Will be published in "Kommunikation in Verteilten Systemen – KiVS '95". W. Kalfa, U. Huebner, K. Franke (Editors), Proceedings, Chemnitz-Zwickau, Feb. 22–24, 1995.

# The DFN Remote Procedure Call Tool for Parallel and Distributed Applications

Rolf Rabenseifner

Computer Center University of Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany, Tel. ++49 711 6855530, e-mail: rabenseifner@rus.uni-stuttgart.de

**Abstract.** DFN-RPC, a remote procedure call tool, was designed to distribute scientific applications accross workstations and compute servers. This document describes the methods in which the DFN-RPC tool supports parallel and distributed applications. Asynchronous RPC's are enhanced into parallel RPC's and combined with data pipes. The startup of processes required for a distributed environment is integrated. The features and possibilities of DFN-RPC's are compared with those of DCE, PVM, Cronus, Astra and Mercury.
**Key words:** Remote Procedure Call, Parallel applications, Distributed applications, Parallel RPC, DFN-RPC.

## 1 Introduction

Scientific applications are increasingly distributed between the workstation at the desk of the scientist and central resources. Because of this it is possible to realize optimum graphic user interfaces on the one hand and to bring to bear the power of central supercomputer, parallel computer and RISC-cluster resources on complex numerical problems on the other.

DFN-RPC provides a uniform platform for programming communications interfaces in an easily learnable form in order to allow the developers of distributed applications to concentrate on the technical and scientific responsibilities of their disciplines [12, 15, 21].

The DFN-RPC was designed at the University of Stuttgart Computer Center by order of the DFN Association[1]. For this purpose the participants were two pilot groups within the University of Stuttgart (IKE[2] and ISD[3]), and outside the Theoretical Astrophysics department of the University of Tübingen and the Alfred-Wegener Institute for Polar and Marine Research in Bremerhaven, enabling the integration of widely different application and network aspects during the designing and testing of the DFN-RPC. Speeds of 10 - 800 mbit/s over Local Area Network (LAN) Ethernet, FDDI and UltraNet were established. The Wide Area Network (WAN) is made with Ethernet and FDDI on top of 140 mbit/s links in the BelWue, the Scientific Network of the DFN Association with 64 kbit/s and 2 mbits/s, and in the future ATM with 150 mbit/s.

The DFN-RPC simplifies the most important methods of distributed and parallel computing. With Remote Procedure Calls (RPC's) [10], applications may be distributed without changing the program model. Asynchronous and parallel RPC's allow an efficient and easily programmable parallelization. Data

[1] Federal Minister for Research and Technology, Project No. TK 558 VA 005.3
[2] Institute for Nuclear Engineering and Energy Systems
[3] Institute for Statics and Dynamics of Aerospace Structures

pipes facilitate the forwarding of messages. If the computers involved use different numbering systems, the data is automatically converted during the transfer to that representing the destination system. With automated process starting, parallel process structures can easily be built. Additionally the parallel application can be controlled from a graphics monitor.

This provides the applications' programmer with an uniform remote procedure call and message passing tool for developing distributed applications which makes transparent not only the complexity of communication networks but also the differences in hardware platforms.

Performance was enhanced through consequently minimizing the number of operating system calls.

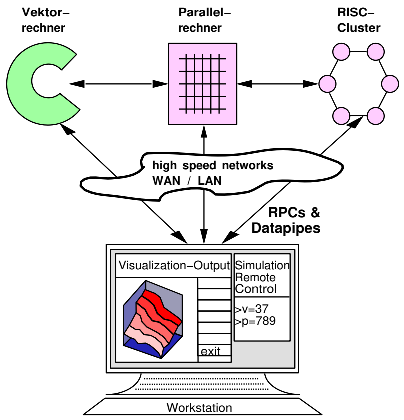


Fig. 1: Portions of the application with different requirements can be stored on the the optimum hardware for each.

## 2 Methods of Communication

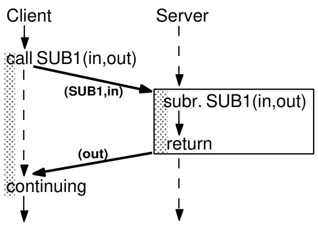


Fig. 2: The Synchronous RPC

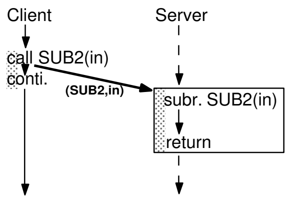


Fig. 3: The Asynchronous RPC

The normal, *synchronous* Remote Procedure Call (RPC) is shown in Fig. 2. The client shown on the left waits until the server routine (box on the right) is finished. The synchronous RPC allows distributing an application accross several processes, however the application is not parallelized because only one of the distributed processes is active at a given time.

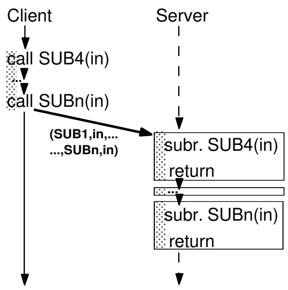


Fig. 4: The Buffered Asynchronous RPC

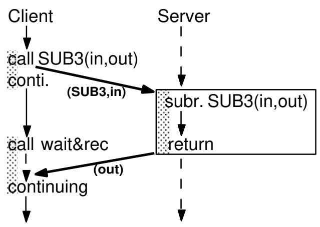


Fig. 5: The Parallel RPC

The parallelization of an application is made possible by the following methods of communication. With *asynchronous* RPC (Fig. 3), the client program can continue as soon as the arguments are sent to the server. However, output arguments can not be returned. With *buffered asynchronous* RPC (Fig. 4), data packets are sent from the client when the internal buffer is full or when a non-buffered RPC follows. Only after the server has received the data packets will the corresponding routines be executed in the order indicated by the client. If data exchange and the interaction of client and server processes occur only via the RPC call, and if the processes use no other common resources, then both asynchronous forms and the synchronous RPC are equivalent to the local procedure call.

The *parallel* RPC (Fig. 5) is a generalization of synchronous and asynchronous RPC's. As soon as the client stub sends off the input arguments, control is returned to the client program. Using inquire and wait routines, the client program can sense that the server routine has finished, after which it can capture the output arguments using a receive routine that writes them directly into the output variables given in the call to the client stub.

All four possibilities require that the RPC *dispatcher* is running in the server process, which receives the RPC's and calls the indicated server stubs, which in turn call the affiliated application server routines. The server routines that are called in a server process are therefore in a common context sequentially executed. Parallelization occurs only between the client and server processes.

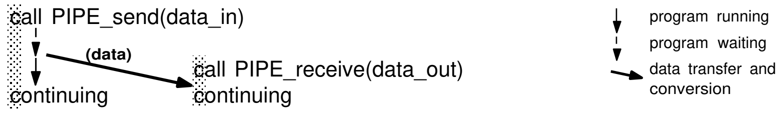


Fig. 6: Message Passing using Data Pipes

The message passing principle is realized along with the RPC principle (Fig. 6). Using *data pipes*, two applications running in parallel can exchange data. Application programs run in parallel after starting several applications' processes. With RPC applications, the application must then allow the RPC dispatcher to control the server processes. Then with parallel or asynchronous RPC's one achieves the parallelization of the application again. A data pipe can then be applied between the client program and the per asynchronous or parallel RPC executed server routines, see Fig. 7. The transport connection for RPC communication is used for this client-server data pipe.

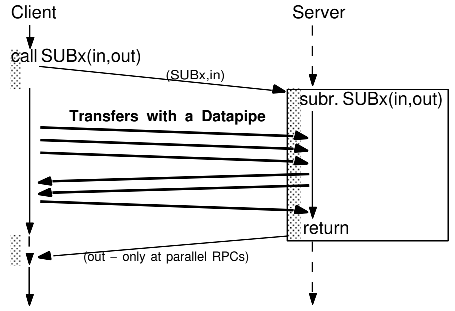


Fig. 7: Client-Server-Data pipe

Data pipes can also be established between server processes, see Fig. 8. For this, additional communications lines between the server processes are required. This can be accomplished by the client using a special RPC-based service routine.

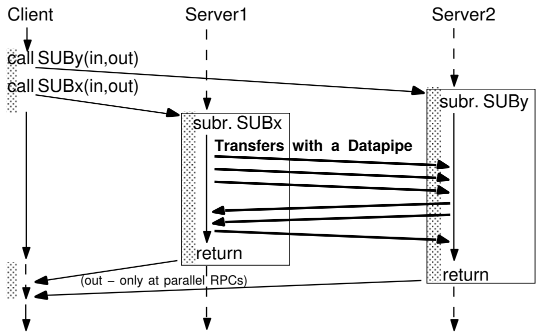


Fig. 8: Server-Server Data Pipe

As with RPC's, data type conversion takes place during the transfer if necessary due to different data representation in an heterogeneous environment.

For each data transfer there is an underlying end-to-end flow control. Because of this a sending process will be blocked if the receiver does not accept the data on time.

## 3 Calling Methods in Context

A client can send RPC's to several servers while a server is handling one client. (A server can also handle several clients, but this is not addressed by this paper.) A server can also be a client for other servers. Regarding *one* client-server pair the following rules apply:

- Asynchronous buffering will be terminated by an asynchronous unbuffered, a synchronous or a parallel RPC.
- If a synchronous RPC or a parallel RPC is received, it must wait until all previous asynchronous server routine calls are processed.
- Receiving of the output of a parallel RPC will occur automatically by a following RPC.
- All methods of RPC can also possibly be callbacks.
- However a callback can only be started from a synchronous server routine.
- The receive library routine for parallel RPC's (rporec, see Fig. 10) calls the receive client stub of the corresponding RPC and stores the output data into the output arguments specified by the call of the parallel RPC.
- A client can use several servers completely independently from each other.
- While using data pipes obvious deadlocks are partially checked.

If data pipes are to be employed between client and server, then communication must occur between the asynchronously or parallelly called server routine and that part of the client application which follows the RPC call and which is processed before another RPC call or the receiving of output arguments from the parallel RPC to this server. See Fig. 10.

With all forms of RPC synchronization it is guaranteed that the server routines will be called *exactly once* if no errors are reported by the RPC library, including the RPC-finish call at the end of the program, and *at most once* in the event of an error.

## 4 Interface Definition

The type of RPC for every interface procedure is defined in the interface definition file. For asynchronous RPC's, the stub generator is checking whether the argument list consists only of input arguments. The distinction between asynchronous or buffered asynchronous can be made during stub generation, or by actual arguments during execution. An example of the interface definition for a parallel RPC is included at the end of Fig. 10. All data types from Fortran 77 are valid as formal arguments, including externals. Alternative return-jumps are only possible with synchronous RPC's.

The following interfaces are available for data pipes:

- The library interface has a send and a receive routine capable of transferring any size of array for the following data types: Integer, Integer*2, Byte, Logical, Logical*2, Logical*1, Real, Real*8, Double Precision and Character
- The stub generator interface allows the application to generate data pipe routines. For an interface *name*(*arguments*), send and receive routines, called *name*S(IN:*arguments*) and *name*R(OUT:*arguments*) are generated. In the interface definition an argument may be marked as a *FLUSH-argument* . At transmission time this value defines whether or not the data will be buffered or sent immediately.

Fig. 12 shows an example with data pipes. Details of the interface definition language and the library routines are described in the handbook [12, 21].

## 5 Process Startup

It is very important for distributed applications that the starting of the involved processes and the establishing of the necessary communication connections is made as easy as possible to serve. With DFN-RPC, process startup, the transfer of address information and the establishing of connections can be separated, but can also be automatically carried out using service routines. The routines that make connections support the concept of *reverse connection establishing*. The initiator – in applications using several service processes this one is typically the client – starts by opening a port. The library routine returns a complete address string. The client then starts its server processes and sends them the address string, for example using the argument list. The connection is then made by the server back to the client. Advantages of this procedure are:

- Any process startup mechanism can be used.
- No daemon is required.
- All processes are running under user accounts.
- Other methods may be applied due to strict separation of process startup, establishing connections and transmission of address strings. For example, the address information required in a multi-client, single server scenario can be sent from the server via directory services to the authorized clients.

Any single-client-multi-server scenario can be set up using a library routine that is included. The application must arrange a start string for each server process. In that string a keyword will be replaced by the address string of the port opened by the client. Also an interactive binding may be chosen, which means the user can send the address strings manually from client to server. The latter is also very useful for debugging, see Fig. 11. With another library routine a freely configurable interconnection network can be established between the server processes; server-server data pipes normally are used there.

Additionally, there is a *Monitor Tool*, with which the user may monitor the status of all of the processes belonging to the distributed application. All RPC's and data pipes are displayed graphically along with history log information.

## 6 Examples

An essential design criteria for DFN-RPC is the ability to easily handle technical scientific applications in the area of high performance computing.

The example shown on the right is to be parallelized on a RISC-cluster. The simulation computations in the inner loop are independent of each other and may therefore be parallelized. simul_part includes the compute-intensive simulation code of a simulation area. The exchange of data between the individual simulation areas is carried out in new_input in the outer loop.

```
program simul
...
do while (¬ sim_fin)
  do i = 1,n_parts
    call simul_part(inp(i),out(i))
  end do
  sim_fin = end_criterion(out)
  if (¬ sim_fin) then
    inp = new_input(out)
  endif
end do
```

Fig. 9: Simulation Program

The following program demonstrates how this application can be parallelized using DFN-RPC. The exchange of data between servers occurs here via the client process inside of the routine new_input. A dynamic load balancing between the servers occurs if the number of servers is fewer than the number of parallel calls in the inner loop (n_parts).

```
C*[BLK_BEG] cb,NAME='c-block', CLIENT_SITE
C*[MOD_USE]    cm, LOCAL
C*[MOD_USE]    sm, CLIENT
C*[BLK_END]
```

With this the stub generator is instructed to split up the source and to generate the necessary stubs. In 'client block' the client portion of the application and the client stub of the module sm are combined together.

```
C*[BLK_BEG] sb,NAME='s-block', SERVER_SITE
C*[MOD_USE]    sm, SERVER
C*[BLK_END]
```

In 'server block' the routines themselves and their server stubs are positioned.

```
C*[MOD_BEG] cm,NAME='client module - main'
  program simul
  ...
```

Client portion of the application.

```
  call rpquci(n_server, obj_id, iret)
```

Server initialization.

```
  do while (¬ sim_fin)
    do i_call=1, min(n_server,n_parts)
      call rpouse(obj_id(i_call), iret)
      call simul_part(inp(i_call),out(i_call))
    end do
```

In every server – selected by rpouse – the simulation routine simul_part is started using parallel RPC's.

```
    do i_rec = 1, n_parts
      timeout = 60000 ! milliseconds
      call rpolry(n_server,obj_id(1:n_server),
           timeout,objrdy(1:n_server),obj1st, iret)
      call rpouse(obj_id(obj1st), iret)
      call rporec(iret)
```

Result collection and further simul_part calls: rpolry determines which parallel RPC is finished. With rpouse the corresponding server is selected and with rporec the output arguments are written into corresponding out(i).

```
      if (i_call < n_parts) then ; i_call=i_call+1      The server is immediately used for
        call simul_part(inp(i_call),out(i_call))        the next run of simul_part,
      endif                                             through which the automatic
    end do                                              load balancing is achieved.
    sim_fin = end_criterion(out)
    if (¬ sim_fin) then                                 Compute the new input data
      inp = new_input(out)                              and termination criteria, etc.
    endif
  end do
  call rpqucf(iret)                                     All servers are terminated.
  end
C*[MOD_END]
C*[MOD_BEG] sm, NAME='server routines mod.'             Server portion of the application.
  program exasrv                                        Special main-program for the
  character*100 client_addr                             server with rpqusp, conterpart
  call getarg(1,client_addr)                            of rpquci and rpqucf.
  call rpqusp(client_addr,iret)
  end
C*[RPI_BEG] PARALLEL                                    Parallel RPC Interface Definition,
  subroutine simul_part(inp,out)                        composed of the subroutine header,
  real inp [IN], out [OUT]                              extended by RPC-specific qualifiers
C*[RPI_NON]                                             in square brackets, e.g. [IN]
  out = simulation_code(inp)                            The subroutine body is without
  end                                                   meaning for the stub generation.
C*[RPI_END]
C*[MOD_END]
```

Fig. 10: Example simul with parallel RPC.

The stub generator separates the software in the client and server portions and generates the necessary stubs. The interface definition, the client and the server software can also be stored in different files. The dynamic load balancing demonstrated in the example requires that no data has to be exchanged between the servers.

Fig. 11 shows an example of a data base for server initialization using rpquci.

```
# Server database for program simul
rsh host_a -l my_username -n simul_sb DFNRPCADDR &
# The following host has 2 processors
rsh host_b -l my_username -n simul_sb DFNRPCADDR &
rsh host_b -l my_username -n simul_sb DFNRPCADDR &
# The following host does allow a telnet session only
echo "'Please start 'simul_sb DFNRPCADDR' on host_c by telnet"'
# n_server in the call of rpquci will be reduced to the number of servers above
-norepeat
```

Fig. 11: Example of server initialization using rpquci.

The example in Fig. 12 shows portions of the programming of a server-server data pipe.

The DFN-RPC installation package includes some examples which demonstrate the usage of all communication methods.

```
...
    call rpouse(obj_id1)                          Software in client:
    call srv_prog1(inp...,out...)
    call rpouse(obj_id2)                          srv_prog1 and srv_prog2
    call srv_prog2(inp...,out...)                 are started with parallel RPC.
    call rpouse(obj_id1) ; call rporec(iret)
    call rpouse(obj_id2) ; call rporec(iret)
...
C*[RPI_BEG] PARALLEL                              Software in server 1:
    subroutine srv_prog1(inp...,out...)
    ... inp... [IN], out... [OUT]
C*[RPI_NON]
    call rpouse(pipe_srv2)                        Switch to the data pipe.
    call rpdr0S(.false., 10000, x(1:10000))       Transmit a Real array without flush.
    call rpdd0S(.true., 30000, d(1:30000))        Transmit a Double precision array.
    call my_3vecR(n, x, y, z)                     Receive of 77 triples with a
    end                                           user-defined data pipe routine.
C*[RPI_END]
...
C*[RPI_BEG] PARALLEL                              Software in server 2:
    subroutine srv_prog2(inp...,out...)
    ... inp... [IN], out... [OUT]
C*[RPI_NON]
    call rpouse(pipe_srv1)                        Switch to the data pipe.
    call rpdr0R(flush, n, x)                      Receive of 10000 Real numbers.
    call rpdd0R(flush, n, d)                      Receive of 30000 Double pr. numbers.
    call my_3vecS(77, x(1:77), y(1:77), z(1:77))  Transmit of 77 triples with a
    end                                           user-defined data pipe routine.
C*[RPI_END]
...
C*[RPI_BEG] PIPE                                  Interface Definition for pipe my_3vec.
    subroutine my_3vec (n, x, y, z)
    real x(1:n), y(1:n), z(1:n)                   The stub generator generates
C*[RPI_END]                                       my_3vecS and my_3vecR
```

Fig. 12: Example of data pipe programming.

## 7 Details of Implementation

With DFN-RPC marshalling the data is copied from the application's memory into a transfer buffer without any further buffering in between. During the copy the data is converted if required. The content of the transfer buffer is written directly in the TCP BSD-Socket using *write*. With unmarshalling the data is read into the transfer buffer using *read* and copied (and reconverted if necessary) from there directly into the memory of the application. By selecting a suitable buffer size, the number of system calls is reduced to a minimum and a maximum data throughput is achieved. With DFN-RPC no further daemon or agent processes are required for the communication between the processes of the distributed application. Daemons are only required when the monitor is turned on. Chapter 8.4 demonstrates that this design provides for a clearly better transfer rate and less delay in comparison to DCE, ONC and PVM. DFN-RPC speed approaches nearly that of a pure TCP-IP socket connection. To achieve higher speeds is it necessary to use a more suitable transfer protocol and implementation, as for example in the Peregrine RPC System [5].

Within one process only **one** transfer buffer is normally set up for marshalling and unmarshalling purposes. Only when operations on several processes of the application are buffered is it necessary to set up further transfer buffers. If an asynchronous RPC or a data pipe send is unbuffered, then a partially full transfer buffer is sent with *write* at the end of the client stub. Otherwise it is filled through further RPC or pipe calls.

Every *read* operation is blocking. The *write* operation can also block through end-to-end flow control. Before receiving the output data of a parallel RPC or of a data pipe, using an unblocking inquiry one can query the status of one or more RPC and data pipe connections, or one can wait until the next packet has arrived or a timeout period given by the application has elapsed. Also, the list of transport connections can be completed by further Unix filedescriptors. A global waiting point can be set on which all communication paths of the application wait for the next incident. This is accomplished internally using *select*.

During implementation the number of system calls is held to a minimum in order to achieve the required functionality. Data conversion is programmed in vectorizeable loops. A distinct improvement in performance can therefore be achieved simultaneously with more functionality in relation to the parallelization and distribution of the application programs in comparison to DCE-RPC, SUN-RPC and PVM.

## 8 Other Platforms for Distributed Applications

Fundamental platforms for designing distributed applications are:

- Open Network Computing RPC (ONC, SUN-RPC) [18],
- Distributed Computing Environment RPC (DCE) [11],
- Parallel Virtual Machine (PVM) [4].

Other extensions of asynchronous RPC's can be found in:

- Cronus [19], introduces the concept of *Futures*.
- Mercury [8], which enlarges this concept by Call-Streams and Promises [7], which allow the sender to initialize several asynchronous calls and to receive their results in the same order, and
- Astra [2], which also enlarges on the concept of Futures (for example that the calls are processed by the server in the professed order) with the choice of a higher throughput or a shorter latency.

In [17] more extensions are described.

### 8.1 Comparison to the possibilities of DCE

**Semantics and Asynchronous Calls:** DCE has available four different semantics: *at most once*, *at least once*, *broadcast* and *may be*, where for example with *may be* the application is not informed of the loss of a RPC. The synchronous DFN-RPC corresponds to the DCE RPC *at most once* semantics. The functionality of the asynchronous DFN-RPC in DCE can be found only with *may be* semantics. Parallel RPC's do not exist in DCE. With DFN-RPC *at most once* is also guaranteed with asynchronous and parallel RPC's and in the following cases also *exactly once*:

- In synchronous RPC, if the client stub registers no errors;
- In asynchronous or asynchronous buffered RPC if no errors are registered from the client stub of the RPC and from all others up to and including the next synchronous RPC or the next parallel RPC's receive;
- In parallel RPC if both the client stub and the receive of the output arguments run without error.

The callback mechanism of DCE assumes a multithreaded client. With the DCE RPC asynchronous RPC functionality can be achieved only by splitting the client into concurrent threads that use synchronous RPC's.

**DCE Pipes:** Asynchronous buffered DFN-RPC's and DCE pipes have similar functionality. A goal of data pipes and asynchronous buffered RPC's is to transfer the entire volume of data in packets without waiting for a response in order to achieve a maximum transfer rate.

DCE data pipes are not comparable to DFN-RPC data pipes in that the programming interface of DCE pipes is made up of RPC callbacks and not with send and receive routines as in DFN-RPC, PVM or Posix pipes. The DCE pipe allows transfers in two directions as follows:

**push** transfers data from server to client (Fig. 13).
The client calls a server-*receive*-application routine via RPC. This server routine prepares the application's data and calls in a loop the *push*-stub, which then – in the sense of DFN-RPC's – via asynchronous buffered callbacks calls the client's *push*-application routine, which then processes the data.

**pull** transfers data from client to server (Fig. 14).
The client calls a server-*send*-application routine via RPC. This server routine then runs the *pull*-stub in a loop, which then calls via callbacks to the client the *pull*-application routine, which then provides the data and returns it in the argument list.
Because of an end-of-data argument in the *pull*-interface, this can also be implemented internally in DCE by asynchronous buffered RPC. Therefore the *send*-client-stub can call the pull-application routine until end-of-data is reached independently of the pull-server calls.
This method does not exist in DFN-RPC. However the asynchronous buffered DFN-RPC's are an alternative with more flexibility.

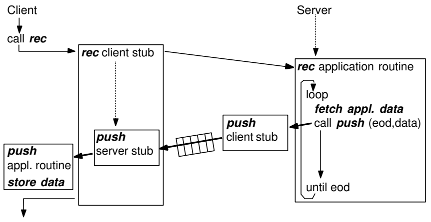


Fig. 13: Push

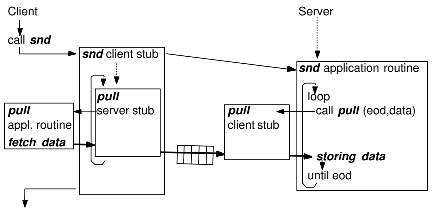


Fig. 14: Pull

The form of internal execution of DCE pipes shown here is only a model. It is used only for comparing the functionality of DCE pipes with asynchronous buffered DFN-RPC's. The model does not necessarily match the real DCE implementation. See [3, 11, Chap. 17.14.4 Pipes].

**Message Passing:**
In DCE the only basic operation with *at-most-once* semantics is the synchronous RPC. Y. Wei and C. Wu describe in [20] how, using RPC, a message passing functionality can be achieved. In comparison to DFN-RPC, either an *empty* request or an *empty* reply is sent with every message. Additionally, another thread must exist on the server side which is implementing the message port. See Fig. 15.

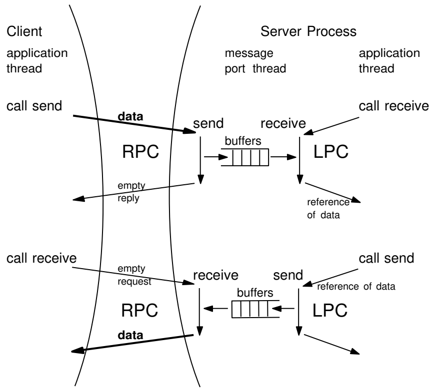


Fig. 15: Message Passing using DCE RPC

Because a synchronous RPC is made up of two messages, with DFN-RPC the message passing user interface is not implemented on top of the RPC user interface, but instead message passing and RPC use the same marshalling and transfer library. Therefore with message passing there are no unnecessary RPC packets in the opposite direction. This provides an optimum throughput and minimal latency.

**Binding and Process Start:** The string binding of DCE is comparable to that of DFN-RPC. The use of the name service in DCE must be achieved in DFN-RPC by the application using explicit storing and reading of the address string with a directory service.

A substantial difference is, that with DCE the setup of connections is made from client to server, while with DFN-RPC the port can be opened by the client or by the server. It is therefore impossible to make a *reverse connection* with DCE as in DFN-RPC (see Chap. 5).

**Language Binding:** DFN-RPC was drafted for Fortran applications, but may also be used for C applications. In a typical workstation environment, a graphical user interface written in C and running on the workstation calls numeric codes implemented in Fortran 77, Fortran 90 or HPF by using DFN-RPC. In [1] an example of an astrophysics application of this type is described.

DCE supports only C. Problems and limitations of DCE for Fortran programs are described in [6]. FIDL is a product that supports the use of Fortran under DCE RPC [9].

## 8.2 Comparison with PVM

PVM provides a message passing library with data conversion, i.e. data pipes. A daemon-based system for starting application's processes is integrated. Because the connenctions are not established in reverse, PVM requires that a bidirectional channel for exchanging address strings is initialized at process startup. For this purpose the standard input and output of *remote shell* (rsh) or *remote exec* (rexec) must be used. Process startup is therefore more complicated and less flexible under PVM as under DFN-RPC.

Other functionalities in PVM are process groups, barriers and message tags, however remote procedure calls are not implemented in PVM. PVM has a C and a Fortran interface. In comparison, DFN-RPC has circa a factor of 2 better performance. Ref. Chap. 8.4.

### 8.3 Comparison with Futures, Call-Streams and ASTRA

Futures from Cronus [19], the asynchronous RPC from ASTRA [2] and Call-Streams and Promises of Mercury System [7, 8] have in part comparable functionality with DFN-RPC.

While with Cronus, ASTRA and Call-Streams the responses of several asynchronous RPC's to one server may be outstanding and may be received in an arbitrary sequence, with DFN-RPC only one parallel RPC may be active to a server at any one time. Therefore it is not necessary to put the responses in temporary storage. Further data handling daemons are not necessary. Such *handlers* would be required when working with large unblocked datasets: if the client sends the requests c1, c2 and c3 and continues with its own application, then the server should be able to return the responses r1, r2 and r3 without the server's application being blocked due to end-to-end flow control. In contrast, with DFN-RPC the application can use data pipes for data communication during a parallel RPC.

### 8.4 Benchmark Comparison with DCE, ONC and PVM

Detailed measurements of DFN-RPC, the RPC of DCE, the RPC of ONC and PVM are available in [13, 14]. Therefore only the most significant data is discused here.

Measurements between two workstations over Ethernet using 64 kbytes of data per RPC, DFN-RPC ver. 1.0.19 realizes up to 755 kbyte/s, where the RPC of ONC attains only 610 kbyte/s (TCP-based) and PVM 3.2.2 only 376 kbyte/s using direct process-to-process connections, and 328 kbyte/s over daemon processes. Reference Tab. 1, left.

| Between 2 SGI Iris with R3000 & R4000 Proc. | | | | | | Bet. RS/6000-530 & -220 | | |
|---|---|---|---|---|---|---|---|---|
| Calls with | Unit | DFN 1.0.19 | PVM 3.2.2 | PVM ind. | ONC TCP | DFN 1.0 | DCE UDP | DCE TCP |
| 64 kbyte In | kbyte/s | 739 | 376 | 328 | 447 | 500 | 42 | 344 |
| 64 kbyte Out | kbyte/s | 755 | 373 | 323 | 610 | 579 | 312 | 311 |
| 3 kbyte In | kbyte/s | 549 | 363 | 200 | 377 | 172 | 128 | 163 |
| 3 kbyte Out | kbyte/s | 427 | 326 | 194 | 322 | 256 | 189 | 209 |
| 1 kbyte In | kbyte/s | 313 | 203 | 95 | 246 | 130 | 88 | 94 |
| 1 kbyte Out | kbyte/s | 283 | 200 | 94 | 214 | 129 | 84 | 96 |
| Average | kbyte/s | 511 | 307 | 206 | 320 | 294 | 141 | 203 |
| Empty Calls | ms/Call | 2.0 | 3.2 | 8.1 | 2.3 | 4.2 | 8.2 | 6.9 |

Tab. 1: Transfer rates and Delays between two Workstations

Measurements between two workstations over two Ethernets connected by a router DFN-RPC achieves in both directions 500 and 579 kbyte/s while DCE RPC (IBM Rel. 1.2) with TCP protocol stack reaches 344 and 311 kbyte/s. DCE RPC using the UDP protocol stack achieves 42 and 312 kbytes/s. This demonstrates a possible problem with large datasets in the end-to-end flow control of the protocol above UDP. Reference Tab. 1, right.

RPC's with empty argument lists were also measured using the same timing program. In the first environment, DFN-RPC required 2.0 ms, a RPC under ONC 2.3 ms, while a transfer with subsequent return transfer (analog to one RPC) under PVM with direct connections takes 3.2 ms and with indirect connections over the daemons 8.1 ms. In the second scenario, DFN-RPC required 4.2 ms while the RPC under TCP-based DCE required 6.9 ms. The worst case was the UDP-based DCE RPC with 8.2 ms. A possible explanation for this is that the end-to-end flow control above UDP is handled by the user process and therefore requires too many context switches between the operating system and the user process.
If client and server processes are running on the same machine, the differences are more clear, see Tab. 2.

| | | On a SGI Iris Indigo with a R3000 Processor | | | | On a RS/6000 220 | | |
|---|---|---|---|---|---|---|---|---|
| Calls with | Unit | DFN 1.0.19 | PVM 3.2.2 | PVM ind. | ONC TCP | DFN 1.0 | DCE UDP | DCE TCP |
| 64 kbyte In | kbyte/s | 2560 | 323 | 457 | 320 | 800 | 222 | 674 |
| 64 kbyte Out | kbyte/s | 2275 | 317 | 468 | 317 | 741 | 221 | 630 |
| 3 kbyte In | kbyte/s | 982 | 450 | 292 | 441 | 403 | 305 | 341 |
| 3 kbyte Out | kbyte/s | 1071 | 459 | 292 | 425 | 450 | 322 | 322 |
| 1 kbyte In | kbyte/s | 417 | 220 | 135 | 235 | 264 | 120 | 127 |
| 1 kbyte Out | kbyte/s | 429 | 224 | 135 | 242 | 261 | 113 | 122 |
| Average | kbyte/s | 1289 | 332 | 297 | 330 | 487 | 217 | 370 |
| Empty Calls | ms/Call | 1.7 | 3.1 | 5.6 | 2.3 | 3.1 | 8.0 | 7.6 |

Tab. 2: Transfer Speeds and Delays with Client and Server are running on the same Workstation

The better, sometimes even considerably better results achieved with DFN-RPC demonstrate that in the design efficient optimization strategies were applied:

- Minimization of the context switches to the operating system using large internal buffers and implementation of the synchronous RPC exclusively using blocked *read* and *write* directly via TCP sockets,
- Minimizing software-overhead in the stubs, for example through converting data in vectorizable loops with only one copy in the application process.

### 8.5 Handling of the User Interface

DFN-RPC is primarily a tool for use in technical and scientific simulations, however it can be used in other areas. It is desirable that an engineer or scientist can implement distributed applications using, if possible, program models he is familiar with. Even for parallelization the modifications should be easy to learn and still have powerful functionality.

Application with only one server can therefore be distributed with DFN-RPC at each pure routine interface[4] without modification of the program. For applications with several servers, the concept of *object identification* is used. Using a library routine (rpouse), every server object can be selected as *current object*. Every following RPC will be sent to this object. The methods of calling *asynchronous RPC* and *asynchronous buffered RPC* for parallelization are normally equivalent to a local procedure call. With *parallel RPC*, the output arguments for the client are first available after the server routine has completed. Using library routines the client program can determine if one or more servers have finished or it can wait until one of them has finished its RPC routine.

For more complex scenarios in which, for example, server-server communication is required, data pipes are available which, for the applications' programmer, are comparable to unformatted I/O to a *named Unix-Pipe*.

Because DCE-RPC with the *at-most-once* semantics is always synchronous, it requires a client program parallelized using thread programming in order to distribute a job accross several servers. Extra objects are required by Futures from ASTRA and Cronus as well as Call-Streams in Mercury for the receiving of output arguments. PVM has the easiest user interface because no RPC functionality is made available. PVM has only a message passing library.

## 9 Availability and Future Development of DFN-RPC

DFN-RPC is not just a research project. It is a product which is available for a wide palette of computer platforms.[5] Further information is available via Mosaic or anonymous ftp in [21].

DFN-RPC has been made available and is supported in the frame of the Regional Testbed for High Speed Networks in DFN and has been included in several applications' projects. The completely installed software for each platform can be downloaded with AFS, DFS or ftp.

Further development plans are to optionally provide the sending and receiving of data through the application's process itself or through a concurrently running thread. This will not only provide parallel execution of the application; it will also allow communication to be parallelized. This is important for those applications which have a graphic user interface and therefore should only briefly allow the stubs to be in control.

## References

1. Ralf Allrutz, Rolf Rabenseifner: Der DFN-RPC, ein Remote Procedure Call Tool. Proceedings, 15. DECUS München Symposium, 1992, S. 523-532.
2. A.L. Ananda, B.H. Tay, E.K. Koh: ASTRA - An Asynchronous Remote Procedure Call Facility. In Proc. 11th Intl. Conf. on Distributed Computing Systems (ICDCS-11), IEEE, Arlington, Texas, USA, 1991, pp. 172-179.
3. Developing DCE Applications for AIX and OS/2. IBM International Technical Support Center, Austin, July 1993, Chap. 12.0 Pipes.

[4] I.e. without argument transfer by global storage.

[5] In Release 1.0.60 Alliant, Apollo, Convex ieee and native, CRAY-2 and -YMP, DEC Mips, Alpha and Vax, Fujitsu, HP, IBM RS 6000, Intel Paragon, Silicon Graphics, and Sun, and PCs running Linux.

4. Al Geist et al.: PVM 3 User's Guide and Reference Manual. ORNL/TM-12187, Oak Ridge National Laboratory, Tennessee, 5/1993.
5. David B. Johnson, Willy Zwaenepoel: The Peregrine High-performance RPC System. Software-Practice and Experience 23(2), 1993, pp. 201-221.
6. Werner Kollak: Distributing FORTRAN Applications using DCE RPC. White Paper. Computer Center University of Stuttgart, March 5, 1993.
7. Barbara Liskov, Liuba Shira: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. ACM SIGPLAN Conf. on Programming Language Design and Implementation 1988, pp. 260-267.
8. Barbara Liskov, T. Bloom, D. Gifford, R. Scheifler, W.E. Weihl: Communication in the Mercury System. Proc. 21st Annual Hawaii Conference on System Science, Jan. 1988, pp. 178-187.
9. Roland Laifer, Andreas Knocke: fidl – a tool for using DCE from Fortran. In Alexander Schill (Ed.), DCE - The OSF Distributed Computing Environment, International DCE Workshop, Proceedings, Karlsruhe, Germany, Oct. 7-8, 1993, pp. 78-88.
10. Bruce-Jay Nelson: Remote Procedure Call. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsilvania, May 3, 1981.
11. Open Software Foundation (OSF): OSF DCE Version 1.0, DCE Application Development Guide, Part 3: DCE Remote Procedure Call (RPC). Revision 1.0, Dec. 31, 1991.
12. Rolf Rabenseifner et al.: The DFN Remote Procedure Call Tool. Reference Manual, Ver. 1.0.60, Computer Center University of Stuttgart, Dec. 15, 1994.
Rolf Rabenseifner et al.: Das DFN Remote Procedure Call Tool. Benutzerhandbuch, Rel. 1.0.60, Rechenzentrum Universität Stuttgart, 15. Dez. 1994.
13. Rolf Rabenseifner, Werner Kollak: DFN-RPC im Vergleich mit DCE, ONC und PVM. DFN-Mitteilungen Nr. 34, März 1994, S. 18f.
14. Rolf Rabenseifner, Armin Schuch: Comparison of DCE RPC, DFN-RPC, ONC and PVM. In Alexander Schill (Ed.), DCE - The OSF Distributed Computing Environment, International DCE Workshop, Proceedings, Karlsruhe, Germany, Oct. 7-8, 1993, pp. 39-46.
15. Rolf Rabenseifner, Hans Dieter Reimann: Verteilte Anwendungen mit dem DFN-RPC. DFN Mitteilungen Nr. 31, März 1993, S. 23-25.
16. Rolf Rabenseifner: Distributed Applications between Workstation and Supercomputer using ISO/OSI Protocols. Proceedings, Twenty-Seventh Semi-Annual Cray User Group Meeting, London. April 22-26, 1991, pp. 80-84.
17. Alexander Schill: Remote Procedure Call: Fortgeschrittene Konzepte und Systeme – ein Überblick. Teil 2: Erweiterte RPC-Ansätze. Informatik Spektrum 15, pp. 145-155, Juni 1992.
18. SUN microsystems: Network Programming Guide. Part Number 800-3850-00, 1990.
19. Edward F. Walker, Richard Floyd, Paul Neves: Asynchronous Remote Operation Execution in Distributed Systems. In Proc. 10th Intl. Conf. on Distributed Computing Systems (ICDCS-10), IEEE, Paris, France, 1990, pp. 253-259.
20. Yi-hsiu Wei and Chuan-lin Wu: Integrating RPC and Message Passing for Distributed Programming. In Alexander Schill (Ed.), DCE - The OSF Distributed Computing Environment, International DCE Workshop, Proceedings, Karlsruhe, Germany, Oct. 7-8, 1993, pp. 192-206.
21. ftp://ftp.rus.uni-stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html contains all informations about the DFN-RPC, accessible by Mosaic and anonymous ftp.

ftp://ftp.rus.uni-stuttgart.de/pub/rus/dfn_rpc/kivs95_dfnrpc_german.ps
contains the German translation of this paper.