Data Handling Division
DD/87/13
T.J. Berners-Lee
June 1987

EXPERIENCE WITH REMOTE PROCEDURE CALL IN DATA ACQUISITION AND CONTROL

(Submitted to the 5th Conference on Real-Time Computer Applications in Nuclear Particle and Plasma Physics, San Francisco, 12-14 May 1987)

DD-jt

Experience with Remote Procedure Call in Data Acquisition and Control

T.J. Berners-Lee
C.E.R.N., 1211 Geneva 23, Switzerland

Abstract

Remote Procedure Call is a flexible technique for building distributed systems. Experience with systems including a variety of processor types and communication media has demonstrated some of its advantages, and pointed out special facilities which are required and simplifying restrictions which are acceptable.

Introduction

Remote Procedure Call (RPC) allows a program on one machine to call a subroutine on another, without knowing that it is remote. Looked at the other way round, any remote operation may be represented by a user subroutine call in a high level language. The principle of RPC is discussed in detail in [1], and practical details of this particular system in [2], so neither is elaborated here. This paper presents the experience gained from our particular system, the facilities we found to be required, and the design choices which these entailed.

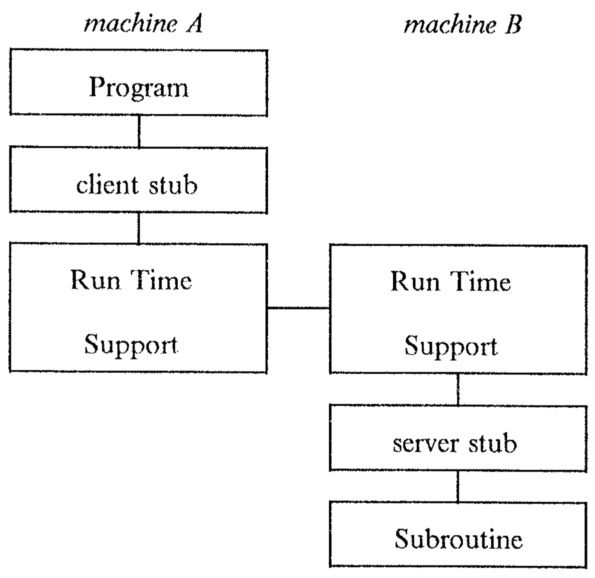

Fig. 1. A Remote Call: Software Components
A vertical line indicates a procedure call

Applications

RPC is being used by the Online group in the Data Handling division at CERN for a variety of applications in control and data acquisition systems for physics experiments and related test equipment. These systems must be distributed for three reasons: the severe geographical constraints, the system topology required to handle very large data rates, and the fact that the design teams are distributed between collaborating departments. In our case, processors involved are VAX ™ minicomputers, and 32 bit (M68000 series) and 16 bit (M6809) microprocessors.

Examples of the use of RPC include access by M68000 based test systems [3] to files and terminals on minicomputers, and allowing tasks on minicomputers to monitor and control M68000 or M6809 based equipment. User-written programs may consist of modules, some of which reside on a minicomputer, and some of which reside in microprocessors in FASTBUS [4] or VMEbus [5] environments.

Transparency

One of the advantages in using RPC as the basis for a distributed system is that it removes the need for users to call communications facilities explicitly. Therefore, as communication protocols and hardware evolve, application code need not be recompiled. This is important in our enviornment, where DECNET ™, raw Ethernet, ISO Class 4 Transport, and RS232 links are currently used, but where direct communication over FASTBUS and parallel cables is foreseen for the future.

As well as giving transparency to the communication media, RPC gives transparency to data representations. Because the RPC system is aware of the data types involved, it can automatically convert byte order and real number formats as required.

Description Language

Given a formal description of each subroutine and its parameters, an RPC system generates the code necessary to perform all the communication between calling and called modules.

In our case, we chose an Ada ™ –like language to describe the procedure interfaces, as it was the only real time language in use locally which provides sufficient information about each parameter, namely its type and direction. From the Ada definition file, the RPC compiler produces two "stub" modules. One stub emulates the called procedure on the calling machine, and one emulates the calling module on the called machine. (The RPC compiler itself is written in, and produces, Pascal, though application modules may be written in other languages). The calling process is known as the "client", and the called process the "server" for this call.

Message format

Standardisation of the format used in RPC messages is still an open question, despite its obvious advantages for interworking. One of the reasons for this is that different RPC systems have been optimised according to different, conflicting, criteria. There is a trade-off, for example, in the coding scheme between generality, economy of message length, and encoding speed.

In our case, we chose the Xerox Courier [6] format, in which data sizes are defined at compilation time, and data is marshalled into bytes most significant byte first. The overhead of encoding data according to the X.409 standard [7] was felt to be too high, both in time and space. Some systems deliberately check whether the communicating machines happen to be of the same type, and in this case skip the conversion of data into standard form. The time, space and complexity of this was not felt to be justified by the relatively small amount of time spent converting data formats. (This is especially so when the conversion routines called by the stub code are expanded in-line by a good optimising compiler.)

Data Types

In reponse to local demand, FORTRAN data types are supported. Other languages are supported with some restrictions. An additional restriction on the total size of all parameters to any remote call has permitted the use of simplified and faster buffer management.

We had to extend the Courier data types to include a real number type (sent in IEEE format). We also found that user program portability is greatly enhanced by the addition of a type which is represented as the native integer on any machine, but is constrained to lie in a 16 bit range, and is passed as 16 bits.

Addressing

The Courier message format which we adopted provides for two levels of addressing, these being the index of a procedure within the "package", ("program" in Courier terms) and the number of the package within the server. To these two layers of address must be added the medium to be used for communication, and the network address of the server task, making a total of four layers.

Unlike in some systems, however, no numbers or identifiers need be globally unique. The procedure number within the package is only known by the client and server stub modules, and is allocated at compilation time. The other three layers are allocated at load time.

The package number is generated by, and only relevant in the context of, the server on which it it provided. It could be used as a capability in a capability-based authorisation system. The client stub module knows only a logical name for its server counterpart: this is used to look up the full address of the remote package. Although this process currently uses local tables which are set up when the system is configured, provision is made for the search to be extended to a heirarchical sequence of name servers.

We find that a distributed system is rarely built as a monolithic whole, but grows by the progressive amalgamation of independently designed sub systems. It is therefore important that a system may function independently as much as possible, while also being part of a larger system. The principle of information hiding in modular software then applies equally to distributed software, and requires that one does not rely on globally unique identifiers or centralised administration.

Call Time vs. Load Time

Trade-offs exist when designing the run-time support software for RPC. There is a distinction between systems designed for an occasional isolated call, and those designed to handle large amounts of traffic between particular partners. In our case, the goal was to optimise run-time performance by doing as much work as possible at compilation time (type checking, for instance), and at program load time.

At load time, any necessary connections are established between tasks. The media to be used, and the remote addresses, are looked up at this time. This allows a system to be reconfigured without recompilation or even relinking the executable images. Load time is also the appropriate time for starting server tasks, and checking client authorisation.

Under the VAX/VMS ™ operating system, facilities exist for the automatic intialisation of software modules when they are loaded. In this case, any stub modules declare themselves to the run-time support at load time. The remote binding is then carried out quite invisibly to the programmer, who does not have to write any code to do it himself.

Although it is possible for a program to expressly relink itself to different modules while it is running, in general any remote call is made over an established communication path to an established server module, and can therefore execute with no additional delay. This method of working also allows us to use connection-oriented communications services (which happen to be available locally and are at a more advanced stage of standardisation than connectionless services) without a prohibitive overhead.

The call-time performance is then determined by parameter copying time and by operating system overheads. Without any assembly coding in these areas, a dummy call with n bytes of user data takes, across Ethernet, for example, about (5+0.08n) ms between VAX and an M68000 based system.

The importance of copying time has prompted the development (in progress) of 32 bit wide parallel interfaces [8] between VAX, VMEbus and FASTBUS which will perform byte and word reordering in hardware as data is transferred directly between user data areas.

Portability

The run-time support library consists of three parts. Its kernel code is is written in Pascal. It deals only with keeping track of server and client stubs, and does not contain any communications protocol. (This was chosen for its compatabilty with FORTRAN on several systems, its ability to handle pointers, and established use within the group. Ada and C were also candidates.) The same master source is used to generate code for all the target environments.

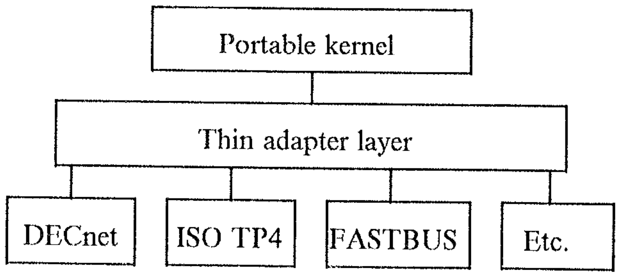

Fig. 2. Run-Time Support software structure

The second part is a thin "adapter" layer, which selects at run-time the communications facility to be used. It also interfaces the calling convention used by the kernel to the local conventions (operating system calls, etc) of the machine. This layer is rewritten when the code is ported onto a new machine.

The third part consists of the underlying communications services, which often existed already. On the microprocessors, these had to be written where they did not exist. Most of the communication facilities used are connection-oriented and reliable. In the case of raw Ethernet, however, a simple protocol is included in the adapter layer.

Extra Facilities

Although the primary goal was simply transparent procedure call between machines, experience has lead to a number of extra features, beyond the basic RPC function. These include detailed trace outputs about remote calls and their parameters, to help in the debugging of distributed applications.

When the confidence in the remote machine, the remote application software, or the communication protocol is low, users have asked for optional application level timeouts on remote operations. In this case, the maximum time allowed for a remote operation is specified when the stubs are compiled.

Also selectable at compilation time is the option of remote operations, once started, continuing concurrently with the calling program. This only applies to procedures which return no data. By adding a little pipelining, this option increases the performance of some control applications.

Multiple servers: Dynamic Binding

A basic distributed system consists of a number of software modules on different machines linked together much as individual program modules are bound together on the same machine. However, in some cases, a client may have the choice, at run-time, of many different, though congruent, server modules. For example, a control program may have access to identical servers on different parts of the apparatus. For these applications, facilities are provided for a client program to reconfigure itself to use different server modules, and to switch the binding rapidly between them. Obviously, in this case, the application program does become aware of the addresses of server modules.

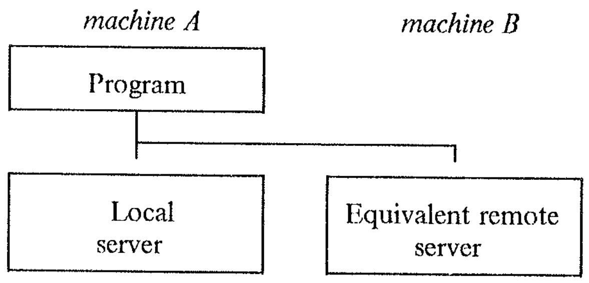

Fig. 3. Multiple instances of compatible servers

One example of this use is when an instance of the server exists locally. In this case, the procedure calls may be switched to be operate directly without any context switches, just like normal procedure calls. For example, an M68000 based system may have a local I/O system, but for remote access it may be reconfigured at run-time to use one on a VAX.

Intertask communication

Writing a program which will call remote procedures (the client) is straightforward. However, designing the server allows a number of alternatives. In the simplest case, a server task waits in a loop, and services calls made from other machines. In a more complex case, a separate subprocess may be started to handle each request.

A third case is one in which the server is a separate application program, which may itself be a client of other modules. This allows a powerful form of intertask communication, in which certain subroutines within a program are made accessible to other programs. It declares itself to be a server, and requests are then handled either asynchronously as they come in, or when specifically accepted. In this way, for instance, one may remotely control the parameters of a monitoring program as it is running, add objects to queues of requests which it is processing, and so on.

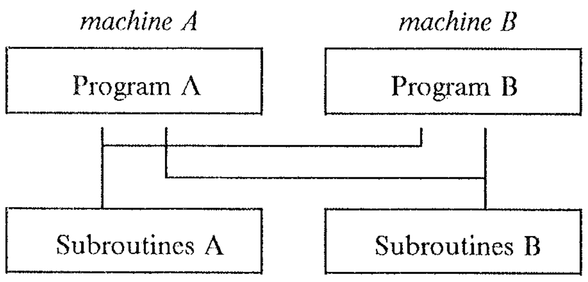

Fig. 4. Intertask communication by RPC

When one server task services more than one client, a server routine can find out by whom it was called. It may also establish a connection to the caller, and call back procedures in the caller. For example, one can call a remote monitoring program function "return statistics" which in turn may call back procedures to display or store information, and log errors on the caller's process. A call back may be done either during the original call, or later on (for instance, to signal completion of an asynchonous operation).

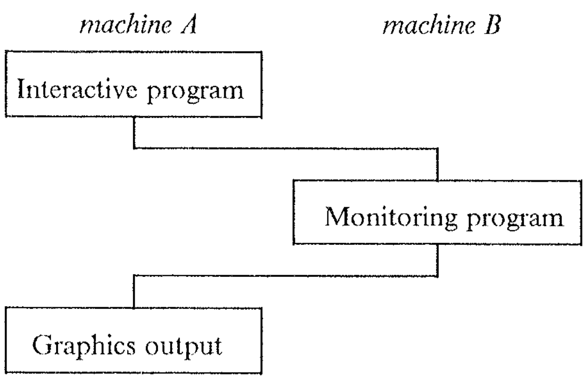

Fig. 5. Call back

Conclusions

Where we have used RPC, it has been possible to design and test application software before the final communication facilities have been available. The resulting code is largely independent of communications software, operating system and processor type.

Remote procedure call has proved a powerful and flexible tool in our environment, solving several of the problems inherent in distributed software engineering. Although various extensions have proved necessary, the application level procedure call has remained the representation of any remote operation.

Acknowledgements

Many people have contributed to the work outlined in this paper. A. Pastore (Turin University) wrote the RPC compiler; T.J. Adye (Rutherford Lab, UK) and D. Gosman (NIKHEF-H, Amsterdam), ported the run-time communications support to the M6809 and the M68000 under RMS68k respectively; I. Martinez (Santander University) and R.J. Jones (CERN) extended the communications media supported. I am also indebted to B. Segal and to D.M. Sendall, Online Group leader, for their support, interest and advice.

References

[1] B.J. Nelson, "Remote Procedure Call", XEROX PARC CSL-81-9, May 1981

[2] T.J. Berners-Lee CERN/DD, "RPC User Manual" available from the author.

[3] T.J. Berners-Lee, C. Parkman, Y. Perrin, J. Petersen, L. Tremblet, CERN, "The VALET-Plus, a VMEbus based microcomputer for Physics Applications", these proceedings

[4] "FASTBUS, A Modular High-Speed data Acquisition System for High-Energy Physics and Other Applications", U.S. NIM Committee, December 1983

[5] IEEE Standard 1014, "VMEbus: A Standard Specification for a Versatile Backplane Bus"

[6] "Courier: The Remote Procedure Call Protocol", XEROX Corporation, XSIS 038112, December 1981.

[7] CCITT "Red Book" Volume VIII, Recomendation X.409: "Message Handling Systems: Presentation Transfer Syntax and Notation"

[8] C.F.Parkman, CERN DD/OC, "The VMEbus/Host Interface I/O Register Module 'HyperVior' User's Manual" April 87, private communication;
K. Hollingworth, CERN/DD, "CERN Host Interface I/O port specification", private communication