be changed independently and dynamically during the execution of a program to reduce the vulnerability of a diminished troupe. This is done using mechanisms such as process migration and checkpointing. A shortcoming of Circus might be the high overhead from the head module overhead. A replicated procedure call between an m-module client troupe and an n-module server troupe requires totally 2(m+n) messages (requests and replies) to be transmitted between the
client and server troupes.

In contrast to Circus, the resilient Remote Procedure Call (rRPC) system [Aschmann 88] employs the primary-standby approach to provide resilient distributed computation. The system depends on the broadcast protocol to exchange messages. A client or server is replicated on several nodes, one replica acting as the primary and the others as backups. To invoke a remote procedure, the client primary sends the request and the server primary receives it, executes the procedure and returns the reply. The backups monitor the communications over the network. The client backup logs the message sent and received by the client primary and uses the interaction log to move to the crash point when the client primary crashes. The server backup stores the call message to its primary in the list of pending calls and opens a log for that call. If the server primary crashes, the server backup starts executing all the pending calls. The log for a call can be used to skip a finished call by simply using its results. The crash and recovery at the client or server side is transparent to the server or client side. An important feature of the rRPC system is its checkpointing scheme that is based on a logical level (call level) rather than physical level (processor level). The interference with bystanders of a call due to broadcast messages might be a disadvantage of the system although the broadcast protocol facilitates the communications between primaries and backups.

The approach used in the Fault Tolerant Remote Procedure Call (FTRPC) system described in [Yap 88] is a combination of the modular redundancy approach and the primary-standby approach. A procedure is replicated on a set of nodes, called a cluster. Copies of a procedure (known as incarnations) form a linear chain. The head of a chain is designated as the primary incarnation while others are secondary incarnations. A cluster may be both a callee and a caller, in which case the cluster makes a call to another remote procedure. Like Circus, all the incarnations of a cluster execute the procedure simultaneously, but message exchanges between the caller cluster and the callee cluster is through the primary incarnation of the each cluster, rather than all the incarnations in both clusters, as in Circus. Within a cluster, messages are propagated from the primary incarnation to secondary incarnations. If a secondary incarnation fails, messages are propagated to its immediate subordinate. If the primary fails, its subordinate can assume its role and act as the primary very quickly since all the incarnations are active concurrently. The system uses the RPC mechanism itself to detect the failure of an incarnation. All messages (except acknowledgements) are acknowledged. If no acknowledgement for a message is received within a defined time period, the sender checks the status of the receiver using a dummy RPC call and timeout. Duplicate messages are detected using a network-wide unique sequence number consisting a unique program-ID and a counter value. The method for generating network-wide unique sequence number here does not involve the complexity of managing tokens in Le Lann’s circulating token method [Le Lann 77] and clock synchronization in Lample’s synchronized clock method [Lample 78]. The FTRPC system avoids the disadvantages of both the primary-standby approach and the modular redundancy approach. It does not require complicated checkpointing and logging mechanisms of the primary-standby method and a large number of message exchanges between the clusters of the modular redundancy method. A call between an m callee cluster and an n callee cluster requires O(m+n) messages rather than O(m*n) messages, as in Circus. However, the system efficiency is limited by the message propagation scheme. The response time of a call is increased linearly with the increase of the degree of the replication as the primary callee does not execute the procedure until the propagation finishes. Unlike Circus, the system is not reconfigurable dynamically.

The systems examined previously in this section aim at hiding application programs from communication and node failures. That is, they attempt to guarantee a normal termination of a call (the reception of a reply message from the called server) even if failures occur. The approaches for this purpose adopted in these systems are all based on replication. In this subsection, we shall describe an RPC mechanism that addresses the fault tolerance problem in a different angle, Rajdoot [Panzieri 88]. Rajdoot makes no attempt to guarantee a normal termination of a call under all failure conditions but it rather guarantees an abnormal termination of a call (no reply received from the called server) under certain failure conditions (e.g., server crashes). The major objective of Rajdoot is to detect and kill orphans, and this is also a major difference between Rajdoot and most other existing RPC mechanisms, which usually do not support orphan detection and killing. Orphans are unclaimed computations left on server nodes due to communication and node failures. They are considered harmful because they can interfere with subsequent computations and consume computation resources on server nodes. Rajdoot employs three mechanisms to handle orphans: deadline, crashcount, and terminator. The deadline mechanism is used to cope with orphans due to communication failures. Each call message contains a deadline telling the server the maximum allowed time for
executing that call. If the deadline expires, then the execution is aborted and the call terminated abnormally on the client side. Therefore it is guaranteed that the computations created by a call are purged if the call terminates abnormally. The crashcount mechanism is used to detect and kill orphans due to a crash of client node before executing a call from the recovered node. Every node maintains a variable whose value is the local clock value at the node's last reboot time as its crashcount and a table of crashcount values and server lists for its client nodes since its last reboot time. A newly created server compares the crashcount value in a call with the corresponding value in the table. If the former is greater, then the server purges all other servers servicing the client node before executing the call. Otherwise the server adds itself, possibly (if no entry for the client node yet) with the crashcount to the table. The terminator mechanism handles orphans on a server node due to crashed client nodes that remain crashed or make no further calls to the server node. Every node has a terminator process that regularly queries suspected nodes about their crashcount values and purges the relevant orphans if it found any crashes. The orphan detection and killing mechanisms of Rajdoot have a few advantages. First, they require little or no stable storage, unlike other orphan handling techniques as in [Nelson 81]. The crashcount needs only a stable clock, which is usually built in modern computers, and the table for recording crashcount values and server lists requires no stable storage. Second, the shared table does not become a performance bottleneck because it needs to be accessed and occasionally updated by newly created servers only for the first call and the terminator process. Third, they require no special recovery facilities. Last, they are cheap because the overhead for the deadline mechanism is little and the terminator process is activated infrequently. A disadvantage of Rajdoot is that the deadline mechanism requires the client to estimate the deadline of a computation.

7. Exception Handling

The reliability issue discussed in the previous section concerns mainly the mask of failures resulting from client and server node crashes as well as communication faults. Closely related to this issue is the notion of exception handling, which in the RPC context concerns the exposure, rather than mask, of the server-side errors to the client and the client's handling of these errors. Reporting an error to the caller is called raising an exception and the caller's reaction to the exception is called handling the exception. In fact, the concept of exception handling is not new and was proposed during the late 60's and the early 70's for dealing with exceptions such as division by zero, array index out of range, etc. in uniprocessor programs [Goodenough 75]. Although there have been some controversies on the usefulness of exception handling and the nature of exceptions [Black 82], [Bull 83], the exception handling mechanism has been provided as a language construct by some modern programming languages such as Ada [DoD 83], Mesa [Mitchell 80], and CLU [Liskov 79], [Liskov 81]. It is more desirable to have an exception handling mechanism in RPC systems, because apart from those exceptions occurring to uniprocessor programs there are other uncertainties such as server crashes and communications broken encountered in RPC programs. The exception handling mechanism allows clients to take some fix-up actions when a handleable error is detected and reported by the server or to terminate to prevent them from hanging forever when a non-handleable error is encountered or when a remote procedure never returns due to some error. In general, an RPC exception handling mechanism works as follows. When an error is detected during a remote call, instead of a result of the procedure, an error code, possibly with some arguments, is returned to the client by the server or the RPC runtime system. Based on the error code, the client invokes an appropriate exception handling procedure to fix the error or terminates if it cannot fix the error. The RPC exception handling mechanisms can be quite different in terms of their complexity and ability to handle exceptions. In the following, we will examine a few RPC exception handling mechanisms provided by some existing RPC systems.

The exception handling mechanism of Cedar RPC is based on that of the Mesa language. There are two types of exceptions: those raised by the callee, which are defined in the interface exported by the callee, and a call fail exception raised by the RPC runtime, which is due to some communication difficulty. When a server detects an exception while executing a procedure, it suspends and returns an exception packet, rather than a result packet, to the caller machine. The RPC runtime on the caller machine raises an exception in the caller process. The exception handling procedure (called catch phrase) in the call stack is executed, with possible arguments conveyed in the exception packet. If the exception handling procedure returns normally, its results are sent back to the server and the execution in the server is resumed. Otherwise the server is informed of the abnormal termination, and the appropriate procedure activations in the server are unwound. Exceptions not defined in the service interface are not handled by the exception handling mechanism and are expected to be handled while debugging. The Cedar RPC exception handling mechanism is powerful
and this is attributed to the exception handling mechanism of the Mesa language. The Cedar RPC system itself requires only facilities to support the transmission of the exception packets and results of exception handling procedures, which can be easily achieved in the transport protocol.

In Courier RPC, a remote program consists of a set of remote procedures and a set of remote errors those procedures can raise. The error data type is used to model remote errors. An error can have arguments of primitive or constructed data type. Like Cedar RPC, remote errors are specified in the service interface. In addition to argument list and results, the definition of a remote procedure in the interface includes a list of errors it may raise. The abort message is used to raise a remote error to the client. Unlike Cedar RPC, Courier RPC provides no special language construct for handling exceptions. However, the client program can use exception handlers written in conventional language constructs to handle exceptions. If the client program does not handle a reported error, the client program is terminated.

The programming language used in Mayflower RPC is Concurrent CLU, extended from CLU [Liskov 81], which has a powerful exception handling mechanism [Liskov 79]. Naturally, Mayflower RPC simply uses the CLU exception handling mechanism as its exception handling mechanism. Potential errors a procedure may raise are specified in the header of the procedure in the interface. Errors can contain arguments of arbitrary type. There are two types of errors in an RPC program: soft errors and hard errors. Soft errors are recoverable errors such as timeout or request congestion and imply it is worth retrying. Therefore they are signaled by procedures with MAYBE call semantics. Hard errors are nonrecoverable errors such as failure to contact the server node and imply there is no need to retry. Exceptions are handled in clients using the language construct for this purpose provided by CLU. The caller of a remote procedure can handle the exception itself, known as masking the exception, or signal the exception up one level to its own caller, known as propagating the exception.

Unlike the RPC systems described above, Sun RPC does not provide an explicit exception handling mechanism. Exception handling depends on the status code returned in the reply message, which is used to distinguish between a successful call and various errors. A set of common errors such as unable to decode arguments or requested procedure unavailable are defined by the RPC system and are reported to the client by a set of server-side RPC library routines. The client checks the status of each call and if there is an error, it invokes a client-side RPC library routine to process the error, usually just printing a canned message on the terminal to bring the exception to the user’s attention. Errors particular to a service can be defined by the user as a set of status codes and reported in a discriminated union in the result parameter of the call by the server. Thus an error can have arguments of any type, just like a result. The client has to check the status code in the result parameter to obtain a successful result or handle an error appropriately.

8. Server Management

Server models vary from implementation to implementation. Server processes are different in terms of the ways of creating them, their lifetime, their client population and whether they share the same address space. There are at least four strategies which can be used to organize server processes to perform procedures. First, server processes can be created statically, remain in existence indefinitely, potentially serve multiple clients and have no shared variables among them. Second, server processes are created on demand by a server manager process, which runs at a well-known address on the server machine. Normally each server process serves only a single client and terminates after finishing a session with that client and it does not share variables with other server processes. Third, a distributor process and a pool of worker processes are created statically and running continuously on the server machine. The distributor process is responsible for receiving call requests and distributing them to worker processes while worker processes execute the calls and return results to clients. A worker process finishing a call goes back to the pool waiting for the next job. A worker process serves multiple clients rather than devotes itself to a particular client. Usually the distributor process and all worker processes have shared variables. This is called lightweight process model. Last, a server is created for each call request and terminates after completing the call.

Sun RPC is an example of the RPC systems that adopt the first server management strategy. Servers are started statically as background processes. Once started, server processes run until they are killed explicitly or crash. They execute call requests from multiple clients on an FCFS basis.

Courier RPC and Rajdoot RPC are typical example systems using the second server management strategy. In Courier RPC, a Courier daemon is running on each node, listening on a well-known XNS
Sequenced Packet Protocol (a connection-oriented protocol) port for client connection requests. A client making remote calls first sends a connection request to the daemon. Once receiving a connection request from a client, the daemon forks a child process to handle the connection and then goes back to listen for connection requests from other clients. The child process verifies that the client is a Courier client and then waits for the client's first call. On receiving the first call, the child process executes the appropriate server program, becoming the server process. Subsequent calls and replies are transmitted directly between the client and the server process. When finishing all the calls, the client closes the connection, causing the server process to terminate normally. Similarly, in Rajdoot RPC, a manager process is running at a well-known address on each node and responsible for creating server processes. The first call from a client to a node is converted by the RPC mechanism into a create-server call to the manager at that node. The manager spawns a server process and continues to receive other create-server calls. The spawned server process sends its own address to the client and the client uses it to send calls directly to the server process. The process of creating a server is transparent to the client program. The RPC mechanism also has facilities to detect possible spurious servers for the same client and abort them. The RPC in Newcastle Connection [Brownbridge 82] has a similar server model.

The method used to organize servers in Cedar RPC is similar to the third server management strategy. Each node maintains a pool of idle server processes and runs an Ethernet interrupt handler as a distributor. The server processes and the interrupt handler have shared memory. Each call/reply packet contains the caller/callee process identifier as its source process identifier and the callee/caller process identifier as its destination process identifier. The destination process identifier in the first call packet of a client is obtained by guess. Based on the destination process identifier, the interrupt handler dispatches a call packet to a server waiting for it (if any) or an idle server process. If the packet is a simple call, the server process executes the call, sends the client a reply and goes back to its idle state. If the packet is a part of a multi-packet call, the server sends the client an acknowledgement and waits for the next packet. The interrupt handler on the client machine delivers packets similarly. In addition, server processes can be created dynamically to meet the demand of a burst of calls and killed to reduce the number of excess idle server processes due to such calls. This scheme reduces the cost of process creation and process swaps, resulting in good performance. Another example using the lightweight process model is the Modula/V RPC system. The server model is based on the notion of teams in the V Kernel [Cheriton 82]. A team consists of a set of processes (with different process identifiers) sharing the same space (team space). One process in the team is designated as the dispatcher process by the RPC runtime routines to receive and dispatch calls. If the call is easy to handle (e.g., compute-bound with no further remote calls), the dispatcher does the job itself. Otherwise, the call is dispatched to a worker process within the team for execution. Due to the constraints of the V IPC primitives, the system provides no good way for worker process to reply to the client directly and to notify the dispatcher to reply.

Although the one-process-per-call strategy is not common, it is used in the RPC in Argus, which is a programming language and system developed to support the implementation and execution of distributed programs [Liskov 83a], [Liskov 83b]. In Argus, a program consists of one or more models called guardians. A guardian consists of a collection of data objects and processes to manipulate on them. Access to the objects within a guardian from other guardians is through a set of operations called handlers. When a handler call (i.e., remote procedure call) arrives, the Argus runtime system creates a process to execute it. Once the call is finished, a reply is sent back to the caller and the process is terminated. The runtime system is responsible for delivery of the reply. In this scheme, there is no need to retain any state between calls since they are executed in different processes. This scheme results in simplified and safe programs.

9. Transport Protocols

RPC packages are built on top of a transport layer. The transport layer is responsible for transmitting call requests and replies between clients and servers. There are two types of transport layer protocols: connection-oriented and connectionless. The connection-oriented protocols are reliable since they guarantee that a packet arrives exactly once, in the right order and uncorrupted. A typical example of this type is TCP (Transmission Control Protocol). The connectionless protocols are unreliable in that they provide no such guarantees as the connection-oriented protocols do. A typical example of this type is UDP (User Data Protocol).
An RPC implementation can use a protocol of either type as its underlying transport protocol. The choice between the protocol types can be made by an RPC designer at the time an RPC system is designed or by an application programmer at the time an RPC application program is written. In the former case, the RPC system is implemented on a fixed transport protocol, giving no choice of protocols to programmers. In the latter case, the RPC system is implemented on a set of transport protocols and allows programmers to choose one of them as the transport protocol.

The underlying transport protocol of an RPC system may impose constraints on applications that use the RPC system. For example, a connectionless transport protocol may have a limit on the maximum size of RPC messages, but a connection-oriented protocol can transmit RPC messages of any size. On the other hand, in some systems, a connection-oriented transport protocol may impose a limit on the number of connections opened by an application at any one time, while there is no such restriction with the connectionless protocol. The transport protocol may also have an effect on the performance of RPC systems. For example, with the connectionless protocol, the maximum packet size may have a significant impact on the response time of a remote call that consists of multiple packets. On the other hand, although the connection-oriented protocol can transmit large messages, establishing and releasing a connection introduces overhead that increases the response time of a remote call. The performance penalty caused by this overhead will become more significant when a client makes only one call or when a separate connection is set up for each call. Obviously, the connectionless protocol favours remote calls with small sized arguments and results while the connection-oriented protocol favours remote calls with large sized arguments and results. A detailed speed comparison among three commercial RPC systems, Sun RPC, Apollo RPC and Newwise RPC [Netwise 88], supported by different transport protocols, can be found in [Levy 89]. Different transport protocols also result in different requirements for the RPC layer. With the connectionless protocol, the RPC layer or the application must implement facilities to handle timeout, retransmission, sequencing, and duplicate detection. However, with the connection-oriented protocol, there is no need to implement such facilities on the RPC layer or in the application since they are provided on the transport layer.

Courier is an RPC system that uses only the connection-oriented transport protocol for transmitting messages between the client and server. The transport protocol used is the XNS (Xerox Network Systems) Sequenced Packet Protocol (Xerox 81b). A connection based on this protocol provides a reliable, ordered, flow-controlled, bi-directional, logical communication channel between the client and server through software ports called sockets.

Most RPC systems use connectionless transport protocol only. This is because there is no overhead due to connection establishment and release with them and mostly the arguments in a procedure call can fit within a single packet, resulting good performance. Some examples in this category are Cedar RPC, Apollo RPC, Mayflower RPC, and Athena RPC. Cedar RPC uses a datagram protocol designed specially for RPC to gain high efficiency. Currently, Apollo RPC uses two different connectionless protocols: UDP and Domain DDS protocol, which is a proprietary network communications protocol of Apollo. The RPC runtime library can automatically select a proper protocol based on the server's socket address in the binding information. Therefore, the selection of the underlying transport protocol is transparent to the client program. A client program can access different servers that use different transport protocols, without having to know what transport protocol is to be used. Mayflower RPC can run on both the Cambridge Ring and the Ethernet. It uses the Basic Block Protocol on the Cambridge Ring and UDP/IP on the Ethernet. Like Apollo RPC, the selection of the transport protocol is made at RPC bind time. The Athena RPC runs over UDP/IP.

Sun RPC currently uses both UDP and TCP as its transport protocol. It allows the application programmer to select the underlying transport protocol for client and server programs through the RPC library routines. Usually a client program uses UDP through the higher level RPC library routines, with the default values for timeouts and retransmissions. If the client wants to run over UDP with values other than the defaults, it can be constructed using the lowest level RPC library routines. The maximum size of a call request or reply over UDP is limited to 8 Kbytes. Unlike some other datagram based RPC systems such as Cedar RPC, there is no way to gather multiple packets into a single call. If a client or server wants to transmit large sized messages or requires higher reliability, it can use TCP through the lowest level RPC library routines. The advantage of the Sun RPC transport mechanism is its flexibility. This allows the programmer to have greater control over performance, although it makes the transport untransparent to the programmer.
10. RPC Exchange Protocols

As described in [Spector 82], there are three message exchange protocols that are useful for implementing various types of RPC. They are the request (R) protocol, the request/response (RR) protocol and the request/response/acknowledge-response (RRA) protocol. In the R protocol, the client sends a call request message to the server and the server neither acknowledges the receipt of the request nor returns any results or status of the call to the client. The R protocol is useful for procedure calls with maybe semantics and without a returned result.

In the RR protocol, the client invokes a remote procedure by sending a request message to the server and the server returns a result in a response message after executing the procedure. A response message from the server can be considered as an acknowledgement of the request message and a new request from the same client as an acknowledgement of the response message. This protocol can be used to implement many types of RPC. For example, if the client retransmits the request message after a certain period of time, either at least once or at most once call semantics can be achieved, depending on the server’s handling of duplicate request messages. If the server does not detect and suppress duplicate requests, at least once call semantics are provided; otherwise the server can support at most once call semantics by buffering and retransmitting response messages. At most once semantics require that the server save information (e.g., sequence numbers) to detect duplicate requests and buffer responses (unacknowledged implicitly by a new request) for potential retransmissions. This may require large buffer space when the server is serving a large number of clients.

The RRA protocol is similar to the RR protocol except that the client acknowledges the receipt of a response from the server. Acknowledgement of a response indicates to the server that all the previous responses including the current one have been received by the client. This implies that the loss of an acknowledgement message is harmless. An acknowledgement allows the server to release the storage used for storing those responses just acknowledged by it. This protocol can be used to achieve those semantics supported by the RR protocol. In addition, it is especially useful for nonblocking remote calls with at least once or at most once semantics. With nonblocking remote calls, a new request does not imply that the previous response has been received, so a separate message is required to acknowledge the receipt of a response.

Many RPC systems use the RR protocol. A typical example is Courier RPC. In Courier, a client sends a call message to a server to invoke a remote procedure. The server sends back a return, reject or abort message as a positive or negative reply to the call request. The return message indicates a successful execution of the procedure and returns the results of the procedure if any. The reject message reports a reason indicating why the server has rejected the call request, e.g., no such procedure. The abort message reports an error specified in the service interface, e.g., no such file, as in a file service interface. Some other examples using the RR protocol are Sun RPC, Athena RPC, and Raydoo RPC.

Some systems use an improved version of the RR protocol to achieve the effect of the RRA protocol. For example, in Cedar RPC, usually a response acknowledges the receipt of a request and a subsequent request implies the receipt of the response. However, if there is no new request arriving within a certain period of time, the server will retransmit the response and ask for an explicit acknowledgement. The client will then send an explicit acknowledgement to confirm the receipt of the response. In addition, if the execution of a call lasts too long, the client will retransmit the request or the last packet of the request (in the case of multi-packet request) asking for acknowledgement. The server will confirm this by sending an explicit acknowledgement immediately. Apollo RPC also employs a similar protocol.

Mayflower RPC uses the RRA protocol. There are four types of messages: Calls, Replies, Errors and Terminators. The client sends the same call message every two seconds until it receives a reply from the server or detects a fatal error. The server transmits the response every two second until it receives a terminator message from the client for the call or detects a fatal error. The terminator message allows the server to discard the reply information. The server sends an error message in response to a call if it receives a duplicate call while executing the call, is congested or does not support a procedure. Error messages due to duplicate calls inform the client that work is in progress; those due to server congestion tells the client that retransmission of the call may succeed; and those due to unsupported procedures allows the client to abort the call. No acknowledgement messages are sent for error messages.
11. Interface Definition Languages

In RPC based distributed systems, clients usually access remote services through a remote interface. A remote interface is a set of logically related procedures provided by a server. It defines the externally visible characteristics of a set of remote procedures in an interface definition language. These characteristics include datatype declarations, procedure declarations, and exception declarations. In addition, an interface also defines its attributes such as interface names and versions to facilitate human reference and upgrade or modification of the remote services. It may also import and export definitions to facilitate the reusability of objects defined in this or other interfaces. Datatype declarations define data types used to specify the data types of parameters and results of the procedures. Procedure declarations specify the names of the procedures, the types and directions of their parameters, the types of their results, the types of their exceptions and other attributes of the procedures. Such attributes may include call semantics (e.g., at most once), the names of the caller-side procedures to be called back, whether the binding or unbinding is done on a per call basis, etc. Exception declarations define the errors that may be reported by the procedures in the interface. These errors may contain parameters of any type, as seen previously in the discussion on exception handling.

The specification of an interface is called an interface definition. An interface definition provides an agreement between the client and server on what services are available from the server and the types of the parameters and results of each procedure defined in the interface. This allows the arguments in the call statement and those of the called procedure to be type-checked. The interface definition can be used to automatically generate client stubs and server stubs that marshal and unmarshal the arguments and results of the procedures. It is a trend that distributed systems are constructed in an object-oriented way. In object-oriented distributed systems, data objects can only be accessed through a set of well-defined procedures in an interface. The interface definition serves as a basis for the development of such systems.

An interface definition language is used to specify interface definitions. To do so, the interface definition language should satisfy a few requirements. First, it should have the expressive power to specify declarations for remote procedures, parameters, datatypes and exceptions and allow explicit specification of attributes for datatypes, parameters and interfaces. In particular, it should provide a set of built-in data types such as integer, real, boolean, character and enumerated as well as facilities for defining aggregated types such as record, array, discriminated union and string and for specifying the varying length of arrays and strings and the range of integers and reals. Second, the interface definition language should be independent of programming languages but capable of mapping to specific programming languages. Third, it should be machine processable to facilitate the automatic generation of stubs. Fourth, it should provide facilities for importing and exporting objects defined in interfaces. And finally, the definition language should allow for future extensibility when new notions are needed. Although many interface definition languages are existing, few really satisfy all these requirements due to the intended application environment in mind when designing them. For example, some interface definition languages do not provide explicit constructs for defining exceptions because the target application programming language has no exception handling mechanism. Basically, there are two approaches to the design of an interface definition language. One is to use an existing programming language, possibly with some additions and modifications of constructs for specifying notions inherent in RPC interfaces. Although simple, this approach may have difficulty in expressing all the notions for RPC interfaces. The other approach is to design an interface definition language for the purpose of defining RPC interfaces starting from scratch. This approach can overcome the above difficulty but may force the application programmer to learn another language.

Closely related to the interface definition language are client and server stubs, which are procedures for marshalling or unmarshalling parameters or results and sending or receiving messages using the underlying transport mechanism. They hide the details of parameter encoding and transmission from the application programmer. Stubs can be generated manually or automatically. In the manual case, the application programmer builds stubs using translation procedures as building blocks from a library provided usually by the RPC system. Stubs handling complicated data types such as linked lists and trees can be written this way. In the automatic case, an interface definition is processed by a stub generator to produce stubs, possibly with some header files defining data types used for specifying parameters and results. Usually the stub generator does not generate stubs for handling data types containing pointers such as linked lists and trees, but allows the programmer to supply procedures to handle such data types. The automatic generation of stubs releases the programmer from the tedious and mechanical programming work for constructing stubs and
reduces the possibility of errors crawling into the stubs. The design of a stub generator varies. The stub generator can be designed for single or multiple target programming languages, or even multiple machine types and multiple languages.

Now we examine some existing interface definition languages and stub generators. Sun RPC defines an interface definition language called Remote Procedure Call Language (RPCL). The syntax and the built-in data types of RPCL are very similar to those of the C programming language. In addition to those built-in types of C, RPCL has an explicit Boolean type. The RPCL syntax for aggregate data types is similar to its C counterpart. However, RPCL provides facilities to define variable length arrays, variable length strings and fixed or variable length, uninterpreted data. RPCL unions are discriminated unions and their definition is a combination of a C-union and a C-switch. The procedure definition in RPCL is different from that in other definition languages in that it specifies a single argument of the procedure as in the operation definition in Remote Operation Notation (CCITT 88). Multiple arguments must be packed into a structure as a single argument of the procedure. RPCL provides no explicit syntax for defining exceptions and importing or exporting definitions. It has a construct for specifying interface attributes such as the program number, version numbers, but none for procedure attributes. Sun RPC provides a library of translation procedures and a stub generator called rpugen for stub generation. The programmer can use either these translation procedures to construct the stubs manually or rpugen to generate them automatically. Rpugen is a single-targeted interface compiler, which accepts an interface definition in RPCL as input and generates only C code for data definitions, client stubs, a server skeleton, and routines for translating objects of the types defined in the interface definition into their external representations.

The Courier language has a set of predefined types (Boolean, cardinal, long cardinal, integer, long integer, string and unspecified) and provides a rich set of constructs to define constructed types (enumeration, array, sequence, record, choice, procedure and error). Actually, procedure types are procedure declarations while error types are exception declarations. Neither procedures nor errors can be passed as arguments or results of remote procedures or arguments of remote errors. Multiple arguments, results and errors can be specified in procedure declarations but attributes of arguments and procedures cannot. The Courier language has constructs to reference definitions in other interfaces but no explicit exportation facilities. One of the shortcomings of the Courier language is its lack of the real type such that float-point numbers have to be represented by two integers, causing inconvenience. The stub generator in Courier RPC is called xnsccourier, similar to rpugen.

Matchmaker [Jones 85] provides a stub generator capable of generating stubs in different programming languages. Matchmaker is an interface specification language used in the SPICE environment at CMU. It has built-in types and type constructors similar to those of the Courier language and Sun RPCL, but it is more flexible in specifying data types and remote procedures. For example, it is possible to specify the range of integers, the direction of the arguments, the timeout values and asynchronouosity of remote procedures, etc. The Matchmaker system provides a multi-targeted compiler that compiles interface definitions into stub code in four different languages used in the SPICE environment: C, PERQ Pascal, COMMON LISP and Ada. The Matchmaker compiler is internally constructed with multiple different back ends capable of producing code for different languages supported. If new languages are added to the environment, the back ends for these languages can be incorporated into the compiler without difficulty. Matchmaker supports interlanguage calls between the client and server.

The Horus interface definition language is an extension of the Courier language [Gibbons 87]. Its stub generator can generate code for different languages and different machine types. Unlike other systems, Horus adopts a quite different approach to the design of stub generators. The approach is to separate the knowledge of the target programming languages and machines from the code for the stub generator. Such knowledge is specified in the language and machine specifications. The language specification provides instructions on how to produce code for type declarations and marshalling for each language data type as well as how to produce code for sub procedures. The machine specification supplies the representational information of each machine data type needed to generate code for marshalling scalar language data types. The Horus stub generator accepts an interface definition, together with these two specifications as input and produces type declarations and stub code for the language specified in the language specification and for the machine type specified in the machine specification. One of the major advantages of this approach is that it is easy to add new languages and machine types to the environment without changes to the stub generator by simply supplying the language and machine specifications.
12. Data Representation

Different Computer architectures have different internal representations for various primitive data types such as character, integer and real. The data representations are different in their machine type order (e.g., big-endian, little-endian), size (e.g., 16, 32 bits), bit representation (e.g., one’s complement, two’s complement, ASCII, EBCDIC), etc. Since the RPC client and server may run on machines with different architectures, the client’s and server’s representations for the same RPC argument or result will be different. Thus, some translations between the two representations are necessary when the client and server send data to each other; otherwise, the data cannot be interpreted correctly by the receiver. For example, if a client with big-endian representation sends a two-byte integer 1 to a server with little-endian representation without translation, this integer will be misinterpreted by the server as a different two-byte integer 256, rather than 1.

Basically, there are two approaches to the translation between different representations. A common approach is to define and use a standard external data representation for a set of data types. In this approach, before sending a message into the network, the client or server stub converts the message’s internal representation into its external representation while on receiving the message from the network the server or client stub converts the message’s external representation into its internal representation before interpreting it. This approach has the advantage of simplicity. As far as data representation is concerned, the stub only needs to know how to translate between the internal representation on its system and the standard external representation. The addition of systems with new architectures to the environment does not affect the existing systems. The disadvantage of this approach is inefficiency. A message may experience two translations even if the sender and the receiver have the same internal representation. However, this might not be a serious problem since the time taken for translation is not significant, compared with the time spent in communication. The simplicity of this approach and its openness for accommodating heterogeneity outweigh its inefficiency, especially in a heterogeneous environment, making it outshine the approach described below.

The alternative to the standard external representation approach is the so-called “receiver makes it right” approach. In this approach, the message to be sent is encoded in the internal representation used in sender’s system and preceded by a label identifying the representation used to encode it. If the receiver finds from the label that the representation of the message is different from the one used locally, it converts the message to its local representation. The advantage of this approach is that it requires at most one translation. However, every receiver must be prepared to translate incoming messages in any representation into its local representation, implying that every receiver has a set of translation routines for each representation. This is practically hard to achieve due to the diversity of existing computer architectures, let alone future computers. Therefore only a set of common representations can be supported in practice. Besides, adding systems with new architectures to the environment involves changes to the existing systems.

The external representation of a data value can be either explicit typing or implicit typing and have either variable length or fixed length. With explicit typing, an identifier or tag is used to indicate to the receiver the type of a value. This provides self-describing information for the receiver to decode the value and to do run-time type checking. However, this increases the processing time for encoding and decoding and transmission time due to the tag field. With implicit typing, only the value of a data object is encoded and transferred. This avoids the processing and communication overheads due to the tag field. Because of absence of typing information, the receiver is required to have prior knowledge of the format of the messages it receives in order to interpret them. Such knowledge can be obtained from the protocol governing the interactions between the sender and receiver. For example, RPC stubs are equipped with such knowledge while being generated from an interface definition, which specifies the interactions between the client and server. Thus, the stubs know what type of RPC arguments or results they expect to appear in due course while interpreting an incoming message. Variable-length representation uses a field to indicate the length of a data value. This allows data of any size to be represented, but introduces processing and communication overheads, like explicit typing. Fixed-length representation encodes data into fixed size fields, for example 16 or 32 bits. This is simpler and more efficient, but has limitations on the representable maximum data values.

Sun RPC uses the standard external representation approach and defines an external data representation called (XDR) [Sun 87] to represent RPC arguments and results. XDR supports all RCP data types and
uses the IEEE format to represent floating-point numbers. It uses the big-endian byte order, implicit typing and 4-byte fixed-length fields as basic data units. Courier RPC has the same approach but uses Courier representation as the external representation. Courier representation is similar to XDR in many aspects such as big-endian, implicit typing and fixed length (16-bit units), but has some differences in data types (e.g., no floating-point type).

Apollo RPC uses the "receiver makes it right" approach and supports a set of representations defined by Network Data Representation (NDR). NDR requires all representations to be implicit typing and fixed-length. It supports two byte orders (big-endian, little-endian), two character sets (ASCII, EBCDIC), and four floating formats (IEEE, VAX, Cray, IBM). A 4-byte format label is used to identify the particular data representation (i.e., a combination of the above formats) in use.

The X.409 or ASN.1 transfer syntax [CCITT 84], [ISO 87] is an explicit typing and variable-length data representation, but no reported RPC systems use it to represent RPC arguments and results.

13. Security

The Security issue may or may not be critical to a distributed application, depending on the nature of the application. However, it is desirable to provide some form of security facilities within the RPC system so that RPC-based distributed systems are able to implement their security policy using these underlying security facilities. The RPC security facilities usually involve authentication and encryption. In an RPC-based application, the client may want to verify that the server is a genuine one while the server may need to make sure that the client is the one who claims to be. This is what authentication concerns. The arguments and results of a remote procedure call may be so sensitive that they should only be interpreted by the principals concerned, i.e., the client and server. Therefore it is necessary to encrypt the data transferred between the client and server to protect them from intruders' interception and manipulation. The provision of security facilities in the RPC system certainly introduces performance penalty since it takes time to transfer and verify the authentication credentials and to encrypt the message. Thus, the security mechanism should be designed in such a way that applications are allowed to select the level of security or bypass it to avoid the extra cost if so desired. Only a few existing RPC systems implement security facilities. In the following, we will examine two such systems, Cedar RPC and Sun RPC.

Among those RPC systems implementing security mechanisms, Cedar provides the strongest security facilities. The Cedar security mechanisms [Birrell 85] are based on the use of encryption, private keys and an authentication service (i.e., key distribution center). The Data Encryption Standard (DES) [NBS 77] is used for encryption and Grapevine is used as authentication service. Each principal (client or server) has a private key known to itself and the authentication service. The messages transferred between two principals are encrypted using a conversation key obtained from the authentication service.

The Cedar security mechanisms work as follows. To obtain a conversation key, the client sends the authentication service (via the RPCRuntime) the client's and server's names and the client's private key in plain text. The authentication service looks up the client's and server's private keys and returns a conversation key, the server's name and an authenticator in cipher text (formed using the client's key), where the authenticator itself is encrypted data (formed using the server's key) containing its creation time, the client's name and the conversation key. The RPCRuntime records the authenticator and conversation key with a unique conversation identifier and returns the client a conversation handle. When the client makes a call using this handle, the RPCRuntime uses it to find the conversation identifier and conversion key. Then it encrypts the message using the conversation key and sends the encrypted message along with the encrypted conversation identifier. On receiving the call, the RPCRuntime on the server's host looks up the conversation identifier in a table mapping conversation identifiers to their corresponding principal's names and conversation keys. If the lookup succeeds, it decrypts the message using the conversation key and passes the server stub the decrypted message with the information for determining the caller's identity. Otherwise, it requests for the authenticator by making a callback to the client, decrypts it using the server's key, records it, and then decrypts the original call message as above. Similarly, the RPC results are encrypted and decrypted using the conversation key.

This system is secure in that the client and server assure each other of their identity and intruders cannot eavesdrop, undetectedly modify or replay calls. The reality of a principal's identity is guaranteed by Grapevine's access controls and the secure communication among the Grapevine servers themselves. The
prevention of various intrusions is guaranteed by means of encryption, checksum, the uniqueness of the conversation identifier and the lifetime of the authenticator. The system is flexible because the client can bypass the security mechanisms completely or just use the authentication without encrypting the calls. The system is convenient because the client does not have to worry about the details of authentication and encryption that are done by Grapevine and the RPCRuntime.

Sun RPC provides an open-ended authentication mechanism. The current implementation of Sun RPC supports three types (flavors) of authentication: Null authentication, Unix authentication and DES authentication, giving different levels of security. New authentication types can be added to the system to meet the particular application’s needs.

In Sun RPC, each call message includes a credential field and a verifier field. The client uses a credential to identify itself to the server and a verifier to provide information that the server uses to verify the reality of the client’s credential. Each reply message has a verifier field but no credential field. The client uses the server verifier to authenticate that the server is really the one it wants to talk to. The absence of server credential field is because the client must know the server’s identity before talking to it. With Null authentication, the credential and verifier are null. The server allows any client to access its service. With Unix authentication, the client includes as its credential the client’s host name, user ID, group ID and the list of groups to which the client belongs, but provides a null verifier, meaning the server cannot verify the client’s credential. The server can impose access controls based on the credentials, but such controls can be trusted only if users do not use fake credentials. The reply verifier may carry null or a shorthorn for the original credential that the client may use in subsequent calls. DES authentication is based on the Data Encryption Standard and is the strongest of the three authentication types. Although different in encryption, the client’s credential is similar to the authenticator in Cedar. It consists of the client’s name, a conversation key created by the client and encrypted with a public key scheme, and the lifetime of the credential encrypted with the conversation key. Unlike the lucky Cedar security system, which can rely on the authentication service’s guarantee of the reality of the principals’ identity, DES authentication uses timestamp-based verifiers to authenticate the client and server. The client encrypts the current time (synchronized with the server’s time) using the conversation key as its verifier. The server uses as its verifier the client’s timestamp minus one encrypted with the conversation key and an unencrypted ID. The client may use this ID in place of the credential in subsequent calls to improve efficiency. Since only the client and server know the conversation key, only they can encrypt the timestamp correctly. Thus, there are sufficient reasons for them to trust each other. The lifetime of a credential and timestamps are used to prevent intruders from replaying calls.

The Sun RPC security mechanism provides great flexibility in choosing the level of security. The client can use the default Null authentication by doing nothing or either of the other two by creating a corresponding authentication handle with the appropriate library routine. The open-ended feature of the mechanism facilitates adding new authentication protocols to the environment. However, the mechanism does not include facilities to encrypt the RPC arguments and results, so the client or server itself must do it if it desires so.

14. Summary

In this report, we have discussed the fundamental issues in designing and implementing RPC systems and examined the various approaches employed by existing systems to tackle these issues. Dynamic binding is common among RPC systems due to its flexibility. Most systems support non-parallel, synchronous, atmost-once remote procedure calls. These call semantics are close to those of local calls and can be achieved with reasonable cost. Although exactly once semantics are desirable, they are generally hard to achieve. Effort have been made to provide fault-tolerant remote calls using replication-based techniques in some systems while orphan handling receives little attention. Not many systems provide explicit exception handling mechanisms due to lack of such mechanisms in languages themselves. Server management shows great diversity, with the server-created-on-demand strategy being relatively common. Transport and message exchange protocols have a direct impact on performance and buffer management. Because of LAN’s high reliability, the connectionless transport and request/response protocols gain popularity. Interface definition languages and their compilers should be provided to facilitate the development of object-oriented distributed applications and release the programmers from the encoding and communication details. The standard (canonical) external data representation makes accommodating heterogeneous systems easier and the implicit-typing encoding makes contribution to efficiency, consequently prevailing among RPC systems.
Although the provision of security mechanisms within RPC systems facilitates implementing security policies of distributed services, few systems include security mechanisms.

RPC systems exhibit many differences in design decisions, implementation details and supporting environments although they have also many similarities. Even if two systems adopt the same design decision (e.g., dynamic binding), they may have different implementations and supporting facilities (e.g., binding services). Consequently, it is very difficult to interconnect various different RPC systems to allow calls to be made across them although HRPC (Bershad 87) has made a significant attempt in this direction. On the other hand, international standards bodies are standardizing RPC to allow interoperability of RPC mechanisms.

Although RPC systems have been developed and used for some years and the general concepts of RPC have been learned from the existing systems, there still exist some subtle issues that have received little attention in these systems. Many systems use a single binding policy (e.g., binding before the first call), but support may be needed for other binding policies such as rebinding and multiple bindings. Little has been done about multicast calls, call signaling such as call cancellation, and context management, which concerns maintaining state between calls. Although some systems support callbacks using separate bindings, a general mechanism is required for callbacks using the same binding or chained calls (like linked operations of ROSE). Work on transport independence and fault tolerance will continue. Interface definition languages also require more work, for example, on parameter types, procedure attributes, notation for data values, etc. Consideration needs to be given to RPC’s relation to OSI and ODP (Open Distributed Processing) models.

Acknowledgments

I would like to thank Bob Kummerfeld, Alan Fekete and Doan Hoang for their helpful suggestions and comments. I am also grateful to Eng Chew for providing me with useful documents.

References


Lamport, B. W., "Remote Procedure Calls," In *Distributed Systems-Architecture and Implementation: An Advanced Course (Lecture Notes in Computer Science*


