Deutsche Übersetzung zu:
Rolf Rabenseifner: The DFN Remote Procedure Call Tool for Parallel and Distributed Applications. Proceedings, Kommunikation in Verteilten Systemen – KiVS '95, W. Kalfa, U. Huebner, K. Franke (Editors), Chemnitz-Zwickau, Feb. 22-24, 1995.

Das DFN Remote Procedure Call Tool für parallele und verteilte Anwendungen

Rolf Rabenseifner

Rechenzentrum der Universität Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany, Tel. ++49 711 6855530, e-mail: rabenseifner@rus.uni-stuttgart.de

Abstract. Der DFN-RPC, ein Remote Procedure Call Tool, wurde entwickelt, um wissenschaftlich technische Anwendungsprogramme zwischen Workstation und Compute Server verteilen zu können. Dieser Bericht beschreibt die Methoden, mit denen das DFN-RPC Tool parallele und verteilte Anwendungen unterstützt. Asynchrone RPCs wurden zu parallelen RPCs erweitert und mit Datenpipes kombiniert. Die für eine verteilte Umgebung benötigten Prozeßstarts sind integriert. Die Möglichkeiten des DFN-RPCs werden mit denen von DCE, PVM, Cronus, Astra und Mercury verglichen.
Key words: Remote Procedure Call, parallele Anwendungen, verteilte Anwendungen, paralleler RPC, DFN-RPC.

1 Einleitung

Zunehmend werden wissenschaftlich-technische Anwendungsprogramme zwischen einer Workstation am Arbeitsplatz des Wissenschaftlers und zentralen Resourcen verteilt. Hierdurch können einerseits optimale graphische Benutzerinterfaces realisiert werden und andererseits zentrale Höchstleistungsrechner, Parallelrechner und RISC-Cluster zur Berechnung komplexester numerischer Probleme herangezogen werden.

Damit sich die Entwickler derart verteilter Anwendungssysteme auf die technisch-wissenschaftlichen Aufgaben ihrer Disziplin konzentrieren können, wurde mit dem DFN-RPC eine einheitliche Plattform geschaffen, die in einer leicht erlernbaren Form die Programmierung der Kommunikationsschalen ersetzt [12, 15, 21].

Der DFN-RPC wurde am Rechenzentrum der Universität Stuttgart im Auftrag des DFN-Vereins entwickelt[1]. Hierbei waren zwei Pilotnutzer innerhalb der Universität Stuttgart (IKE[2] und ISD[3]), sowie die Theoretische Astrophysik der Universität Tübingen und das Alfred-Wegener-Institut für Polar- und Meeresforschung in Bremerhaven eng eingebunden. Damit konnten sowohl unterschiedlichste Anwendungsaspekte als auch Netzwerkaspekte beim Design und Test des DFN-RPCs integriert werden. Hierbei werden im Local Area Network (LAN) Ethernet, FDDI und UltraNet mit 10 - 800 Mbit/s eingesetzt. Technologien im Wide Area Network (WAN) sind VBN-Strecken im Landesnetz BelWue (140 MBit/s), das Wissenschaftsnetz des DFN-Vereins (WIN) mit 64 kbit/s und 2 Mbit/s und in Zukunft ATM mit 150 Mbit/s.

Der DFN-RPC vereinigt die wichtigsten Methoden zum verteilten und parallelen Rechnen. Mittels Remote Procedure Calls (RPCs) [10] kann eine Anwendung verteilt werden, ohne daß das Programmiermodell geändert wird. Asynchrone und parallele RPCs ermöglichen eine effiziente und einfach zu programmierende Parallelisierung. Datenpipes ermöglichen ein Versenden von Messages.

[1] BMFT-Förderkennzeichen TK 558 VA 005.3
[2] Institut für Kernenergetik und Energiesysteme
[3] Institut für Statik and Dynamik der Luft- und Raumfahrtkonstruktionen

Wenn die beteiligten Rechner unterschiedliche Zahlendarstellungen verwenden, werden bei sämtlichen Methoden die Daten automatisch während des Transfers in die Darstellung des Zielsystems umgewandelt. Mittels automatisierter Prozeßstarts kann problemlos eine parallele Prozeßstruktur aufgebaut werden. Außerdem kann die parallele Anwendung mit einem graphischen Monitor kontrolliert werden.

Damit steht dem Anwendungsprogrammierer ein einheitliches Entwicklungswerkzeug für seine verteilten Anwendungen zur Verfügung, das sowohl die Komplexität der Kommunikation in Rechnernetzen als auch die Unterschiede heterogener Hardwaresysteme abstrahiert.

Durch konsequente Minimierung der Anzahl von Betriebsystemaufrufen konnte eine gute Performance erreicht werden.

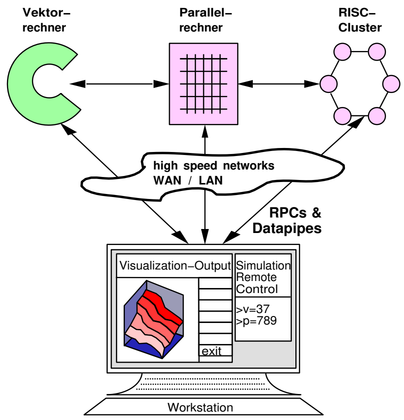

Abb. 1: Anwendungsteile mit unterschiedlichen Anforderungen können innerhalb eines Programms auf die jeweils optimale Hardware ausgelagert werden.

2 Kommunikationsformen

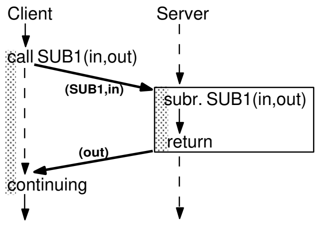

Abb. 2: Der synchrone RPC

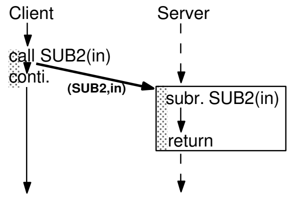

Abb. 3: Der asynchrone RPC

Der normale, *synchrone* Remote Procedure Call (RPC) ist in Abb. 2 dargestellt. Der links dargestellte Clientstub wartet bis die Serverroutine (rechte Box) fertig ist. Der synchrone RPC ermöglicht zwar die Verteilung einer Anwendung auf mehrere Prozesse, aber die Anwendung wird nicht parallelisiert, da immer nur einer der beteiligten Prozesse aktiv ist.

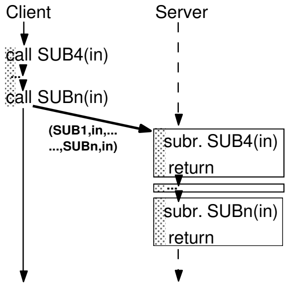

Abb. 4: Der asynchron gepufferte RPC

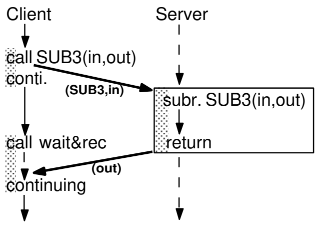

Abb. 5: Der parallele RPC

Die folgenden Kommunikationsformen ermöglichen eine Parallelisierung der Anwendung. Beim *asynchronen* RPC (Abb. 3) kann das Clientprogramm sofort weiterrechnen, nachdem die Parameterliste zum Server abgesandt wurde. Es können hierbei keine Outputparameter zurück gegeben werden. Beim *asynchron* *gepufferten* RPC (Abb. 4) werden die Datenpakete vom Clientstub erst abgeschickt, wenn der interne Puffer gefüllt ist oder wenn ein nicht gepufferter RPC nachfolgt. Erst wenn die Datenpakete beim Server ankommen, werden die entsprechenden Routinen in der vom Client gegebenen Reihenfolge hintereinander aufgerufen. Wenn der Datenaustausch und die Interaktionen von Client- und Serverprozeß nur über den RPC-Aufruf erfolgen und wenn die Prozesse auch sonst keine gemeinsame Resource benutzen, dann sind beide asynchrone Formen wie der synchrone RPC mit dem lokalen Procedure Call gleichwertig.

Der *parallele* RPC (Abb. 5) ist eine Verallgemeinerung des synchronen und asynchronen RPCs. Nachdem der Clientstub die Inputparameter abgesandt hat, gibt er die Kontrolle wieder an das Clientprogramm zurück. Mittels Inquire- und Waitroutinen kann das Clientprogramm die Beendigung der Serverroutine erfahren und anschließend mit einer Receiveroutine die Outputparameter empfangen und direkt in die beim Aufruf des Clientstubs angegebenen Outputvariablen schreiben.

Alle vier Möglichkeiten setzen voraus, daß im Serverprozeß der RPC-*Dispatcher* läuft, der die RPCs entgegennimmt und die entsprechenden Serverstubs aufruft, welche wiederum die zugehörigen Serverroutinen der Anwendung aufrufen. Die Serverroutinen, die in einem Serverprozeß gerufen werden, werden somit in einem gemeinsamen Kontext sequentiell abgearbeitet. Parallelität besteht nur zwischen dem Client- und dem Serverprozeß.

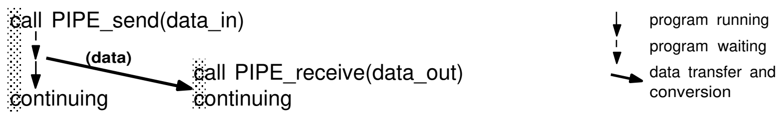

Abb. 6: Message Passing mittels Datenpipes

Neben dem RPC-Prinzip ist auch das Message-Passing-Prinzip realisiert (Abb. 6). Mit den *Datenpipes* können zwei parallel laufende Anwendungsprogramme Daten austauschen. Wenn man mehrere Anwendungsprozesse startet, hat man zuerst parallel laufende Anwendungsprogramme. Bei RPC-Anwendungen muß danach die Anwendung in den Serverprozessen die Kontrolle an den RPC-Dispatcher abgeben. Mit parallelen oder asynchronen RPCs erreicht man dann wieder die Parallelität der Anwendung, d.h. eine Datenpipe kann dann zwischen dem Clientprogramm und den per asynchronem oder parallelem RPC gerufenen Serverroutinen angewandt werden, s. Abb. 7. Für diese Client-Server-Datenpipe wird die Transportverbindung der RPC-Kommunikation eingesetzt.

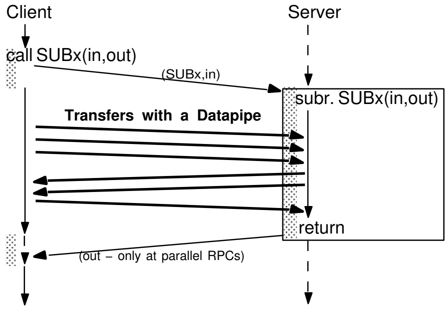

Abb. 7: Client-Server-Datenpipe

Datenpipes können auch zwischen Serverprozessen eingesetzt werden, s. Abb. 8. Hierfür sind zusätzliche Kommunikationsverbindungen zwischen den Serverprozessen zu etablieren. Dies kann vom Client mit einer speziellen RPC-basierten Serviceroutine durchgeführt werden.

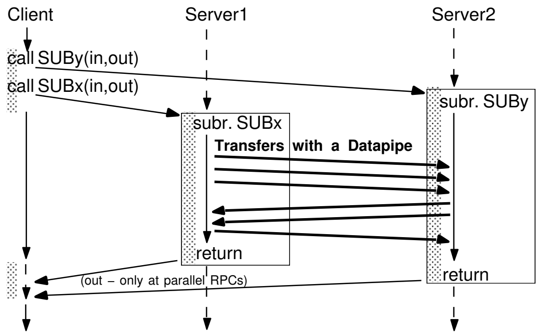

Abb. 8: Server-Server-Datenpipe

Wie bei den RPCs werden auch bei den Datenpipes die Daten bei der Übertragung automatisch konvertiert, wenn dies aufgrund unterschiedlicher Datenrepräsentationen in einer heterogenen Umgebung notwendig ist.

Jegliche Datenübertragung unterliegt einer Ende-zu-Ende-Flußkontrolle. Dadurch wird der sendende Prozeß blockiert, wenn der empfangende die Daten nicht rechtzeitig abnimmt.

3 Die Aufrufformen im Kontext

Ein Client kann RPCs zu mehreren Servern senden, während ein Server nur einen Client bedient. (Ein Server kann auch mehrere Clients bedienen, doch auf diesen Fall wird im weiteren nicht eingegangen.) Ein Server selbst kann wiederum Client zu weiteren Servern sein. Bezüglich *eines* Client-Servers Paares gelten folgende Regeln:

- Die asynchrone Pufferung wird beendet durch einen asynchron ungepufferten, einen synchronen oder parallelen RPC.
- Durch einen synchronen RPC oder beim Receive eines parallelen RPCs wird auf die Abarbeitung aller zuvor asynchron gerufenen Serverroutinen gewartet.
- Das Receive zu einem parallelen RPC wird automatisch von einem nachfolgenden RPC ausgelöst.
- Sämtliche RPC-Formen sind auch als Callback möglich.
- Ein Callback kann aber nur von einer synchron gerufenen Serverroutine gestartet werden.
- Die Receive-Library-Routine für den parallelen RPC (rporec, s. a. Abb. 10) ruft den Receive-Clientstub des zugehörigen RPCs und speichert die Outputwerte in die beim Aufruf des parallelen RPCs angegebenen aktuellen Outputparameter.
- Ein Client kann mehrere Server vollständig unabhängig voneinander benutzen.
- Der Einsatz von Datenpipes bzgl. offensichtlicher Deadlocks wird teilweise kontrolliert.

Will man Datenpipes zwischen Client und Server einsetzen, so muß die Kommunikation zwischen der asynchron oder parallel gerufenen Serverroutine und dem Teil der Clientanwendung erfolgen, der dem RPC-Aufruf folgt und vor einem weiteren RPC-Aufruf oder dem Receive der Outputparameter eines parallelen RPCs zu diesem Server liegt, s. a. Abb. 10.

Bei sämtlichen RPC-Synchronisierungsformen ist garantiert, daß die Serverroutinen *exactly once* gerufen wurden, wenn das Programm von der RPC-Library (einschließlich dem RPC-Finish Aufruf am Ende des Programms) keine Fehler gemeldet bekam. Im Fehlerfalle wird *at most once* garantiert.

4 Interface Definition

Die Art des RPCs wird für jedes Procedureinterface im Interface-Definition-File festgelegt. Der Stubgenerator überprüft bei asynchronen RPCs, daß die Parameterliste nur Inputparameter enthält. Die Unterscheidung zwischen asynchron und asynchron gepuffert kann bei der Stubgenerierung, aber auch zur Laufzeit per aktuellem Parameter entschieden werden. Abb. 10 enthält am Ende ein Beispiel für die Interface Definition eines parallelen RPCs. Als formale Parameter sind alle Datentypen von Fortran 77 möglich, einschließlich Externals. Alternative Rücksprünge sind nur bei synchronen RPCs möglich.

Für die Datenpipes gibt es folgende Interfaces:

- Das Libraryinterface enthält für die Datentypen Integer, Integer*2, Byte, Logical, Logical*2, Logical*1, Real, Real*8, Double Precision und Character je eine Sende- und eine Empfangsroutine, um ein beliebig langes Array zu übertragen.
- Das Stubgenerator-Interface ermöglicht es dem Anwender Datenpipe-Routinen zu generieren. Zu einem Interface *name*(*parameterliste*) wird die Senderoutine *name*S(IN:*parameterliste*) und die Empfangsroutine *name*R(OUT:*parameterliste*) generiert. In der Interface-Definition kann ein Parameter als *FLUSH-Argument* ausgezeichnet. Dessen aktueller Wert bestimmt dann sendeseitig, ob die Daten gepuffert bleiben oder sofort abgeschickt werden sollen.

Abb. 12 enthält ein Beispiel mit Datenpipes. Die Interface Definition Language und die Library Routinen sind ausführlich im Handbuch beschrieben [21, 12].

5 Prozeßstart

Für verteilte Anwendungen ist es auch wichtig, daß der Start der beteiligten Prozesse und der Aufbau der nötigen Verbindungen möglichst einfach zu bedienen ist. Beim DFN-RPC können Prozeßstart, Übermittlung der Adreßinformation für den Verbindungsaufbau und der Verbindungsaufbau selbst getrennt werden, aber auch durch Serviceroutinen automatisch durchgeführt werden. Die Routinen zum Verbindungsaufbau unterstützen das Konzept des *rückwärtigen Verbindungsaufbaus*. Der Initiator – bei einer Anwendung mit mehreren Serverprozessen ist dies typischerweise der Client – öffnet zuerst einen Port. Die Libraryroutine gibt einen vollständigen Adreßstring zurück. Der Client startet nun über einen beliebigen Mechanismus seine Serverprozesse und gibt ihnen beim Start, z.B. über die Parameterliste, den Adreßstring mit. Die Verbindung wird dann vom Server zurück zum Client aufgebaut. Vorteile dieses Verfahrens sind:

- Es können beliebige Prozeßstart-Mechanismen eingesetzt werden.
- Es wird kein Daemon benötigt.
- Alle Prozesse laufen unter User-Account.
- Durch die strikte Trennung von Prozeßstart, Verbindungsaufbau und Übermittlung des Adreßstrings können auch andere Methoden angewandt werden, z.B. kann die Adreßinformation zum Aufbau eines Multi-Client / Single-Server Szenariums vom Server mittels eines Directory-Services an berechtigte Clienten übermittelt werden.

Mit einer mitgelieferten Libraryroutine kann ein beliebiges Single-Client / Multi-Server Szenarium aufgebaut werden. Für jeden Prozeß hat der Anwender einen Startstring zur Verfügung zu stellen. In diesem wird ein Schlüsselwort durch den Adreßstring des vom Client eröffneten Ports ersetzt. Wahlweise kann auch ein interaktives Binding durchgeführt werden, d.h. der Anwender überträgt den Adreßstring manuell vom Client zum Server. Letzteres ist vor allem zum Debuggen sehr geeignet, s. a. Abb. 11. Mit einer weiteren Library-Routine kann ein beliebig konfigurierbares Interconnection Network zwischen den Serverprozessen aufgebaut werden; hier werden i. Allg. Server-Server-Datenpipes eingesetzt.

Zusätzlich gibt es ein *Monitortool*, mit dem der Anwender den Zustand aller an einer verteilten Anwendung beteiligten Prozesse beobachten kann. Sämtliche RPCs und Datenpipetransfers werden graphisch dargestellt und ggf. protokolliert.

6 Beispiele

Ein wesentliches Designkriterium für den DFN-RPC ist die einfache Handhabbarkeit für wissenschaftlich-technische Anwendungen im Bereich High Performance Computing.

Das nebenstehende Beispiel soll auf einem RISC-Cluster parallelisiert werden. Die Simulationsberechnungen in der inneren Schleife sind unabhängig voneinander und können daher parallelisiert werden. simul_part enthält den rechenintensiven Simulationscode für ein Simulationsgebiet. Die äußere Schleife realisiert in new_input den Datenaustausch zwischen den einzelnen Simulationsgebieten.

```
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
```

Abb. 9: Simulationsprogramm

Das folgende Programm zeigt, wie diese Anwendung mit dem DFN-RPC parallelisiert werden kann. Der Datenaustausch zwischen den Servern erfolgt hierbei über den Clientprozeß innerhalb der Routine new_input. Eine dynamische Lastbalancierung zwischen den Servern findet automatisch statt, wenn die Anzahl der Server kleiner ist als die Anzahl der parallelisierbaren Aufrufe in der innersten Schleife (n_parts).

`C*[BLK_BEG] cb, NAME='c-block', CLIENT_SITE`
`C*[MOD_USE]    cm, LOCAL`
`C*[MOD_USE]    sm, CLIENT`
`C*[BLK_END]`
 	
`C*[BLK_BEG] sb, NAME='s-block', SERVER_SITE`
`C*[MOD_USE]    sm, SERVER`
`C*[BLK_END]`
	Hiermit wird der Stubgenerator angewiesen, die Source aufzuspalten und die nötigen Stubs zu generieren. Im 'client block' werden der Clientteil der Anwendung und die Clientstubs des Moduls sm zusammengefaßt, während im 'server block' die Routinen selbst und deren Serverstubs plaziert werden.

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

`  call rpquci(n_server, obj_id, iret)`	Initialisierung der Server.

`  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 jedem Server – angewählt mit rpouse – wird mittels parallelem RPC die Simulationsroutine simul_part gestartet.

do i_rec = 1, n_parts	Ergebnisse sammeln und weitere simul_
timeout = 60000 ! milliseconds	part-Aufrufe: rpolry ermittelt, welcher
call rpolry(n_server,obj_id(1:n_server),	parallele RPC fertig ist. Mit rpouse
timeout,objrdy(1:n_server),obj1st, iret)	wird der entsprechende Server ausge-
call rpouse(obj_id(obj1st), iret)	wählt und mit rporec werden die Out-
call rporec(iret)	putparameter in entspr. out(i) ge-
if (i_call < n_parts) then ; i_call=i_call+1	schrieben. Der Server wird sofort wie-
call simul_part(inp(i_call),out(i_call))	der für die nächste Berechnung von simul_part
endif	genutzt, wodurch eine automatische
end do	Lastbalancierung erreicht wird.
sim_fin = end_criterion(out)	
if (¬ sim_fin) then	Berechnung der neuen Inputdaten
inp = new_input(out)	und des Abbruchkriteriums, etc.
endif	
end do	
call rpqucf(iret)	Sämtliche Server werden beendet.
end	
C*[MOD_END]	
C*[MOD_BEG] sm, NAME='server routines mod.'	Serverteil der Anwendung.
program exasrv	Spezielles Hauptprogramm für den
character*100 client_addr	Server mit rpqusp, dem Gegenstück
call getarg(1,client_addr)	zu rpquci und rpqucf.
call rpqusp(client_addr,iret)	
end	
C*[RPI_BEG] PARALLEL	Parallele RPC Interface Definition,
subroutine simul_part(inp,out)	bestehend aus Subroutine-Kopf,
real inp [IN], out [OUT]	erweitert um RPC-spezifische Qualifier
C*[RPI_NON]	in eckigen Klammern, z.B. [IN]
out = simulation_code(inp)	Der Subroutine-Rumpf ist für die
end	Stub-Generierung ohne Bedeutung.
C*[RPI_END]	
C*[MOD_END]	

Abb. 10: Beispiel simul mit parallelem RPC

Der Stubgenerator trennt die Software in Client- und Serverteil und erzeugt die benötigten Stubs. Die Interface Definition, die Client- und die Serversoftware können aber auch auf getrennten Files abgespeichert sein. Die in dem Beispiel dargestellte dynamische Lastbalancierung setzt voraus, daß keine Daten direkt zwischen den Servern ausgetauscht werden müssen.

Abb. 11 zeigt ein Beispiel einer Datenbasis für die Serverinitialisierung mittels 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
```

Abb. 11: Beispielfile zur Server-Initialisierung mittels rpquci

Das Beispiel in Abb. 12 zeigt Ausschnitte der Programmierung einer Server-Server-Datenpipe.

Zu ieder Installation des DFN-RPC werden auch mehrere Beispiele mitgeliefert, die in das Arbeiten mit allen Kommunikationsformen einführen.

...	
call rpouse(obj_id1)	Software im Client:
call srv_prog1(inp...,out...)	
call rpouse(obj_id2)	Mit parallelen RPC werden die Routinen
call srv_prog2(inp...,out...)	srv_prog1 und srv_prog2 gestartet.
call rpouse(obj_id1) ; call rporec(iret)	
call rpouse(obj_id2) ; call rporec(iret)	
...	
C*[RPI_BEG] PARALLEL	Software im Server 1:
subroutine srv_prog1(inp...,out...)	
... inp... [IN], out... [OUT]	
C*[RPI_NON]	
call rpouse(pipe_srv2)	Umschalten auf die Datenpipe.
call rpdr0S(.false., 10000, x(1:10000))	Senden eines Real-Arrays ohne Flush.
call rpdd0S(.true., 30000, d(1:30000))	Senden eines Doubleprecision-Arrays.
call my_3vecR(n, x, y, z)	Empfangen von 77 Trippel mit einer
end	selbstdefinierten Datenpipe.
C*[RPI_END]	
...	
C*[RPI_BEG] PARALLEL	Software im Server 2:
subroutine srv_prog2(inp...,out...)	
... inp... [IN], out... [OUT]	
C*[RPI_NON]	
call rpouse(pipe_srv1)	Umschalten auf die Datenpipe.
call rpdr0R(flush, n, x)	Empfangen von 10000 Real-Zahlen.
call rpdd0R(flush, n, d)	Empfangen von 30000 Doublepr.-Zahlen.
call my_3vecS(77, x(1:77), y(1:77), z(1:77))	Senden von 77 Trippel mittels
end	selbstdefinierter Datenpipe.
C*[RPI_END]	
...	
C*[RPI_BEG] PIPE	Interface Definition zur Pipe my_3vec.
subroutine my_3vec (n, x, y, z)	
real x(1:n), y(1:n), z(1:n)	Der Stubgenerator generiert hierzu
C*[RPI_END]	my_3vecS und my_3vecR

Abb. 12: Beispiel zur Programmierung von Datenpipes

7 Details zur Implementierung

Beim DFN-RPC Marshalling werden die Daten vom Memory der Anwendung in einen Transferpuffer kopiert und hierbei ohne nochmalige Zwischenspeicherung bei Bedarf konvertiert. Der Inhalt des Transferpuffers wird direkt mittels *write* in die TCP BSD-Socket geschrieben. Beim Unmarshalling werden die Daten mittels *read* in den Transferpuffer eingelesen und von dort in das Memory der Anwendung kopiert und hierbei ggf. dekonvertiert. Durch die Wahl einer geeigneten Puffergröße wird die Anzahl der Systemcalls minimal gehalten und ein maximaler Datendurchsatz erreicht. Beim DFN-RPC werden für die Kommunikation neben den Prozessen der verteilten Anwendung keine weiteren Daemon- oder Agentenprozesse benötigt. Nur bei eingeschaltetem Monitor werden ausschließlich hierfür Daemonprozesse eingesetzt. Im Kap. 8.4 wird gezeigt, daß dieses Design im Vergleich zu DCE, ONC und PVM zu einer deutlich besseren Übertragungsrate und zu einem geringeren Delay führt. Der DFN-RPC erreicht nahezu die Geschwindigkeit einer reinen TCP-IP Socket-Verbindung. Um höhere Geschwindigkeiten zu erreichen sind geeignetere Transportprotokolle und -Implementierungen nötig, wie z.B. im Peregrine RPC System [5].

Innerhalb eines Prozesses wird normalerweise nur **ein** Transferpuffer für alle Marshalling- und Unmarshalling-Aufgaben eingesetzt. Nur, wenn Operationen zu mehreren Prozessen von der Anwendung gepuffert werden, müssen mehrere Transferpuffer eingesetzt werden. Wenn ein asynchroner RPC oder das Senden auf einer Datenpipe ungepuffert sein soll, dann wird ein nicht vollständig gefüllter Transferpuffer mit *write* am Ende des Clientstubs abgesandt, andernfalls wird er durch nachfolgende RPC-, bzw. Pipe-Aufrufe weiter gefüllt.

Sämtliche *read*-Operationen sind blockierend. Das *write* kann durch die Ende-zu-Ende-Flußkontrolle ebenfalls blockiert werden. Vor dem Empfangen der Outputdaten eines parallelen RPCs oder einer Datenpipe kann man mittels eines nicht blockierenden Inquiry den Zustand einer oder mehrerer RPC- und Datenpipe-Verbindungen abfragen, oder auch bis zum Eintreffen des nächsten Pakets oder dem Erreichen eines von der Anwendung bestimmten Timeouts warten. Die Liste der Transportverbindungen kann ggf. um weitere Unix-Filedescriptoren ergänzt werden. Hierdurch kann ein globaler Wartepunkt realisiert werden, an dem auf allen Kommunikationspfaden der Anwendung auf das nächste Ereignis gewartet wird. Dies ist intern mittels *select* realisiert.

Bei der Implementierung wurde darauf geachtet, daß zum Erreichen der beschriebenen Funktionalität nur eine minimale Anzahl von Systemaufrufen gemacht wird. Außerdem wurde darauf geachtet, daß die Datenkonversion in einer vektorisierbaren Schleife stattfindet. Dadurch konnte eine deutlich höhere Performance im Vergleich zu DCE-RPC, SUN-RPC und PVM erreicht werden bei gleichzeitig umfangreicherer Funktionalität bzgl. der Parallelisierung und Verteilung von Anwendungsprogrammen.

8 Andere Plattformen für verteilte Anwendungen

Wesentliche Plattformen zur Entwicklung verteilter Anwendungen sind

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

Weitere Ansätze zu asynchronen RPCs findet man in

- Cronus [19], das das Konzept der *Futures* einführt.
- Mercury [8], das dieses Konzept um sog. Call-Streams und Promises [7] erweitert, die es dem Sender erlauben, mehrere asynchrone Calls zu initiieren und anschließend deren Ergebnisse in der gleichen Reihenfolge zu empfangen, und
- Astra [2], das ebenfalls das Konzept der Futures erweitert (z.B. daß die Calls in der vorgegebenen Reihenfolge beim Server durchgeführt werden) und wahlweise einen hohen Durchsatz oder eine geringe Latency ermöglicht.

In [17] sind noch weitere Ansätze beschrieben.

8.1 Vergleich mit den Möglichkeiten von DCE

Semantik und asynchrone Calls: DCE kennt vier verschiedene Semantiken: *at most once*, *at least once*, *broadcast* und *may be*, wobei z.B. bei *may be* der Verlust eines RPCs der Anwendung nicht zurückgemeldet wird. Der synchrone DFN-RPC entspricht im DCE dem RPC mit *at most once* Semantik. Die Funktionalität des asynchronen DFN-RPCs findet man im DCE nur mit der *may be* Semantik. Parallele RPCs fehlen in DCE. Beim DFN-RPC ist dagegen auch bei asynchronen und parallelen RPCs *at most once* garantiert und in folgenden Fällen auch ein *exactly once* feststellbar:

- Beim synchronen RPC, wenn der Clientstub keinen Fehler meldet;

– beim asynchronen oder asynchron gepufferten RPC, wenn kein Fehler gemeldet wird vom Clientstub dieses RPCs und von allen weiteren bis einschließlich dem nächsten synchronen RPC oder dem nächsten Receive eines parallelen RPCs;
– beim parallelen RPC, wenn sowohl Clientstub als auch das Receive der Outputparameter fehlerfrei ablaufen.

Der Callback-Mechanismus des DCE setzt einen multithreaded Client voraus. Beim DCE RPC kann die Funktionalitaet asynchroner RPCs nur erreicht werden, indem man den Client in mehrere Threads aufspaltet, die wiederum synchrone RPCs benutzen.

DCE Pipes: Asynchron gepufferte DFN-RPCs und die DCE Pipes haben eine ähnliche Funktionalität. Datenpipes und asynchron gepufferten RPCs haben zum Ziel, das ganze zu übertragende Datenvolumen als eine Folge von Paketen ohne Rückantworten zu übertragen, um eine maximale Übertragungsgeschwindigkeit zu erreichen.

Die DCE-Datenpipes sind aber nicht mit den DFN-RPC Datenpipes vergleichbar, da ihr programmiertechnisches Interface mit RPC-Callbacks gebildet ist und nicht mittels Send- und Receiveroutinen, wie beim DFN-RPC, bei PVM oder bei den Posix-Pipes. Die DCE Pipe stellt zwei Transferrichtungen zur Verfügung:

push überträgt Daten vom Server zum Client (Abb. 13).
Der Client ruft eine Server-*Receive*-Anwendungsroutine per RPC. Diese Serverroutine stellt die Anwendungsdaten bereit und ruft dann in einer Schleife den *push*-Stub, welcher dann – im Sinne des DFN-RPCs – per asynchron gepufferter Callbacks im Client die eigentlich *push*-Anwendungsroutine ruft, die die Daten verarbeitet.

pull überträgt Daten vom Client zum Server (Abb. 14).
Der Client ruft eine Server-*Send*-Anwendungsroutine per RPC. Diese Serverroutine ruft dann in einer Schleife den *pull*-Stub, welcher dann per Callback beim Client die *pull*-Anwendungsroutine ruft, die die Daten bereitstellt und in der Parameterliste zurückgibt.
Aufgrund eines End-of-Data-Parameters im *pull*-Interfaces kann auch diese Form beim DCE asynchron gepuffert gelöst werden. Daher kann unabhängig von den pull-Serveraufrufen der *Send*-Client-Stub die pull-Anwendungsroutine so oft aufrufen, bis End-of-Data erreicht ist.
Diese Form gibt es beim DFN-RPC nicht. Die asynchron gepufferten DFN-RPCs sind aber eine Alternative mit insgesamt größerer Flexibilität.

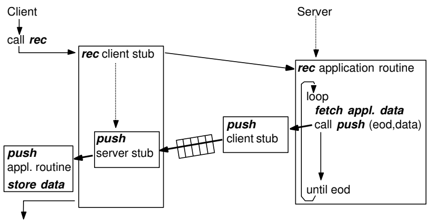

Abb. 13: Push

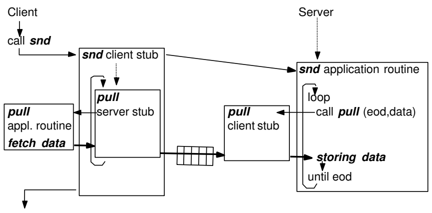

Abb. 14: Pull

Die hier dargestellte Form der internen Realisierung der DCE Pipes ist nur als Modell zu verstehen. Sie dient nur dem Vergleich der Funktionalitäten von DCE Pipes und asynchron gepufferten DFN-RPC. Das Modell muß nicht mit der realen DCE-Implementierung übereinstimmen, s. [3, 11, Chap. 17.14.4 Pipes].

Message Passing:

DCE besitzt als einzige Grundoperation mit *at-most-once* Semantik den synchronen RPC. [20] beschreibt, wie man mittels RPC eine Message Passing Funktionalität realisieren kann. Im Vergleich zum DFN-RPC wird hierbei bei jeder Message entweder eine *leere* Anforderung oder ein *leeres* leeres Reply gesandt. Außerdem muß serverseitig eine weitere Thread existieren, die den Message Port realisiert, s. Abb. 15.

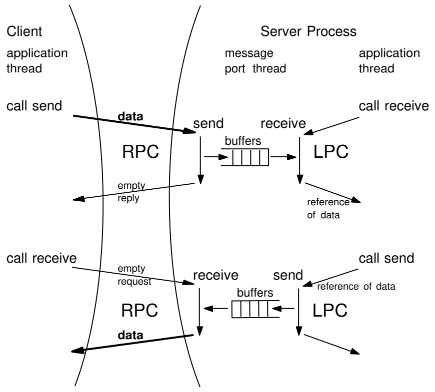

Abb. 15: Message Passing mittels DCE RPC

Da ein synchroner RPC aus zwei Messages besteht, wurde beim DFN-RPC das Message Passing User Interface nicht oberhalb des RPC User Interfaces implementiert sondern Message Passing und RPC verwenden die gleiche Marshalling- und Transfer-Library. Dadurch fallen beim Message Passing keine unnötigen RPC-Pakete in der Gegenrichtung an. Dadurch ist ein optimaler Throughput und eine minimale Latency möglich.

Binding und Prozeßstart: Das Stringbinding des DCE entspricht dem Binding des DFN-RPC. Der Einsatz des Name Service im DCE muß beim DFN-RPC von der Anwendung durch explizites Abspeichern und Lesen des Adreßstrings mittels eines Directoryservices realisiert werden.

Ein wesentlicher Unterschied ist, daß beim DCE der Verbindungsaufbau vom Client zum Server gemacht wird, während beim DFN-RPC der Port wahlweise beim Client oder beim Server eröffnet werden kann. Damit ist beim DCE ein *rückwärtiger Verbindungsaufbau* im Gegensatz zum DFN-RPC (s. Kap. 5) nicht möglich.

Language Binding: Der DFN-RPC wurde für Fortran Anwendungen entworfen, kann aber auch für C-Anwendungen eingesetzt werden. In einer typischen Einsatzform liegt auf der Workstation eine in C geschriebene graphische Benutzerschnittstelle, die via DFN-RPC auf die in Fortran 77, Fortran 90 oder HPF implementierten Numerikcodes zugreift. In [1] ist eine derartige Anwendung aus der Astrophysik beschrieben.

DCE selbst unterstützt nur C. Probleme und Grenzen beim Einsatz von DCE für Fortran-Programme sind in [6] beschrieben. FIDL ist ein Produkt, das den Einsatz des DCE RPC unter Fortran unterstützt [9].

8.2 Vergleich mit PVM

PVM stellt eine Message Passing Library mit Datenkonversion, d.h. Datenpipes, zur Verfügung. Ein daemon-basiertes System zum Start der Anwendungsprozesse ist integriert. Da die Verbindungen nicht rückwärts aufgebaut werden, ist es bei PVM notwendig, daß beim Prozeßstart ein bidirektionaler Übertragungskanal für den Adreßstring aufgebaut wird. Hierzu wird der Standardinput und -output von *remote shell* (rsh) oder *remote exec* (rexec) verwendet. Der Prozeßstart ist daher bei PVM komplizierter und weniger flexibel als beim DFN-RPC.

Weitergehende Funktionalitäten in PVM sind process groups, barriers und massage tags, jedoch sind Remote Procedure Calls in PVM nicht realisiert. PVM hat ein C- und ein Fortran-Interface. Der DFN-RPC hat vergleichsweise eine ca. um den Faktor 2 bessere Performance, s. Kap. 8.4.

8.3 Vergleich mit Futures, Call-Streams und ASTRA

Die Futures von Cronus [19], der asynchrone RPC von ASTRA [2] und die Call-Streams und Promises im Mercury System [7, 8] besitzen teilweise gleiche Funktionalität wie der parallele DFN-RPC.

Während bei Cronus, ASTRA und Call-Streams die Antworten mehrerer asynchroner RPCs zu einem Server ausstehen können und in unterschiedlicher oder gleicher Reihenfolge wie die Calls von der Client-Anwendung empfangen werden können, kann beim DFN-RPC zu einem Server jeweils nur ein paralleler RPC aktiv sein. Dafür werden die Antworten nicht nochmals zwischengespeichert. Es sind keine zusätzlichen Prozesse nötig. Solche *Handler* wären nötig, wenn man große Datenmengen ohne Blockierungen bearbeiten möchte: schickt man vom Client an den Server die Aufträge c1, c2 und c3 und rechnet man dann weiter, dann sollte der Server in der Lage sein, die Antworten r1, r2, r3 hierzu zurückzusenden ohne, daß die Anwendung im Server aufgrund einer Ende-zu-Ende-Flußkontrolle blockiert wird. Für einen ggf. nötigen Datenaustausch während eines parallelen RPCs sieht der DFN-RPC Datenpipes vor.

8.4 Benchmark-Vergleich mit DCE, ONC und PVM

In [13, 14] wurden detailierte Meßergebnisse zum DFN-RPC, dem RPC von DCE, dem RPC des ONC und PVM veröffentlicht. Daher seien hier nur die wesentlichsten Daten dargestellt.

Bei Messungen zwischen zwei Workstations auf einem Ethernet mit 64 kByte Daten pro RPC erzielte der DFN-RPC 1.0.19 bis zu 755 kByte/s, der RPC des ONC 610 kByte/s (TCP-basiert) und PVM 3.2.2 nur 376 kByte/s bei direkten Prozeß-zu-Prozeß-Verbindungen, bzw. 328 kByte/s bei zwischengeschalteten Daemon-Prozessen, s. Tab. 1, linke Seite.

zw. zwei SGI Iris mit R3000, bzw. R4000 Proc.						zw. RS/6000-530 u. -220		
Calls mit	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
Durchschnitt	kByte/s	511	307	206	320	294	141	203
Leerer Call	ms/Call	2.0	3.2	8.1	2.3	4.2	8.2	6.9

Tab. 1: Übertragungsgeschwindigkeit und Delay zwischen zwei Workstations

Bei Messungen auf zwei über einen Router verbundenen Ethernets erreichte der DFN-RPC in den beiden Übertragungsrichtungen 500 bzw. 579 kByte/s, während der DCE RPC (IBM Rel. 1.2) mit TCP-Protokollstack nur 344 bzw. 311 kByte/s erreichte. Beim DCE RPC mit dem UDP-Protokollstack wurden

42 und 312 kByte/s erziehlt. Dies weist auf mögliche Probleme in der Ende-zu-Ende-Flußkontrolle bei dem oberhalb von UDP realisierten Protokoll bei großen Datenmengen hin, s. Tab. 1, rechte Seite.

Mit dem gleichen Meßprogramm wurden auch die Delays von RPCs mit leerer Parameterliste gemessen. Bei ersterer Umgebung benötigt ein DFN-RPC 2.0 ms, ein RPC im ONC 2.3 ms, während eine Datenübertragung mit anschließender Rückübertragung (analog zu einem RPC) bei PVM mit direkten Verbindungen 3.2 ms und indirekt über die Daemons 8.1 ms benötigte. Im zweiten Szenarium benötigte ein DFN-RPC 4.2 ms während der RPC bei TCP-basiertem DCE 6.9 ms brauchte. Am schlechtesten schnitt der UDP-basierte DCE RPC mit 8.2 ms ab. Eine mögliche Ursache hierfür könnte sein, daß die Ende-zu-Ende-Flußkontrolle oberhalb von UDP im Benutzerprozeß stattfinden muß und daher ein zu häufiger Kontextwechsel zwischen Betriebssystem und Benutzerprozeß nötig ist.

Wenn Client- und Serverprozeß sich auf dem gleichen Rechner befinden, sind die Unterschiede noch deutlicher ausgeprägt, s. Tab. 2.

		auf einer SGI Iris Indigo mit R3000 Processor				auf einer RS/6000 220		
Calls mit	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
Durchschnitt	kByte/s	1289	332	297	330	487	217	370
Leerer Call	ms/Call	1.7	3.1	5.6	2.3	3.1	8.0	7.6

Tab. 2: Übertragungsgeschwindigkeit und Delay, wenn Client und Server auf der gleichen Workstation laufen

Die besseren und zum Teil deutlich besseren Ergebnisse beim DFN-RPC zeigen, daß die bei seiner Entwicklung angewandten Optimierungsstrategien sinnvoll sind:

- Minimierung der Kontextwechsel zum Betriebssystem durch große interne Puffer und Implementation des synchronen RPCs ausschließlich mittels blokkierendem *read* und *write* direkt mittels TCP-Sockets,
- Minimierung des Software-Overheads in den Stubs, z.B. durch Konversion der Daten in vektorisierbaren Schleifen mit nur genau einem Umkopiervorgang innerhalb eines Anwendungsprozesses.

8.5 Handhabung der Benutzerschnittstelle

Der DFN-RPC soll in erster Linie ein Werkzeug im Bereich der wissenschaftlich-technischen Simulationsrechnungen sein, sein Einsatz ist aber auch in vielen anderen Bereichen möglich. Der Ingenieur oder Naturwissenschaftler soll verteilte Anwendungen möglichst mit den ihm bekannten Programmiermodellen erstellen können. Auch bei der Parallelisierung sollten die Erweiterungen möglichst leicht erlernbar und trotzdem in der Funktionalität mächtig sein.

Anwendungen mit nur einem Server können daher ohne Programm-Modifikationen an jedem rein prozeduralen Interface[4] mit dem DFN-RPC verteilt werden. Für Anwendungen mit mehreren Servern wird der Begriff der *object identification* eingeführt. Mittels einer Libraryroutine (rpouse) kann jedes Serverobjekt als *current object* ausgewählt werden. Sämtliche nachfolgende RPCs werden zu diesem Objekt gesandt. Die zur Parallelisierung eingeführten Aufrufformen *asynchroner RPC* und *asynchron gepufferter RPC* sind dem lokalen Procedure Call normalerweise äquivalent. Beim *parallelen RPC* kommt hinzu, daß die Outputparameter für den Client erst nach Abschluß der Serverroutine zur Verfügung stehen und, daß das Clientprogramm mittels Libraryroutinen untersuchen kann ob einer oder mehrere Server fertig sind, bzw. daß darauf gezielt gewartet werden kann.

Für komplexere Szenarien, in denen z.B. eine Server-Server-Kommunikation nötig ist, stehen Datenpipes zur Verfügung, die für den Anwendungsprogrammierer vergleichbar sind mit unformatted I/O auf einer *named Unix-Pipe*.

Beim DCE RPC ist normalerweise schon ein mittels Thread-Programmierung in sich parallelisiertes Clientprogramm nötig, um eine Aufgabe auf mehrere Server zu verteilen, da der einzelne RPC mit *at-most-once* Semantik immer synchron ist. Bei den Futures von ASTRA und Cronus und den Call-Streams in Mercury dagegen werden zusätzliche Objekte für das Empfangen der Outputparameter eingeführt. Das einfachste Userinterface besitzt PVM, da hier keine RPC-Funktionalität bereitgestellt wird. PVM besitzt nur eine Message-Passing-Library.

9 Verfügbarkeit des DFN-RPCs und Ausblick

Der DFN-RPC ist nicht nur ein Forschungsprojekt, sondern auch ein Produkt, das für eine breite Palette von Rechnern[5] verfügbar ist. Weitere Informationen findet man per Mosaic oder anonymous ftp in [21].

Das DFN-RPC-Tool wird im Rahmen der Regionalen Testbeds für Hochgeschwindigkeitsnetze im DFN zur Verfügung gestellt und gewartet und in mehreren Anwendungsprojekten eingesetzt. Hierbei kann man die Software für sämtliche Plattformen fertig installiert mittels AFS, DFS oder ftp beziehen.

Geplant ist auch, das Versenden und Empfangen der Daten wahlweise durch den Anwendungsprozeß oder durch eine nebenläufige Thread zu ermöglichen. Hierdurch wird nicht nur eine parallele Abarbeitung der Anwendung ermöglicht, sondern auch eine parallele Durchführung der Kommunikation. Dies ist vor allem für Anwendungen wichtig, die beim Client ein graphisches Benutzerinterface besitzen und daher die Kontrolle nur kurzzeitig an die Stubs abgeben sollten.

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] d.h. ohne zusätzliche Parameterübergabe über globale Datenbereiche

[5] In Release 1.0.60 Alliant, Apollo, Convex ieee und native, CRAY-2 und -YMP, DEC Mips, Alpha und Vax, Fujitsu, HP, IBM RS 6000, Intel Paragon, Silicon Graphics und Sun und PCs unter 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. Rechenzentrum Universität 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.: Das DFN Remote Procedure Call Tool. Benutzerhandbuch, Rel. 1.0.60, Rechenzentrum Universität Stuttgart, 15. Dez. 1994.
Rolf Rabenseifner et al.: The DFN Remote Procedure Call Tool. Reference Manual, Ver. 1.0.60, Computer Center University of Stuttgart, Dec. 15, 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
enthält alle Informationen zum DFN-RPC und ist über Mosaic oder anonymous ftp erreichbar.

ftp://ftp.rus.uni-stuttgart.de/pub/rus/dfn_rpc/dfnrpc_kivs95.ps
enthält das englische Original dieses Artikels.
ftp://ftp.rus.uni-stuttgart.de/pub/rus/dfn_rpc/dfnrpc_kivs95_german.ps
enthält diese deutsche Übersetzung.