Building distributed systems with remote procedure call

by Steve Wilbur and Ben Bacarisse

Remote procedure call is gaining popularity as a simple, transparent and useful paradigm for building distributed systems. Ideal transparency means that remote procedure calls are indistinguishable from local ones. This is usually only partially achievable. This paper discusses those implementation decisions which affect transparency and intrude on the design of distributed applications built using remote procedure call.

1 Introduction

Distributed applications which concentrate on point-to-point data transmission can often be adequately and efficiently handled using special-purpose protocols such as those familiar for remote terminal access and file transfer. Such protocols are tailored specifically to the one application and do not provide a foundation on which to build a variety of distributed applications such as distributed operating systems, electronic mail systems, computer conferencing systems etc. While conventional transport services can be used as the basis for building such applications, the programmer is still left with many organisational problems even where the interaction between the processes reduces to a simple request-response exchange. Among these problems are the use of different representations in different machines, synchronisation and the provision of a simple programming paradigm.

Distributed systems are usually assumed to contain a number of processors interconnected by communications networks at data rates up to tens of megabits per second. A variety of machine architectures is usually found, each having its own internal representation for primitive data types, its own address alignment rules and its own operating system. Three of the key problems in building distributed systems are dealing with this heterogeneity, dealing with partial failures of the system and providing adequate process synchronisation.

However, one simplification is afforded by noting that a large proportion of applications use a request and response interaction between processes where the initiator is idle until the response is returned. This can be modelled by a procedure call mechanism between processes. A remote procedure call (RPC) mechanism is usually a type-checked mechanism which permits a language level call on one machine to be turned automatically into a language level call in a process on another machine (Ref. 1). If the RPC mechanism is in possession of the type specifications of the remote procedures and their parameters, a presentation layer can convert data from the format required by the calling machine to that required by the target machine. Furthermore, the facilities provided by the host operating system for communication and synchronisation between processes and machines are encapsulated in one simple abstraction which is available in most programming languages: that of a procedure call. To provide the programmer with a familiar type-safe mechanism for building distributed systems is one of the primary motivations for developing RPC-like services. While they are not a universal panacea, they do provide a valuable set of services on which a significant number of distributed applications can be built.

Although standards are emerging for RPC-like services (Refs. 2 and 3) there are a number of RPC design and implementation decisions which can significantly affect the design of distributed applications. Moreover, there are several extant RPC systems claiming to be *de facto* standards. Each offers subtly different semantics. This paper describes the basic concepts of RPC and some of these issues. Much of the paper covers the general principles and issues, but where necessary examples are drawn from extant implementations. This paper does not provide a guide to implementing RPC; Birrell and Nelson have written a comprehensive paper showing one implementation in great detail (Ref. 4).

2 Remote procedure call

2.1 Principles

Remote procedure call is a mechanism for providing synchronised type-safe communication between two processes. It is a special case of the general message passing model of interprocess communication (IPC). In the simplest case, message-based IPC involves a process (the *client*) sending a message to another process (the *server*). It is not necessary for the processes to be synchronised either when the message is sent or received. It is possible for the client to transmit the message and then begin a new activity, or for the server's environment to buffer the incoming message until the server is ready to process a new message. Remote procedure call, however, imposes

tighter constraints on synchronism because a mechanism is provided which models the local procedure call by passing parameters in one direction, blocking the calling process until the called procedure is complete, and then returning the results. RPC thus involves two message transfers, and synchronisation of the two processes for the duration of the call.

Bindings between processes in message passing systems can be very flexible, for example one-to-one, one-to-many etc. For RPC, binding is normally one-to-one to mirror the familiar language primitive, although some experimental one-to-many implementations exist. Binding in RPC is usually performed at run time.

In some message passing systems messages bear a type, and can only be received if a suitably typed request is outstanding at the receiver. Such message typing is to be found in all good RPC implementations, but is not universal in message passing systems. Furthermore, the RPC mechanism may be closely integrated with one or more programming languages. If a remote call has exactly the same syntax as a local one then the RPC mechanism is said to be *syntactically transparent*. When the semantics of a remote call are identical to those of a local call the mechanism is said to be *semantically transparent*. The degree to which semantic transparency can be achieved is one of the major topics of this paper.

It is the familiarity of the procedure call paradigm which gives RPC its value, so good implementations are well integrated with the programming environment. Although in most environments total semantic transparency is impossible, enough can be done to ensure that the programmer feels comfortable. RPC represents a significant step away from most mechanistic IPC systems towards building higher-level abstractions for distributed programs.

2.2 The mechanism

The RPC mechanism is usually implemented as follows, and is shown in Fig. 1.

The two parts of the application are split across a procedural boundary, and a dummy procedure with the same name as that in the server is placed in the client process. This dummy procedure, usually known as a *stub*, is responsible for taking the calling parameters and packing them in a suitable transmission format before sending them to the server. It then merely awaits the server's reply, unpacking the results before passing them in the local representation back to the calling procedure.

At the server end the situation is somewhat similar, except that the server may be able to service any one of several call types. Thus the main program loop of the server will await an incoming message, decide which procedure to call, re-format the parameters for local consumption, and call the server procedure. When the procedure returns, its results are packed and transmitted back to the client. It can be seen that there is only one thread of control in the application program despite there being two machines and processes in the system. It should also be noted that there is an underlying assumption that the client and server exist in different address spaces, and thus all parameters must be passed by value.

2.3 Stubs and interfaces

The purpose of the client and server stubs is to manipulate the data contained in a call or reply message so that it is suitable for transmission over the network or for use by the receiving process. The stubs can be thought of as implementing the calling convention for remote procedures, taking the place of the standard procedure prologue and epilogue generated by a compiler for a local call.

Stubs can be generated either by the programmer or automatically. In the manual case the RPC implementor will provide a set of translation functions from which the user can construct his/her own stubs. This method requires the server implementor to do more work but it is simple to implement and can easily handle very complex parameter types. The Sun Microsystems mechanism uses this approach (Ref. 5). Some very early examples of RPC did not see stub generation as an issue, the emphasis being on the protocols and system structuring. For example, in the Newcastle Connection (Ref. 6), remote access to UNIX† system calls was provided through hand-crafted stubs.

Automatic generation of stubs is usually done by having a *parameter description language* which is used to define the interface between client and server in terms of the procedures

†UNIX is a trademark of AT&T Bell Laboratories.

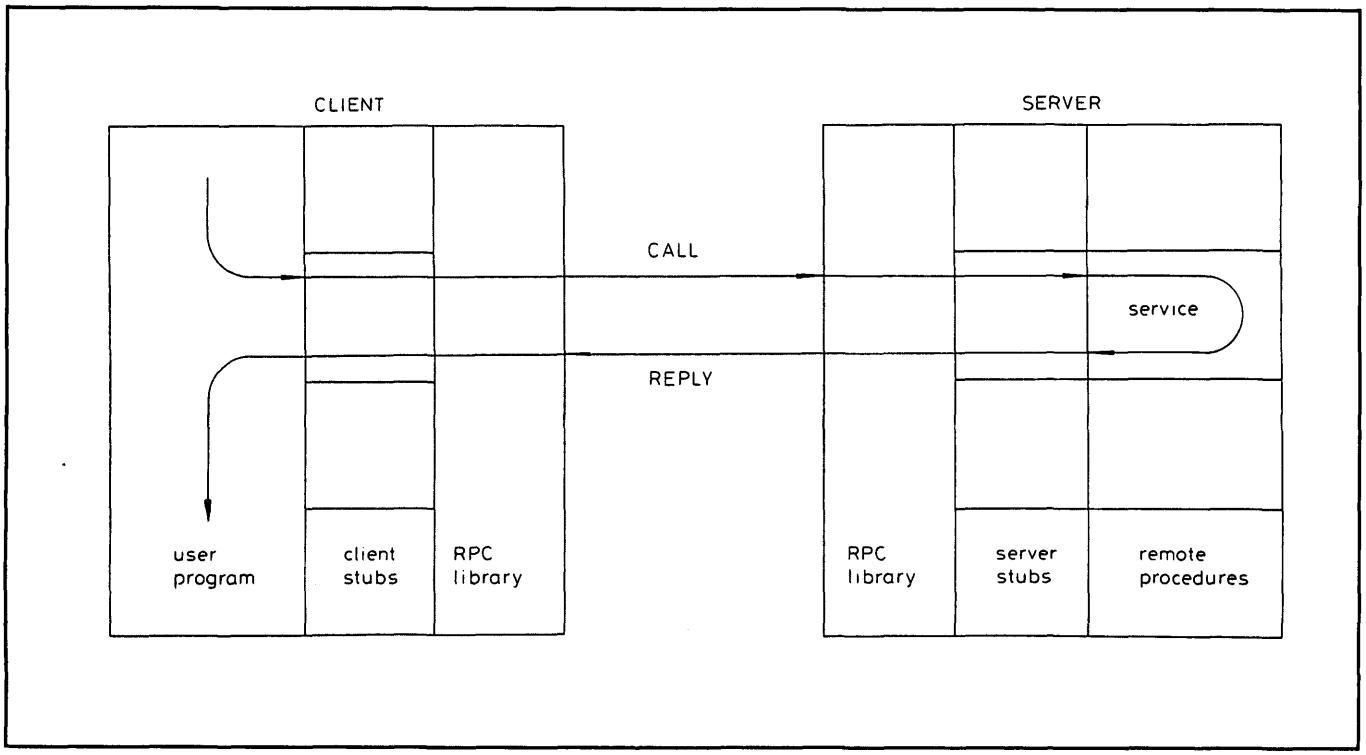

Fig. 1 The RPC mechanism

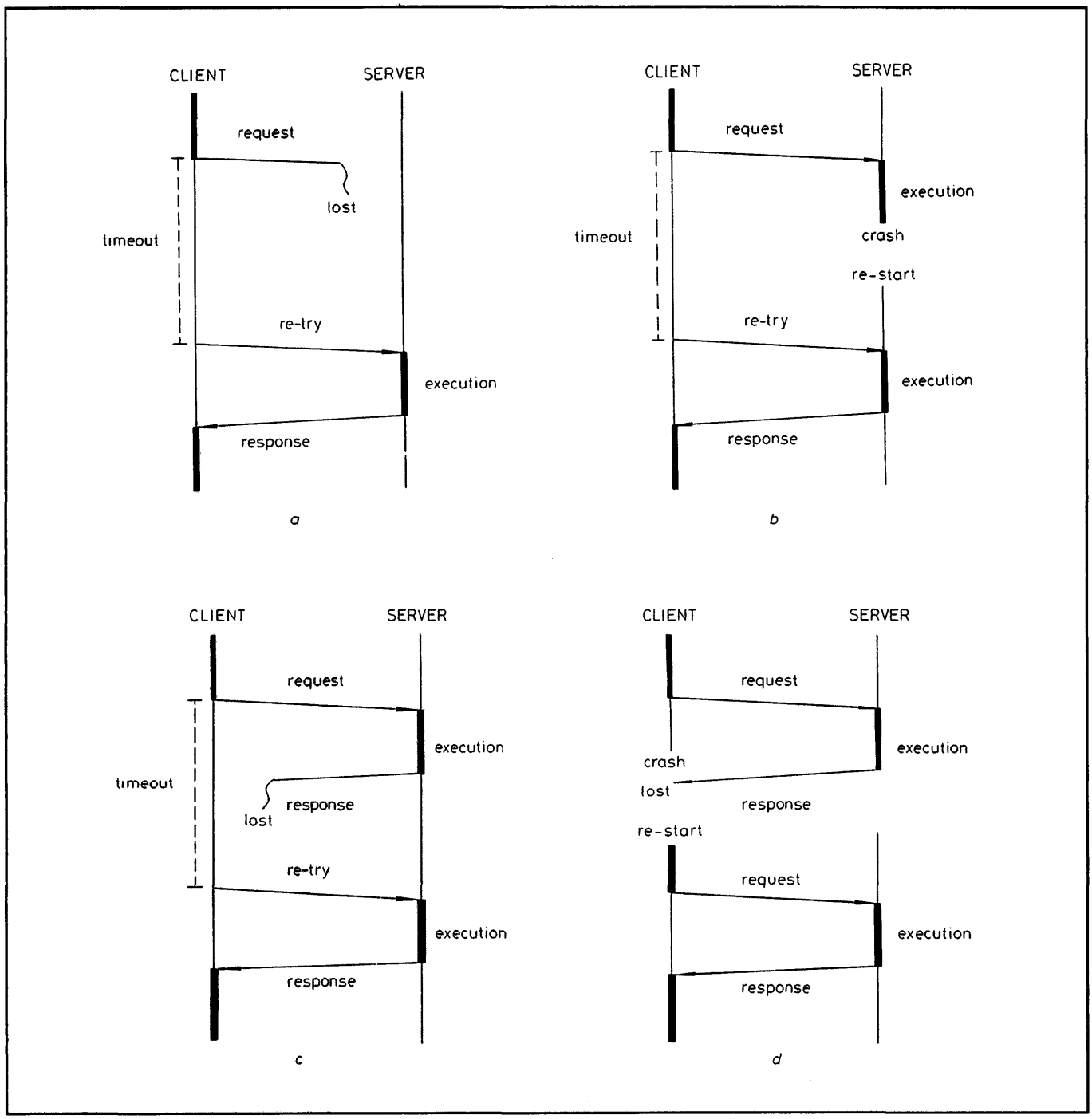

Fig. 2 Failures in transactions between processes
a Outbound message lost *b* Server crash
c Response lost *d* Client crash

provided and their parameters. A set of basic types will be provided, together with mechanisms to construct more complex types from the basic ones. The interface definition is processed to generate the appropriate stubs automatically, which can then be compiled and linked in the normal way with the client or server code. This technique can be extended so that the stub generator also reads a description of the machine hardware and the programming language to be used for the stubs (Ref. 7).

In some cases procedures are grouped into *interfaces* (Refs. 4 and 8), where the procedures are related and typically operate on some shared data structure. For example, a file server interface might provide a set of procedures for creating, deleting, reading and writing files. The interface description language in such systems allows clients to specify the interface they require rather than the procedure. This ensures that a client may call related routines and be sure that they will be executed within the same server. This is the first step towards an *object-oriented* system, where program objects can only be accessed through a specific set of procedures.

2.4 Data representation

Most implementations define a common intermediate data format that specifies how the basic and compound types are represented in a call or reply message. Although this approach leads to two translations being performed for each message sent, the translations can be made relatively efficient, taking about 20 microseconds per byte on average on modern machines. If the client stub can determine the type of machine it is sending to, the call need only involve one translation. Implementations like this are more complex to maintain since each stub must be capable of generating all possible message formats. This approach also has implications for systems that allow a call to be sent to multiple destinations.

The transmitted representation (or *standardised* represen-

tation) built by the stubs can be either self-describing (tagged) (Ref. 9) or untagged (Refs. 4 and 8). Untagged representations will make more efficient use of the network bandwidth and require less processing time to encode and decode.

Type checking cannot be performed until run time with remote procedure calls, since the stub cannot determine what process will receive the message it builds. However, type safety can be achieved with both tagged and untagged representations.

2.5 Call execution

While a distributed system may offer potential for parallelism, it is also prone to partial failures. It is possible in such a system for either the client or server to fail independently, and later to be re-started. The *call semantics* determine how often the remote function might be performed under fault conditions. The weakest of these semantics are *possibly*, which are not really appropriate to RPC but are mentioned for completeness. They correspond to the client sending a message to the server without waiting for a reply or acknowledgment. Where local area network (LAN) technology is being used the probability of successful transmission is high and may be adequate for some applications.

Another weak form is *at-least-once* semantics. In this case the client re-transmits the request if the response does not return within a pre-determined time. Clearly from Fig. 2, it is impossible to determine whether the failure was due to a server crash or to loss of the request or response. It is thus possible for the procedure in the server to be executed several times or even partially. This behaviour is unlike that of a local procedure call, but can be provided by a very simple protocol. If it is used semantic transparency is lost, and the server must be implemented using *idempotent* functions, where the programmer must ensure that multiple executions of the same call are identical to a single execution with the same parameters.

Acceptable semantics are *at-most-once*. In this case the server functions are eventually executed once if the server does not crash. The underlying protocol has to be designed to filter out duplicate requests and return the previous results when reply messages get disrupted. The protocol is more complex, and typically needs to retain results from some previous calls. If the server crashes the procedure may be partially executed, and the application program or an operator would be expected to make the affected data consistent.

The strongest semantics are *exactly-once*, in which the procedure is guaranteed to be performed exactly once even if the server crashes. This corresponds to the provision of *atomic actions*, and is rarely found even in non-distributed programming environments other than through an enhancement package. In the International Standards Organisation Open Systems Interconnection standards, the commitment, concurrency and recoverability mechanisms of the common applications service elements (layer 7) provide this enhancement.

It must be pointed out that the definition of these terms is not yet stable. Some sources attribute the opposite meanings to the terms at-most-once and exactly-once (Ref. 10).

2.6 Binding

It is necessary for the client to know the location of the server before the remote procedure call can take place. There are several ways in which this can happen. For example, the server's network address can be compiled into the client code by the programmer, it can be found by looking up the server's name in a file or by soliciting the information from an operator. These methods are not very flexible but are useful in certain limited cases. A more flexible approach is for the client to contact a *binding* service. Typically the binder will be a server with a well known address, probably located using one of the simpler methods described above. When a server starts executing it registers its location with the binder together with information about the procedures that it supports. This operation is often called *exporting*. The client will then contact the binder in order to find the network address of the procedures it requires (*importing*), and can then use the returned address for the indefinite future. Thus it is not necessary for the client to consult the binder before each call is made to the server. The binder is a key difference between the local and remote procedure call mechanisms. Several servers may register the same procedures or interfaces; it is up to the client to decide on the most suitable one, or just to choose an arbitrary one.

When servers are about to be withdrawn from service they are usually required to de-register themselves with the binder. De-registration only prevents future attempts at binding to the deceased server from succeeding; existing client-server bindings will still remain. It is therefore important that the underlying transport protocol should be able to detect when the server no longer exists and inform the client.

2.7 Server management

In some implementations of RPC several *instances* of a server may be installed on the same or different machines to provide either load balancing or some measure of resilience to failure. The binder mechanisms so far described would allow an arbitrary one of these instances to be selected when a client attempts to import the appropriate interface. Such static servers generally remain in existence indefinitely and may retain state between successive procedure calls. Moreover, the server may interleave the requests from a number of clients and thus have to manage concurrently several sets of state information.

In some implementations there is a server manager, which can create servers on demand. The client contacts the binder in the usual way, which returns the address of the server manager. The client then contacts the server manager with a suitable CreateServer( ) call, whereupon the server manager passes back the address of a server of the required type to the client for later, private use. In some environments the server manager acts as resource manager and chooses an idle server from a pool of servers created earlier; in others a new server may be spawned on demand. In both cases the client has exclusive use of a server for the duration of a transaction or session. In some cases the server manager can create a variety of servers in response to suitable CreateServer( ) calls. Such a server is often known as a *generic server*.

A third, less common server management strategy is where each call to the server results in a new server instance. Each instance terminates when its call completes. This might be termed the *instance-per-call* strategy.

The static server approach is the most rudimentary, but requires the implementor to handle concurrent state management within the server. It is valuable for standard services, but load balancing between different instances is difficult. With server managers, each server normally only services a single client, so the need for load balancing disappears, and the code implementing the server only has one set of state to manage. In the final case there is no state retained between calls, even from the same client, resulting in a stateless server.

2.8 Underlying protocol

A transport-level protocol must provide the request-response service implied above. Usually, at-most-once call semantics are provided. With such a protocol in place, mechanisms to manage binding and to manage servers can be built from it. The only difference between access to the binder and other servers is that the binder must be located by some non-standard means, often

by having binders at fixed addresses.

The transport-level protocol may impose restrictions on the data (parameter) size of a message. Where this occurs the parameter size may be limited to a few hundred bytes, forcing the user to design his/her interface within these limits.

2.9 Issues affecting the user

Although RPC mechanisms are not completely transparent, they are sufficiently useful to cope with a large class of problems. By its nature, this paper emphasises the difficulties rather than the successes. Nonetheless, RPC is being used extensively as part of many systems, and similar techniques have been used to specify the CCITT X.400 mail system (Ref. 2). In the remainder of the paper the semantics, ease of use, limitations, possible extensions and inter-working aspects of RPC will be explored.

3 Semantics

A major aim of many RPC mechanisms has been to provide a transparent mechanism with which to access remote services. The current generation of programming languages have mostly been designed for the single-machine environment, where language constructs pay no attention to the external environment other than through the human-oriented input/output mechanism. Within this framework it is possible to provide reasonable syntactic transparency for remote procedure calls through the use of a parameter description language and stub generator. The semantics of a remote call, especially where there are failures, are sufficiently different that they cannot be completely hidden from the user. At best they can be sugared, but the sugaring is usually reflected in minor syntactic differences between local and remote calls. Future generations of programming languages should include constructs to overcome the representational and failure recovery deficiencies of current languages.

Three kinds of semantics can be distinguished as important to the user of RPC mechanisms: parameter passing semantics, call semantics and server semantics. Each represents a conscious design choice which the user must be aware of in order to engineer properly his/her application.

3.1 Parameter passing semantics

Because the client and server exist in different address spaces, possibly even on different types of machines, passing of pointers or passing parameters by reference is meaningless. Most RPC mechanisms therefore pass parameters by value; i.e. all parameters and results are copied between client and server through the intervening network. For simple compact types such as integers, counters etc. this poses no problem. Small arrays are also little problem, but larger arrays, and especially multi-dimensional arrays which would normally be passed by reference, can consume much time for transmission of data that may not be used. For example, a 1000 byte array might typically take 20 milliseconds for type conversion and transmission through a local area network.

In some RPC mechanisms there are limitations imposed by the underlying message passing transport service on the actual size of parameters and results passed. This is usually related to some link level packet size and is likely to be of the order of 500 to 1500 bytes. In some poor implementations these limitations are network dependent, so that an application tailored to exploit the 1500 byte limit of a local network (say) would fail if a link with a 500 byte limit were used. There is, however, an argument in favour of a limited parameter size, and that is that it forces the user to be aware of the expense of remote procedure calls for large parameter lists, and may force a more careful appraisal of the actual interface needed between client and server to minimise the passage of unnecessary data. Thus a first result of RPC parameter passing semantics is a careful re-structuring of application interfaces so that parameters become more specific, with minimal data being transmitted.

The address space problem also recurs when data types with embedded pointers are to be passed as parameters. A simple, linear linked list is one example of this, and two approaches are possible. The first would be to 'linearise' the list and pass it as an array of list elements. The links themselves have no meaning in the address space of the receiving process, but they can be regenerated relatively easily. In order to perform such regeneration, type information must be passed between client and server, indicating not only that a linked list is being passed, but which elements contain pointers and to which elements they point. This process of flattening and shaping of linked lists also extends to other structured data such as trees. However, it is important to note that if the flattening and shaping is to be performed automatically the stubs must be able to trace the links of the structure and translate each element encountered to or from the standardised representation.

Some systems (Ref. 11) require the programmer to define the packing and un-packing routines for each new parameter type introduced. A library of primitive packing functions, together with routines for building packing routines for complex types like arrays, records and unions out of simpler ones, is usually provided. Such a system can handle arbitrarily complex types but is hard to use and requires the programmers writing the packing routines for both client and server to use the same conventions. Herlihy and Liskov (Ref. 12) describe a strategy for transmitting values of abstract data types between modules that may use different representations for the type. The implementor of the type must specify a transmissible representation as well as an ordinary representation, together with a mapping between them.

The second approach to dealing with linked lists is to inspect the operations performed on the list and provide a more object-based rather than data-type-based interface. This will tend to produce an interface which is less dependent on the actual machine representation as well as one with smaller parameters. For example, a print server holding its queue locally as a linked list might be scanned by a client first asking for a copy of the queue and then searching the list, i.e.:

```
Client:  GetPrintQ(Q);
         for each element of Q
         {       if (Q.Value = "search string")
                 {       //process Q element
                 }
         }
```

A better approach, which hides the representation of the queue structure from the user, and prevents needless transmission of the list might be:

```
Client:  SearchQ("search string", QElement);
             //process Q element
```

which returns the queue element which matches "search string". This second approach is the same as that advocated for dealing with large arrays, and is a structuring technique which is widely advocated for building any program, while providing re-usable piece parts for the future. The re-usability comes from the concentration on objects rather than data structures. Thus application of an accepted software engineering principle eases some of the parameter passing problems associated with RPC. Automatic flattening and shaping of structures may still be useful for passing elements of a complex data type.

A few RPC mechanisms do allow passing of parameters by reference. These are usually closed systems, where a single

address space is shared by all processes in the system. De-referencing a parameter typically causes a page of memory to be swapped via the network, and accesses then take place on the local page. Clearly, support must be provided to ensure that only one copy of a given page exists within the system even in the face of machine failures, and there is an underlying assumption that all machines (and operating systems) are identical (Ref. 13). It could be possible to allow automatic de-referencing of parameters from one process back into the calling process. However, with the present low performance of transport services it seems acceptable to trade transparency for user awareness of performance issues.

Procedures as parameters give rise to even greater problems. If the RPC mechanism is being used with an interpreted language then there might be some point in trying to send the code to the server for execution. When compiled procedures in a heterogeneous machine environment are involved then another solution is needed. One approach is to introduce the idea of a server reference or *handle* which uniquely identifies a particular server and provides enough information for a client to make calls to that server. By allowing these handles to be passed as parameters to remote calls, a client can pass to any server the handle of a server within the client itself. Provided that the implementation allows general *nested* remote calls, the called server may make calls back to procedures within the original client. This is illustrated in Fig. 3.

The semantics of this operation are not the same as passing a procedure across to the server, but the mechanism is simple to implement and powerful to use. Very large parameters can also be treated in this way, with the recipient getting only the handle of a server that can deliver portions of the parameter as they are needed. Server handles have a wider application than simple call-back and are described in more detail in Section 4. The ability for a server to call its client back is very important and care is needed in the design of the RPC protocols to ensure that it is possible.

3.2 Call semantics

Although at-most-once call semantics are most desirable and are relatively easy to achieve, some RPC mechanisms provide only at-least-once semantics. This is very easy to implement over a connectionless protocol, requiring essentially a timer and re-try mechanism built into the client transmission routines. In a multi-programming operating system, such facilities can be provided in user space rather than in the operating system kernel, making the RPC package more portable. The main disadvantage of these cheap semantics is that they force the applications user to design idempotent interfaces; i.e. procedures must produce the same results and side-effects when re-tries, caused by loss of the results in transit, cause the procedure to be executed one or more times with the same parameters.

As an example, consider a sequential file of fixed-size records. A suitable procedure to read successive records might be:

```
ReadNextRecord(File)
```

which returns the next record from the named file after each call (we shall ignore initialisation and end-effects). Clearly, this is not idempotent and the server needs to keep track of the 'current record' for each client that may be accessing the file. An idempotent interface for sequential file access might be:

```
ReadRecordN(File, N)
```

which returns record N from the specified file. In this case the server needs to retain no client-related state, the so-called stateless server; it is up to each client to keep its own records. However, not all interfaces can be so easily transformed to an idempotent form. If we consider the addition of new records to the same sequential file, the usual approach might be with a procedure of the form:

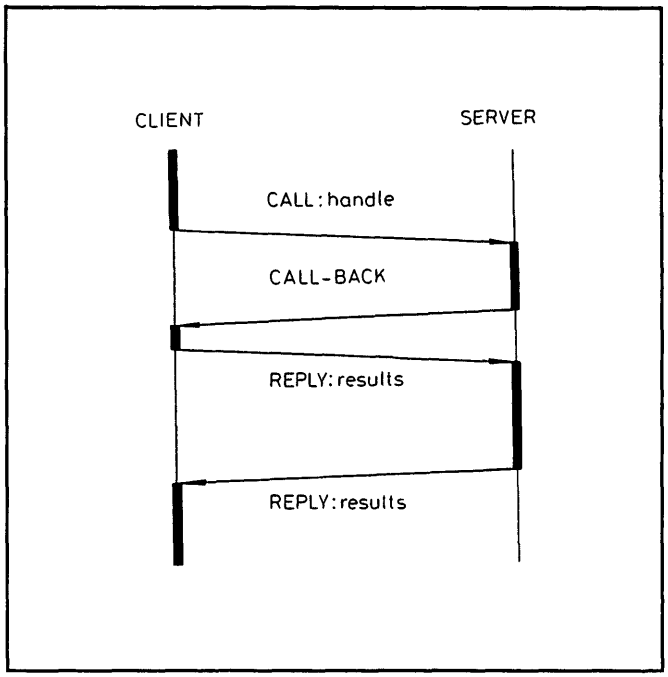

Fig. 3 Nested call-back

```
AppendRecord(File, Record)
```

which adds the specified record contents at the end of the existing file. It is clearly not idempotent since repeated execution will add further copies of the same record to the file. To convert this to an idempotent form at least two procedures are needed:

```
LastRecordNo(File)
WriteRecordN(File, N, Record)
```

the first of which returns the record number of the last record currently in the file, and the second of which writes a specified record. The client would use them as follows:

```
K := LastRecord(File)
WriteRecordN(File, K, Record)
```

In a single-client environment, network or machine crashes will be accommodated using at-least-once semantics. However, if multiple clients are to access the server at random times to perform updates the interface needs to include additional functions to form these calls into an atomic transaction (for example a locking mechanism).

Provided the AppendRecord procedure is atomic this application would not need a locking mechanism with at-most-once RPC semantics. Thus, while at-least-once semantics need only a simple protocol for implementation, and they minimise the amount of client-related state which needs to be retained in the server, in some cases they do force the applications user to more complex interfaces. At-most-once semantics are found in most RPC mechanisms. They closely model the behaviour of a local procedure call except that there will be some irrecoverable errors which have to be signalled back to the client or server. Such errors might be a client or server process crashing during a call. This is dealt with below.

3.3 Server semantics

Unlike a local procedure, the compiled code that implements a remote procedure is not linked into the client process, and as a result it is not loaded into memory when the client starts executing. It must reside in a separate process, the server, which will wait to execute the procedure on the client's behalf. In fact, the choice as to where the remote procedure will be executed is

often left until run time, and so it is rare (except in embedded systems) for the server code to be identified and loaded into an appropriate machine at the same time as the client. Usually the server is installed before the client ever runs or it is created (implicitly or explicitly) on the client's behalf when it is needed. Different implementations have chosen different mechanisms for creating RPC servers, giving rise to a wide variety of server semantics.

At one extreme we have a server that exists only for the duration of a single call. It is created by the RPC run time support system only when a call message arrives. After the call has been executed the server is deleted. Clearly, for it to be of use, the server must have some effect outside its own process, like reading or writing a file, telling the time or at least making another (nested) remote call. Because all these operations will involve the local operating system, they will be comparatively expensive. Only in the simplest cases can the server be created in advance of the call because it is not known what procedures might be called, and in addition the server cannot cache any important data in memory between calls. Any state that has to be preserved across several calls must be entrusted to the supporting operating system. In order to avoid this overhead the programmer may design the interface to the server so as to pass this state information to and from the server with each call, thereby losing the data abstraction across the client-server interface. As a result the RPC mechanism loses a lot of its attractiveness to the programmer.

At the other extreme is a persistent server, usually shared by many clients. A server of this sort can retain useful state, in memory, between calls and so can present a cleaner, more abstract interface to its clients. If the server is shared by several clients then the remote procedures that it offers must be designed so that interleaved or concurrent requests from different clients do not interfere with each other. For example, several clients searching a table using the following sequence will interfere with one another unless only one client is allowed to access the table at a time:

```
StartSearch(table)
until (item := GetNextEntry()) = EndOfTable do
        // Process item.
```

The server would have to lock the table when StartSearch is executed and only unlock it when GetNextEntry returns EndOfTable. It is unlikely that this option is acceptable, especially since a bug in such a client could lock the table indefinitely. The archetypical example of a shared server is an operating system kernel, where the problem is often solved by introducing the idea of a *descriptor*, allocated by the system, to identify the resource on which operations are to be performed. The same method is often seen in RPC servers that interleave operations on behalf of several clients. The example above can be re-written as:

```
desc := StartSearch(table)
until (item := GetNextEntry(desc)) = EndOfTable do
     //Process item.
```

provided the server uses the descriptor to identify the state of each sequence of requests.

Intermediate behaviour is possible by combining the use of persistent servers with the generic server mentioned in Section 2.5. The generic server is one that offers a server creation procedure, returning a handle that uniquely identifies the new server. The new server is *private* to the client that asked for it since it alone is in possession of the handle to it. The server can maintain state between calls but cannot share its data with other clients. The generic server can also be used to model the server per call semantics described earlier, with a new server being created each time a call is made.

4 Binding

Before a program can make a remote call it must possess a *handle* for the remote procedure. The exact nature of the handle will vary from implementation to implementation, but typically it will consist of the address of the server process together with enough information to identify a procedure within the server. The process by which the client gets hold of this handle is known as *binding*. The term is slightly misleading since the association between the client and the server is very weak. In many implementations the server may be quite unaware that a client has bound to it.

Binding is superficially analogous to link editing in a conventional programming language, but may differ in three interesting respects:

- The binding can be done at any time prior to making the call and is often left until run time.
- The binding may change during the client program's execution.
- It is possible for a client to be bound to several similar servers simultaneously.

The way in which these differences are exploited by an implementation greatly affects the way the user perceives the remote procedure call mechanism.

4.1 Bind time

The client can be provided with server handles by the programmer or operator prior to execution. This is useful if the application configuration is static or only changes infrequently, and where the extra cost of binding immediately prior to the call may significantly affect performance, as might, for example, be the case in raising alarms in a process control application.

Alternatively, the client can obtain a server handle at run time. This can happen in many ways. Consider, for example, a print command that can queue a file for printing on any specified host. The command might construct a handle for the required print server from the given host name and a service number (port):

```
Server := MakeHandle(HostName, PrintServerNumber)
RpcBindTo(Server)
```

This style of binding suffers from three major shortcomings. First, there must be an agreement that the print server always uses the same port number. This may not be possible for technical or administrative reasons on some systems. Secondly, there is no guarantee that the server exists and is healthy, and lastly by using the host name in this way it is not possible to install multiple (identical) servers on the same machine.

A binding service:

In order to allow more flexibility in the design of applications using RPC, many implementations include some form of *binding service* that can act as a broker between clients and the services that they require. When a service wants to make its facilities available to remote clients, it registers its handle with this binding service along with an identifying key. A client can obtain the server handle by quoting the appropriate key to the binding service. For simplicity the binder is usually accessed by remote calls. The more rigid forms of binding referred to above can be used to locate it.

How an application decides to make use of the keys is of paramount importance. If the format of the key is rigid and chosen by the RPC system then much of the flexibility lost. As an example, if the binding service uses keys that are simply

character strings, then one print service might register as:

```
RegisterAs("printer: slow, high quality, 1st floor", MyHandle())
```

and another as:

```
RegisterAs("printer: basement, fast, low quality", MyHandle())
```

A key of this sort is particularly useful if the binder offers a pattern match facility. The example below assumes that '*' will match any sequence of characters in the registration key:

```
RpcBindTo(Lookup("printer*fast*"))
PrintFile("myfile")
```

Naturally the more sophisticated the binder is, the more flexibility the application has in its naming.

In addition to any functional requirements, the binder *must* be robust against failures and should not become a performance bottleneck. Distributing the binding function among several servers and replicating information among them can satisfy both of these criteria. Unfortunately, a complex distributed database management package is needed to meet all these goals, and for this reason the functionality offered by many binders is lower than might be hoped for.

Server handles as parameters:

The binding service described above is simply a server that returns a handle to another service as a result of a remote call. By providing a data type to represent handles that may be passed from a client to a server as a parameter, or returned as the result of a call, the programmer may perform such operations for him/herself. To see how this might be used, consider once more the simple print server example above. If the print server runs on a machine with a separate file name space to the client, then a simple file name, passed to the server, will not refer to the correct file. The problem can be solved by making the client itself act as a simple file server. This server only implements read calls on an already open file. Before calling the print server, the client creates a server of this type running locally with the desired file open and ready for reading:

```
File := CreateFileServer(FileName)
```

The handle for the server is returned. The client can then call the print server, giving it the handle for the newly created file server:

```
PrintFile(File)
```

All the server needs to do is to bind to the file server and repeatedly call its read procedure:

```
RpcBindTo(File)
until ReadData(buffer) = EOF do
        Print(buffer)
```

Server handles used in this way greatly enhance the power of a basic remote procedure call mechanism.

4.2 Changing bindings

There is no reason why a client program should not change a binding during execution. This could be the response to a failed call or it could be a deliberate attempt to cause the same action to occur more than once, for example updating multiple copies of some replicated data. The previous binding can be cached in the client, so switching between similar servers that have all been bound to can be made very efficient.

It is also possible for the server to alter the binding. This might be necessary if the service needs to move to another host or to allow a new version of the server to be installed. Care is needed to ensure that any state data held by the server is no longer needed or can be duplicated in the replacement server. For example, a file server must be replaced when no files are open, or the new server must arrange to have the same set of open files, each correctly positioned. This sort of replacement is hard to do in general, but can be done quite easily in certain cases. The call semantics provided, and the amount of state kept by the server between calls will determine how practical it is.

4.3 Multiple simultaneous bindings

We have seen that there may be many servers available to service a remote call. Clearly a client can bind to any number of them in turn but it might bind to more than one *simultaneously*. Logically, a binding of this sort gives rise to multi-cast communication. The implication is that when a call to the given routine is made, several servers all process the call.

We have found a simple form of multi-cast to be very useful, but it is not clear how to integrate the facility into a high-level language, partly because the paradigm is alien to most current programming languages, and partly because handling results depends on how many responses are needed. For example, an *N*-modular redundancy algorithm would need a majority of servers to respond positively, while in other applications the first response from *N* requests is adequate.

5 Other issues

5.1 Type consistency

It is often the case that the various co-operating processes that make up a distributed application were developed quite separately. Basic principles of software engineering require that the interface between the processes be unambiguously defined to ensure that the finished components are compatible. Fortunately, this requirement coincides with that of the RPC system to know the order and type of the parameters and results of each procedure in the interface. The implementation can use this information to provide the server (and the client) with a degree of type safety, so that a call will not be accepted unless it appears to have come from a client that used the same interface definition as the server. Type safety is of particular importance to servers since they should be able to survive being sent corrupt call requests. Furthermore, in a distributed system where components may be changed without re-starting, clients and servers should be able to tell if they are incompatible. Also, provided the interface remains the same, the server's implementation should be able to change, say to remove a bug, without requiring that all its clients be re-compiled or modified in any way.

If the remote procedure's parameters are packed using a *tagged* representation (one in which the type of each field is encoded along with the corresponding value) then it is a simple matter for the server to check the type of the data as it arrives. However, a tagged representation is more expensive than an untagged one, both in terms of the quantity of data transferred and the time it takes each end to pack and unpack it. A simple alternative is to send a checksum, derived from the procedure parameter types, to the server which can be tested against the checksum of its own parameter types. If the checksum is obscure and covers a large value space, say 32 bits, then a very high degree of confidence can be achieved. This method is used in the implementation at UCL (Ref. 8) and has proved to be satisfactory.

The above discussion assumes that the RPC mechanism is being used from a type-checked language. If it is not, or the type checking is weak, then errors can be inadvertently introduced by passing incorrect data to the RPC stub routines. For example, a language like C, which uses untagged union types, will ultimately depend on well behaved programmers to ensure that the correct interpretation is placed on the union when it is passed to a stub.

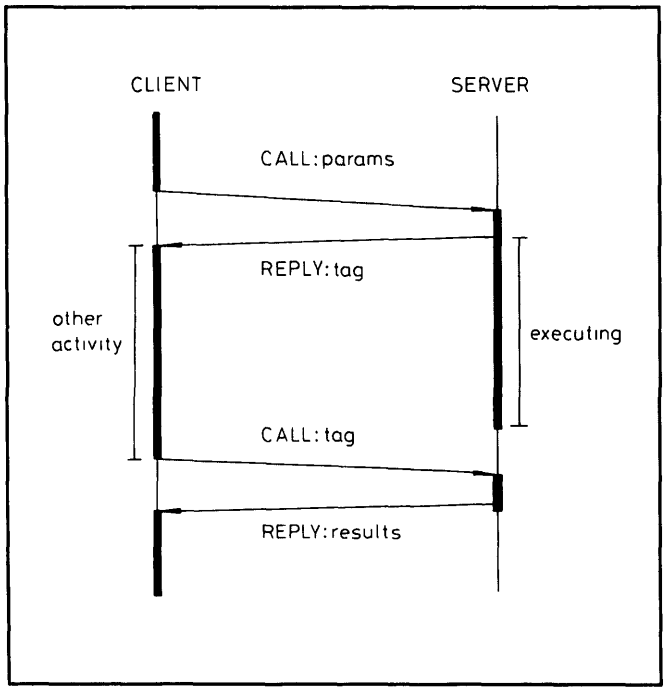

Fig. 4 Early reply

5.2 Error handling

When a programmer makes use of a local procedure call he/she is unlikely to consider the possibility of the call mechanism itself failing. If the call did it would be the result of low-level machine fault like an address error, for which there is unlikely to be any effective corrective action. The program might detect this error and re-start, but little else. The mechanism needed to support a remote procedure call is considerably more complex and involves many hardware and software components, any of which might fail independently of the rest. The situation is often worse in an inter-network environment where transient conditions like congestion can also cause a call to fail. Only in the simplest cases is it acceptable to treat these failures like a low-level hardware fault and abort the calling program. The programmer must be provided with some way to catch and identify the various errors and to take some corrective action.

For example, the client of a replicated database accessed by remote procedure calls will want to know if an access was unsuccessful so as to try another copy of the database. Furthermore, if at-most-once semantics are provided by the calling mechanism, such a client may well want to distinguish between a call that failed to get through and one that failed to reply, since in the latter case the procedure may have been executed.

Most programming languages do not provide any facilities for trapping and handling errors of this sort. Only in languages like Clu (Ref. 14) that provide an extensible mechanism for handling user-defined errors can an elegant and simple solution be found. Argus (Ref. 15), which is built on top of the Clu language and provides the programmer with nested atomic actions rather than simple remote procedure calls, goes much further in this respect, but the call overhead is consequently greater. In other cases, where RPC is provided as an addition to an existing language, extensions are sometimes provided to define handlers for the various classes of error.

To make matters worse, unless the transport protocols used to implement the remote call have access to reliable information about machine and process crashes and network errors, some failures will only show up as remote calls that do not return in a 'reasonable' time. Furthermore, if the called routine contains a bug that causes it to loop and never return, the client will be locked up waiting for a reply unless some provision is made for timing out calls. Only the application writer can choose the time-out since he or she alone knows how long a call might legitimately take.

Usually the RPC run time system will see a large number of different failures. For example, the target machine may be down, the server may not understand the message, or a reply might not be able to get back to the client, but in general what interests the programmer is whether the remote procedure was executed or not. If at-most-once semantics are used then there

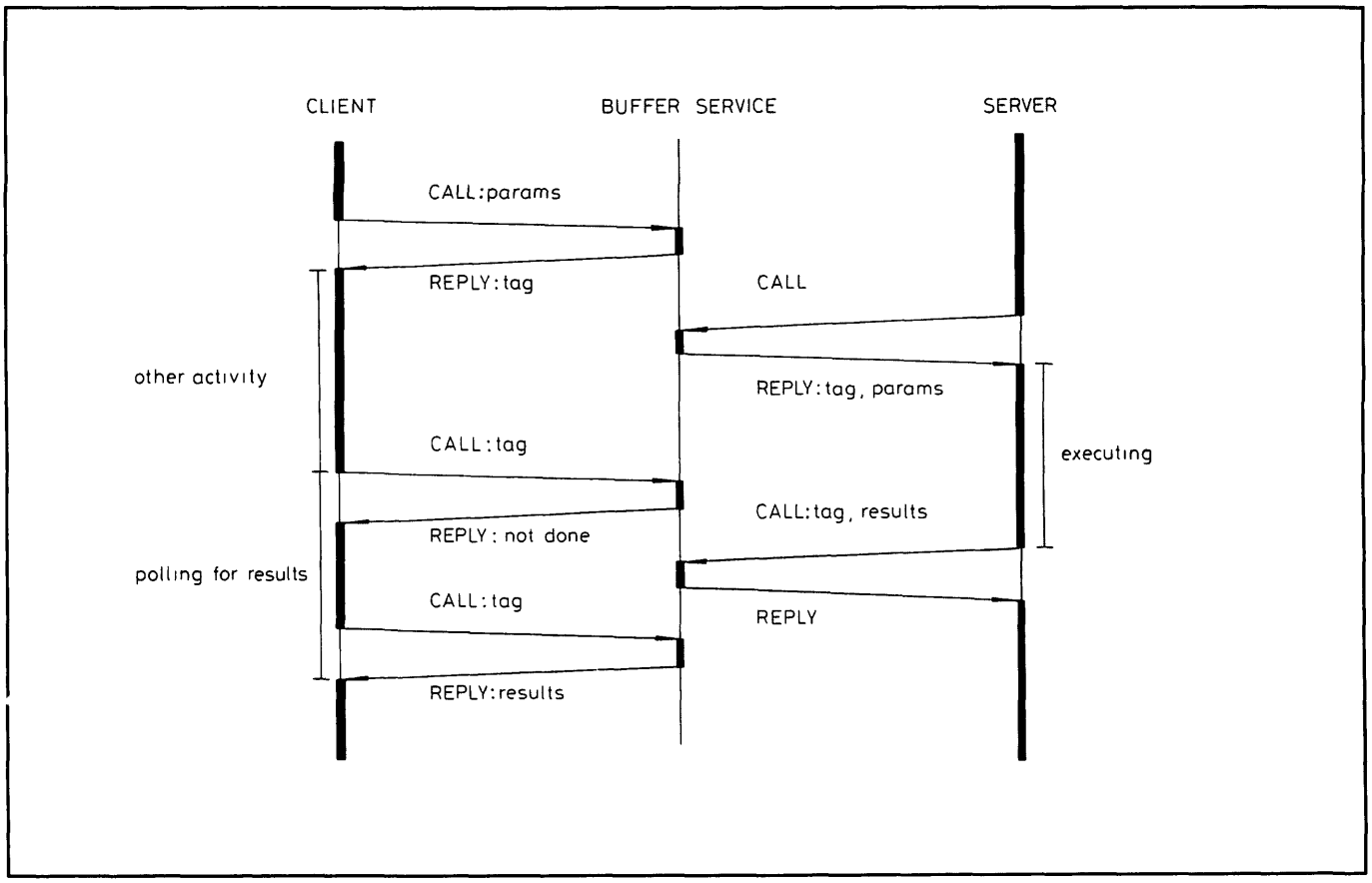

Fig. 5 Call buffering

will occasionally be some doubt since a server that crashes immediately after receiving a call request is indistinguishable from one that crashes immediately prior to replying to the call.

Servers, too, may need to be notified of client failures, particularly if the client ties up some valuable resource in the server. If the client crashes or abandons the call the resource should be freed automatically. Unfortunately because of the inherent asymmetry of the procedure call this is not a simple problem. Consider a client that opens a file in a server, executes a few read operations on it and then exits. If the server is informed that the client has exited it can close the open file, but the client may have given the server handle to another process, intending it to read the rest of the file. Solving the general garbage collection problem for server handles in a distributed system is very costly.

5.3 Efficiency considerations

Although RPC is frequently used as a familiar paradigm with which to split single-process applications across several machines (for example where the user interface is to be moved to a personal computer, and the computation is to be done on a larger, specialised machine), the use of servers shared by several clients is a frequently encountered distributed system structure. Clearly, RPC provides a suitable inter-process communications mechanism for such systems, but efficiency needs to be considered as well as familiarity and transparency.

If we consider a simple server which makes no calls on other servers, to execute a call it may need access to a resource which is temporarily unavailable, for example a shared file which is currently locked elsewhere. Such *local delays* degrade the performance of the system if it is not possible for the server to begin other incoming calls during the delay. A second form of delay, known as *remote delay*, can occur when a server calls a remote function which involves a considerable amount of computation to complete or involves a considerable transmission delay. In the absence of these delays servers will be maximally efficient if incoming calls are serviced serially. However, good RPC implementations must provide mechanisms to allow an efficient system to be built despite these delays, and this implies that the server must be able to service multiple requests simultaneously, or to pass an exception back to the client so that it might either locate another server or start some other activity.

Liskov *et al.* (Ref. 16) consider the cases of environments supporting static or dynamic tasks, and RPC or send/receive mechanisms, for providing adequate expressive power to cope with local and remote delays. They define expressive power to be whether common problems can be solved in a straightforward and efficient manner. They consider that the combination of RPC and static task structures does not provide adequate expressive power, arguing that although a server can be made up of a family of identical tasks to handle client requests, permanently allocating a task to each client is not feasible for many applications, and use of a task manager imposes extra delay as well as requiring clients to be well behaved about notifying the manager when they have finished with the server. Thus the client code must reflect the structure of the server implementation, thus exhibiting poor expressive power.

Thus a good RPC system will provide dynamic task/process creation at least for server implementation. However, the use of operating system provided processes can significantly reduce the server response as seen by the client because such processes take tens or hundreds of milliseconds to be created. Clearly, this cost is only acceptable when it can be amortised over many subsequent calls by the client. For this reason it is usual to find support for 'cheap processes' (or tasks or co-routines) within processes which take of the order of a millisecond to create. Their cheapness derives from their non preemptable nature, and because the code segment and address space is shared by all such tasks.

Such tasks can also provide a means of overcoming a fundamental characteristic of RPC, namely its lack of concurrency. Although one of the benefits of RPC is its synchronisation property, many distributed applications can benefit from concurrent access to multiple servers. Tasks in clients can provide a suitable mechanism, provided the addressing in the underlying protocol is rich enough to provide correct routing of responses.

An alternative to tasks in clients is to use the *early reply* approach (Fig. 4). With this, the call is split into two separate RPC calls, one passing the parameters to the server and the other requesting the results. The results of the first call may be a tag which can be passed back in the second call to associate them and identify the correct results. In principle, the client may interpose a delay between the two calls, and may indeed make several other RPC calls. However, if the request for results is delayed, it may cause congestion or unnecessary delay at the server.

As a final variant, it is worth noting the call buffering approach of Gimson (Ref. 17) (Fig. 5). An RPC call to service X is made to a call buffering server, where the request parameters together with the name of X and the client are buffered. Thereafter, the client will make periodic requests to the call buffering server to see if the call has been executed, and if so it will recover the results. In the meantime, servers also poll the call buffer server, to see if there are any calls awaiting them. If so, the parameters are recovered, the call is executed and a call is made back to the call buffer server to store the results.

Considering these various examples relating to process support for RPC, we can see that ideally a cheap mechanism for dynamic task creation should be provided as an adjunct to RPC. This provides a powerful and expressive mechanism to overcome some of the efficiency limitations of RPC in server-based environments. Other alternatives, such as early reply techniques, can ameliorate the problem but force the user to adopt a less elegant and more intrusive programming style.

5.4 Performance

Fig. 6 shows the average time taken for RPC calls with varying parameter sizes. In that particular RPC implementation (Ref. 8) a call to a null procedure with no parameters and no results took about 8 milliseconds to execute. The transmission of parameters and results took about 13 microseconds per byte. These times are for calls between two processes on lightly loaded Sun 3 workstations over a 10 Mbit per second Ethernet. In other implementations (Ref. 4), considerable effort was expended to reduce the null call cost to about 1 millisecond, while others using a connection-oriented transport-level substrate only achieved a null call time of 35 milliseconds. Thus,

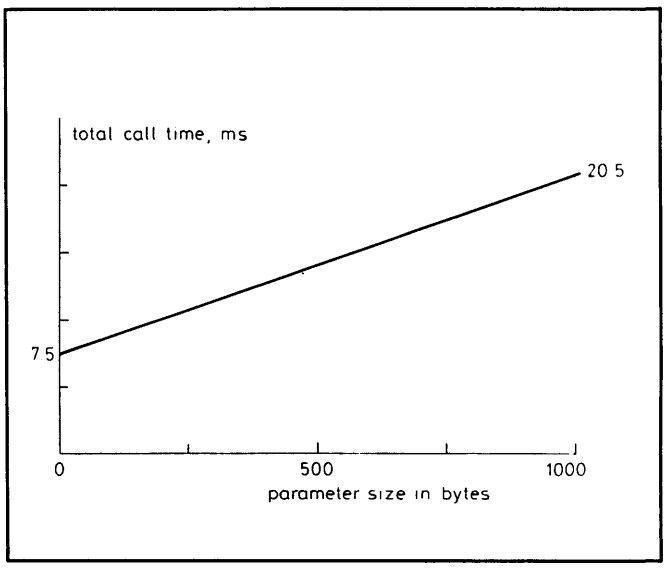

Fig. 6 Typical RPC performance

although RPC might provide a general-purpose mechanism, in practical applications remote calls will incur an overhead between two and three orders of magnitude greater than local calls. The impact of this is to force the user to carefully consider the separation of systems into modules and to design those interfaces which span machines to have minimal traffic flows. Such separation will often force the user to a more object-based view of servers, calling on the server to perform operations rather than transferring data structures to the client. Not only does this improve the performance of the application, but it improves modularity and data hiding properties of servers.

If we briefly look at the costs involved in making a remote procedure call in the absence of network errors:

call time = parameter packing + transmission queuing + network transmission
+ server queuing and scheduling + parameter unpacking + execution
+ results packing + transmission queuing + network transmission
+ client scheduling + results unpacking

This might be re-written as:

call time = parameter transformations
+ network transmission
+ execution
+ operating system delays

For a general-purpose heterogeneous RPC mechanism, parameter transformations are necessary, although knowledge of the system environment may allow local optimisations to be used to reduce this cost component. For calls with few parameters, the dominant overhead component will be the operating system delays. This is made up of context switching, operating system routing functions, network queuing, process scheduling and swapping if necessary. Where swapping of process images to disk is not involved, these delays reduce to an inverse function of the processor speed. Thus, moving the UCL RPC implementation from Sun 2 to Sun 3 processors has reduced the call time from 18 milliseconds to 7.5 milliseconds, and the cost per byte from 50 microseconds per byte to about 13 microseconds per byte.

Where larger parameter lists or results are involved, the transformation and transmission costs become more significant. Efficient transmission techniques have been devised (Ref. 18) which provide data rates exceeding conventional file transfer rates. Alternative approaches have been to use RPC as a mechanism for controlling the transfer of bulk data by a highly optimised protocol. For example, in one system a side-effect of certain RPC calls was to transfer a file between server and client machines. Such mechanisms may gain performance at the expense of transparency.

We have, so far, looked at the costs per call. Most servers implement a collection of related functions, and during a client-server association the client will typically make several calls to them. From the client's point of view the cost of the association will be:

association time = server location time
+ server creation time
+ $N \times$ call time
+ server termination notification time

for an association with N calls. In a system with pre-created servers which are at known addresses the location, creation and termination times reduce to zero. Where binding must take place, the server location time will take typically one to three times the call time. Dynamically created servers may take an extra call time to invoke the server manager together with the process or task creation or allocation time, and an extra call to terminate them. Thus for dynamic servers the call cost is very approximately:

effective call time = $((N + 4) \times$ call time + process creation time$)/N$

Thus for servers having only a small number of calls per association, the effective call time may be increased to several times the minimum.

Applications requiring more than about 50 to 100 remote calls per second may not be feasible with most RPC implementations. However, some applications which require higher call rates can be accommodated if *batching* of parameters is used. This can only be done where a sequence of calls do not return any results. The client stubs recognise these procedures and add their parameters into a transmission buffer, only sending them after either a pre-determined interval or when a suitable number of parameters are packed or when a call is made to one of the server's procedures which returns results. Thus several calls are made at the cost of a single call plus the usual per byte cost. Although this optimisation retains syntactic transparency, it may produce obscure timing-related effects where other clients are accessing the server simultaneously.

5.5 Inter-working

There is a large variety of RPC mechanisms in existence, many of them proprietary. It would be useful if it were possible to build a universal gateway which would allow all components written for one mechanism to inter-work with complementary components for another mechanism. Several aspects of the specific RPC mechanisms may make this extremely difficult:

- Clients and servers must agree on data types and representations for parameters and results. It may be that the base types in the two representations are different, and each set would need to be extended to include those from the other set. In cases where the standardised representation does not include type markers (tags) in the transmitted representation, a gateway would be unable to provide universal translation of parameters and results.
- It may not be possible to pass difficult data structures such as linked lists in some mechanisms.
- The call semantics may be different in the two mechanisms; automatic mapping is not feasible.
- Binding mechanisms may be very different in the two systems; for example, one may allow server handles to be passed as parameters, another may not.

Providing a universal RPC gateway may be possible between fairly similar mechanisms, but is not generally feasible. It is, however, possible to build specific gateways between clients using one RPC mechanism and a service implemented using another. The gateway builder has the responsibility for resolving the difficult mappings identified above.

An interesting RPC mechanism has been built recently at the University of Washington (Ref. 19). Workers there have identified five key areas of an RPC implementation: compile time support (interface descriptor language, stub generation etc.), binding protocol, transport protocol, call control protocol and data representation. They have managed to provide sufficiently rich procedural interfaces to these mechanisms that they can write applications which use their heterogeneous RPC (HRPC) to access or be accessed by any of three other existing RPC mechanisms. The stubs in the HRPC component adapt to the complementary component's RPC style at run time, apparently with virtually no performance penalty. The RPC mechanisms with which HRPC inter-works all appear to have at-most-once call semantics and to be rather similar. Nonetheless, this represents a useful step forward in moving to RPC standards with

acceptable cut-over costs.

6 Conclusions

In this paper we have presented those key aspects of RPC mechanisms which affect the user; i.e. we have tried to highlight where syntactic or semantic transparency between local and remote procedures is impaired. An RPC mechanism can be seen as being composed of a simple transaction protocol taking in all layers up to the presentation level of the OSI reference model, together with an environment in which clients and servers exist. Different RPC mechanisms do not just differ in the representation of parameters or in the underlying transport protocols. They differ also in the way processes can be bound, the richness of the binding, and in the way in which system structures and call semantics force the splitting of state between client and server. Such considerations make it very difficult to provide any automatic means of inter-working between RPC mechanisms other than where there are close similarities.

RPC is a useful structuring tool for distributed applications. Nonetheless, to make it an efficient and elegant mechanism, there must be a means of creating processes dynamically at run time. The cost of operating system processes is usually so great that a language-supported task structure or a tasking package within a process is an essential adjunct. Although techniques such as call buffering and early reply can solve the same problems, they are more disruptive of conventional programming styles.

Although the performance of remote procedure calls is significantly poorer than that of local calls, attention to the structuring of interfaces can minimise this effect for many applications. In those cases demanding even better performance, transfers via highly optimised protocols can be triggered as side-effects of RPC or batching of parameters and calls can be arranged.

Run time binding, which is commonly used, allows enhanced facilities over those found in most language systems. Relatively little has yet been done to explore the power and difficulties of an RPC mechanism including multi-cast calls and responses. There are many applications which could benefit from this approach, but before techniques like call batching and multi-cast remote calls become a valuable tool for application programmers language-level constructs must be provided for controlling them that do not intrude on the simplicity of the basic remote procedure call mechanism.

The object-oriented style of programming has been mentioned in this paper. By virtue of its 'information hiding' approach it is particularly suited for use in building distributed heterogeneous systems. The user is presented with a functional interface, and only its parameters need be transported between machines. However, if new objects can be defined from existing ones (inheritance) the mechanism has valuable software engineering properties too. Our current approach is leading in this direction.

In computer science it sometimes seems that the problems arising in a new environment bring insights and rationalisation of previously existing problems which had hitherto been solved on an *ad hoc* basis. Although we have given many reasons why RPC is not transparent in use, inclusion of support for RPC in programming languages may bring a new generation of languages which also solve existing problems. For example, the use of type specifications for interfaces and hiding internal representations behind procedural interfaces not only allow for distribution but make programs more portable and easier to modify. Mechanisms used for recovery from host or network errors are also useful as general-purpose programming constructs (Ref. 14), as are more dynamic binding mechanisms. Thus, rather than striving for total transparency within existing programming frameworks, we see RPC as highlighting the facilities to be provided in new procedural languages.

7 Acknowledgments

The authors wish to acknowledge the financial support for this work from the Alvey Directorate and the UK Science and Engineering Research Council. The contributions of our Alvey Admiral project collaborators from British Telecom Research Laboratories, GEC Research Laboratories and the University of London Computer Centre are also gratefully acknowledged.

8 References

1 HAMILTON, K. G.: 'A remote procedure call system', Ph.D. Thesis, Technical Report 70, Computer Laboratory, University of Cambridge, Cambridge, England, Dec. 1984
2 'Remote operations: model, notation and service definition', CCITT X.ros0 or ISO/DP 9072/1, Geneva, Switzerland, Oct. 1986
3 'ECMA distributed application support environment', European Computer Manufacturers Association, TC32-TG2/86/61, July 1986
4 BIRRELL, A., and NELSON, B. J.: 'Implementing remote procedure calls', *ACM Transactions on Computer Systems*, 1984, **2**, (1), pp. 39–59
5 'Remote procedure call protocol specification', *in* 'Networking on the Sun Workstation', Sun Microsystems, Inc., Mountain View, CA, USA, Feb. 1986
6 BROWNBRIDGE, D. R., MARSHALL, L. F., and RANDELL, B.: 'The Newcastle Connection or UNIXes of the world unite!', *Software — Practice & Experience*, 1982, **12**, pp. 1147–1162
7 GIBBONS, P. B.: 'A stub generator for multi-language RPC in heterogeneous environments', *IEEE Transactions on Software Engineering*, 1987, **SE-13**, (1), pp. 77–87
8 BACARISSE, B.: 'Remote procedure call: user guide for release 2 and 3', Internal Note IN-1936, Department of Computer Science, University College London, London, England, Apr. 1986
9 'Specification of basic encoding rules for abstract syntax notation one (ASN.1)', ISO TC97, Draft International Standard ISO/DIS 8825, June 1985
10 'Specification of protocols for common application service elements. Part 3: Commitment, concurrency and recovery', ISO TC97 SC21 WG6, Working Paper ISO/OP 8650/3, Oct. 1986
11 'External data representation protocol specification', *in* 'Networking on the Sun Workstation', Sun Microsystems, Inc., Mountain View, CA, USA, Feb. 1986
12 HERLIHY, M., and LISKOV, B.: 'A value transmission method for abstract data types', *ACM Transactions on Programming Languages and Systems*, 1982, **4**, (4), pp. 527–551
13 LEACH, P. J., LEVINE, P.H., DOUROS, B.P., HAMILTON, J. A., NELSON, D.L., and STUMPF, B.L.: 'The architecture of an integrated local network', *IEEE Journal on Selected Areas in Communications*, 1983, **SAC-1**, pp. 842–857
14 LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SHAFFERT, C., SCHEIFLER, R., and SNYDER, A.: 'CLU reference manual', Lecture Notes in Computer Science (Springer-Verlag, 1981)
15 LISKOV, B., and SCHEIFLER, R.: 'Guardians and actions: linguistic support for robust distributed programs', *ACM Transactions on Programming Languages and Systems*, 1983, **5**, (3), pp. 381–404
16 LISKOV, B., HERLIHY, M., and GILBERT, L.: 'Limitations of remote procedure call and static process structure for distributed computing', Programming Methodology Group Memo 41, Massachusetts Institute of Technology, Cambridge, MA, USA, Sept. 1984
17 GIMSON, R.: 'Call buffering service', Technical Report 19, Programming Research Group, Oxford University, Oxford, England, 1985
18 CROWCROFT, J., and RIDDOCH, M.: 'Sequenced exchange protocol', Internal Note IN-1824, Department of Computer Science, University College London, London, England, Aug. 1986
19 BERSHAD, B.N., CHING, D.T., LAZOWSKA, E.D., SANISLO, J., and SCHWARTZ, M.: 'A remote procedure call facility for heterogeneous computer systems', Technical Report 86–09–10, Department of Computer Science, University of Washington, Washington, DC, USA, Sept. 1986

S. R. Wilbur and B. Bacarisse are with the Department of Computer Science, University College London, Gower Street, London WC1E 6BT, England.