Submitted to S-P&E

BC/ww 5.8.1983

Experience with remote procedure calls in a real-time control system

B.E. Carpenter and R. Cailliau
CERN
CH - 1211 Geneva 23, Switzerland

Summary

This paper decscribes several years' practical experience with remote procedure calls as a primary tool for application software in a large and complex distributed real-time process control system. We motivate the use of remote procedure calls as an effective technique for the use of local area networks, describe our implementation, and discuss its advantages and disadvantages.

Key words : remote procedure call, local area network, distributed computing, process control.

Introduction

In 1976, it was decided to implement a distributed real-time computer control system for the 28 GeV Proton Synchrotron accelerator, and several associated devices, at CERN, the European Laboratory for Particle Physics, in Geneva [1,2]. Upon evaluation, this project proved to require a local area network containing about twenty 16-bit minicomputers, in addition to microprocessors, and to involve at least 80 person-years of applications software. The ease with which application programmers could exploit the network therefore became a key issue in the design of system software.

An important characteristic of the system is the fact that certain computers - those controlling particular pieces of synchrotron hardware - are specialised, whereas others - those controlling the operators' consoles - are general-purpose and identical. Application programs must embed and reflect this structure.

Conventional local networks offer essentially only two facilities to real-time application programmers: remote file operations, and task-to-task communication (message passing). In many networks, remote file operations are limited to the complete transfer of a file between computers, not allowing access to individual records. Consider the options open to a programmer required to implement one or more programs to:

1) interact with an operator through any one of several general-purpose consoles (each with its own computer)

and 2) throughout the interaction, make calls to specialised device drivers (controlling, say, power supplies for large electromagnets) in several specific specialised computers

and 3) throughout the interaction, give immediate feedback to the operator.

It should be noted that with a few substitutions, this requirement describes a very wide variety of distributed computing problems, for example distributed data bases or distributed mail systems.

The programmer, then, conventionally has only two tools with which to satisfy the requirements: remote files and task-to-task messages. Typical implementations are:

1) implement a task in each computer involved; design task-to-task communications sequences between the interactive task and each of the other tasks (which in turn communicate with device drivers)

or 2) use remote file access to modify and read back data in each specialised computer; ensure that tasks run regularly in each computer to move data between the files concerned and the device drivers

or 3) agree with all past and future programmers about details of task-to-task messages, and collectively write very general tasks for each computer (i.e. subsume the requirements of each individual application into a general scheme).

These methods are complex, cumbersome and inflexible. Except in the first one, the work of various programmers is closely coupled, a fundamental error in any large system involving, say, 50 programmers over the years.

Remote procedure call

Remote procedure calls (RPC) have been discussed recently in the literature largely in the context of distributed operating systems [3,4,5,6,12]. The basic principle of an RPC is that a procedure call (with its parameters) is intercepted and instead of being executed in the same computer, it is transmitted to another computer for execution. Meanwhile, the user program is blocked at the point of call. When the procedure ends, in the remote computer, the return (with its parameters) is also intercepted and transmitted back to the calling computer. At this time, the user program is unblocked and continues as if the call had been executed locally.

This process, in an ideal situation, is transparent to the user program (except for a real-time delay). Indeed, in the "Newcastle connection"[3], total transparency has been achieved by adroit use of the UNIX naming scheme, but on the other hand only operating system calls are handled by RPC.

In the implementation described here, where application programs must control real hardware which has a given geographical location, an RPC is transparent to the application programmer except that the remote computer must be identified explicitly. This is supported by the syntax of P+, our primary programming language for applications in the control system [7]. Thus, to call a device driver for an electromagnet power supply one statement is required:

```
CPS & POW(V,WFLAG,N,CCV,3,STATUS);
```

Executed in any computer in the network, this causes a call to procedure POW in the computer whose identifier is CPS. The details of the parameters of POW are irrelevant; they are shown in order to make the example completely realistic.

It is clear that the design and implementation of the distributed interactive application described above is dramatically simplified by the availability of an RPC facility. The programmer writes a single task which interacts with the operator and calls device drivers as it needs to. At run time, execution switches automatically between computers as required. Application programs are conceptually simplified, and largely decoupled from one another.

Implementation

An RPC facility has three essential phases:

1) provision of an RPC protocol

2) provision of a convenient interface to each programming language in use

3) provision of an initial set of remote procedures.

We now describe our implementation of each of these phases.

RPC protocol

For managerial reasons, and because of performance constraints, our system uses a local-area network developed during 1974-75 for control of another CERN accelerator, the 400 GeV Super Proton Synchrotron [8]. It is a special-purpose packet-switched network with a star topology, giving extremely high performance (end-to-end packet transmission time is 5 to 7 msec). On the other hand, its high-level protocols are essentially limited to file transfer and remote execution of interpretive code; task-to-task communication is not provided.

The remote execution protocol requires some comment. The network was originally developed for control of a relatively slow device - the 400 GeV synchrotron is pulsed at a rate of once about every ten seconds. For this reason, a pure source code interpreter could be used without crippling the performance of application programs. Like the network, the interpretive language, NODAL [9], was developed specially for CERN. The programmer requests remote execution of part of his program by use of specialised syntax described in the references [8,9], and then this part of his source program is transmitted, in ASCII code, to a server in the remote computer. This server is itself a copy of the interpreter which receives and interpretively executes the transmitted source code. Specialised syntax is also provided to remit results back to the calling program. The network provides an access method for both the calling interpreter and the server to support these transactions. Thus an application programmer wishing to call a power-supply driver remotely will write, for example,

```
1.1    EXEC(CPS)10 V WFLAG N CCV STATUS
1.2    WAIT(CPS)
       ...

10.1   POW(V,WFLAG,N,CCV,3,STATUS)
10.2   REMIT STATUS
```

(Note: some syntactic details have been omitted or simplified in this example to clarify the exposition. Line 1.1 causes lines 10.1 and 10.2 to be transmitted, with several variables, for remote interpretation.)

The relative complexity and proneness to programming errors of this sequence should be contrasted with the simplicity of the preceding single statement in P+, which achieves the same effect.

This interpretive remote execution protocol, available to us as part of the network, was used as the basis for our RPC protocol. An access routine named REM was coded in low-level language. REM accepts a run-time call descriptor, encodes the descriptor in interpretive code, and uses the network access method. The intepretive code is essentially identical to the two lines above numbered 10.1 and 10.2; REM directly simulates the network calls made by lines 1.1 and 1.2.

The network access method, when used by REM, transmits a datagram which contains the necessary interpretive code in ASCII, plus binary values of the parameters. After interpretation by the server - consisting of a single procedure call - the result parameters (if any) are returned as another datagram to REM, which in turn passes them back to the calling program. The remote server is unaware that the call was generated by REM: it simply receives and interprets a small piece of source code (plus variables).

Thus, in the context of a network already supporting remote interpretation, implementation of an RPC protocol as such was reduced to about 400 lines of assembly programming (excluding comments). The time overhead for a null RPC is about 20 msec, of which some 15 msec are spent in the network hardware and software, and the remainder in the interpreter (the execution time of the REM routine itself is negligible). An additional 4 msec of overhead is caused by each actual parameter.

Interfaces to programming languages

For practical reasons, application programs in our system are coded in one of three compiled languages, not considering the interpreter which is too slow for general use, the pulse time of the 28 GeV synchrotron being about one second. The three languages are P+ (see[7]), PASCAL and NORD-PL, the computer manufacturer's machine-oriented language [10]. The RPC access routine, REM, requires a run-time descriptor which may be described in extended BNF by:

<descriptor>::=<procedure name><computer number>
{<tag>[<size>]<reference>} <end tag>

where <procedure name> is the name, in ASCII, of the remote procedure

<computer number> is the logical number of the remote computer

<tag> is encoded to indicate

- parameter type (word, real, string, word array, real array)
- parameter mode (read-only, read-write, write-only)

<size> indicates the size of array parameters

<reference> is the address of the actual parameter

<end tag> indicates the end of the descriptor.

[ ] indicates an optional item

{ } indicates repetition zero or more times

Note that abstract data types are mapped onto one of the parameter types mentioned above, which are the only ones directly supported by the interpreter.

For low-level application programs in NORD-PL, the programmer must provide this descriptor explicitly; this is simplified by pre-declared mnemonics and macros. In Pascal, a local compiler extension allows the programmer to write, for example,

```
FLAG:=WFLAG;P:=CCV;L:=3;
REMOTE('POW',CPS,RO V,RO FLAG, RO N, RO P, RO L, WO STATUS);
```

Only variables are allowed as the actual parameters, whence the necessity for several preliminary assignments. RO and WO indicate read-only and write-only respectively.

In P+, the RPC reduces to the single statement given above :

```
CPS & POW(V,WFLAG,N,CCV,3,STATUS),
```

The extended Pascal compiler, and the P+ compiler, both produce code to generate a descriptor in the above format and a call to REM. The cost of these two compiler features was one or two person-months each.

Provision of remote procedures

The convenience and usefulness of an RPC protocol obviously depends on the provision of a full set of remote procedures to be accessed by RPC. In our case, these procedures fall into four clases:

- general system routines (e.g. start a task, store in a mailbox)
- device driver routines, known as 'equipment modules', (e.g. POW as mentioned above)
- routines for specific groups of applications programs.
- routines which are an integral part of a single distributed program

Such routines are written to conform to a set of interface rules for compatibility between interpretive and compiled calls, and may be coded in low-level language or in P+. The principle restrictions imposed on them (due to the compatibility rules and to the network) are :

- certain esoteric parameter types impossible (e.g. variants)
- not more than 8 actual parameters whose combined size is somewhat less than 1 Kbyte
- procedure name unambiguous in first 6 characters
- execution time limited to 2 elapsed seconds.

Practical Experience

Some 400 remote procedures have been implemented in our system during the period 1979-1983, essentially all of which are in frequent use. Most of these routines have been coded by part-time or full-time application programmers, rather than by system specialists.

Some 150 compiled application programs (i.e. not counting interpreter code) have been implemented during the same period, all by application programmers, accelerator scientists, or technicians; they all rely on RPCs for use of the network.

Although precise statistics are not available, approximate measurements show that on average, round the clock and throughout the year, at least 5 RPCs are executed per second somewhere in the network, representing a total of about 5×10^8 remote procedure calls since the network went on-line in October 1980.

These figures speak for themselves: RPC has become one of the most fundamental tools for our application programmers, and probably the most cost-effective tool in terms of implementation effort. Any programmer may provide a general facility by coding it as a remote procedure. Any programmer may use such a procedure, in a program running on any computer, without particular knowledge of the network and without coding multiple tasks. In cases where multiple tasks are needed for other reasons, they typically communicate via mailbox routines called either locally or remotely: the design of the tasks is hardly affected by their distribution accross the network.

Clearly, the RPC facility has some disadvantages in practice. Principally, these are due to the restrictions mentioned above - restriction on parameter types, numbers and size; restricted length of names; and above all limited execution time. The latter is due to timeouts inherent in the network protocols[8], which lack end-to-end flow control and therefore impose timeouts to avoid emptying the buffer pools. The direct consequence of this problem is that network timeouts

must be treated as normal, expected events by application programmers and remote calls tend to be followed in all programs by statements such as

```
IF  IOERROR=TIMEOUT  THEN...
```

A disadvantage of our present implementation is that if a remote procedure generates a fatal run-time error during debugging, the RPC server will be aborted: no further RPCs may then be executed in the computer concerned, and fatal network protocol errors may also occur. It is undesirable to automatically restart the server, because this leads to the loss of debugging information.

Finally, one or two applications programs with special requirements are constrained by the fact that the calling program is blocked for at least 20 msec during RPC execution. It is clear that the Ada[11] rendezvous mechanism, in a distributed implementation, overcomes this constraint, but is probably more expensive and complex to implement than RPC.

Conclusions

Our four years' experience with remote procedure calls as a tool for a large team of real-time application programmers has been extremely positive. We intend to continue the use of RPC and to extend its domain of application to include many microprocessors. Apart from improvements in the underlying local area network, we intend to overcome some of the disadvantages of the preset implementation by introducing a dedicated RPC server capable of more efficient and less restricted interpretation of RPC datagrams.

Acknowledgements

We would like to thank our many colleagues, and in particular our project leader B. Kuiper, for help, encouragement and useful criticism.

References

[1]) B. Kuiper et al, "The improvement project for the CPS controls", IEEE Trans. Nucl. Sci. NS-26 3272 (1979).

[2]) B. Carpenter, A. Daneels and F. Perriollat, "The planned replacement of a functioning control system", IEE Conf. "Trends in on-line computer control systems", 1979.

[3]) D.R. Brownbridge, L.F. Marshall and B. Randell, "The Newcastle Connection", Software Practice and Experience **12** 1147-162 (1982).

[4]) F. Panzieri and S.K. Shrivastava, "Reliable remote calls for distributed UNIX", Proc. 2nd Symposium on reliability in distributed software and database systems, July 1982.

[5]) G.R. Andrews and F.B. Schneider "Concepts and Notations for Concurrent Programming", Computing Surveys **15** 3-43 (1983).

[6]) "Courier, the remote procedure call protocol", XSIS 038112, Xerox Corporation, 1981.

[7]) R. Cailliau, B. Carpenter, I. Killner, "Early experience with the programming language P+", ACM International Computing Symposium, Nürnberg, March 1983.

[8]) J. Altaber, "Real-time network for the control of a very large machine", IEEE Conf. "Computer Network trends and applications", 1976.

[9]) M.C. Crowley-Milling & G. Shering, "The Nodal system for the SPS", CERN 78-07, Geneva, 1978.

[10]) "NORD-PL User's Guide", ND-60.047.03, Norsk-Data A.S., Oslo, 1976.

[11]) J.D.A. Ichbiah et al, "Reference Manual for the Ada Programming Language", ANSI MIL-STD 1815A, Castle House Publications, London, 1983.

[12]) B.J. Nelson, "Remote Procedure Call", Ph.D Thesis, CSL-81-9, XEROX Corporation, 1981.