NoWait–RPC: Extending ONC RPC to a fully compatible Message Passing System*

Thomas Hopfner Franz Fischer Georg Färber
Laboratory for Process Control and Real–Time Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
{Thomas.Hopfner,Franz.Fischer,Georg.Faerber}@lpr.e-technik.tu-muenchen.de

Abstract

Locally and functionally distributed applications realized on different system architectures demand a universal, portable and intuitive software utility for interprocess communication. Remote Procedure Calling (RPC) fulfills these requirements but suffers unnecessarily strict synchronization and the danger of deadlocks with complex client/server relations. Using message passing to avoid the inherent problems of RPC, however, requires additional knowledge and sometimes a complete structural redesign. This paper presents ***NoWait–RPC****, an extended but fully compatible version of SUN's Open Network Computing Group's* ***ONC RPC****, which adds* ***message passing capabilities*** *to form an easy to use programming environment for robust system integration. NoWait–RPC was developed to resolve potentially arising deadlocks in an already RPC–based complex application. It is designed to be a* ***plug-and-play substitute*** *for ONC RPC and consists of the library and the extended protocol compiler* ***nwrpcgen****. Additionally, applications using the asynchronous (non–blocking) features of NoWait–RPC may experience a major speedup compared to ONC RPC through pipelining calls to different servers. It has successfully been employed in a research programme dealing with the development of autonomous mobile robots.*

1 Introduction

Sharing resources using electronic networks is becoming a new paradigm in technological development. New concepts such as distributed user facilities and team–based development are rapidly evolving, since present systems tend to consist of a large number of modules. Especially in robotics applications these modules include sensor or motor control, knowledge management and model bases, sensor data preprocessing, but also more complex tasks like navigation, manipulation or exploration of unknown environments. They tend to have small interfaces and thus lend themselves very well to distributed development. The key problem, though, lies in the need for a robust and easy to use system integration utility which allows the distributed development of system modules and their integration even in case of a heterogeneous computing environment. Another demand is the support of some degree of parallelization to allow for different timing constraints of sensors and actors without a large communication overhead or the danger of deadlocks.

Most of these problems do not arise with *message passing* systems like *Parallel Virtual Machine (PVM)* [2], a free available software package for parallel processing in heterogeneous computer networks. But PVM is mainly aimed at parallelization by dividing programs into small tasks and not at building client/server–structures which are necessary for distributed robotics applications.

Another library offering message passing support is *MPI* (*Message Passing Interface*) [5]. The free–ware version MPICH depends on PVM or a similar product as communication layer and acts just as an interface. Since the MPI–standard is the result of trying to combine existing message passing libraries, the functionality offered is very high; on the other hand MPI became quite large, complex and therefore difficult to understand.

Remote Procedure Calling (RPC) hides many details of interprocess communication like connection establishment, synchronization or data representation from the application programmer and hence is very convenient to use: The client simply calls a (usually automatically generated) local stub procedure, which encodes the arguments, sends the request to the appropriate server process, waits for the server to execute the procedure and returns the result to the client after decoding the server's response. This enables even unexperi-

*The work presented in this paper is partially supported by the *Deutsche Forschungsgemeinschaft* as part of an interdisciplinary research project on "Information Processing in Autonomous Mobile Robots" (SFB 331).

enced programmers to rapidly familiarize themselves with RPC and implement distributed client/server applications. However the fundamental drawback of RPC is that — as is true with local procedure calls — the calling process is blocked until the reception of the result. Furthermore, complex scenarios where a server also acts as a client (in the following termed a "clientserver") are not easily realized due to the potentially arising deadlocks with complex and cyclic client/server relations. These limitations are especially true for ONC or Sun RPC [7], which was originally designed for client–server communication in the Sun NFS network file system and for this reason is available on a wide range of architectures and operating systems.

To overcome some of the problems caused by ONC RPC's strict synchronization, there are other RPC–packages. Beside commercial ones [1], which were not acceptable because one demand was to have open sources to be able to integrate RCP over a serial line, free available packages are either not very helpful (like *TI–RPC* [6] which uses threads and therefore the application has to be thread-safe too) or incompatible with ONC RPC (e.g. *DFN–RPC* [8]).

To avoid the redesign of already existing ONC RPC based applications, *NoWait–RPC* has been developed. As it is fully compatible to ONC RPC — i.e. NoWait–RPC–programs can interact with clients and servers linked with the standard ONC RPC library, — it preserves all its advantages, but additionally supports non–blocking remote procedure calls. That enables the development of complex client/server–structures without deadlocks and the parallelization of communication and computation to speed up the application. Existing knowledge about RPC and system structures based on RPC can be reused without problems.

The following section describes the main mechanisms and the programming interface of NoWait–RPC compared to ONC RPC. Section 3 illustrates the power of this package including performance comparisons, by describing exemplary robotics applications, which could not have been realized with ONC RPC, before the paper closes with conclusions.

2 NoWait–RPC extensions to ONC RPC

For a better understanding of the advantages of NoWait–RPC, at first the principles and limitations of ONC RPC are outlined (Section 2.1), followed by a description of the concepts and implementation of NoWait–RPC (Section 2.2).

2.1 ONC RPC

Using a RPC package simplifies the development of client/server applications significantly, because RPC resembles a local procedure call and hides interprocess communication details from the application programmer.

In the case of ONC RPC this is accomplished by the *RPC library*, and the automatic code generator *rpcgen*. The library provides the basic functions for establishing the connection to the server — ONC RPC uses the Internet protocols UDP/IP or TCP/IP for data transfer, — sending and receiving data and converting basic data types from the architecture dependent local representation into the machine independent XDR (*External Data Representation*) format and vice versa. This format takes into account e.g. endianness, floating point formats and alignment restrictions.

To implement a client/server application, the programmer first has to specify the server's interface, i.e. declare the exported procedures and data structures for their arguments and return values, using *RPCL* (RPC Language), a C–like language. Using this interface definition as input, *rpcgen* generates

1. the stub procedures for the client side,
2. the stub procedures for the server side,
3. procedures to convert the specified data structures from the internal representation into XDR format and
4. a skeleton for the exported procedures[2].

For the server side, the programmer simply has to fill the generated skeletons with the procedure bodies to implement the exported procedures and compile and link these with the server stubs, the XDR procedures and the RPC library. The client has to be completed with the calls to establish and shutdown the connection to the server (possibly running on another computing node) and linked with the library and the generated client stubs. These encode the arguments, send the request to the server, wait for the server to execute the procedure and return the result to the client after decoding the server's response.

Exactly this property causes the major disadvantage of RPC: Just like one has to wait for the execution of a normal procedure, a client process has to wait until the RPC is finished and can therefore be blocked for a substantial time. In complex systems this means that the application designer has to take care that no deadlock conditions arise if two processes communicate with each other (cyclic structure). Also the client process can not continue processing while being blocked.

2.2 NoWait–RPC

Avoiding the deadlock situations described above while keeping compatibility to existing applications was the major motivation for the development of NoWait–RPC. It should combine the ease of use of ONC RPC (especially *rpcgen*)

[1] e.g. *Netwise RPCTOOL*, *DCE–RPC* [3] or *NCS RPC* [4]

[2] This feature is not available with old versions of rpcgen.

with the capability of asynchronous, non–blocking communication. The additional flexibility should allow even scenarios which would have caused deadlocks before.

2.2.1 Internals

The basic concept is to split the former *clnt_call()* into two separate functions *clnt_send()* and *clnt_recv()* and to replace the several distinct waiting points in the original RPC library with one central waiting point *nw_work_select()*. In order to timely separate the application's calls from the actual network communication, RPC requests to be sent and the server's responses are buffered in FIFO queues on the client side. The application just adds data to or retrieves data out of the FIFO buffers when using clnt_send() and clnt_recv() (Figure 1).

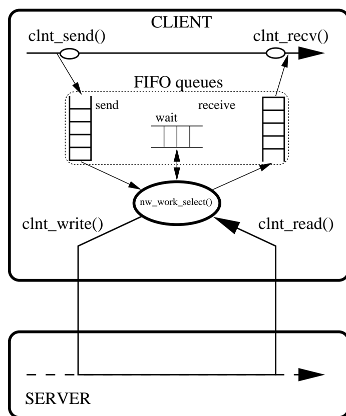

Figure 1. Buffering client calls

The actual network I/O is done each time nw_work_select() is called: In the case of a client RPC, requests from the send FIFO queues are sent to the respective servers (represented by their socket file descriptor) with clnt_write(), and available responses are read with clnt_read() and enqueued in the receive FIFO. On the server side, the requests read from the network are not buffered, nw_work_select() instead decodes the request and calls the respective service procedure to process the RPC and send back the result.

Because nw_work_select() dispatches any pending RPC request to the appropriate service procedure, calling this routine — e.g. while processing a time consuming RPC — results in a cooperative scheduling. That means, the processing of the current request is preempted or interrupted by another (Figure 2). This feature is important to bound the maximum response time of RPC servers to urgent requests. In a robotics application e.g., some exploration procedures with long execution times have to be preemptable by an emergency stop RPC, which obviously has to be executed "immediately". As indicated in Figure 3 clientserver

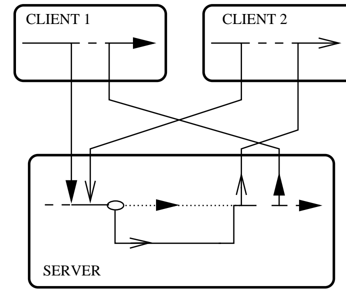

Figure 2. Using nw_work_select() in servers

applications can use this feature to parallelize RPCs. However, it is up to the programmer to provide a mechanism to avoid that server procedures never finish, because they are interrupted all the time or recursively call themselves.

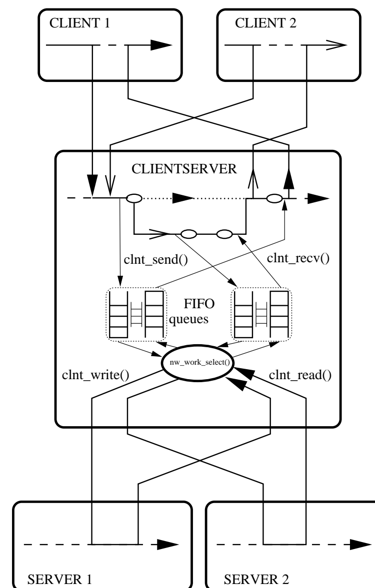

Figure 3. A complex clientserver using NoWait–RPC

Handling other input or output is significantly simplified with NoWait–RPC, as nw_work_select() allows the application programmer to specify arbitrary file descriptors to be checked for being ready to read from or write to or for exceptional conditions (cf. the select(2) system call). This allows e.g. a NoWait–RPC server to provide a graphical user interface based on a typical X Window System Toolkit library by specifying the X Server's file descriptor when calling nw_work_select() and afterwards call the toolkit's event handler if a X event is pending.

2.2.2 Migrating existing Applications

As mentioned above, NoWait–RPC is fully compatible to ONC RPC. That provides an easy migration path for RPC application programmers:

In a first step, the application's source code is left unchanged, but *nwrpcgen* is used to translate the RPCL interface definition into C code and the application is linked with *libnwrpc.a*.

The second step is to use the features of NoWait–RPC by replacing all calls of *clnt_call()* with *nwclnt_call()* which already resolves deadlock conditions by using the central waiting point described above.

Then in a third step the programmer may use the message passing functionality of NoWait–RPC by splitting *nwclnt_call()* in two functions *nwclnt_send()* and *nwclnt_recv()*, which makes it possible to parallelize calls to several different servers. While calling a server procedure the client process can continue to work and some time later get the answer of the RPC by calling the receiving function. This may speed up complex applications a lot as shown later on. The application designer can decide to use this feature only on demand.

2.2.3 Current Status

NoWait–RPC has been ported to 64 bit[3] DEC Alphas, SUNs (SunOS 4.1.3) and Intel–based Linux machines. Further ports should be easily possible as demonstrated with the original ONC RPC sources. The port has been successfully tested with existing applications according to the migration path outlined above.

3 Applications

Within the joint research project "Information Processing in Autonomous Mobile Systems" a group of laboratories of the *Technische Universität München* is working on mobile robots that are to plan and execute tasks in service or production environments autonomously[1]. The system modules were developed in parallel by the different groups and at first integrated using ONC RPC. With the increasing complexity of the systems, though, the limitations of ONC RPC became more and more apparent, which led to the development of NoWait–RPC. In the following, some applications are outlined which now have become possible.

3.1 Parallelized image processing

Figure 4 depicts a system structure for a mobile robot autonomously exploring its environment. The *Sequencer* coordinates the different sub-tasks by calling the corresponding server processes. For localization, an image is requested of the framegrabber server, then lines are extracted and matched to model lines provided by the model base, Using NoWait–RPC it was possible to realize parallelized processing. In this case the time for image grabbing and transfer could be used by another process to work on the last picture.

[3]This port is based on a DEC Alpha version of the Linux libc.

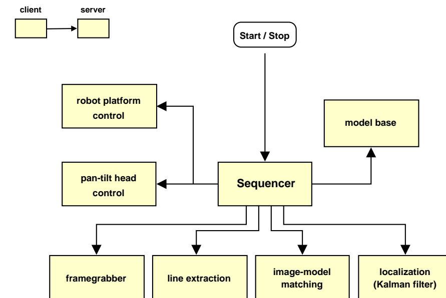

Figure 4. System structure for autonomous navigation

By prefetching the different calls with NoWait–RPC (Figure 5) frame rates of about 10 Hz were reached, which is a speed-up by a factor of 2 compared to the ONC RPC version.

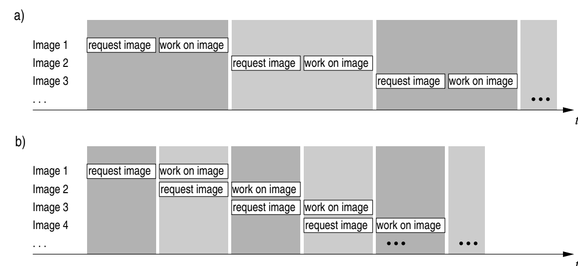

Figure 5. Image processing a) without and b) with asynchronous RPCs

3.2 Cyclic client/server–structures

In addition to speed up processes, asynchronous, non-blocking RPCs now make it possible to resolve cyclic client/server–structures. Consider two clientserver processes calling procedures from each other: The first clientserver process is blocked while waiting for the response of the second one and in case the second clientserver process has to call the first one to finish, the system is obviously deadlocked when using ONC RPC.

With NoWait–RPC, on the other hand, these structures present no problem, because the client is unblocked by the

Execution Times [ms]		Protocol / Argument Size [bytes] UDP/7	TCP/7	TCP/34000
local	ONC RPC	0.516	0.867	15.428
	NoWait–RPC	0.764	1.184	14.169
remote	ONC RPC	0.510	0.845	15.380
	NoWait–RPC	0.782	1.183	14.070

Table 1. Average execution times of a simple RPC

server's RPC, processes its request and then blocks again waiting for its own previous RPC to complete.

3.3 Responding to asynchronous events

The sequencer in Figure 4 should respond to an asynchronous start/stop call in some hundred Milliseconds. This was not possible before, because it was sometimes blocked while waiting for the answer of a previous RPC. With NoWait–RPC it is possible to respond to this asynchronous signal while waiting for an answer.

3.4 Performance

Table 1 compares the average execution times of a simple RPC (library function *clnt_call()*) for different transport protocols, argument sizes[4] and server locations.

The times were measured on two identically configured 100 MHz Intel Pentium PCs with 32 MByte main memory, running Linux 1.3. The machines were connected by an otherwise unloaded 10 MBit Ethernet using NE 2000 compatible Ethernet adapter cards.

In the case of an RPC to a server on the same machine, NoWait–RPC is about 50 % slower than ONC RPC for small amounts of data (FIFO management overhead), while it shows a slight better performance (3 %) than the original implementation for the large argument string due to memory allocation in bigger chunks. If the server on the remote node is used, network latency slows down RPC throughput for both versions. In this case NoWait–RPC is approximately 20 % and 3 % slower than ONC RPC for small and large arguments, respectively.

The performance costs of NoWait–RPC compared to ONC RPC are clearly outweighed if a process can actually engage in useful work after issuing the non–blocking RPC request and before receiving a reply. Furthermore the benefits of avoiding deadlocks and easier programmability of client/server code (e.g. compared to using callbacks) have proved to be a valuable consideration.

[4] Argument type "string", return type "int".

4 Conclusion

RPC is a user friendly way to develop and integrate distributed applications in heterogeneous environments. It is fast, already available on almost all architectures, there is a protocol compiler which generates the necessary C code for the network communication and data conversion out of an interface description and the source code is freely available. Unfortunately with complex and therefore maybe cyclic structures the danger of deadlocks arises.

NoWait–RPC extends the functionality of ONC RPC to resolve these deadlocks and to allow parallelization of RPC requests, while still being fully upward compatible, i.e. existing, unmodified ONC RPC servers and clients can remain untouched. This is done by using a central waiting point and adding an asynchronous, easy to use message passing capability. Nevertheless it is still up to the programmer to use these extensions with care.

NoWait–RPC is an easy and powerful tool for robust distributed development and integration of complex systems with the focus on the functionality of the systems and not on the communication problems. In such scenarios it is easier to use than more complex message passing libraries.

References

[1] C. Eberst, D. Burschka, A. Hauck, G. Magin, N. O. Stöffler, and G. Färber. A System Architecture Supporting Multiple Perception Tasks on an Autonomous Mobile Robot. pages 1–8, 1996.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. Also available on-line at http://www.netlib.org/pvm3/book/pvm-book.html.

[3] M. M. Kong. *DCE: An Environment for Secure Client/Server Computing*. Hewlett Packard Journal 12/95, S.6 ff, 1995.

[4] T. Lyons. *Network Computing System: Tutorial*. Prentice–Hall, 1991.

[5] MPI Forum. MPI: A Message Passing Interface Standard. *Int. Journal of Supercomputer Applications*, 8(3/4), 1994. Version 1.1 of this document is available on-line at http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html.

[6] ONC. *Solaris ONC, Design and Implementation of Transport–Independent RPC*. Sun Microsystems, Inc., 1991.

[7] Open Network Computing Group ONC. *Remote Procedure Programming Guide*. Sun Microsystems, Inc., 1988.

[8] R. Rabenseifner and H. D. Reimann. *Verteilte Anwendungen über Hochgeschwindigkeitsdatenkommunikation*. Rechenzentrum der Universität Stuttgart, 1992.