EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH
CERN - SPS DIVISION

CERN LIBRARIES, GENEVA

CERN SPS/89-45 (ACC)

CM-P00061479

Tools for Remote Computing in Accelerator Control

Pal S. ANDERSSSEN and Veronique FRAMMERY
CERN, CH-1211 Geneva 23, Switzerland

Rainer WILCKE
ESRF, F-38043 Grenoble Cedex, France

Abstract

In modern accelerator control systems the intelligence of the equipment is distributed in the geographical and the logical sense. Control processes for a large variety of tasks reside in both the equipment and the control computers. Hence successful operation hinges on the availability and reliability of the communication infrastructure. The computers are interconnected by a communication system and use remote procedure calls and message passing for information exchange. These communication mechanisms need a well-defined convention, i.e. a protocol. They also require flexibility in both setup and changes of the protocol specification. The Network Compiler is a tool which provides the programmer with a means to establish such a protocol for his application. Input to the Network Compiler is a single Interface Description File provided by the programmer. This file is written according to a grammar, and completely specifies the inter-process communication interfaces. Passed through the Network Compiler, the Interface Description File automatically produces the additional source code needed for the protocol. Hence the programmer does not have to be concerned about the details of the communication calls. Any further additions and modifications are much easier, because all the information about the interface is kept in a single file.

Presented at the International Conference on Accelerator
and Large Experimental Physics Control Systems,
Vancouver, British Columbia, Canada,
October 30 - November 3, 1989.

Geneva, November 1989.

1. Introduction

Control systems for large particle accelerators constructed today are almost exclusively based on distributed computing systems. As computer networks and cheap computers became available during the last two decades, designers turned to distributed computing as a cost effective solution to the controls problem for large plants. The advent of the minicomputer, followed by the microcomputer, has been well matched by the development of Local Area Networks (LAN) to promote this trend. Today's accelerator control systems illustrate the evolution with the construction at CERN of the SPS control system in the early seventies [1] and the LEP control system [2] recently brought into operation, as well as the design for the ESRF [3].

Implementation has also become simpler. Early systems were based on proprietary hardware and software technology, which locked the evolution of the system into a particular manufacturer's product line. Only a fortunate few had the necessary resources and courage to break out.

The situation is different today. Industrially accepted standards are established for LAN technology, and compliant equipment is available in a competitive market. At the same time there is a convergence towards a single UNIX operating system, offered by all major computer manufacturers. It is indeed possible to interconnect equipment from different sources, and to make them talk together.

With these tools it has become possible to not only use and share distributed units, but also to implement distributed control systems. In such a system [4] it is hidden from the user that his software may be split up for execution between several processors, as all the handling of the necessary communication is done by the control system software itself.

There has been an extensive treatment of the advantages and the drawbacks of distributed systems in the literature (see e.g. [5]). We will not repeat this discussion here, for the simple reason that for large accelerator control systems a distributed approach is unavoidable anyway: the size of the accelerator leads for a centralized system to response times that are unacceptable. In addition, today's "off-the-shelf" subsystems often already provide their own local computerized control system and require computer connectivity to handle them. Therefore communication between processes on distributed control units is needed. It is then desirable to keep the communication interface as transparent to the user as possible, as he should concentrate on the problems of the application and not worry about details of the communication. This transparency can be achieved by the use of Remote Procedure Calls (RPC) [6 - 9].

2. Principles of Remote Procedure Calls

RPC are based on the client-server principle of interprocess communication: a client is any application process that wants something done (a service) by another process (the server). In the case of RPC, the client issues its service request in the form of a normal procedure call and also receives its results when the call returns, but the service procedure is executed remotely.

The underlying mechanism is shown in figure 1. The client, when issuing the call, actually calls a local routine ("stub routine"), which packs the arguments into a message and hands them over to the transport layer of the client. They are then sent over the network to the server, received and unpacked. The server calls the service routine in a normal procedure call with the received arguments, executes it, gets back the results, packs and sends them to the client, where they are received, unpacked and handed to the client application at return time.

Fig. 1 The Remote Procedure Call mechanism: control- and data-flow beyond the application-RPC interface.

In the ideal case, the user would write only the client application and the server routine, leaving everything else to the stub and transport routines, which are provided by the control system. Unfortunately, this would require that all parameters of the communication are completely fixed in advance, including the server address. In addition, the control system would have to provide a large library of stub routines, many of which might never be needed on a particular computer.

This scheme is clearly too rigid and cumbersome. The user should have the freedom to specify the communication parameters, and the stub routines should only be installed where needed. However, if the user were to write the stub routines himself, he would have to dive far too deep into the details of the communication calls.

3. Implementation of the Remote Procedure Calls

The method chosen for this implementation of the RPC mechanism was therefore the development of a code-generating program - the Network Compiler (NC) - that takes a formal description of the client-server interface and produces the necessary stub files. The generated code is C. The user compiles these stub files as well as his application program, and links them with the library containing the transport routines to produce the executable client and server modules.

This formal description of the client-server interface is contained in a separate file, the Interface Description File (IDF). A great variety of interface attributes can be specified, thus giving the desired flexibility. The average user, however, can keep his IDF very simple, as the NC provides default values for most attributes.

The features that can be specified include:

- 1) Connection attributes: connection length timeouts, host name and port number of the server, protection against unauthorized use.
- 2) Transport protocol: as default, the RPC uses a custom-developed transport protocol to pass arguments and return values between client and server. Other options are TCP/IP [10] and System-V-Messages.
- 3) Data types: there are predefined data types (similar to those in C). Additionally, the NC provides constructs that allow the user to define his own data types. All data translation between client and server is handled by the RPC software.
- 4) Runtime arguments: it is possible to assign fixed values for all attributes of the client-server interface in the IDF. These values then become part of the stub code, and if the interface changes (e.g. the server is moved to another host), then one must also change the IDF and regenerate the stub files and executables. To allow more flexibility, attribute values can be assigned at runtime as arguments of a service activation call, and the user declares in the IDF which attributes will be defined in this way.
- 5) Client-server communication mechanism: the RPC-mechanism described above is a "blocking" mechanism, where the client waits until the answer from the server returns. Other mechanisms are available and sometimes preferable:
 - asynchronous calls: the client sends the argument with a "send" call, continues execution and collects the return values later with a separate "receive" call;
 - interrupt calls: the client only sends values and continues execution, it never expects nor gets return values;
 - message passing: client and server can both send and request data, as often as desired and in any sequence. The flow of information in this communication scheme depends entirely on the application programs and must be specified by the user himself, but the RPC still handles data conversion and transport mechanism.

The choice of possible options has been influenced both by historical requirements and general remote processing considerations. The RPC implementation with an IDF and built-in default values makes it easy to introduce new features without disturbing existing applications; moreover, by changing the default values a whole site could actually be adapted to new environmental conditions (e.g. installation of a different transport layer) without the users noticing much of it.

4. Usage of the RPC Facility

To generate an executable client and/or server, the application programmer must:

- 1) design and write the application programs;
- 2) add the service statements;
- 3) write the Interface Description File;
- 4) compile the IDF with the NC to generate the stub routines;
- 5) compile (with C-compiler) and link application programs and stub routines into executable modules.

These steps will now be further detailed. However, it is impossible to give a full description in the frame of this paper; for this the NC/RPC User Guide should be consulted [11]. We will instead try to explain the principles of usage, restricting ourselves to the "classical" RPC, and talking of one procedure and one service only. A service can comprise many procedures and a client use several services, moreover a server can itself use another service and thus become client of this server; but the same principles of usage will be applicable.

Design and coding of the application programs for a remote procedure call can normally be done in exactly the same way as for local procedure calls - this transparency was after all the goal of the RPC mechanism. There are peculiar situations in which a valid local procedure call will fail when executed remotely (see e.g. [12]), but they rarely affect the normal user.

The service statements are only few.

For the client:

- activate the service;
- use the service;
- free the service.

For the server:

- activate the service;
- wait for client request.

For the client, these statements can be contained anywhere in the program, as long as the above sequence is preserved during execution. For the server, they must be in the main routine. Activation of the service is done by calling the service activation routine defined in the IDF. Use of the service by the client is done by the procedure call itself. Once the service is activated, the procedure can be called any number of times. After the last call, the client frees the service to clean up and close the client-server connection and release the corresponding system resources. For the server, waiting is done by calling a special (RPC-provided) wait routine, which will wait for a client request, execute the service routine, and wait again for the next request. The server main routine is usually almost dummy and contains just the two service statements.

To write the Interface Description File, a special description language is used. It is based on the MESA language from XEROX [13] and its grammar is fully specified in [11], using the Bachus- Naur-Format. The layout of the file is shown in fig. 2. The most important sections are

- the server section, which specifies the server's attributes, in particular the name and arguments of the service activation routine;
- the client section, which does the same for the client;
- the procedure definition, which specifies name and type of the remote routine, its arguments and its return values.

```

/* IDF header, defines the service name */

task =

/* IDF interface information */

    statistics = begin
        /* statistics information */
    end
    server = begin
        /* server specific definitions */
    end
    client = begin
        /* client specific definitions */
    end
    directory = begin
        /* name server definitions */
    end
    protocol = begin
        /* protocol specific definitions */
    end

/* IDF data type and procedure definitions */

begin
    /* data type definitions */
    /* procedure definitions */
end

```

Fig. 2 Layout of the IDF for the service "task". "statistics", "server", etc. are keywords and have to be given as shown, as well as "begin", "end" and the "=".

The specification of the client- and server-attributes is illustrated in the following example:

```

client = begin
    connection = import[host,timeout absolute];
    timeout relative = 120;
end
server = begin
    connection = export;
    port = 1530;
end

```

These attributes describe a server, which will listen at port 1530, and a client with a relative timeout of 120 (sec). The "connection" attribute tells that the client will specify the host to which it wants to connect (i.e. the server computer) and its own absolute timeout at service activation time, whereas the server will specify no attributes at service activation time. The names of the service activation routines are combined from the "connection" attribute and the service name; if the name of the service were "task", then the client would activate the service by a call like

```
sid = import_task("neptun",120);
```

and the server by

```
sid = export_task();
```

The only client- and server-attributes that must always be given are "connection", "host" and "port". For all other attributes default values are provided.

The procedure definition is illustrated in the following example:

```
begin
    treat_text:
    procedure[input_text: string]
    returns[output_text: string];
end
```

This defines the routine "treat_text" that takes one argument of type "string" and also returns a string. The client would call this routine as e.g. in the following fragments of a C-program:

```
char text_buffer[100];
char *return_buffer_pointer;
/* some client code, including the "import_task" call */
return_buffer_pointer = treat_text(text_buffer);
```

and the service routine would look like:

```
char *treat_text(in_text)
char in_text[ ];
{
    static char out_text[100];
    /* server procedure code */
    return(out_text);
}
```

Note that the names "input_text" and "output_text", which are declared in the IDF procedure definition, need not be used by the application programs (they are used internally in the stub- files).

When compiled by the NC, the IDF for the above service "task" will generate four C-code files:

task_client.c	client side stub files
task_server.c	server side stub files
task_defs.h	header file with declarations for client and server
task_support.c	data translation routines for client and server

The file task_defs.h must be included by the client- and server-application routines. The file task_client.c is compiled and linked with the client application routines to produce the executable client; in the same way task_server.c together with the server main and the service routine yields the server. The file task_support.c will be included automatically.

5. Present Utilization and Performance

Work on the NC/RPC package started at CERN in 1987 [14]. It is a key component in the control system for the LEP accelerator, where it is used for general services like personnel access, water supply and vacuum control. It also serves to acquire measurements from the beam instrumentation, and to monitor and control the power converters and the high frequency power plant. In addition, the NC/RPC is used in the SPS control system. For this purpose, special gateways have been provided to allow remote procedure calls to the already existing SPS control system. The package will also be used in the control systems of the ESRF and the Sincrotrone Elettra in Trieste.

To facilitate this widespread utilization, the NC and the transport protocol routines have been written in C and are thus portable between different computer hardware and even operating systems. Experience has shown, however, that for each new system some adaptations are necessary. To avoid divergence into many incompatible versions, these adaptations are all implemented into the same source code, using C-preprocessor statements. At present, the RPC package has been ported to

- Intel 80286 and 80386 processors with XENIX;
- IBM-RT computers with AIX;
- HP 9000 series 300 and series 800 computers with HP-UX;
- APOLLO DN 3000 series computers with DOMAIN;
- Motorola 680X0 processors with OS-9;
- DEC-VAX computers with VMS.

Stating performances for a software package like the NC/RPC is difficult, as it uses and therefore depends on a whole range of system parameters, both hardware and software, over which the RPC routines have no direct control. Any number measured today might be obsolete tomorrow with e.g. a new release of the operating system or the availability of more efficient network interfacing. Nevertheless, the application programmer must be able to judge whether his requirements can be satisfied when using the RPC. For this, three values are important:

- connect time: open connection, issue one remote procedure call, close connection;
- call time: issue one remote procedure call on an already open connection;
- throughput: maximum transfer rate for a big data buffer (40 000 bytes).

The performance achieved on a small range of available computers are shown in table 1. Because the measurements were made in an operational installation, the variations were as high as 20%.

Computer	Operating System	Transport Protocol	M e a s u r e m e n t		
			Connect Time [ms]	Call Time [ms]	Throughput [Bytes/s]
Apollo DN 3500	DOMAIN IX	RUDP	85	48	25 000
Apollo DN 4500	DOMAIN IX	RUDP	45	23	37 000
Olivetti M380C	SCO XENIX	RUDP	79	45	25 000
Apollo DN 3500	DOMAIN IX	TCP/IP	—	15	85 000
Apollo DN 4500	DOMAIN IX	TCP/IP	—	11	100 000

Table 1: Performance measurements for the RPC. The definition of the measured values is given in the text. Statistical variations on these measurements were as high as 20%. RUDP is our custom developed protocol. Connect time for TCP/IP has not yet been measured.

6. Plans for Future Development

Although the NC/RPC package in its present form has proven to be a valuable tool, there are features which we would like to incorporate:

- dynamic address generation and loading of servers;
- automatic creation of a server main program from the IDF;
- interfaces to other languages.

Dynamic address generation and loading of servers: to connect to a server, the application program must specify the server address. This is inconvenient, as it disturbs the desired transparency of the RPC concept. Moreover, it is also undesirable for the layout of a control system as it prevents dynamic relocation of servers in the case of a breakdown of a server computer. Both problems could be solved by the implementation of a central directory server. Every server would at startup time register itself with the directory server, giving its service name and address. An application program then only needs to know the service name and would inquire for the address at the directory server. A second feature, following similar thoughts, is to implement on every computer a load server, which is able to start a server. If a particular server ceases running for any reason, a client trying to contact it gets a specific error code. It could then request the load server on the server host to restart this server. The load server could also be used for some rarely used servers, which would then not run all the time, consuming system resources, but only be started when needed and stopped again afterwards.

Automatic creation of a server main program from the IDF: as already mentioned, the server main program is almost dummy; in most cases the application programmers simply copy an existing one and change only the service name. This main program could easily be generated by the NC along with the server stubs.

Interfaces to other languages: in the original proposal, an interface to FORTRAN had been foreseen. However, no request for this has been made. Our present intention is therefore to not invest any effort into FORTRAN, but rather consider other languages; among them ADA, which is increasingly used and might soon show up in accelerator control systems.

Acknowledgements

Several people have contributed to this work. I.B. Kristiansen designed and wrote the initial version of the Network Compiler, the stub generator as well as all the support libraries. K. Kostro wrote parts of the interface to the transport layer. P. Defert provided additional functionality to the Network Compiler. P. van der Stok helped making the package more robust. G. Turchányi wrote the TCP/IP version of the run-time libraries.

References

- [1] M.C. Crowley-Milling, The Design of the Control System for the SPS, CERN 75-20 (1975)
- [2] P.G. Innocenti, The LEP Control System, these Proceedings (1989)
- [3] W.D. Klotz and C. Hervé, The Conceptual Design of the ESRF Control System, in: S. Tazzari (ed.), European Particle Accelerator Conference (Rome) vol. 2, p. 1197, World Scientific Publishing Co., Singapore (1989)
- [4] This use of the term "Distributed System" follows the definitions in [5] and [12].
- [5] J. Alves Marques, The Computational Models of Some Distributed Operating Systems, in: C. Verkerk (ed.), Proceedings 1987 CERN School of Computing, p. 110, CERN, Geneva (1988)
- [6] A. Birrel and B. Nelson, ACM Transactions on Computer Systems 2 (1984), 39
- [7] Courier: The Remote Procedure Call Protocol, Xerox System Integration Standard 038112, Xerox, Stamford (1981)
- [8] Remote Procedure Call Protocol Specification, SUN Microsystems Inc., Mountain View (1984)
- [9] Network Computing System (NCS) Reference, Order No. 010200, Apollo Computer, Chelmsford
- [10] MIL-STD 1778, Transmission Control Protocol, in: DDN Protocol Handbook vol. 1, DDN Network Information Center, SRI International, Menlo Park (1985)
- [11] P.S. Anderssen, V. Frammery, and G. Morpurgo, User Guide to the Network Compiler Remote Procedure Call (NC/RPC), CERN LEP Controls Note 97, unpublished (1989)
- [12] A.S. Tanenbaum, Computer Networks, Prentice Hall Inc., Englewood Cliffs (1988)
- [13] J.G. Mitchell, W. Maybury, and R. Sweet, Mesa Language Manual (Version 5.0), Technical Report CSL-79-3, Xerox Palo Alto Research Center (1979)
- [14] I.B. Kristiansen, System Design Draft for the SPS/LEP Controls Network Services, unpublished (1987)

Trademarks:

UNIX is a trademark of AT&T Bell Laboratories.
 DEC, VAX and VMS are trademarks of Digital Equipment Corporation.
 AIX is a trademark of International Business Machines Corporation.
 XENIX is a trademark of Microsoft Inc.
 OS-9 is a trademark of Microware Systems Corporation and Motorola Inc.