1989

Design Considerations for Distributed Scientific Software Systems

Paul E. Buis

Report Number:

89-909

DESIGN CONSIDERATIONS FOR DISTRIBUTED SCIENTIFIC SOFTWARE SYSTEMS

Paul E. Buis

CSD-TR-909
September 1989

Design Considerations for Distributed Scientific Software Systems

Paul E. Buis

Abstract

In this paper we enumerate the various design issues in distributed computing systems. In particular, we look at the issues facing the conversion of non-distributed scientific systems into distributed systems. The discussion focuses on interface specification, binding, and protocols.

1 Introduction

In this paper we enumerate the various design issues in distributed computing systems and give examples of how they have been dealt with in existing systems. We look at the issues facing the conversion of two particular types of non-distributed scientific systems into distributed systems. We consider systems based on libraries, such as IMSL, and systems based on software parts technology, such as ELLPACK. The discussion focuses on interface specification, the *what* of message passing; binding, the *where* of message passing; and protocols, the *how* of message passing.

2 Interface Specification

In a subroutine library, the interfaces are specified by the subroutine parameters. When converting such systems for distributed use, the most natural paradigm is remote procedure call (RPC). RPC is a common form of interprocess communication (IPC) where input parameters are converted into a message sent to a server and a client waits for output parameters to be returned in a message. The most convenient form of interface specification is one that can be compiled into code to do the message passing. Several interface description languages (IDLs) have been built with associated code generators for RPC. The code is generated as subroutines that replace computation code with message passing code (called *stub* routines). Ideally, one would like to use the parameter specifications in the original subroutine's source code as an interface specification for the RPC system. To the best of my knowledge, no such compilers yet exist.

In a software parts technology based system, the interface between modules takes the form of structures located in global memory and subroutines containing user-supplied functions. One would still like to have interface specifications that are compilable into message-passing code even if the message passing paradigm is not as simple as in the RPC case. Fortunately, one should be able to use the tools developed for RPC since the set of input (or output) parameters for a subroutine can be viewed as a structure and vice versa.

Fortran 8X, currently a draft proposal for a replacement for FORTRAN, includes a new piece of parameter description syntax called the "INTENT" statement. With this new syntax, one can specify whether a parameter is intended to be used as an input parameter and/or an output parameter. This allows the compiler to do additional safety checking and optimization. However, it will also allow Fortran 8X interface modules (another new piece of syntax) to be used as RPC interface specifications. For example, to specify a subroutine that adds two floating point numbers and returns their sum, one could write:

```
interface
subroutine add(a, b, c)
real a, b, c
intent (in) a, b
intent (out) c
end interface
```

Xerox has developed an RPC system called Courier that contains an IDL based on the Mesa language. To the best of my knowledge, the Courier system does not support floating point numbers [15], but this has probably been added by others since Xerox. This IDL was used at Cornell as the basis for a C language stub generator under UNIX, and modified for use with more heterogeneous systems at the University of Washington [11]. In Courier, one could specify a routine to add two integers as follows:

```
Add : PROCEDURE[ a: LONG INTEGER, b: LONG INTEGER ]
 RETURNS [ c: LONG INTEGER ] ;
```

Netwise has developed an RPC system called RPC TOOL that uses annotated ANSI C as its IDL. This system is targeted at inter-operating system heterogeneity with support for MS-DOS, OS/2, several flavors of UNIX, and VMS [10,14]. With it, one could declare a procedure to add two floating point numbers as:

```
    add( float [in] a, float [in] b, float [out] *c);
```

Sun Microsystems has developed an RPC system with an IDL called RPCL based on the C language. RPCL is used to specify the interface for their Network File System (NFS) and Yellow Pages (YP) services [4,5,6,8]. This system would be particularly useful in software parts systems because it uses the equivalence between structures and parameters lists. One could specify the addition of two floating point numbers as follows:

```
struct add_in{ float a, b; };
struct add_out{ float c; };
PROGRAM {
        VERSION {
                add_out add(add_in) = 1;
        } = 1;
} = 0x20001234;
```

3 Binding

In a non-distributed system, binding is done by the compiler and the linker. The compiler binds names to code and data, and the linker binds these to addresses. In a distributed system, (machine name, program name) pairs must be bound to (machine address, program address) pairs. The problem of binding can be broken down into 5 issues: machine naming, program naming, port determination, activation, and duration.

3.1 Machine Naming

Machine names need to be bound to network addresses. For BSD UNIX systems this means Internet Protocol (IP) addresses. The most primitive scheme for this binding is to simply have a file of all known (machine name, machine address) pairs called /etc/hosts on every machine. This file is accessed via the gethostent() system call. As machines are added to a network, updating all these files becomes an administrative nightmare, so a system to span a local network was developed by Sun Microsystems called the Yellow Pages. It uses RPC to provide the functionality of the gethostent() call (and similar administrative tasks). However, to be able to access the huge number of machines on the entire Internet, even this is too simple a scheme. More sophisticated protocols have been developed to cope with this problem [21,22]. The binding of machine names to addresses need not be a deep concern as long as one limits oneself to UNIX systems on the Internet.

3.2 Program Naming

In a non-distributed environment, a program is typically referred to by the name of the disk file which contains the executable form of the code for the program. This may be inappropriate in the distributed case. If the distributed system is heterogeneous at the operating system level, the naming of files may differ from one system to another and lead to confusion. The entire name of a disk file may be quite long and/or not describe the use of the program it contains.

One extreme solution to program naming is to avoid the issue by making distributed programs accessible only by *a priori* knowledge of their program addresses. SUNRPC uses a technique that is one step less crude: each program is identified by a program number that is mapped directly to a port number by a binding agent at run time. More sophisticated systems provide a global namespace that is searched by a binding agent to determine both machine and program addresses for servers. In such systems, the binding agent may become a reliability problem and/or a performance bottleneck.

3.3 Port Determination

Once the namespace has been searched and a particular destination (machine, process) name pair for a message has been identified, one need further refine the address beyond the machine name to a particular communication port number.

One solution is to assume all port number assignments are well-known. The file `/etc/services` contains well-known port numbers for UNIX systems. This technique leads to the same administrative difficulties as does the use of `/etc/hosts` for machine addresses: as the set of servers grows (or, heaven forbid, changes dynamically), updating the files on all machines becomes an unmanageable chore.

Another solution is to use a single well-known port on all machines that is attached to a process that either activates servers (such as `inetd` on BSD UNIX systems) or forwards messages.

Finally, one can set up port determining agents (with well-known addresses) that know all (process name, port number) pairs for a machine or set of machines. SUNRPC uses a program called the portmapper that is told the port number of a server by the server when the server starts up. However, its knowledge is limited to the local machine, so clients must know the name of the server's machine *a priori*. Also, if the server dies, the portmapper will incorrectly inform clients to use a port already claimed by a process other than the server.

3.4 Activation

The activation of the program that is to receive input is an issue. The questions are who and when will activate the program. If a program is active all the time, it will be consuming machine resources such as virtual memory space. If a program is not activated until bind-time, there will be a delay while the program is activated. In SUNRPC it is assumed that the program is already active. The UNIX programs listed in `/etc/inetd.conf` are not activated until a message is sent to a daemon program (`inetd`) that acts as an activation agent.

Also, in UNIX systems, security arrangements such as file access are associated with the activator of programs. Hence, programs activated by `inetd` have superuser privileges. In the SUPERSRV system, each supplier of a server activates an activation agent which must be contacted to activate the server. Thus, each server runs with the privileges of its supplier.

The tradeoff is between the simplicity of leaving programs running all the time and the complexity of using activation agents. Manually activating programs (via `rsh`) is only suitable for the crudest systems.

3.5 Duration

The duration of the binding is also an issue. If the binding only lasts for the duration of the receipt of a message (or a receive/reply), the overhead of binding may be significant. However, if the binding is more permanent, either the server must be constructed so it can communicate with multiple clients simultaneously (via UDP for instance) or be limited to dealing with one client at a time and denying/delaying service to others. The ISIS system has a subsystem for lightweight processes to allow multiple simultaneous clients within the same server process [13]. With such a system, one must be careful that the multiple clients do not adversely interact.

4 Protocols

The question of how messages get from source to destination can be broken into three components: the transport protocol, the data representation protocol, and the control protocol. The transport protocol is the lowest level and deals with simply getting bytes from here to there. The data representation protocol deals with the meaning of those bytes and how to translate structures into byte strings. The control protocol deals with the bookkeeping needed to be sure the bytes get where they are supposed to go.

4.1 Transport Protocols

The transport protocol provides a higher level interface to the physical communication channel. Various physical communication channels can be used, such as disk files, shared memory, ethernet, token-rings, or telephone lines. The simplest transport protocols to use are those that are directly supported by the operating system.

In UNIX, intermachine transport is provided by the TCP and UDP protocols. TCP supplies a reliable ordered byte stream and UDP supplies an unreliable, unordered packet stream. Thus TCP is usable as is, where UDP requires a higher level transport protocol to obtain reliable, sequenced packet streams. However, TCP can be inefficient due to the overhead needed to provide reliable byte streams when reliable packet streams would suffice [12]. Protocols based on UDP may also have potential advantages in that they may not report failure when the network appears to have failed.

If the programs needing to communicate happen to be on the same machine, many systems provide interprocess communication facilities that can be used instead of intermachine protocols. BSD UNIX provides named pipes, and System V UNIX provides shared memory. On some UNIX derivatives, both pipes and shared memory can be used as a transport protocol.

Other alternatives are to exchange data via disk files, which can be convenient if all machines in the system share filespace. This may be the only practical method if source and destination machines are not particularly compatible. In this case, file exchange protocols such as FTP must be invoked [20]. Some systems have even used electronic mail over 1200 baud telephone lines in a file-exchange-based transport protocol.

4.2 Data Representation Protocol

The data representation protocol deals with how to convert language constructs to and from bytes sent via the transport protocol. The data representation must be compatible with the transport protocol. For instance, some transport protocols will only handle printable ASCII characters [25]. Or, if file transfer is used, an end-of-file character will need to be avoided on some systems.

When in a heterogeneous hardware environment, it will not be practical to use the internal representation of data in the transmitted byte stream. Source and destination machines may use different internal representations and it will be necessary for either the sender or receiver to do a conversion. To simplify the conversions, all data can be transmitted in a common format. The printed representations of values would suffice if all machines used the same character set. Even if this were the case, printed representations are not as compact as non-printable representations. Sun Microsystems' XDR standard [2] specifies use of IEEE encodings for basic data types.

The transmission of compound and abstract data types such as records and lists also poses difficulties. Instances of compound data types can simply be broken down into their basic components and transmitted one component at a time. Transmission of instances of abstract data types is much more complex. Objects such as lists contain references (pointers) to other objects which also must be transmitted. Under some fairly reasonable assumptions, this type of transmission can be done [17]. Abstractions such as continuous functions that are represented as code rather than data are virtually impossible to transmit. This can be dealt with either by transmitting a discrete approximation to the function or by allowing the function to be accessed via RPC.

Typing can be either explicit or implicit. In an explicitly typed system, the transmission of each data value is preceded by a code for the type. In an implicitly typed system, no type information is transmitted. This results in a more compact encoding, but the receiver is required to anticipate the type of each data value in order to convert it to the correct internal format. Usually implicit typing suffices, and when one needs explicit typing, one can emulate it with explicit tags in the data values. Both the Sun XDR and Xerox Courier RPC data representation standards call for implicit typing.

An example of explicit typing is the National Bureau of Standards (NBS) Computer Based Message System (CBMS) standard [15,24].

4.3 Control Protocol

The control protocol determines the reliability and flexibility of the overall system. A message has not been reliably delivered simply by reliably transmitting bytes from here to there. Another level of reliability is obtained when there is an acknowledgment that the receiver has understood the message and processed it. Flexibility is obtained by allowing for transmission of the same message to a group of receivers. Additional flexibility is obtained by giving control over when the message is to be delivered.

4.3.1 Reliability

In an RPC-based system, receipt of output parameters by the client also serves as an indication that the server successfully processed the input message. The difficult part of maintaining reliability in such a system is handling the case of an error. Errors are signaled by the lack of receipt of the output message. One may rely on a timeout mechanism. That is, if a response is not received within a certain amount of time, then either an error is assumed to have occurred, or the control protocol attempts to verify that the server is still functioning. For less catastrophic errors, the client must be prepared to receive a message indicating that the server was unable to process the input.

The reliability of message delivery will affect the semantics of the message passing system. Three different semantics can be provided: at-most-once delivery, at-least-once delivery, and exactly-once delivery. At-most-once semantics are the simplest to implement; the control protocol tries to deliver the message to at most one server, and if this fails, simply gives up. At-least-once delivery requires the control protocol to try to deliver the message to any server that will accept it until one of them acknowledges a successful delivery. Exactly-once delivery is difficult to implement, since it requires not only the delivery of the message at least once, but also the cancellation of the effects of any unacknowledged multiple receives. This implies a multi-stage acknowledgment and the capacity of the receiver to roll back any effect of undesired state changes.

4.3.2 Flexibility

Sometimes it will be useful to have the same message delivered to a set of destinations rather than a single destination. If this set includes all machines on a network, this is called *broadcasting*. If this set is smaller but contains more than one destination, this is called *multicasting*. Some transport protocols may offer one or both of these options. For instance, on UNIX systems the superuser may use UDP to do a broadcast to all machines on the same physical network. Otherwise, these actions are the responsibility of the control protocol.

The timing of the delivery of a message relative to the end of the execution of the user-invoked subroutine can also vary. If the user code blocks until the delivery is complete, the message passing is said to be *synchronous*. If not, the message passing is said to be *asynchronous*. If the transport protocol provides for asynchronous delivery, synchronous delivery can be simulated by waiting for an acknowledgment by the sender. If the transport protocol does not provide for asynchronous delivery, it can be simulated by sending the message to an intermediate delivery agent that is guaranteed to receive the message virtually instantaneously. Some transport mechanisms allow for only limited size messages to be delivered asynchronously, and will lead to blocking if a message buffer becomes full. UNIX pipes are an example of a mechanism with such limited asynchrony.

An extreme form of asynchrony is the provision for messages to be delivered when some event occurs. For instance, it may be useful for a server to arrange for a message to be sent to a binding agent whenever the server dies. This sort of arrangement requires a process external to the user's process to monitor such events and arrange for the message delivery. The ISIS system includes a control protocol for this extremely flexible form of message passing [13].

5 Conclusion

When building a distributed system, one must choose appropriate tools to build the system. Different tools provide different degrees of integration. Several systems offer integration on the language or operating system level [9,18,19]. These are mostly inappropriate for our purposes since they require rewriting the entire system in a new language and/or investing in hardware capable of supporting the operating system. Other systems supply distributed communication facilities as a "black-box" with no control over how the communication is performed. Such systems may be acceptable for prototyping, but for a finished product, we felt we wanted more control. Alternatively, one can implement a distributed system from scratch, a project involving too much effort on issues that have no direct bearing on the application at hand. The route we find most acceptable is a combination of the last two: take "black-box" components, and combine them oneself.

The projects we are attempting have the following restrictions:

1. We are limited to BSD-like UNIX operating systems on a variety of hardware types.

2. The application code we want to interface with is written in FORTRAN on at least one of the endpoints of communication.

3. We have virtually no money to buy commercial software and for portability, we cannot use software without source code.

4. We do not require multicasting.

The first restriction is both a blessing and a curse. We need not accommodate operating system heterogeneity to the degree that the HRPC project at the University of Washington does [11,23]. However, different BSD-like systems have differing degrees of similarity. Since our environment has a fair degree of homogeneity, we can do more sophisticated things.

Dealing with FORTRAN code means we need to interface FORTRAN to C, as FORTRAN bindings to UNIX system calls are not widely available. However our communications interface need not be as general as the multilanguage RPC stub generator of Gibbons [16]. Since FORTRAN does not yet support a a pointer data type, we can avoid much of the difficulty faced by Herlihy and Liskov [17]. To mass produce FORTRAN interfaces for a RPC library, we are working on a stub generator that will accept such Fortran 8X interface specifications as input and produce C code as output. This will allow interfaces definitions to be constructed in a semi-automatic manner.

Lack of funds and a desire for source code has made me fond of using parts of the SUNRPC package. SUNRPC is available without charge from Sun Microsystems in source code form. In addition, all of the parts needed are already compiled and installed on most machines running Sun's NFS [1,3,7]. Hence, we can modify the parts we don't like and can assume the rest are installed and correct.

The SUNPRC package has several weaknesses that require writing subsystems from scratch:

1. Lack of a global binding agent. I have been working on a program to use with SUNRPC that will have knowledge of a set of machines. However, the current version has no provision for redundancy if the machine it resides on fails. This project may be abandoned if a workable pre-written binding/activation package is found.

2. Assumption that servers will be active all of the time and will only support one client at a time. A compromise that I have used with some of my SUNRPC programs is to instruct the server to `fork()` at bind-time, creating multiple (heavyweight) processes that last as long as the client requires. This ensures that the clients will not interact at all, which may not be desired in all circumstances.

3. Overly simple control protocol based on timeouts. If the server takes too long to compute results, the client will assume it has crashed. Turning off the timeout mechanism will cause an RPC request to wait forever if the server machine crashes while an RPC request is pending.

4. Lack of provision for a callback during an RPC request. The server cannot call a procedure located in the client while processing an RPC request from that client.

If multicasting were required, the control protocol would become sufficiently complex as to mandate the use of a "black-box" control protocol such as the one included in ISIS. However, the ISIS system has a different set of weaknesses.

References

[1] *Concentrix NFS Installation Guide and Release Notes.* Alliant Computer Systems Corp., Littleton, Mass., May 1987. Part No: 300-40004-B.

[2] External data representation standard: protocol specification. RFC 1014, June 1987.

[3] *Network File System Reference.* Tektronix, Beaverton, Oregon, February 1988. Part No: 070-6628-00.

[4] *Network Programming.* Sun Microsystems, Inc., Mountain View, California, May 1988. Part No: 800-1779-10.

[5] *Networking on the Sun Workstation.* Sun Microsystems, Inc., Mountain View, California, February 1986. Part No: 800-1324-03.

[6] NFS: Network file system protocol specification. RFC 1094, June 1988.

[7] *Programming and Protocols for NFS Services.* Hewlett-Packard, Fort Collins, Colorado, January 1989. Part No: 50969-90010.

[8] *Yellow Pages Protocol Specification.* Sun Microsystems, Inc., Mountain View, California, February 1986.

[9] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[10] David L. Andrus. Netwise remote procedure call application development tools. *NetWare Technical Journal*, 28–31, October 1988.

[11] Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[12] Andrew D. Birell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[13] K. Birman, R. Cooper, T. Joseph, K. Kane, and F. Schmuck. *The ISIS System Manual, Version 1.1.* The ISIS Project, June 1989.

[14] Susan Breidenbach. Netwise: In center stage with RPC tools. *LAN Week*, January 1989.

[15] A. L. DeSchon. Survey of data representation standards. RFC 971, January 1986.

[16] Phillip B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77–87, January 1987.

[17] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[18] Barbara Liskov. On linguistic support for distributed programs. *IEEE Transactions on Software Engineering*, SE-8(3):203–210, May 1982.

[19] Barbara Liskov and Robert Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[20] Oliver A. McBryan. *Using Supercomputers as Attatched Processors*, pages 130–146. SIAM, 1987.

[21] Paul Mockapetris. Domain names – concepts and facilities. RFC 1035, November 1987.

[22] Paul Mockapetris. Domain names – implementation and specification. RFC 1034, November 1987.

[23] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 259–273, March 1988.

[24] National Bereau of Standards. Specification for message format for computer based message systems. RFC 841, January 1983.

[25] B. Teufel. System design for the remote execution of library routines. *The Computer Journal*, 30(3):254–257, 1987.