Das DFN Remote Procedure Call Tool

Benutzerhandbuch
Band I - Einführung und Handbuch

Rolf Rabenseifner
Jens Bönisch, Frank Deuble, Chris Dodge, Bernd Hesse, Mandana Karimian
Jan Kayser, Werner Kollak, Hans Dieter Reimann

15. Dezember 1994

Release 1.0.60

Rechenzentrum der Universität Stuttgart
Prof. Dr.-Ing. R. Rühle
Allmandring 30
70550 Stuttgart
(Tel.: 0711-685-5530, FAX 0711-682357)

Gefördert vom
Verein zur Förderung eines Deutschen Forschungsnetzes (DFN),
und Bundesministerium für Forschung und Technologie (BMFT)
unter den Förderkennzeichen TK 558 VA 005.1 und VA 005.3
'Verteilte Anwendungen über Hochgeschwindigkeitsdatenkommunikation'

Interne Dokumentationsbezeichnung

dfnrpc/1.0/src/doc/*.tex (Text)
MAC/rabenseifner/DFNRPC_1.0/pics/* (Fig.-Src)
dfnrpc/1.0/src/doc/pics/*.showcase (Fig.-Src)
dfnrpc/1.0/src/doc/pics/*.eps (EPS-Abb.)

Als Postscript-File zum Ausdrucken:
Printable Postscript Files:

$DFNRPCROOT/dfnrpc/1.0/dfnrpc.ps (Druckvorlage, Band I, deutsch)
$DFNRPCROOT/dfnrpc/1.0/dfnrpc_obsolete.ps (Druckvorlage, Band II, deutsch)
$DFNRPCROOT/dfnrpc/1.0/dfnrpc_e.ps (Druckvorlage, englisch)

Inhalt

Projektleitung: Rolf Rabenseifner. Projektstart: 1. Sept. 1988

Gefördert vom
Verein zur Förderung eines Deutschen Forschungsnetzes (DFN) und Bundesministerium für Forschung und Technologie (BMFT) unter dem Förderkennzeichen TK 558 VA 005 „Verteilte Anwendungen über Hochgeschwindigkeitsdatenkommunikation“ und TK 558 VA 005.3 „Erweitertes RPC-Tool für verteilte Anwendungen“.

1 Einleitung

Wissenschaftlich-technisches Rechnen wird in Zukunft vorwiegend über **Workstation-Rechner** und zugeordnete Dienstleistungsrechner durchgeführt. Diese dezentrale Datenverarbeitungsstruktur drückt sich auch in den Empfehlungen des Wissenschaftsrats, der DFG und des EDV-Gesamtplans III Baden-Württemberg aus. Dies bedeutet, daß der Wissenschaftler dann direkt am Arbeitsplatz auf seiner Workstation arbeitet, von der aus er Zugriff auf Dienstleistungsrechner (Server) für spezielle Aufgaben hat. Der Einsatz von Workstations mit ihren leistungsfähigen Graphikmöglichkeiten ist heute für die Visualisierung wissenschaftlich-technischer Ergebnisse unabdingbar.

Da der Einsatz der schnellsten Rechner für den Erhalt und den Ausbau unserer Position im weltweiten Forschungsmarkt essentiell ist, aber aus ökonomischen Gründen nicht jeder Arbeitsplatz mit einer "Super-Workstation" ausgestattet werden kann, ist der unmittelbare Zugriff auf zentrale **Compute-Server** über **Hochgeschwindigkeits-Daten-Netze (HDN)** notwendig. Als Compute-Server dienen heute Vektorrechner wie Parallelrechner und Workstation-Cluster. Nur die Hochgeschwindigkeitsdatenkommunikation erfüllt die Anforderungen an eine schnelle graphische Darstellung von wissenschaftlich-technischen Ergebnissen.

Die Möglichkeiten, die hier in Zukunft den Wissenschaftlern eröffnet werden, wurden bei der Einweihung der 140 MBit Glasfaserverbindung zwischen Stuttgart und Karlsruhe demonstriert. Doch sind mit den heute vorhandenen Software-Werkzeugen nur Spezialisten in der Lage, Workstation und Compute-Server über Hochgeschwindigkeits-Daten-Netze zu koppeln und dieses Potential für die Berechnung und Darstellung ihrer wissenschaftlich-technischen Probleme zu nutzen. Das Arbeiten mit Workstation und Supercomputer unter Ausnutzung der Hochgeschwindigkeitsdatenkommunikation ist von Natur aus wesentlich komplexer als die Bedienung eines Universalrechners von einem Terminal aus. Daher müssen den Wissenschaftlern Werkzeuge zur Verfügung gestellt werden, damit sie ohne Spezialwissen ihre auf einem Compute-Server erzeugten Resultate direkt über Hochgeschwindigkeitsdatenkommunikation auf ihrer Workstation visualisieren können.

Mit dem Remote Procedure Call Tool des Deutschen Forschungsnetzes (**DFN-RPC**) wird dem Entwickler einer Anwendung ein Werkzeug gegeben, mit dem er sein Anwendungsprogramm zwischen Workstation (Client) und Compute-Server entsprechend seinen Erfordernissen verteilen kann. Der Aufwand für den Anwendungsprogrammierer und die notwendigen kommunikationstechnischen Kenntnisse (Kommunikationsprotokolle, usw.) wird hierbei möglichst klein gehalten. Er kann dabei auch Test-Tools (z.B. Debugger) auf den beteiligten Rechnern einsetzen. Die fertige Anwendung kann auch so gestartet werden, daß dem Anwender die Slave-Prozesse verborgen bleiben.

Das nachfolgend beschriebene Remote Procedure Call Tool ermöglicht dem Wissenschaftler eine geeignete **Verteilung seines Anwendungsprogramms** zwischen Workstation und Compute-Server.

Den tar-File und sämtliche Dokumente zum DFN-RPC findet man mittels anonymous ftp oder Mosaic in
ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/ bzw.
ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html

2 Installation des DFN-RPC

ACHTUNG: Die Groß- und Kleinschreibung ist signifikant.

Die Installation des DFN-RPC besteht aus folgenden Schritten:

1. Bereitstellen der entsprechenden Betriebssystemmittel, wie Plattenplatz, Compiler, etc.
2. Holen des tar-Files.
   Der DFN-RPC wird möglichst über existierende Netze ausgeliefert.
3. Entpacken des tar-Files.
   Der DFN-RPC wird normalerweise als Sourcecode ausgeliefert. Er ist aber ohne interne Dokumentation (Kommentare).
4. Schreiben des eigenen Installationsparametersatzes.
   Es werden Installationsparametersätze für verschiedene Rechner und Betriebssystemkonfigurationen mitgeliefert. Meist kann ein solcher direkt oder mit leichten Modifikationen benützt werden.
5. Anpassen und Aufruf des Navigations-Scripts dfnrpc_set.
   Hiermit kann man sowohl bei der weiteren Installation, als auch später im Gebrauch des DFN-RPC leicht durch die Directories des DFN-RPC navigieren.
6. Installation einer Konfiguration.
   Hier werden die Bibliotheken und Beispiele kompiliert und gebunden.

Nach diesem Schritt ist der DFN-RPC installiert.

Abschließend werden in diesem Kapitel noch zwei weitere Aspekte erörtert:

- Einfahren von Fehlerkorrekturen,
- Installation auf einem Fileserver.

2.1 Voraussetzungen

Berechtigung	Normaler User, Schreibrecht auf einer beliebigen Directory.
Plattenplatz	35 – 55 MByte (komplett, incl. Beispiele) Im einzelnen: 3 MByte für den komprimierten Tar-File 8 MByte für die Source (zumindest während der Installation) 4 MByte für eine Installation ohne Beispiele 20 – 40 MByte für die Beispiele, Platzbedarf hängt vom Betriebssystem ab
Betriebssystem	Posix (oder VMS)
Compiler	Fortran und C
Transport-Layer	TCP/IP (und/oder ISO/OSI TP4, s.u.)

2.2 Holen des tar-Files

Der DFN-RPC, Release 1.0.60 wird unter Unix in Form eines komprimierten tar-Files ausgeliefert:

`dfn_rpc_1.0.60.tar.z`

Dieser File kann mittels *ftp* (im *binary* mode) oder Mosaic bezogen werden von:
ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/dfn_rpc_1.0.60.tar.Z

2.3 Entpacken des tar-Files

Man kann den DFN-RPC in jeder beliebigen Directory des Filesystems installieren. Diese Directory heißt DFNRPCROOT. In den folgenden beispielhaften Kommandosequenzen wird als DFNRPCROOT die Directory */usr/3rdparty/dfn* benützt.

Entpacken des tar-Files:

```
cd    /usr/3rdparty/dfn
zcat  /..._tar_file_directory_.../dfn_rpc_1.0.60.tar.Z | tar xvf -
```

Der hierbei entstehende Subtree *dfnrpc/1.0/src* enthält die Software zum DFN-RPC vollständig als Sourcecode (ohne interne Dokumentation, bzw. Kommentare):

```
Files  in /usr/3rdparty/dfn/dfnrpc/1.0
                  / dfnrpc.ps        diese Dokumentation als Postscript-File
                  / dfnrpc_set       csh-script zum navigieren durch d. Softw.

Files  in /usr/3rdparty/dfn/dfnrpc/1.0/src
                  / README           vor der weiteren Installation zu lesen
                  / inst_SUN4        Installations-Script fuer eine Sun 4
                  / inst_CRAY2       Installations-Script fuer eine Cray-2
                  / install          Zentrales Installations-Script
                  / install_...      Hilfsfiles hierfuer


Subdirectories in /usr/3rdparty/dfn/dfnrpc/1.0/src

 .../ makefiles   / bin_makefile     Generischer makefile fuer sys/.../bin
                  / lib_makefile     Generischer makefile fuer sys/.../lib
                  / include_makefile Generischer makefile fuer sys/.../include
 .../ ct4         / ...              Source zur DFN-RPC Library
 .../ ipsm        / ...              Source zum Interactiv Process Start Module
 .../ stubgen     / ...              Source zum Stubgenerator (1.Teil)
 .../ rpc         / dfnrpcgen        Source zum Stubgenerator (2.Teil)
                  / rpcmacro.m4
                  / dfn...           Hilfsfiles und Utilities

                                    Mitgelieferte Beispiele:
 .../ examples/sum / README          Einfuehrungsbeispiel in Fortran
                  / source_makefile
                  / sum.rpi
```

```
.../ examples/csum/ README          Einfuehrungsbeispiel in C
                  / source_makefile
                  / csum.rpi
                  / csum_lb.c
                  / csum_cb.c
                  / csum_sb.c
.../ examples/awi / ...             Beispiel des AWI
.../ examples/ben / ...             Benchmark zum DFN-RPC
.../ examples/cav / ...             Ueberstroemung eines Hohlraumes
.../ examples/cvttest / ...         Test der Konvertierungsroutinen
.../ examples/exa / ...             RPI-Beispile aus diesem Handbuch
.../ examples/moni / ...            Beispiel zum Monitor
.../ examples/para / ...            Demoprogramm parallele RPCs
.../ examples/ph1 / ...             Einfaches PHIGS-Beispielprogramm
.../ examples/ph2 / ...             wie ph1, nur mit herstellerabh. Initial.
.../ examples/phi / ...             Verteiltes PHIGS
.../ examples/pipe / ...            Datenpipes
.../ examples/rfa / ...             Remote-File-Access
.../ examples/rpcpipe / ...         Beispiel RPC und Datenpipes
.../ examples/test / ...            Testprogramm Datenuebertragung
.../ examples/ua1a / ...            Beispiel - Remote Write - interaktiv
.../ examples/ua1b / ...                                    - mit NQS
.../ examples/ua1c / ...                                    - mit rsh
.../ examples/ua1d / ...                                    - mit IPSM
```

2.4 Plattformen

Der DFN-RPC steht nun für in Tab. 2.1 genannte und hierzu kompatible Plattformen zur Verfügung. Die Bezeichnungen MANU, ARCH und OPER in den Inst_...-Files entsprechen den Optionen -M, -A und -O von dfnrpcgen.

Der DFN-RPC kann normalerweise mit den Angaben von MANU, ARCH und OPER auch auf anderen kompatiblen Plattformen installiert werden, z.B. bei Unicos 8.0 wird mit ARCH=unicos_7.0 installiert, oder bei IRIX 5.2 wird mit ARCH=IRIX_5.1 installiert.

Der DFN-RPC unterstützt außerdem folgende Möglichkeiten (s.a. Tab. 2.2):

F2C_CC — Compilation von Fortran-Files mit dem Public Domain Fortan-C-Konverter f2c und anschließender C-Compilation mit dem C-Compiler des Herstellers. Wenn F2C_CC nicht angegeben ist, wird der Fortran-Compiler des Herstellers benutzt. S.a. Release Notes Kap. 2.11.6 auf Seite 17.

ULTRA — Zusätzlicher UltraNet-Native-Protokollstack, wenn ULTRA=true. Sonst muß ULTRA=false gesetzt sein.

OSI — Zusätzlicher OSI-Protokollstack; Werte sind false (kein OSI installiert), sunlink und sunnet (Produkte von Sun), cray (Cray OSI 1.0) und cray2 (Cray OSI 2.0). ULTRA und OSI schließen sich gegenseitig aus.

PHIGS — Beispielprogramme mit Phigs

Die Phigs-Beispiele werden bei der Installation nicht mehr automatisch übersetzt. OSI und PHIGS werden nicht mehr gewartet.

Bezeichnungen in den inst-Files			Fortran-	mitgelieferte	zugehörige
MANU	ARCH	OPER	Compiler	inst_...Files	AFS-Arch.
alliant	FX	Concentrix_5.0	fortran	inst_ALLI	
apollo	DN10000	SR10.2	ftn	inst_APO	
cray	cray2	unicos_4.0	cf77		
		unicos_5.0	cf77		
		unicos_6.0	cf77		
		unicos_7.0	cf77		
	ymp	unicos_7.0	cf77	inst_CRAYYMP	c90_uni8
convex	C120[1]	ConvexOS_V7.1	fc	inst_CVX	
	C120ieee[2]	ConvexOS_V7.1	fc	inst_CVXI	
dec	vax	ultrix	fort	inst_VAX	vax_ul4
	dec	ultrix	f77	inst_DEC	pmax_ul43
		OSF_1	f77	inst_ALPHA	alpha_osf1
fujitsu	vp	uxpm	frt -Ab	inst_FUJITSU	
hp	pa_risc	hp_ux	f77	inst_HP	hp700_ux90
ibm	RS6000	AIX	xlf	inst_RS6	rs_aix32
intel	paragon	OSF_1	f77	inst_PARAGON	paragon_12
sg	iris3d	IRIX	f77		
	iris4d	IRIX_3.2	f77	inst_IRIS4	sgi_40
		IRIX_5.1	f77	inst_IRIS4_51	sgi_52
sun	sun3	SunOS_3.5	f77		
		SunOS_4.0.3	f77		
		SunOS_4.1	f77		sun3_411
	sun4	SunOS_3.5	f77		
		SunOS_4.0.3	f77		
		SunOS_4.1	f77	inst_SUN4	sun4_411
		SunOS_52	f77		sun4_52

Tab. 2.1: Plattformen des DFN-RPCs

Bezeichnungen in den inst-Files			Fortran-	mitgelieferte	**Zusätzliche**
MANU	ARCH	OPER	Compiler	inst_...Files	**Optionen**
bel.	bel.	beliebig	fc		F2C_CC=Pfad von f2c und fc
sg	iris4d	IRIX_3.2	fc	inst_IRIS4F2C	F2C_CC= /client/bin
convex	C120	ConvexOS_V7.1	fc	inst_CVXU	ULTRA=true
sg	iris4d	IRIX_3.2	f77	inst_IRIS4U	ULTRA=true
cray	cray2	unicos_6.0	cf77		ULTRA=true
sun	sun3	SunOS_4.0.3	f77	inst_SUN3O	OSI=sunnet
sun	sun4	SunOS_4.1	f77		OSI=sunnet
cray	cray2	unicos_6.0	cf77		OSI=cray
cray	cray2	unicos_7.0	cf77	inst_CRAY2	OSI=cray2

Tab. 2.2: Optionale Plattformen

Zusätzlich werden noch Installationsfiles folgender Formen mitgeliefert:

inst_...P. Es ist PHIGS=true gesetzt.

[1] Rechner mit Convex-Native-Realzahldarstellung als Default

[2] Rechner mit IEEE-Realzahldarstellung als Default

Konfigurationsname in dfnrpc_set	Installationsparametersatz	Konfigurations-directory	Alle Files werden bei der Installation abgelegt in dem Teilbaum:
ALLI	inst_ALLI	alliant	.../dfnrpc/1.0/sys/alliant
ALLID	inst_ALLID	alliant	.../dfnrpc/1.0/sys_debug/alliant
ALPHA	inst_ALPHA	alpha	.../dfnrpc/1.0/sys/alpha
ALPHAD	inst_ALPHAD	alpha	.../dfnrpc/1.0/sys_debug/alpha
ALPHA_sw	inst_ALPHA_sw	alpha	/sw/axp_osf1/dfnrpc-1.0.60
APO	inst_APO	apollo	.../dfnrpc/1.0/sys/apollo
APOD	inst_APOD	apollo	.../dfnrpc/1.0/sys_debug/apollo
CRAY2	inst_CRAY2	cray2	.../dfnrpc/1.0/sys/cray2
CRAY2D	inst_CRAY2D	cray2	.../dfnrpc/1.0/sys_debug/cray2
CRAYYMP	inst_CRAYYMP	cray_ymp	.../dfnrpc/1.0/sys/cray_ymp
CRAYYMPD	inst_CRAYYMPD	cray_ymp	.../dfnrpc/1.0/sys_debug/cray_ymp
CRAYYMP_sw	inst_CRAYYMP_sw	cray_ymp	/sw/c90_uni8/dfnrpc-1.0.60
CVX	inst_CVX	convex	.../dfnrpc/1.0/sys/convex
CVXD	inst_CVXD	convex	.../dfnrpc/1.0/sys_debug/convex
CVXI	inst_CVXI	convex_ieee	.../dfnrpc/1.0/sys/convex_ieee
CVXID	inst_CVXID	convex_ieee	.../dfnrpc/1.0/sys_debug/convex_ieee
CVXI_sw	inst_CVXI_sw	convex_ieee	/sw/convex_os10/dfnrpc-1.0.60
CVXU	inst_CVXU	convexultra	.../dfnrpc/1.0/sys/convexultra
CVXUD	inst_CVXUD	convexultra	.../dfnrpc/1.0/sys_debug/convexultra
DEC	inst_DEC	dec	.../dfnrpc/1.0/sys/dec
DECD	inst_DECD	dec	.../dfnrpc/1.0/sys_debug/dec
DEC	inst_DECP	dec	.../dfnrpc/1.0/sys/dec
DECD	inst_DECPD	dec	.../dfnrpc/1.0/sys_debug/dec
DEC_sw	inst_DEC_sw	dec	/sw/pmax_u143/dfnrpc-1.0.60
FUJITSU	inst_FUJITSU	fujitsu	.../dfnrpc/1.0/sys/fujitsu
FUJITSUD	inst_FUJITSUD	fujitsu	.../dfnrpc/1.0/sys_debug/fujitsu
HP	inst_HP	hp	.../dfnrpc/1.0/sys/hp
HPD	inst_HPD	hp	.../dfnrpc/1.0/sys_debug/hp
HP_sw	inst_HP_sw	hp	/sw/hp700_ux90/dfnrpc-1.0.60
IRIS4	inst_IRIS4	iris4	.../dfnrpc/1.0/sys/iris4
IRIS4D	inst_IRIS4D	iris4	.../dfnrpc/1.0/sys_debug/iris4
IRIS4F2C	inst_IRIS4F2C	iris4f2c	.../dfnrpc/1.0/sys/iris4f2c
IRIS4DF2C	inst_IRIS4DF2C	iris4f2c	.../dfnrpc/1.0/sys_debug/iris4f2c
IRIS4	inst_IRIS4P	iris4	.../dfnrpc/1.0/sys/iris4
IRIS4D	inst_IRIS4PD	iris4	.../dfnrpc/1.0/sys_debug/iris4
IRIS4U	inst_IRIS4U	iris4ultra	.../dfnrpc/1.0/sys/iris4ultra
IRIS4UD	inst_IRIS4UD	iris4ultra	.../dfnrpc/1.0/sys_debug/iris4ultra
IRIS4U	inst_IRIS4UP	iris4ultra	.../dfnrpc/1.0/sys/iris4ultra
IRIS4UD	inst_IRIS4UPD	iris4ultra	.../dfnrpc/1.0/sys_debug/iris4ultra
IRIS4_51	inst_IRIS4_51	iris4_51	.../dfnrpc/1.0/sys/iris4_51
IRIS4_51D	inst_IRIS4_51D	iris4_51	.../dfnrpc/1.0/sys_debug/iris4_51
IRIS4_51_sw	inst_IRIS4_51_sw	iris4_51	/sw/sgi_51/dfnrpc-1.0.60
IRIS4_sw	inst_IRIS4_sw	iris4	/sw/sgi_40/dfnrpc-1.0.60
LINUX	inst_LINUX	linux	.../dfnrpc/1.0/sys/linux
LINUXD	inst_LINUXD	linux	.../dfnrpc/1.0/sys_debug/linux
LINUX_sw	inst_LINUX_sw	linux	/sw/linux_1/dfnrpc-1.0.60
PARAGON	inst_PARAGON	paragon	.../dfnrpc/1.0/sys/paragon
PARAGOND	inst_PARAGOND	paragon	.../dfnrpc/1.0/sys_debug/paragon
PARAGON_sw	inst_PARAGON_sw	paragon	/sw/paragon_12/dfnrpc-1.0.60
RS6	inst_RS6	ibmrs6	.../dfnrpc/1.0/sys/ibmrs6
RS6D	inst_RS6D	ibmrs6	.../dfnrpc/1.0/sys_debug/ibmrs6
RS6_sw	inst_RS6_sw	ibmrs6	/sw/rs_aix32/dfnrpc-1.0.60
SUN3O	inst_SUN3OP	sun3osi	.../dfnrpc/1.0/sys/sun3osi
SUN3OD	inst_SUN3OPD	sun3osi	.../dfnrpc/1.0/sys_debug/sun3osi
SUN3O	inst_SUN3O	sun3	.../dfnrpc/1.0/sys/sun3osi
SUN3OD	inst_SUN3OD	sun3osi	.../dfnrpc/1.0/sys_debug/sun3osi
SUN4	inst_SUN4	sun4	.../dfnrpc/1.0/sys/sun4
SUN4D	inst_SUN4D	sun4	.../dfnrpc/1.0/sys_debug/sun4
SUN4O	inst_SUN4OP	sun4osi	.../dfnrpc/1.0/sys/sun4osi
SUN4OD	inst_SUN4OPD	sun4osi	.../dfnrpc/1.0/sys_debug/sun4osi
SUN4	inst_SUN4P	sun4	.../dfnrpc/1.0/sys/sun4
SUN4D	inst_SUN4PD	sun4	.../dfnrpc/1.0/sys_debug/sun4
SUN4_52_sw	inst_SUN4_52_sw	sun4	/sw/sun4_52/dfnrpc-1.0.60
SUN4_sw	inst_SUN4_sw	sun4	/sw/sun4_411/dfnrpc-1.0.60
VAX	inst_VAX	vax	.../dfnrpc/1.0/sys/vax
VAXD	inst_VAXD	vax	.../dfnrpc/1.0/sys_debug/vax
VAX_sw	inst_VAX_sw	vax	/sw/vax_u14/dfnrpc-1.0.60
VMS	inst_VMS	vms	.../dfnrpc/1.0/sys/vms
VMSD	inst_VMSD	vms	.../dfnrpc/1.0/sys_debug/vms

Tab. 2.3: Mitgelieferte Installationsparametersätze

inst_...D	Alle Files werden mit der Debug-Option übersetzt.
inst_...all	Die Beispiele werden nicht nur mit Transferkodierung ieee, sondern auch mit dec, vax und cray compiliert.
inst_...sw	Installation in /sw, s. u. „Zentrale Installation“. Hierbei werden feste Pfade in die makefiles eingetragen. CONF muß hierzu mit / beginnen und darf höchstens 32 Zeichen lang sein.

Installationsfile für **VMS** auf Anfrage.

2.5 Der eigene Installationsparametersatz

Auf der Basis eines der mitgelieferten Installationsparametersätze ist ein eigener zu schreiben. Die in den mitgelieferten gesetzten Optionen können sich mit jedem Release ändern. Daher sollten diese nicht geändert oder direkt benutzt werden. Hierbei sind im wesentlichen zwei Namen frei wählbar:

Konfigurationsname	Der Konfigurationsname wird im weiter unten beschriebenen Shell-Script **dfnrpc_set** benützt, um durch die Dateistruktur der eigenen Installation navigieren zu können. Der Filename des Installationsparametersatzes ist meist (aber nicht immer) **inst_**Konfigurationsname.
Konfigurations-Directory	a) Der hier angegebene Directoryname beginnt <u>nicht</u> mit / , d.h. er ist ein **relativer Pfad**: Unter .../dfnrpc/1.0 werden beim Installieren des DFN-RPC die Directories .../dfnrpc/1.0/**sys** und/oder .../dfnrpc/1.0/**sys_debug** angelegt. In diesen wird für jede Konfiguration, die man installiert, eine Subdirectory mit dem hier gewählten Namen angelegt. Es handelt sich hierbei um eine sog. **lokale Installation**. b) Der hier angegebene Directoryname beginnt mit / , d.h. er ist ein **absoluter Pfad**: Unter diesem Pfad werden die Directories bei der Installation angelegt. Es handelt sich hierbei um eine sog. **zentrale Installation**.

Die mitgelieferte Konfigurationen sind in Tab. 2.3 auf Seite 7 aufgelistet.

Beispiel inst_SUN4

```
#!/bin/csh
#
#    inst_SUN4   -   DFN-RPC Installation Parameter File
#    ---------------------------------------------------
#
setenv DFNRPCROOT `cd ../../..; pwd | sed -e 's|^/tmp_mnt||'`
setenv CONF sun4

setenv MANU sun
setenv ARCH sun4
setenv OPER SunOS_4.0.3
setenv ULTRA false
setenv ONLYSRV false
setenv PHIGS true
```

```
setenv OSI false

setenv TRANS_ieee true
setenv TRANS_cray false
setenv TRANS_dec  false
setenv TRANS_vax  false
setenv SYS_OPTIM true
setenv SYS_DEBUG false

setenv COMP_SYS  true
setenv COMP_EXA  true

umask  2
/bin/csh install
```

ACHTUNG: Am Ende einer Zeile dürfen keine zusätzlichen Leerzeichen stehen!

Der Installationsparametersatz im einzelnen:

- **Filename inst_***SUN4*
  Der zweite Teil des Filenamens ist der sog. *Konfigurationsname*. Er ist frei wählbar.
- **#!/bin/csh**
  Diese Zeile erzwingt, daß dieses Script von einer C-Shell (csh) ausgeführt wird. Diese Zeile sollte nicht geändert werden.
- **# inst_***SUN4* **- DFN-RPC Installation Parameter File**
  Der Konfigurationsname sollte in diese Kommentarzeile eingetragen werden.
- **setenv DFNRPCROOT 'cd ../../..; pwd | sed -e 's|^/tmp_mnt||''**
  Hiermit wird aus der current directory die root diretory des DFN-RPC extrahiert, hier z.B. */usr/3rdparty/dfn*. Mit diesem Kommando müssen sämtliche Prefixe eines Automounters entfernt werden. Als Default wird */tmp_mnt* – falls vorhanden – entfernt.
- **setenv CONF** *sun4*
  Wahl der Konfigurations-Directory: Alle bei der Installation entstehenden Files werden in diese Directory gelegt, d.h. hier in /usr/3rdparty/dfn/dfnrpc/1.0/sys/*sun4* oder in /usr/3rdparty/dfn/dfnrpc/1.0/sys_debug/*sun4*. Der Name der Konfigurations-Directory kann beliebig gewählt werden. Es kann auch ein Teilbaum sein, z.B. *sun/4/osi* (dies geht nur bei MANU=cray, sg, sun).

Für eine zentrale Installation muß ein absoluter Pfad angegeben werden (s.a. Kapitel 2.10 auf Seite 14) Nachfolgende Zeilen hängen von der realen Rechnerkonfiguration ab:

- **setenv MANU** *sun*
  Manufacturer: *alliant, apollo, convex, cray, dec, ibm, sg* (Silicon Graphics), *sun*
- **setenv ARCH** *sun4*
  Architektur: siehe untenstehende Tabelle.
- **setenv OPER** *SunOS_4.0.3*
  Operating System: siehe untenstehende Tabelle.
- **setenv ULTRA** *false*
  Das ULTRA Flag gibt an, ob die ULTRA Compatibility Library benützt werden soll: *true* oder *false*. *true* ist nur möglich, wenn die ULTRA Compatibility Library installiert ist und **OPER** = Concentrix_5.0, ConvexOS_V7.1,UNICOS_5.0 bis 7.0, IRIX_3.2, SunOS_4.03 oder SunOS_4.1 ist.

- **setenv ONLYSRV** *false*

  Das ONLY-Compute-Server Flag: Bei Compute Servern, d.h. bei Systemen ohne eigene Graphik-Konsole, sollte dieses Flag auf *true* gesetzt werden (sonst *false*). Bei *true* werden die Visualisierungs-Blöcke der Beispiel-Programme nicht generiert. Diese Option ist obsolete und sollte auf *false*gesetzt werden.

- **setenv PHIGS** *false*

  Das PHIGS Flag gibt an, ob PHIGS installiert ist: *false* oder *true*. Diese Option ist nur relevant für die mitgelieferten Beispiele cav, phi, ph1, ph2. Diese werden nicht mehr gewartet und auch nicht mehr bei der Installation automatisch übersetzt.

  true ist nur möglich bei **OPER** = SR10.2, Ultrix, AIX, IRIX, IRIX_3.2, SunOS_3.5 bis SunOS_4.1 Bei SUN wird hierzu SunPHIGS 1.2 benötigt.

- **setenv OSI** *false*

  Das OSI Flag gibt an, welche OSI-Software installiert ist:

false	keine OSI-Software
sunlink	SunLink OSI 6.0 od. kompatibel (nur bei SunOS_4.03 und SunOS_4.1 möglich)
sunnet	SunNet OSI 7.0 oder kompatibel (nur bei SunOS_4.03 und SunOS_4.1 möglich)
cray	CRAY OSI Rel. 1.0 (nur bei **OPER** = UNICOS_6.0 möglich)
cray2	CRAY OSI Rel. 2.0 (nur bei **OPER** = UNICOS_7.0 möglich)

  Der OSI-Protokollstack wird nicht mehr gewartet.

- **setenv F2C_CC** *unused*

  Das F2C_CC Flag gibt an, ob statt des proprietären Fortran-Compilers der Public-Domain-Compiler *f2c* benutzt werden soll.
  Mögliche Werte sind:
  unused oder Weglassen dieser Zeile:
  der proprietäre Fortran-Compiler des Rechnerherstellers wird benutzt
  Directory-Angabe, z.B. /usr/local/bin:
  Pfad, unter dem *f2c* und *fc* liegen. Siehe auch Kapitel 2.11.6 auf Seite 17.

Nachfolgende Flags geben an, welche Zweige für die Generierung (kompilieren und binden) vorbereitet werden sollen, d.h. welche Directories angelegt werden sollen. In diese Directories werden entsprechende makefiles gelegt.

- **setenv TRANS_ieee** *true*

  Bei *true* werden die Beispiel-Directories und -Makefiles für die Transferkodierung *ieee* angelegt, d.h. in unserem Beispiel wird
  */usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/examples_***ieee** angelegt.

  Bei *false* wird diese Transfer-Kodierung bei der Generierung der Beispiele ignoriert.

- **setenv TRANS_cray** *false*

  dito. für Transfer-Kodierung cray, d.h. für die Beispiele würde bei *true* die Directory
  ..._**cray** angelegt.

- **setenv TRANS_dec** *false*

  dito. für Transfer-Kodierung dec

- **setenv TRANS_vax** *false*

  dito. für Transfer-Kodierung vax

- **setenv SYS_OPTIM** *true*

  true = Generierung soll *ohne* Debug-Option in der Directory
  */usr/3rdparty/dfn/dfnrpc/1.0/***sys***/sun4* stattfinden.

- **setenv SYS_DEBUG** *false*

  true = Generierung soll *mit* Debug-Option in der Directory
  /usr/3rdparty/dfn/dfnrpc/1.0/**sys_debug**/*sun4* stattfinden.

Nachfolgende Flags geben an, welche Teile der Software beim Installationslauf generiert (d.h. kompiliert und gebunden) werden sollen:

- **setenv COMP_SYS** *true*
  DFN-RPC System Compilation Flag: *true* oder *false*
  Bei *true* werden folgende makefiles aufgerufen:

  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/include; make
  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/lib; make
  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/bin; make

  (sofern SYS_DEBUG, bzw. SYS_OPTIM gesetzt sind)
- **setenv COMP_EXA** *true*
  Examples Compilation Flag: *true* oder *false*
  Bei *true* werden die makefiles der mitgelieferten Beispiele aufgerufen:
  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/examples_ieee/ua1a; make
  make
  ...
  (sofern SYS_DEBUG, bzw. SYS_OPTIM, sowie TRANS_ieee, ... gesetzt sind)
- **umask** *2*
  Mit umask *2* werden alle generierten Files für Sie und Ihre Gruppe schreib- und lesbar, sowie für die Welt lesbar.
- **/bin/csh install**
  Hiermit wird das eigentliche Installations-Script gestartet. Diese Zeile sollte nicht geändert werden.

Bei den mitgelieferten Konfigurationsparametersätzen sind folgende Parameter gesetzt:

Konfigurations-	TRANS_				SYS_		COMP_	
parametersatz	ieee	cray	dec	vax	OPTIM	DEBUG	SYS	EXA
inst ...	true	false	false	false	true	false	true	true
inst ... D	true	false	false	false	false	true	false	false

Die Optionen MANU und ARCH beziehen sich auf die Hardware (bei SparcStations wähle man ARCH = sun4). Für die Option OPER wähle man im Zweifelsfall den untersten der bei dieser Architektur zulässigen Werte. Die weiteren Optionen (ULTRA, ONLYSRV, PHIGS und OSI) sollten im Zweifelsfall immer mit *false* belegt werden.

2.6 Navigieren durch die DFN-RPC Software

Vor allem beim Installieren ist ein leichtes Navigieren durch die Directorybäume der Software angenehm. Hierzu muß die eigene Konfiguration im Shell-Script dfnrpc_set in der Directory */usr/3rdparty/dfn*/dfnrpc/1.0/src eingetragen werden.

Ca. Zeile 119-122: Ein großgeschriebener Name für die eigene Konfiguration sollte in die erste der folgenden Zeilen eingetragen werden:

```
#4       SUN4             (or another configuration, see below.)
#4    Available configurations with optimized executables:
#4     ALLI ALPHA APO CVX CRAY2 DEC FUJITSU HP IRIS3 IRIS4 PARAGON
#4     RS6 SUN3 SUN4 VAX
```

Ca. Zeile 171-122: Die oben erfundene Konfiguration muß analog der folgenden Zeilen *programmiert* werden, wobei die hier angegebene Directory (z.B. *sun4*) identisch sein muß mit der Wahl bei **setenv CONF** *sun4* im eigenen, oben vorbereiteten Installations-Script.

Bei lokaler Installation (relative Pfadangabe in CONF) ist DFNRPCCONF einzusetzen, während bei zentraler Installation (absoluter Pfad in CONF) DFNRPCCENTRAL zu benutzen ist.

```
# The following lines represent a choice of different configurations.
alias ALLI     DFNRPCCONF sys/alliant
alias ALLID    DFNRPCCONF sys_debug/alliant
alias ALPHA    DFNRPCCONF sys/alpha
alias ALPHAD   DFNRPCCONF sys_debug/alpha
alias ALPHA_sw DFNRPCCENTRAL /sw/axp_osf1/dfnrpc-1.0.60
   ...
alias SUN4     DFNRPCCONF sys/sun4
alias SUN4D    DFNRPCCONF sys_debug/sun4
```

Folgende Kommandos können dann in den .cshrc File eingetragen und mit source .cshrc gestartet werden:

```
setenv DFNRPCROOT /usr/3rdparty/dfn
setenv DFNRPCREL   1.0
setenv DFNRPCTRANS ieee
source $DFNRPCROOT/dfnrpc/$DFNRPCREL/src/dfnrpc_set
SUN4     (bzw. die Konfiguration, die Sie oben etabliert haben)
```

Sobald diese 5 Zeilen ausgeführt sind, kann man sich mit dem Kommando *dfnh* die Funktionsweise dieses Scripts erläutern lassen (s. a. Kap.4.11 auf Seite 46).

2.7 Installation einer Konfiguration

Man gebe folgende Kommandos ein:

```
dfns       oder    cd /usr/3rdparty/dfn/dfnrpc/1.0/src
inst_SUN4  bzw.    inst_<Konfigurationsname>
```

Wenn im Installationsparametersatz **inst_***Konfigurationsname* **COMP_SYS** und **COMP_EXA** *true* sind, ist die Installation fertig.

Bei **COMP_SYS false** müssen anschließend von Hand folgende makefiles aufgerufen werden.

Hierzu müssen folgende Environment Variablen gesetzt sein, DFNRPCCONF, DFNRPCSRC. Wenn man dfnrpc_set und seine Konfiguration (wie im letzten Kapitel beschrieben) aufgerufen hat, dann sind diese Variablen richtig gesetzt. Man könnte sie auch von Hand setzen. In unserem Beispiel mit:

```
setenv DFNRPCROOT     /usr/3rdparty/dfn
setenv DFNRPCREL      1.0
setenv DFNRPCTRANS    ieee
setenv DFNRPCSRC      $DFNRPCROOT/dfnrpc/$DFNRPCREL/src
setenv DFNRPCCONF     $DFNRPCROOT/dfnrpc/$DFNRPCREL/sys/sun4
                                              oder  sys_debug/sun4
```

Anschließend kann man die makefiles starten, indem man in die entsprechende Directory geht und *make* aufruft:

```
dfni   oder   cd $DFNRPCINCLUDE
make

dfnl   oder   cd $DFNRPCLIB
make

dfnb   oder   cd $DFNRPCBIN
make
```

Bei **COMP_EXA false** können die Beispiele nachträglich von Hand mittels *make* kompiliert werden:

```
sumb  oder   cd $DFNRPCEXAMPLES/sum
make

csumb  oder   cd $DFNRPCEXAMPLES/csum
make

usw.
```

2.8 Einfahren von Fehlerkorrekturen

Vorläufig werden die meisten Fehlerkorrekturen dadurch vorgenommen, daß ein Sourcefile ersetzt wird. Anschließend sind sämtliche *make*-Aufrufe, wie sie im letzten Kapitel beschrieben sind, nochmals durchzuführen.

2.9 Installation auf einem Fileserver

Da für jede Installation eine eigene Directory eröffnet wird, ist es möglich mehrere Installationen in einer DFNRPCROOT Directory auf einem Fileserver abzuspeichern. Im einfachsten Fall liegt DFNRPCROOT schon auf einem Fileserver und der Zugriff auf diese Directory geschieht von allen beteiligten Rechnern aus mittels NFS.[3] Der Vorteil liegt darin, daß man den Sourcecode des DFN-RPC nur einmal auf der Platte liegen hat und somit Fehlerkorrekturen einfacher eingefahren werden können.

[3] Auf diese Art ist das ganze DFN-RPC System bei den Entwicklern auf die verschiedenen Workstations und Compute Server portiert worden.

2.10 Zentrale Installation

Nutzung des DFN-RPCs über NFS bzw. AFS

Seit einiger Zeit existiert am RUS ein Konzept zur einheitlichen Verwaltung und Bereitstellung von Softwarepaketen. Dadurch wird der Aufwand zur Wartung von zusätzlich zum Betriebssystem installierter Software verringert. Dieses Kapitel beschreibt wie das DFN-RPC Tool mittels NFS bzw. AFS über dieses einheitliche /sw-Installationskonzept genutzt werden kann. Die verfügbaren Architekturen und die zugehörige Methode NFS oder AFS kann man untenstehender Tabelle entnehmen. [6, 7]

Für hier nicht genannte Architekturen besteht selbstverständlich weiterhin die Möglichkeit, den DFN-RPC lokal zu installieren.

Der Unterschied in der Nutzung mittels NFS oder AFS besteht lediglich in der Art und Weise wie die vorhande Software auf den Rechnern verfügbar gemacht wird. Die Nutzung der Software ist dann gleich. Im folgenden sind die Schritte beschrieben wie ein Workstation-Verwalter den DFN-RPC, sowie die über das SW-Installationskonzept verfügbare Software, auf seinem Rechner nutzbar machen kann.

via	<arch>	Hardware/Operating System
AFS	hp700_ux90	HP 9000 Series 700 running HP-UX 9.0
AFS	pmax_ul43	DECStation 2100, 3100 or 5000 (single processor) with Ultrix 4.3
NFS	axp_osf1[4]	DEC/Alpha running OSF/1 T2.0
AFS	vax_ul4	VAX systems running Ultrix 4.2 and 4.3
AFS	rs_aix32	IBM RS 6000/AIX 3.2, 3.2.1, 3.2.
AFS	sun4_411	Sun Sparc/Sun OS 4.1.1, 4.1.2, 4.1.3
AFS	sun4_52	Sun SparcStations running Solaris 2.3
NFS	sgi_40	Silicon Graphics/IRIX 4.0.x
AFS	sgi_52	Silicon Graphics running IRIX 5.2
NFS	c90_uni8	Cray-C90 running Unicos 8.0.x
NFS	paragon_12	Intel Paragon running OSF/1 Rel. 1.2
NFS[5]	convex_os10	ConvexOS, Rel. 10.1

Zur Nutzung mittels NFS ist folgender Eintrag in /etc/fstab erforderlich:

sw.rus.uni-stuttgart.de:/sw_nfs/sw/ ␣/sw␣nfs␣ro,intr,soft,timeo=50,retrans=2,bg␣0␣0

Zum Mounten benötigt man außerdem:

mkdir ␣ /sw
mount ␣ -vat ␣ nfs

Nutzung mittels AFS:

ln ␣ -s ␣ /afs/rus/uni-stuttgart.de/sw ␣ /sw

Das weitere Vorgehen ist für **NFS und AFS** identisch

ln ␣ -s ␣ /sw/clients/<arch> ␣ /client

Für <arch> muß die jeweils richtige Architektur eingetragen werden

[4] wird voraussichtlich umbenannt in alpha_osf1

[5] Nicht über sw.rus.uni-stuttgart.de, sondern vom ZDV, Uni. Tübingen zu beziehen

Die Software des DFN-RPC Tools (und auch jede andere Software, die über das SW-Installationskonzept verfügbar gemacht ist) steht nun im Directory /client/bin zur Verfügung. Die für den DFN-RPC benötigten Libraries bzw. Include-Files stehen im Directory /client/lib bzw. /client/include.

Der Aufruf der DFN-RPC Software erfolgt nun im Gegensatz zu der im Benutzerhandbuch beschriebenen Vorgehensweise nicht über Umgebungsvariablen, sondern direkt. Wenn Sie also /client/bin in der Umgebungsvariablen PATH berücksichtigt haben, können die Programme direkt aufgerufen werden (z. B. dfnrpcgen).

Die Sourcen des DFN-RPCs sind bei der Nutzung mittels AFS/NFS nicht für den normalen Benutzer verfügbar. Dadurch ist ein Navigieren in der DFN-RPC Software wie in Kap. 4.3. des Benutzer Handbuch beschrieben nur eingeschränkt möglich.

Im Directory /client/examples/dfnrpc/examples_ieee befinden sich einige Beispiele für die Verwendung des DFN-RPCs. Die Beispiele sind im Handbuch zum DFN-RPC 1.0, Kap. 9 und hier im Kap. „Beispiele“ auf S. 165 ausführlich beschrieben.

Die Datei /client/doc/dfnrpc-doc/dfnrpc.ps enthält das Handbuch zum DFN-RPC als ausdruckbares Postscript-File. Außerdem finden sich im Directory /client/doc/dfnrpc-doc weitere Informationen und Veröffentlichungen bzgl. des DFN-RPCs in Form von Postscript-Files.

2.10.1 Unterschiede zw. lokaler und zentraler Installation

Eine zentrale Installation wird durchgeführt, wenn im Installationsfile inst_<platform> der Wert von CONF mit Slash „/“ beginnt.

- *dfnrpc_set bei lokaler und zentraler Installation*

Am Start von dfnrpc_set hat sich nichts geändert:

```
setenv DFNRPCROOT /sw/src/dfnrpc-1.0.60
                  (or where you have installed the dfnrpc.)
setenv DFNRPCREL   1.0
setenv DFNRPCTRANS ieee
source $DFNRPCROOT/dfnrpc/$DFNRPCREL/dfnrpc_set
```

Bei einer lokalen Installation ist dann z.B.

```
SUN4
```

oder bei einer beliebigen nicht-debug-installation

```
DFNRPCCONF sys/configuration_directory
```

aufzurufen, wobei die `configuration_directory` der Angabe CONF im inst_...-File entsprechen muß.

Bei einer zentralen Installation ist dagegen

```
DFNRPCCENTRAL configuration_directory      bzw. z.B.
DFNRPCCENTRAL /sw/sgi_40/dfnrpc-1.0.60
```

anzugeben, damit sämtliche Aliase korrekt arbeiten.

Außerdem sind bei einer zentralen Installation die Beispiele in der Directory `$DFNRPCCONF/examples/` zusammengefaßt. Bei einer Installation in /sw zeigt `/client/examples/dfnrpc/` auf diese Directory.

`dfnrpc_set` liegt außerdem in `$DFNRPCCONF/bin`, bzw. in `/client/bin`.

- *Hineinkompilierte Pfade*

Bei der Installation werden an verschiedenen Stellen Zugriffspfade hineinkompiliert. Diese sind bei lokaler und zentraler Installation unterschiedlich.

File	benutzt von	Suchpfad bei lokaler /	zentraler Install.
dfnrpc_stub.inc	INCLUDE-Stat. in den Stubs	current directory	<CONF>/include
dfnrpcm4	dfnrpcgen	current$DFNRPCBIN	<CONF>/bin
rpcmacro_short.m4	dfnrpcm4	cur.$DFNRPCINCLUDE	<CONF>/include
dfnrpc.servers	Routine RPQUCI im Anwendungspr.	current directory	current directory
RPCAGMIB	Monitorprogramme dfnrpcsrv u. Stubs	current directory	current directory
dfnrpc.inc	Kompilieren des Anwendungsprog.	current directory, bzw. gemäß INCLUDE-Statement im Anwendungsprogramm	
libdfnrpc.a	Binden des Anwendungsprog.	current$DFNRPCLIB im Binde-Kommando des Anwenders	<CONF>/lib im Binde-Kommando des Anwenders

<CONF> wird während der Installation ausgewertet und fest hineinkompiliert, während *current* ... erst beim Kompilieren der Anwendung, bzw. beim Start des entsprechenden Programms ausgewertet wird. Bei einer lokalen Installation muß der Anwender dafür sorgen, daß die Environment-Variablen $DFNRPCBIN, $DFNRPCINCLUDE und $DFNRPCLIB immer korrekt gesetzt sind, s. a. [17] S. 18.

Bei einer zentralen Installation im Rahmen des Software-Installations-Konzepts /sw muß der Anwender für <CONF> den Wert /client einsetzen. [6, 7]

2.10.2 Files in /sw – Links in /client

Bei einer /sw-Installation sind sämtliche Files über Links in /client verfügbar:

/client/bin/dfnrpcgen	der Stubgenerator
/client/bin/dfnrpcsrv	der Monitor-MIB-Server
/client/bin/dfnrpcwin	der window-orientierte Monitor
/client/bin/dfnrpc_set	Script zum Navigieren
/client/bin/dfnrpcm4	zur Umwandlung von modifizierten Stubs
/client/bin/dfnrpcagt	(interner Agent zur Kommunikation zw. Stub und dfnrpcsrv)
/client/include/dfnrpc.inc	Include-File mit Fortran-Enumerations
/client/include/dfnrpc_stub.inc	(intern von den Stubs included)
/client/lib/libdfnrpc.a	die DFN-RPC-Library
/client/examples/dfnrpc/examples_ieee	enthält die Beispiele awi, ben, cav, con, crshexa, csum, cvttest, exa, moni, para, ph1, ph2, phi, pipe, rfa, rpcpipe, rshexa, test, ua1a, ua1b, ua1c und ua1d
/client/doc/dfnrpc-doc	Dokumentation zum DFN-RPC.

2.11 Release Notes

Bei folgenden Installationen gibt es es kleine Abweichungen, die zu beachten sind:

2.11.1 Convex

Bei MANU=convex muß man mit getsysinfo feststellen, mit welcher Zahlendarstellung der Rechner arbeitet (native_default oder ieee_default) und entsprechend muß bei der Installation ARCH=C120 oder ARCH=C120ieee angegeben werden, bzw. inst_CVX oder inst_CVXI benutzt werden.

Wenn man den DFN-RPC zusammen mit in C geschriebenen Hauptprogrammen benutzt, dann muß man zur Laufzeit die Environment-Variablen FOR005 und FOR006 setzen, wie dies im Beispiel csum angegeben ist.

2.11.2 DEC RISC ULTRIX

Erst ab „DEC Fortran for Ultrix RISC Systems Version 3.1“ ist der Compiler in der Lage, sämtliche Standard-Fortran 77 Features, die der DFN-RPC einsetzt, korrekt zu übersetzen.

2.11.3 Fujitsu

Bei MANU=fujitsu müssen alle Fortran-Anwendungen und die Stubs mit frt mit der Option -Ab übersetzt werden.

2.11.4 Apollo

Der graphische Monitor und der X-Window-Teil des Beispiels „Test“ können auf Apollo-Rechnern nicht genutzt werden.

2.11.5 Abweichende X11-Installation

Wenn X11 nicht in /usr/lib und /usr/include installiert ist (z.T. bei Sun), dann muß in $DFNR-PCSRC/makes/bin_makefile
und $DFNRPCSRC/examples/test/source_makefile die dann gültige Directory für die X11-Library angegeben werden. Beim Compilieren von monbut.c und testc.c ist ggf. mit -I eine abweichende Include-Directory anzugeben.

2.11.6 *f2c* statt proprietärem Fortran-Compiler

Falls der fc eingesetzt werden soll, ist im Installationsfile folgende Zeile einzufügen:

setenv F2C_CC <Directory, in der fc und f2c stehen>

also z.B.

setenv F2C_CC /usr/local/bin

f2c unterstützt auch INTEGER*1 und LOGICAL*1,2. Dies ist aber nicht dokumentiert. INTEGER*1 hat den Wertebereich 0..255, f77 hat dagegen -128..127. Ein Fehler tritt nur auf, wenn ein INTEGER*1 Wert im Bereich 128 ... 255 benutzt und f2c-compiliert abgeschickt wird, als INTEGER*2/4/8 übertragen wird und beim Empfänger in INTEGER oder INTEGER*2/4/8 abgespeichert wird. Bei der Test-Routine wird dies in den Tests m2 und m3 passieren, wenn test_sb auf einer BYTE-Maschine f2c-compiliert ist und test_cb auf einer Nicht-BYTE-Maschine (z.B. CRAY) läuft.

Defaultmäßig verwendet fc als C-Compiler den gcc von GNU. Der DFN-RPC verwendet einen modifizierten fc, der den proprietären C-Compiler (cc) verwendet.

Anmerkung: Man sollte mindestens das Release f2c-1993.04.28 einsetzen. Der Wert von F2C_CC ist der Pfad von f2c (Konverter) und fc (Compiler = Konverter + gcc).

Bei den in C programmierten Beispielen ist im File dfnrpc/1.0/src/install.m4_h in der Zeile CFLFLAGS = $(FCPATH)/../lib/libf2cF77.a $(FCPATH)/../lib/libf2cI77.a festgelegt, wo die Libraries stehen (s.a. vorvorletzte Zeile von fc).

Falls Ihr f2c die Option -Nl1000 nicht kennt, dann ist dies ebenfalls in install.m4_h zu ändern (ca. 9 Zeilen weiter oben).

Siehe auch Seite 5 und Seite 10.

2.11.7 Abweichende proprietäre Fortran-Compiler

Durch unterschiedliche Hardware kann es sein, daß der Fortran-Compiler u.U. beim Binden auf Ihrem System andere Libraries hinzubindet. Dies muß beim Binden von C-Programmen (wie in den Beispielen csum und crshexa) berücksichtigt werden. Die hierzu benötigten Optionensind in $DFNRPCSRC/install.m4_h für jede Architektur in CFLFLAGS angegeben und müssen dort u.U. geändert werden.

3 sum und csum, zwei Beispiele zur Einführung

3.1 Einleitung

Anhand eines einfachen Beispiels soll auf den folgenden Seiten gezeigt werden, wie eine Anwendung auf zwei Rechner verteilt werden kann.

Die untenstehenden Beispielanwendungen **sum** und **csum** haben die Aufgabe, zwei Zahlen von der Tastatur einzulesen, diese zu addieren und das Ergebnis auszugeben. Die Ein- und Ausgabe soll hierbei auf dem ersten Rechner erfolgen, die Berechnung auf dem zweiten. Das heißt, daß die beiden Zahlen auf Rechner_1 eingelesen werden und anschließend die Subroutine, die die beiden Zahlen addiert, auf Rechner_2 gerufen wird. Der Aufruf der Berechnungsfunktion von Rechner_1 aus auf dem entfernten Rechner_2 ist ein "Remote Procedure Call" und geschieht mittels DFN-RPC.

sum.f:

```
program sum_client
real a,b,result
write(*,*)'please insert first number:'
read(*,*) a
write(*,*)'please insert second number:'
read(*,*) b
call add(a,b,result)
write(*,*)'the sum of ',a,' and ',b,
          ' is ',result
stop
end


subroutine add(a,b,result)
real a, b, result
write(*,*)'computing the sum of ',a,
          ' and ',b
result = a + b
return
end
```

csum_lb.c:

```
void add_(a,b,result)
  float *a,*b,*result;
 {
  printf("...computing the sum of \
          %f + %f...\n",*a,*b);
  *result = *a + *b;
 }

main()
 {
  float a,b,res;

  printf("Please insert first number  : ");
  scanf("%f",&a);
  printf("Please insert second number : ");
  scanf("%f",&b);
  add_(&a,&b,&res);
  printf("The sum of %1.8g plus %1.8g is \
          %1.8g\n",a,b,res);
 }
```

Wie der Name "Remote Procedure Call" sagt, können nur echte Subroutine-Calls über das Netz geschickt werden. Das heißt, daß alle Aufgaben, die auf dem entfernten Rechner ausgeführt werden sollen, in Functions und Subroutinen verpackt sein müssen.

3.2 Vom Local Call zum Remote Procedure Call

Will man ein Programm mit dem DFN-RPC verteilen, so muß man zum einen sein Programm *aufteilen* und zum anderen die entsprechenden *Client-* und *Serverstubs* generieren. Der *Stub-Generator* benötigt hierzu einen *Remote Procedure Interface Definition File*, da die Ausprägung der Stubs von der jeweiligen Anwendung abhängt. Außerdem muß man einige Initialisierungsaufrufe für den *Objekt-Start* (Prozeßstart) hinzufügen.

Für das *Aufteilen der Software in Client- und Server-Part* gibt es zwei Möglichkeiten:

1. Man teilt die Sourcen von Hand oder

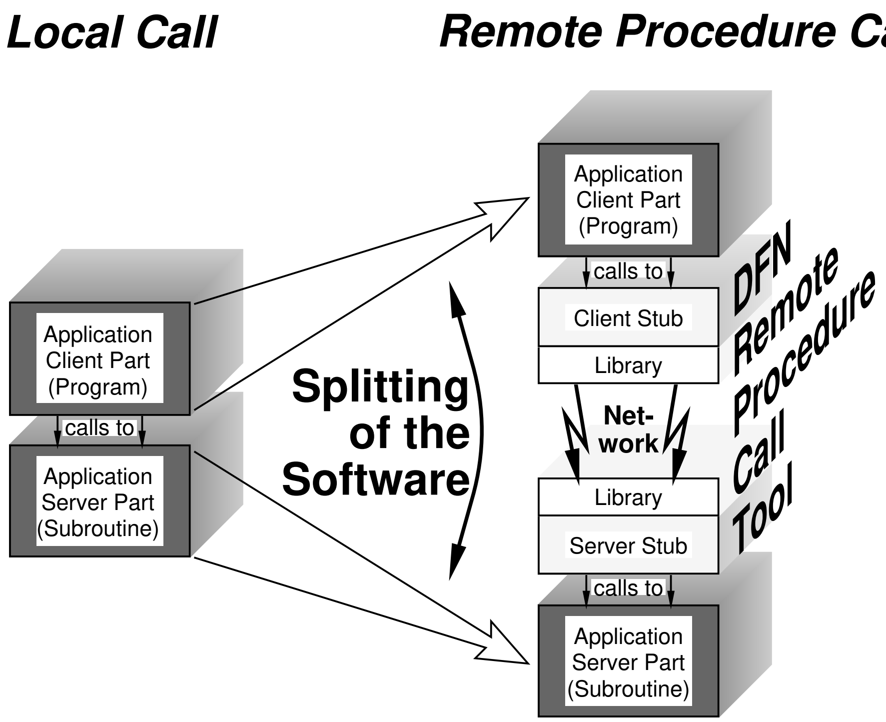


Abbildung 3.1: Vom 'Call' zum 'Remote Procedure Call'

2. man fügt der Source entsprechende Anweisungen hinzu und läßt den Stub-Generator den Client- und Server-Part aus dem so erweiterten Fortran-Source-File herausfiltern.

   Hierbei sollte man gleichzeitig die RPI Definition in die Fortran-Source integrieren. Bei dieser Vorgehensweise wird das Single Source Prinzip erfüllt (s.u.).

Für das Generieren der Stubs benötigt man die *Remote Procedure Interface* (**RPI**) *Definition*, d.h. eine Beschreibung der Schnittstelle, an der die Anwendung verteilt wird. Die RPI Definition kann entweder auf einen gesonderten File geschrieben oder in die Fortran-Source integriert werden (*Single Source Prinzip*), wobei man die zusätzliche Information in die Deklaration der Argumente[1] einstreuen oder auf spezielle Kommentarzeilen schreiben kann.

Der *Stub-Generator* ist das zentrale Werkzeug. Sein Input, ein RPI-File, kann also folgende Informationen enthalten:

- Informationen über die Programmaufteilung,
- Fortran-Quellcode
- RPI Definitionen

und ist im Kapitel 5 auf Seite 53 beschrieben. Folgende Abbildung soll die beiden wesentlichen Möglichkeiten verdeutlichen.

Die Bedienung des Stub-Generators findet man im Kapitel 7 auf Seite 159.

[1] Um Verwechslungen mit dem Fortran PARAMETER Statement zu vermeiden, wird für die formalen und aktuellen Parameter das Wort „Argument“ benutzt, vgl. engl. „dummy argument“

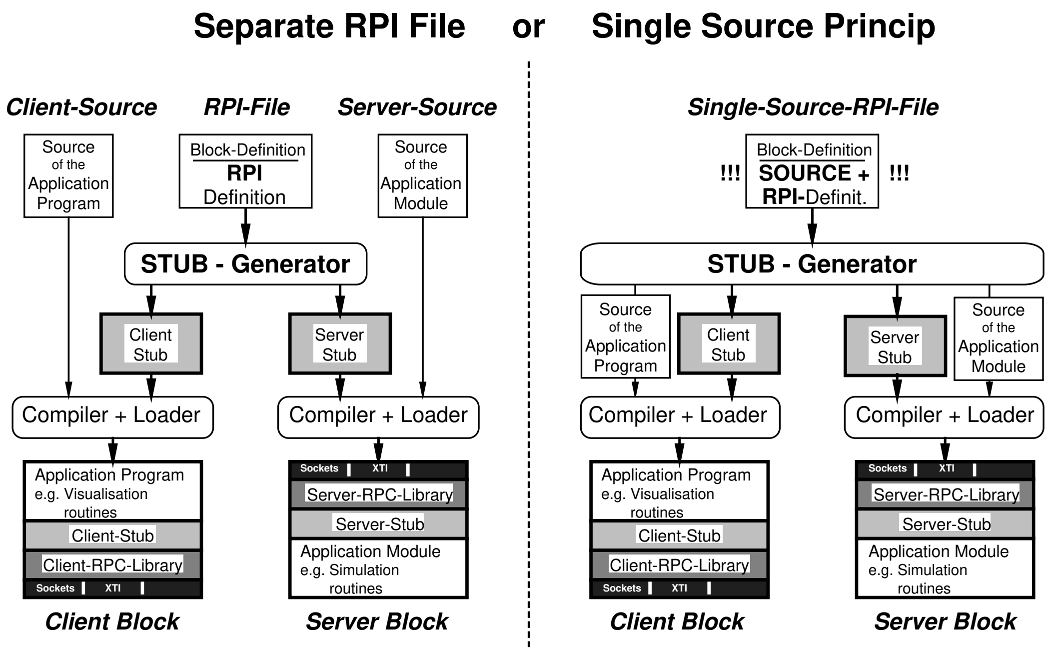


Abbildung 3.2: Gesonderter RPI-File oder Single Source Prinzip

Obwohl der Remote Procedure Call Mechanismus für das Anwendungsprogramm transparent sein soll, kann das damit verbundene *Object Handling* nicht verborgen werden. Der RPC Mechanismus selbst, der Transport-Dienst und die entfernten Server-Prozesse müssen initialisiert werden.

Die entsprechenden Bibliotheksfunktionen sind in den Kap. 4.9 u.6 (S. 43f, 111ff) beschreiben.

3.3 Der Code von sum.rpi

Der erste Rechner ist in diesem Beispiel der **Client**, da auf ihm das Hauptprogramm der Anwendung läuft. Der zweite Rechner ist der **Server**. Auf ihm läuft ein spezielles Serverprogramm, das die vom Client gerufenen Funktionen bearbeitet. Unser Programm gliedert sich folglich in zwei Teile: einen mit dem Code für den Client und einen mit dem für den Server. Entsprechend stellen wir zwei Module zusammen: Das Clientmodul “cm” und das Servermodul “sm”. Zum Clientmodul gehört das Hauptprogramm der Anwendung, zum Servermodul gehören das Serverprogramm, das die vom Client gerufenen Subroutinen empfängt und bearbeitet, und die Subroutinen, die vom Client auf dem Server gerufen werden kann, selbst.

Zur Verteilung der Anwendung muß ein Remote Procedure Interface (RPI) File erstellt werden. Dies kann in Form einer Ergänzung des ursprünglichen Fortran-Source-Files geschehen.

Am Anfang des Files steht der Blockdefinition-Part, in dem die Module zu Blöcken zusammengestellt werden. Diese Informationen werden vom Stubgenerator benötigt. Die beiden Module cm und sm schließen sich an.

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

Im **Client-Block** fügt der Stubgenerator bedingt durch LOCAL den kompletten Code des Clienthauptprogramm-Moduls cm ein. Durch CLIENT wird der Clientstub zu cm eingefügt.

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

Der Serverblock enthält die Source von sm und die Serverstubs.

```
C*[BLK_BEG] lb,     NAME='local block', LOCAL_SITE
C*[MOD_USE]         cm, LOCAL
C*[BLK_END]         sm, LOCAL
```

Der **Local-Block** ist die unverteilte Anwendung und enthält die Sourcen von cm und sm ohne RPC-spezifische Calls.

Die kompilierten Blöcke sind die Ausführbaren (Teil-)Programme der Anwendung. Die Stubs werden vom Stubgene - rator automatisch erzeugt. Sie schicken die RPCs über die Leitung.

```
C*[MOD_BEG] cm,  NAME='sum client module'
      program sum_client
      real a,b,result
      integer iret
```

Beginn des Client-Moduls cm
Beginn des Client-Hauptprogramms

```
C*RPC  integer nserver,sobjid
C*RPC  write(*,*) ' sum_cb - this is the client of a'
C*RPC+   ' distributed fortran program'
C*RPC  nserver=1
C*RPC  call rpquci(nserver,sobjid,iret)
C*RPC  if(iret.ne.0) then
C*RPC    write(*,*)'sum_cb - error in rpc initialisation'
C*RPC    stop
C*RPC  endif
```

rpquci initialisiert den RPC auf Clientseite, startet einen Server und gibt dessen Object-ID *sobjid* (wird bei nur einem Server nicht weiter benötigt) und die Fehlervariable *iret* zurück. Ist *iret*=0, so wurde der RPC erfolgreich initialisiert. RPC-spezifische Zeilen beginnen mit C*RPC und werden so im unverteilten Fall ignoriert.

```
      write(*,*)'please insert first number:'
      read(*,*) a
      write(*,*)'please insert second number:'
      read(*,*) b
      call add(a,b,result)
      write(*,*)'the sum of ',a,' and ',b,' is ',result
```

Einlesen der Argumente für die Routine **add**, die per RPC gerufen wird. Die Aufrufsyntax entspricht der eines lokalen Subroutine-Calls.

```
C*RPC  call rpqucf(iret)
C*RPC  if(iret.ne.0) then
C*RPC    write(*,*)'sum_cb - error in rpc finishing'
C*RPC  else
C*RPC    write(*,*)'sum_cb - finished'
C*RPC  endif
      stop
      end
C*[MOD_END]
```

Die Library-Routine **rpqucf** schließt die Verbindung zum Server und beendet den RPC clientseitig.

Ende des Client-Hauptprogramms
Ende des Clientmoduls

—

```
C*[MOD_BEG] sm,  NAME='sum server_module'
C*RPC  program sum_server
C*RPC  integer iret
C*RPC  character*100 master_address
C*RPC
C*RPC  write(*,*)'sum_sb - this is the server of a '
C*RPC+ ' distributed fortran program '
C*RPC
C*RPC  if(iargc().lt.1) then
C*RPC    master_address= ''
```

Beginn des Servermoduls
Beginn des Server-Hauptprogramms

Der Server benötigt zum Verbindungsaufbau die Adresse des Clients. Wird diese nicht als Kommandozeilenparameter übergeben, so wird **rpqusp** mit einem leeren String als *master_address* gerufen und fragt dann den Anwender nach der Clientadresse.(Die Eingabe der Adresse über

```
C*RPC   else
C*RPC     call getarg(1,master_address)
C*RPC   endif
```

Parameter wird beim automatischen Serverstart benutzt.)

```
C*RPC   call rpqusp(master_address,iret)
C*RPC   if(iret.ne.0) then
C*RPC     write(*,*)'rpc error - server stopped '
C*RPC     stop
C*RPC   endif
C*RPC   write(*,*)'sum_server - rpc correctly finished'
```

rpqusp initialisiert den RPC auf Serverseite, baut die Verbindung zum Client auf, nimmt die RPCs entgegen und baut nach Beendigung des Clients die Verbindung ab.

```
C*RPC   stop
C*RPC   end
```

Ende des Server-Hauptprogramms

—

```
C*[RPI_BEG]
        subroutine add(a,b,result)
          real a
C*RPC+  [IN]
          real b
C*RPC+  [IN]
        real result
C*RPC+  [OUT]
```

Beginn der Interfacedefinition für die RPC-Routine.
Bei der Deklaration der Funktionsparameter muß angegeben werden, ob es sich um Input- oder Outputparameter handelt. Die Angabe erfolgt in einer mit C*RPC+ eingeleiteten Fortsetzungszeile, so daß die Zusatzangaben im unverteilten Fall ignoriert werden.

```
C*[RPI_NON]
        write(*,*)'computing the result of ',a,' and ',b
        result=a+b
        return
        end
```

Mit C*[RPI_NON] wird der Teil des Codes der Subroutine eingeleitet, der vom Stubgenerator nicht gelesen wird, also hier die Berechnung von *result*.

```
C*[RPI_END]
C*[MOD_END]
```

Ende der Interfacedefinition
Ende des Servermoduls

3.4 Bedienung des Beipielprogramms

Für die verteilte Beispielanwendung benötigen Sie zwei Fenster. Starten sie in beliebiger Reihenfolge in einem Fenster das Clientprogramm **sum_cb**[2] und im anderen das Serverprogramm **sum_sb**. **sum_cb** gibt nach dem Starten den String *master's address* aus. Diesen String, der die Rechneradresse des Clients, die Nummer des Ports, der für die Verbindung geöffnet wurde, und das verschlüsselte Passwort enthält, geben Sie nun im Serverprogramm eingegeben ein. Daraufhin wird die Verbindung zwischen Client und Server aufgebaut.

Alternativ kann der Server auch, wie in den Beispielen rshexa und crshexa (Kapitel 9.5, Seite 170), per Remote-Shell rsh gestartet werden.

CLIENT:

```
host1$ sum_cb
sum_cb - this is the client of a distributed
                              fortran program
dfn-rpc - rpquci - quick client initialization
rpquci - please start server no. 1
master's address=
 host1.abc.irgendwo.de:3340:sdfhg3Yzz::
```

SERVER:

```
host2$ sum_sb
sum_sb this is the server of a distributed
                              fortran program
dfn-rpc - rpqusp - quick server program
dfn-rpc - rpqusi - quick server initialization
rpqusi - please type in the master's address:
host1.abc.irgendwo.de:3340:sdfhg3Yzz::
```

[2] $DFNRPCEXAMPLES/sum/sum_cb
bzw. /client/examples/dfnrpc/examples_ieee/sum/sum_cb

```
rpquci - connected with server 1
dfn-rpc - rpquci - client connected to 1 server
please insert first number:
3
please insert second number:
4
                                             computing the sum of 3.0000 and 4.0000
the sum of 3.0000 and 4.0000 is 7.0000
dfn-rpc - rpqucf - releasing all connections  dfn-rpc - rpquci - all rpcs done correctly
dfn-rpc - rpqucf - client correctly finished  dfn-rpc - rpqusf - releasing the connection
sum_cb - finished                             sum_server - rpc correctly finished
host1$                                        host2$
```

3.5 Das Beispiel sum selbst kompiliert

Um das Beispiel zu kompilieren, erzeugen Sie in Ihrem Homeverzeichnis ein Unterverzeichnis mysum. Wechseln Sie in dieses Verzeichnis und kopieren Sie die Datei sum.rpi ins aktuelle Verzeichnis.

Nun wird der Stubgenerator aufgerufen. Er steht im Verzeichnis $DFNRPCBIN.

```
$DFNRPCBIN/dfnrpcgen -T ieee sum
f77 -o sum_lb sum_lb.f
f77 -o sum_cb sum_cb.f sum_cb_stb.f $DFNRPCLIB/libdfnrpc.a
f77 -o sum_sb sum_sb.f sum_sb_stb.f $DFNRPCLIB/libdfnrpc.a
```

Hierbei werden folgende Dateien erzeugt:

sum_lb.f	das unverteilte Programm
sum_cb.f	der Clientteil der Anwendung
sum_cb_stb.f	die Clientstubs zum Remote Procedure Interface
sum_sb.f	der Serverteil der Anwendung
sum_sb_stb.f	die Serverstubs zum Remote Procedure Interface

Anschließend werden die vom Stubgenerator erzeugten Fortrandateien kompiliert. Dabei werden beim Client- und beim Serverblock Programm und zugehörige Stubs zu einer ausführbaren Datei zusammengelinkt. Zusätzlich muß bei diesen beiden Blöcken die DFN-RPC-Library 'libdfnrpc.a' dazugelinkt werden.

3.6 Der Makefile von sum.rpi

Die Makefiles für die Beispielprogramme unterscheiden sich je nach Architektur, Betriebssystem und benutzten Compilern. Sie werden daher bei der Installation des DFNRPC automatisch aus einem maschinenunabhängigen Source-Makefile generiert. Aufgrund dessen enthalten die Originalmakefiles teilweise zusätzliche, nicht benutzte Variablen und Statements. Im folgenden wird daher zur Erläuterung ein auf das Wesentliche verkürzter Makefile verwendet.

```
# makefile for DFN-RPC example "sum"
# =================================
#
# Some hosts prefer SUFFIXES =, others prefer .SUFFIXES: (see below)
SUFFIXES   = .unused
.SUFFIXES:   .unused
```

Statt der eingebauten Regeln des “make” werden sämtliche Abhängigkeiten und Arbeitsschritte explizit im folgenden angegeben.

```
#
# ...commands:
#
# $(FC)     = call to the fortran compiler (predefined macro)
# $(FLD)    = loader for fortran
FC          = f77
FLD         = $(FC)
SHELL       = /bin/sh
RM          = /bin/rm
```

Hier werden die Calls für Compiler und Linker festgelegt. Im Beispiel wird für beides der f77 benutzt. Um Fehler durch vom Anwender umdefinierte Kommandos zu vermeiden, werden für die beiden Aufrufe sh und rm absolute Pfadangaben benutzt.

```
#
# ...directories and files:
#
# From the shell environment the following variables will be imported:
# DFNRPCBIN DFNRPCLIB DFNRPCINCLUDE
#
# or in a central installation they must be defined here, e.g.:
# DFNRPCBIN      = /sw/sgi_40/dfnrpc-1.0.60/bin
# DFNRPCLIB      = /sw/sgi_40/dfnrpc-1.0.60/lib
# DFNRPCINCLUDE  = /sw/sgi_40/dfnrpc-1.0.60/include
# ...flags:
#          -c = only compile, no loader call
#          -g = debug       / -O = optimize (no debug)
FFLAGS  =  -c -g
LFLAGS  =  -g
#          -lbsd  =libraries for TCP
NETLIB  =  -lbsd
DFNRPCTRANS = ieee
```

Hier werden die Flags definiert, mit denen der Compiler und der Linker aufgerufen werden. Durch die Option -c wird beim f77 das Linken unterbunden. Beim Anschließenden Aufruf ohne -c werden die Objektdateien und die Libraries zusammengelinkt. Außerdem werden das Flag für die TCP-Library und die Transferkodierung festgelegt. Wenn die drei benötigten Pfadvariablen nicht gesetzt sind, müssen die entsprechenden Kommentarzeichen entfernt werden.

```
#
# ...targets
#
modules:sum_lb sum_cb sum_sb
#
source:sum_lb.f sum_cb.f sum_sb.f sum_cb_stb.f sum_sb_stb.f
```

Durch Aufruf von **make modules** oder nur **make** werden alle drei Teile der Anwendung erzeugt. Durch **make source** wird nur der Stubgenerator gerufen, der die Source aufspaltet, und die Stubs erzeugt.

```
#
# Client part:
# ===========
#
client: sum_cb
```

```
sum_cb: sum_cb.o sum_cb_stb.o $(DFNRPCLIB)/libdfnrpc.a
        echo "make module      sum_cb.o--->sum_cb"
        $(FLD) $(LFLAGS) -o sum_cb sum_cb.o sum_cb_stb.o \
               $(DFNRPCLIB)/libdfnrpc.a $(NETLIB)
sum_cb.o:sum_cb.f dfnrpc.inc
        echo "make: compilation   sum_cb.f--->sum_cb.o"
        $(FC) $(FFLAGS) sum_cb.f
sum_cb_stb.o:sum_cb_stb.f dfnrpc_stub.inc
        echo "make: compilation   sum_cb_stb.f--->sum_cb_stb.o"
        $(FC) $(FFLAGS) sum_cb_stb.f
```

Kompilieren und Linken des Clientblocks. Dem f77 – im Beispiel als Compiler und Linker benutzt – werden jeweils die entsprechenden Flags mitgegeben. Die beim kompilieren der Source des Client-Anwendungsprogramms sum_cb.f und des Clientstubs sum_cb_stb.f entstehenden Objektdateien werden zusammen mit der DFNRPC-Library und der TCP-Library zum Clientblock zusammengelinkt. Die benötigten Fortran-Sourcen wurden vom Stubgenerator erzeugt (s.u.).

Analog dazu wird der Serverblock erzeugt:

```
# Server part:
# ============
#
server: sum_sb
sum_sb:sum_sb.o sum_sb_stb.o $(DFNRPCLIB)/libdfnrpc.a
        echo "make module      sum_sb.o--->sum_sb"
        $(FLD) $(LFLAGS) -o sum_sb sum_sb.o sum_sb_stb.o \
               $(DFNRPCLIB)/libdfnrpc.a $(NETLIB)
sum_sb.o:sum_sb.f dfnrpc.inc
        echo "make: compilation   sum_sb.f--->sum_sb.o"
        $(FC) $(FFLAGS) sum_sb.f
sum_sb_stb.o:sum_sb_stb.f dfnrpc_stub.inc
        echo "make: compilation   sum_sb_stb.f--->sum_sb_stb.o"
        $(FC) $(FFLAGS) sum_sb_stb.f
#

# Not distributed version:
# ========================
#
local: sum_lb
sum_lb:sum_lb.o
        echo "make module         sum_lb.o--->sum_lb"
        $(FLD) $(LFLAGS) -o sum_lb sum_lb.o
sum_lb.o:sum_lb.f
        echo "make: compilation   sum_lb.f--->sum_lb.o"
        $(FC) $(FFLAGS) sum_lb.f
```

Zur Erzeugung des Localblocks werden einfach nacheinander Compiler und Linker mit den entsprechenden Flags aufgerufen. Source ist sum_lb.f.

```
# Sources :
# =========
#
sum_cb.f sum_cb_stb.f sum_sb.f sum_sb_stb.f sum_lb.f : sum.rpi $(DFNRPCBIN)/dfnrpcgen
        echo "stub generation     sum.rpi --->sum_*.f"
        $(DFNRPCBIN)/dfnrpcgen  -T $(DFNRPCTRANS) sum
```

Hier wird der Stubgenerator dfnrpcgen gerufen, um aus dem RPI-File die Fortran-Sourcefiles zu erzeugen. Als Parameter wird die oben festgelegte Transferkodierung mitgegeben.

Die hierbei in der ersten Zeile angegebene Abhängigkeit von dfnrpcgen garantiert, daß bei Änderungen am Stubgenerator (z.B. neues Release mit optimiertem Kommunikationsprotokoll), die Stubs neu generiert werden.

```
dfnrpc.inc : $(DFNRPCINCLUDE)/dfnrpc.inc
        echo "make: get source    dfnrpc.inc"
        -$(RM) -f dfnrpc.inc ; echo reset_errorflag >> /dev/null
        ln -s $(DFNRPCINCLUDE)/dfnrpc.inc dfnrpc.inc
dfnrpc_stub.inc : $(DFNRPCINCLUDE)/dfnrpc_stub.inc
        echo "make: get source    dfnrpc_stub.inc"
        -$(RM) -f dfnrpc_stub.inc ; echo reset_errorflag >> /dev/null
        ln -s $(DFNRPCINCLUDE)/dfnrpc_stub.inc dfnrpc_stub.inc
```

Vor dem Kompilieren müssen im aktuellen Verzeichnis Links auf die beiden Includefiles dfnrpc.inc und dfnrpc_stub.inc gesetzt werden. Eventuell vorhandene alte Links auf diese Dateien werden vorher gelöscht.

```
.SILENT:
```

Nur die Erläuterungen per “echo” und die Ausgaben der Compiler etc. erscheinen auf dem Bildschirm.

3.7 csum – das selbe in C

Das Beispiel **csum** entspricht in seiner Funktionalität dem Beispiel **sum**, ist jedoch in C geschrieben. Beim Einsatz des DFN-RPC für C-Programme muß die RPI-Definition, bestehend aus dem Block-Definition-Part und der Schnittstellendefinition für die per RPC gerufenen Routinen, in einem separaten File mit der Endung '.rpi' stehen. Der Code für die einzelnen Blöcke steht dann jeweils in einem eigenen File. Die Source von **csum** besteht also aus **csum.rpi**, **csum_cb.c**, **csum_sb.c** und **csum_lb.c**.

Die Fortran-Routinennamen aus der RPI-Definition müssen im Programmcode auf C-Routinennamen umgesetzt werden. Dies geschieht im Beipiel durch Anhängen eines Underline-Zeichens an den (kleingeschriebenen) Funktionsnamen.

csum.rpi

```
C*[BLK_BEG] cb,NAME='csum client block',CLIENT _SITE
C*[MOD_USE] sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb,NAME='csum server block',SERVER _SITE
C*[MOD_USE] sm, SERVER
C*[BLK_END]


C*[MOD_BEG] sm, NAME='csum server routines'
C*[RPI_BEG]
 subroutine add (a,b,result)
 real a [IN]
 real b [IN]
 real result [OUT]

C*[RPI_END]
C*[MOD_END]
```

Durch CLIENT werden die Clientstubs zu sm und der Code von csum_cb.c vom Stubgenerator zum Clientblock zusammengefügt.
Analog dazu besteht der Serverblock aus csum_sb.c und den Serverstubs.
Für den Localblock ist keine Definition nötig, da er nur aus csum_lb.c besteht.
Im Module-Definition-Part von sm erfolgt in Fortran die Schnittstellendefinition für die per RPC gerufenen Routinen, bestehend aus den Funktionsheadern mit zugehöriger Variablendeklaration unter Angabe der Transferrichtung.

csum_cb.c

```
main()
 {
   float a,b,res;
   int nserv,iret,sobjid[2];
   printf(csum_cb - This is the client of ");
                 "a distributed C program\n");
   nserv = 1;
   rpquci_(&nserv,sobjid,&iret);
   if (iret != 0)
    {
      printf(csum_cb - error in rpc initialization\n");
      exit(1);
    }
   printf(Please insert first number : ");
   scanf("%f",&a);
   printf(Please insert second number : ");
   scanf("%f",&b);
   add_(&a,&b,&res);
   printf(The sum of %1.8g plus %1.8g is %1.8g\n",a,b,res);
   rpqucf_(&iret);
   if (iret != 0)
    {
      printf(csum_cb - error in rpc finishing\n");
      exit(1);
    }
   printf(csum_cb - finished\n");
   exit(0);
 }
```

– RPC initialisieren
Da **rpquci** eine Fortranroutine ist, muß sie ebenfalls mit nachgestellter Underline aufgerufen werden. Parameter mit Rückgabewert werden als Pointer übergeben.

Die Parameter der RPC-Routine sind mit Call-by-Reference zu übergeben.
– RPC beenden

csum_sb.c

```
main(argc, argv)
int argc; char *argv[];
 {
  int iret;
  int strlng = 1;
  char* str = ;
  printf(csum_sb - This is the server of a ");
                    "distributed C program\n");
  if (argc>1)
   {
    strlng = strlen(argv[1]);
    str = argv[1];
   }
  rpqusp_(str,&iret,strlng);
  if(iret != 0) exit(1);
  printf(csum_sb - finished\n");
 }
void add_(a,b,result)
float *a,*b,*result;
 {
  printf(csum_sb ...computing the sum of %1.8g + "
                                  "%1.8g...\n",*a,*b);
  *result = *a + *b;
 }
```

Im Fall, daß kein Parameter über geben wird, ist der String mit der Masteradresse leer.

Andernfalls wird der String mit der Masteradresse gleich dem Parameter gesetzt und die Länge des Strings bestimmt.

Bei Strings steht an der selben Stele wie in der Fortran-Parameterliste ein Pointer auf den String, am Ende als Call-by-Value die Länge.
Bei der per RPC zu rufenden Subroutine ist zu beachten, daß die Parameter entsprechen dem Funktionsaufruf als Pointer zu deklarieren sind. Ansonsten wird die RPC-Routine wie eine lokale Funktion programmiert.

Anmerkung:
Die oben benutzte Syntax für die Umsetzung der C-Routinen auf die Fortran-Interfaces des DFN-RPC (nachgestellter Underscore) stellt ein Beispiel dar und ist in Wirklichkeit systemabhängig. Die genaue Syntax für ihr System finden Sie im Kapitel über C-Programmierung auf Seite 163.

Auf einer Convex müssen zur Laufzeit der Programme folgende Environmentvariablen gesetzt sein, wenn die Stubs mit dem proprietären Fortrancompiler übersetzt wurden:

setenv FOR005 SYS\$INPUT
setenv FOR006 SYS\$OUTPUT

Kompilieren von csum

Vor dem kompilieren müssen wie bei sum.rpi die Links auf die Includefiles angelegt werden. Die Kompileraufrufe und die Linkerflags sind systemabhängig. Die Linkerflags für Ihr System finden Sie im Kapitel 8 "'C-Programmierung"' auf Seite 163. Den benötigten Fortrankompiler für Ihr System finden Sie in Tab. 2.1 auf Seite 6. Sie können auch ersatzweise den Fortran-to-C-Konverter f2c einsetzen – siehe Kap. 2.11.6 auf Seite 17 und auch Seiten 5 und 10.

```
dfnrpcgen -n csum.rpi

f77 -c csum_cb_stb.f
cc -c csum_cb.c
cc -o csum_cb csum_cb.o csum_cb_stb.o $DFNRPCLIB/libdfnrpc.a \
   -l... (systemabhaengige Linkerflags)

f77 -c csum_sb_stb.f
cc -c csum_sb.c
cc -o csum_sb csum_sb.o csum_sb_stb.o $DFNRPCLIB/libdfnrpc.a \
   -l... (systemabhaengige Linkerflags)

cc -o csum_lb csum_lb.c
```

4 Übersicht über die DFN-RPC Software

Die wesentlichen Funktionalitäten des DFN-RPC Tools sind:

- synchrone RPCs
- asynchrone und asynchron gepufferte RPCs
- parallele RPCs
- Datenpipes (Message Passing)
- Stubgenerator
- zentraler Wartepunkt
- Konvertierungslibrary
- Remote-File-Access-Library
- graphisches Monitortool
- integrierter Start von Serverprozessen
- automatisierter Aufbau beliebiger Kommunikationsnetze zwischen den Serverprozessen

Der DFN-RPC und sein Einsatz sollen an folgender Übersicht erläutert werden:

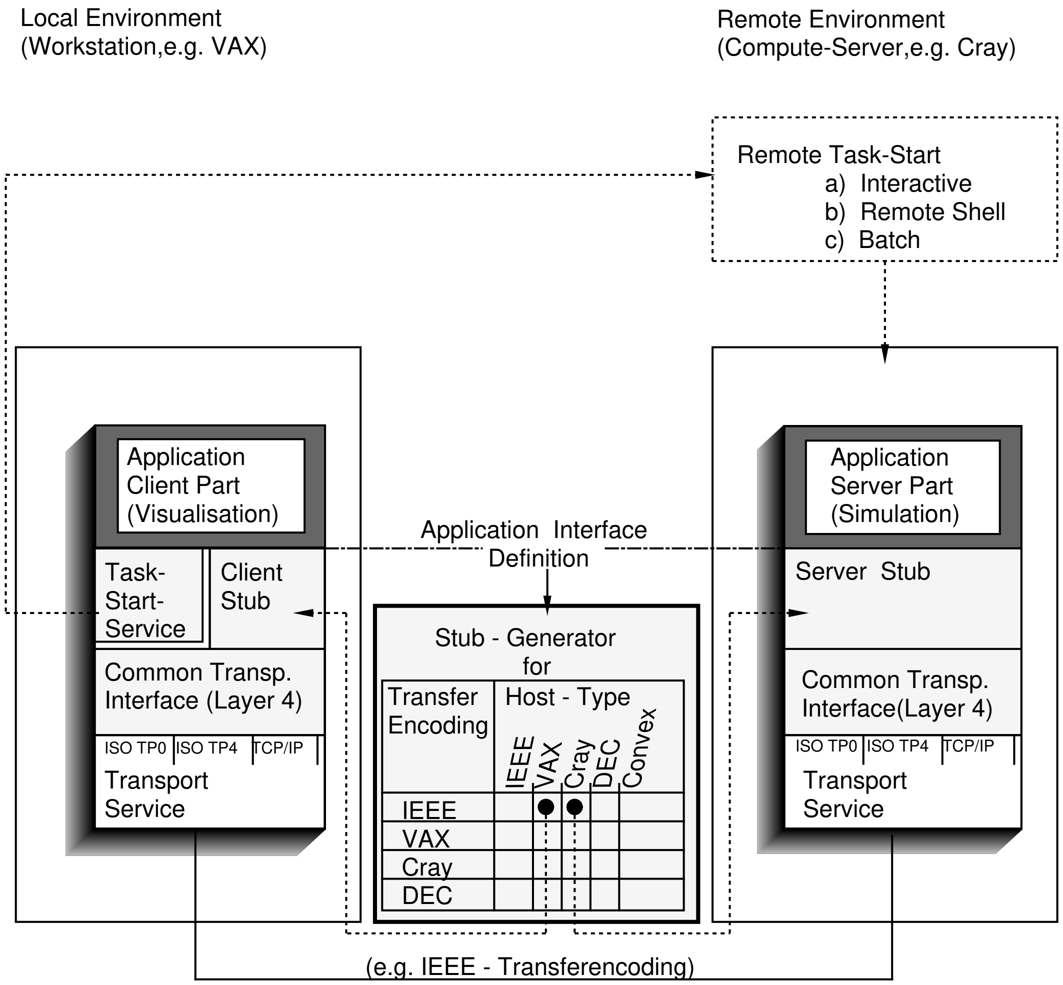


Abbildung 4.1: Gesamtkonzept, ein beispielhaftes Szenarium

Die zum DFN-RPC gehörenden Module sind *hellgrau* unterlegt. Die Module des Anwendungsprogramms sind *dunkelgrau* hervorgehoben und die System-Dienste sind *weiß* dargestellt.

Der Anwender teilt sein Anwendungsprogramm (dunkelgrau) in zwei Teile, das Hauptprogramm[1] (*Client Part*) und ein Modul aus Unterprogrammen (*Server Part*). Der Client ruft Unterprogramme im Server mittels Remote Procedure Call (RPC).

Dies ermöglicht das RPC-Tool (hellgrau). Wenn der Client Part via RPC eine Routine xyz im Server Part rufen möchte, dann ruft er in Wirklichkeit eine gleichnamige Routine im *Client Stub*. Dieser schickt ein Datenpaket, bestehend aus den Input-Parametern und einer Kennung für die Routine xyz. Hierfür wird eine Transferkodierung verwandt, die z.B. eindeutig festlegt, wie eine Realzahl bei der Übertragung dargestellt werden soll.

Der *Server Stub* empfängt dieses Datenpaket, reserviert Speicher für die Parameterliste und schreibt die Input-Parameter dort hinein. Dann ruft er die echte Routine xyz im Server Part auf. Nach Beendigung von xyz werden die Output-Parameter vom Server Stub auf die gleiche Weise zurückübertragen und anschließend die Client Stub Routine xyz beendet. Damit kann der Client Part des Anwendungsprogramms wieder weiterrechnen.

Die Übertragung der Datenpakete übernimmt das *Common Transport Interface* . Dieses ist eine verallgemeinerte Schnittstelle zu den verschiedenen Transport-Diensten. Das Common Transport Interface verdeckt die Unterschiede zwischen verschiedenen Protokollen und unterschiedlichen Betriebssystemen und realisiert den Fortran-C-Übergang.

Die Stubs sind zwar anwenderspezifisch, können aber mit dem *Stub-Generator* automatisch generiert werden. Hierzu benötigt er eine Application Interface Definition, welche im wesentlichen die Parameterliste der mittels RPC zu exportierenden Routinen des Server Parts beschreibt. Außer den im Anwendungsprogramm schon vorhandenen Angaben, wie z.B. der Datentyp, müssen auch RPC-spezifische Angaben gemacht werden, wie z.B. ob es sich um einen Input- oder Output-Parameter handelt, da der RPC kein Call-by-Reference sondern ein Call-by-In-and-Out-Copy durchführt. Durch solche optionalen Angaben kann eine beträchtliche Effizienzsteigerung im Sinne der Anwendung erreicht werden.

In den Stubs ist auch die Konvertierung der Zahlen realisiert, wenn die beteiligten Rechner unterschiedliche interne Zahlendarstellungen verwenden. Der Anwender kann die Transferkodierung frei wählen.

Optimierung gemäß Rechenzeit führt zu einer der Workstation entsprechenden Transferkodierung (IEEE oder VAX), während bei Rechenkostenoptimierung die Kodierung des Compute Servers (z.B. CRAY) transferiert wird.

Als letztes Modul sei der *Task-Start-Dienst* erläutert. Der Anwender kann mit dem Task-Start-Dienst automatisch den oder die Server-Prozesse starten, ohne daß er sich von Hand interaktiv auf dem Compute-Server einloggen muß. Hierbei kann der Anwender jeden beliebigen Service, wie z.B. Remote Shell (rsh) oder das Network Queuing (nqs) einsetzen.

Beim Einsatz mehrerer Server kann jede Kommunikationstopologie aufgebaut werden, wie z.B. Ring, Gitter, Torus, kubisches Gitter, Hypercube, Binärbaum oder vollständiger Graph. Siehe Kap. 4.1-4.10 und 6.

Es gibt Anwendungsszenarien, bei denen die Parallelität von Prozessen physikalische Sachverhalte oder deren mathematisches Modell widerspiegelt. Die nun vorhandenen Kommunikations- und Synchronisierungsformen, d.h. die asynchronen und parallelen RPCs, die Datenpipes und der zentrale Wartepunkt, erschließen dem RPC-Anwender vollständig die beim verteilten Rechnen inhärent vorhandene Parallelität von Client- und Serverprozessen.

Der graphische Monitor gibt auch bei komplexen Multiserver-Anwendungen einen Überblick über die Aktivitäten der beteiligten Prozesse. Siehe Kap. 11.

Mittels des DFN-RPC wurde eine Remote-File-Access-Library entwickelt. Siehe Kap. 10.
Sämtliche Möglichkeiten werden durch mitgelieferte Beispiele veranschaulicht. Siehe Kap. 9.

[1] einschließlich der Unterprogramme, die auf dem Client angebunden gehören

4.1 Kommunikationsformen im Überblick

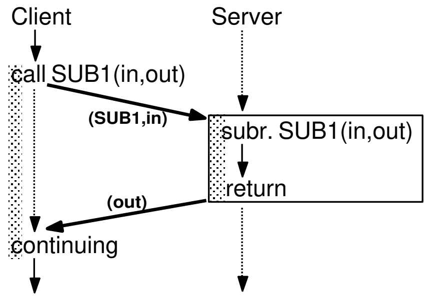


Abb. 4.2: Der synchrone RPC

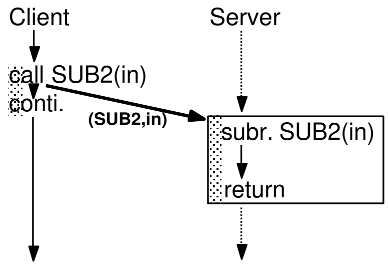


Abb. 4.3: Der asynchrone RPC

Der normale (synchrone) Remote Procedure Call (RPC) ist in Abb. 4.2 dargestellt. Der links dargestellte Clientstub wartet bis die Server-Routine (rechte Box) fertig ist. Das Einführungsbeispiel auf Seite 19 enthält einen synchronen RPC.

Beim asynchronen RPC (Abb. 4.3) kann das Client-Programm sofort weiterrechnen, nachdem die Parameterliste zum Server abgesandt wurde. Es können hierbei keine Outputparameter zurückgegeben werden.

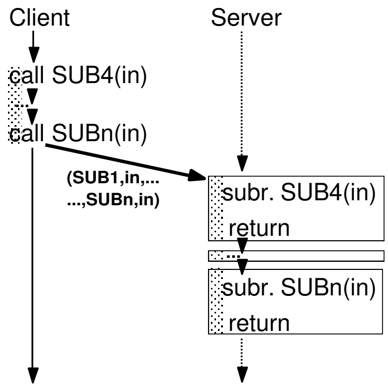


Abb. 4.4: Der asynchron gepufferte RPC

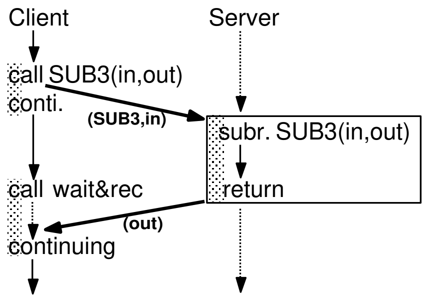


Abb. 4.5: Der parallele RPC

Beim asynchron gepufferten RPC (Abb. 4.4) 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.

Der parallele RPC (Abb. 4.5) ist eine Verallgemeinerung von synchronem und asynchronem RPC. Nachdem der Clientstub die Inputparameter abgesandt hat, gibt er die Kontrolle wieder an das Clientprogramm zurück. Mittels Inquire- und Wait-Routinen kann das Clientprogramm die Beendigung der Serverroutine erfahren und anschließend mit einer Receive-Routine die Outputparameter empfangen und direkt in die beim Aufruf des Clientstubs angegebenen Outputvariablen schreiben.

Bei parallelen RPCs ist prinzipiell clientseitig das Senden der Inputparameter und das Empfangen der Outputparameter in einer parallelen Thread möglich. Hierbei sind dann die Inputvariablen ab dem Aufruf des Clientstubs und die Outputvariablen ab dem Start des Receive-Vorgangs jeweils bis zur Beendigung der Receive-Routine als für die Anwendung gesperrte Ressource zu behandeln.

Ob die Inputparameter in einer parallelen Thread gesendet werden sollen, wird in der Interface-Definition festgelegt. Ob die Outputparameter mit einer parallelen Thread empfangen werden, wird zur Laufzeit durch den Aufruf der entsprechenden Receive-Routinen festgelegt. Die entsprechenden Interfaces sind vorhanden, die Funktionalität ist aber derzeit noch nicht implementiert.

Alle vier Möglichkeiten setzen voraus, daß im Server-Prozeß der RPC-Dispatcher läuft, der die RPCs entgegennimmt und die entsprechenden Server-Routinen der Anwendung aufruft. Die Server-Routinen, die in einem Server-Prozeß gerufen werden, werden somit in einem gemeinsamen Kontext sequentiell abgearbeitet. Parallelität besteht nur zwischen dem Client- und dem Server-Prozeß.

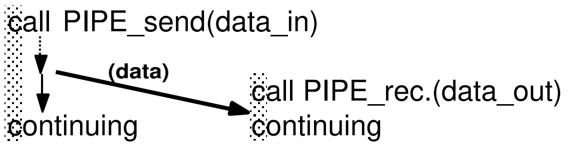


Abb. 4.6: Message Passing mittels Datenpipes

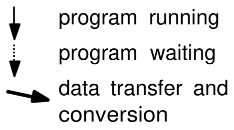


Abb. 4.7: Legende

Neben dem RPC-Prinzip ist auch das Message-Passing-Prinzip realisiert. Mit den Datenpipes können zwei parallel laufende Anwendungsprogramme Daten austauschen. Wenn man mehrere Anwendungsprozesse startet, hat man automatisch parallel laufende Anwendungsprogramme. In Verbindung mit RPCs erreicht man dies mit parallelen oder asynchronen RPCs. Datenpipes können sowohl zwischen Client und Server, als auch zwischen verschiedenen Servern eingesetzt werden.

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.

4.2 Die Aufrufformen im Kontext

Ein Client kann RPCs zu mehreren Servern senden, während ein Server nur einen Client bedient. 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.
- 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 Server-Routine und dem Teil der Clientanwendung erfolgen, der dem RPC-Aufruf folgt und vor einem weiteren RPC-Aufruf, einem RPORCT oder einem RPOREC-Aufruf zu diesem Server liegt.

Für die Datenpipe zwischen Client und Server wird die gleiche Transportverbindung wie für die RPCs eingesetzt. Für Datenpipes zwischen verschiedenen Servern müssen direkte Transportverbindung aufgebaut werden.

Bei sämtlichen RPC-Synchronisierungsformen ist garantiert, daß die Server-Routinen *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.

4.3 Parallele RPCs

Das RPC-Interface einer Subroutine kann in der Interface Definition als *parallel RPC* definiert werden. Wenn man eine solche Serverroutine per *parallelem RPC* ruft, gibt der Clientstub die Kontrolle an die Anwendung zurück, nachdem die Inputparameter zum Serverprozeß übertragen sind. Danach läuft das Anwendungsprogramm im Client- und im Serverprozeß parallel weiter. Natürlich stehen die Outputparameter dieses parallelen RPCs zu diesem Zeitpunkt noch nicht zur Verfügung. Mit der RPC-Library-Routine RPOREC kann der Clientprozeß auf das Ende der Serverroutine warten und die Outputparameter empfangen. Sie werden dabei in den aktuellen Parametern abgespeichert, die beim parallelen RPC-Aufruf eingangs angegeben waren.

```
            CLIENTPROGRAM                        .
                   |                             .
Appl. is computing |                             . dispatcher
                   |                             . is waiting
              CALL XXX   Input arguments         .
             clientstub-->-->-->-->-->-->-->--serverstub
                return                             |
                   |                               SUBR. XXX
Appl. is computing |                                   |
                   |                                   |
             CALL RPOREC                               |
                   .                               RETURN
rporec is waiting  .     Output arguments          |
                storing--<--<--<--<--<--<--<--serverstub
                return   (Return package)        .
                   |                             .
```

RPOREC wartet auf das Ergebnis des parallelen RPCs, falls dieses noch nicht eingetroffen ist, und schreibt die Werte der Outputparameter in die aktuellen Variablen, die beim Aufruf des RPCs angegeben waren.

Bevor die Clientanwendung RPOREC aufruft, kann sie mit weiteren RPC-Library-Routinen die Verbindung zum Server oder zu mehreren Servern bearbeiten:

RPORCT kann[2] das Empfangen des Ergebnisses eines parallelen RPCs in einer parallelen Thread initiieren. Dieses Empfangen muß durch den Aufruf von RPOREC abgeschlossen werden. Zwischen den Aufrufen von RPORCT und RPOREC darf die

[2] Nicht möglich, wenn das Betriebssystem keine Threads unterstützt

Anwendung die aktuellen Outputparameter dieses parallelen RPCs nicht verwenden. RPORCT ist – im Gegensatz zu RPOREC – nicht blockierend. RPORCT ist im derzeitigen Release nur als Dummy-Routine implementiert.

RPOCUR gibt den Objectidentifier des aktuellen Servers, d.h. des *current object's* zurück. Außerdem wird bei Server-Prozessen der Objectidentifier, mit dem sie ihren Client adressieren können, zurückgegeben.

RPOUSE schaltet zwischen verschiedenen Serverprozessen um, d.h. das in der Parameterliste angegebene Objekt wird zum *current object*, d.h. zum aktuellen Server.

RPOCRY wartet bis der aktuelle Server das Return-Paket zu einem parallelen RPC oder ein Datenpaket (s.u. unter Datenpipes) schickt. Die maximale Wartezeit ist zwischen 0 und unendlich vielen Millisekunden einstellbar.

RPOLRY wie RPOCRY, nur daß eine Liste von Serverobjekten angegeben werden kann.

RPORRY wie RPOLRY, nur daß zusätzlich eine Liste von Filedescriptoren angegeben werden kann, die selektiert werden, wenn bei ihnen Daten für eine Read-Operation anstehen. Hiermit kann man gleichzeitig warten bis ein paralleler RPC fertig ist, ein Datenpaket angekommen ist, auf dem Standardinput eine Eingabe erfolgte oder ein X-Window-Event an einem Display stattfand. Hierzu benötigt man die Unix-Filedescriptoren, 0 beim Standardinput, ConnectionNumber(XOpenDisplay()) beim X-Window Display.

RPOGRY wie RPOLRY, nur daß zusätzlich noch weitere Sockets angegeben werden können. Hier steht die volle Funktionalität des Unix-Systemcalls *select* zur Verfügung. Siehe auch Kap. „Select(2)“ auf Seite 154.

RPOGSD ermittelt den Filedescriptor der DFN-RPC-Verbindung zum aktuellen Server. Dieser Filedescriptor wird benötigt, wenn das Anwendungsprogramm andere Wait-Utilities oder den Systemcall *select* einsetzten will.

RPOISP wartet bis ein Paket angekommen ist und ermittelt, ob es sich um ein Datenpaket oder das Return-Paket eines parallelen RPCs handelt.

RPD..S/R senden bzw. empfangen Datenpakete (s.a. unter Datenpipes).

S.a. Beispiel *para* auf Seite 168.

4.3.1 Interface Definition für parallele RPCs

Für parallele RPCs muß man im RPI-File (Stubgenerator-Input) bei C*[RPI_BEG] oder im ENTRY-Statement als Synchronisierungs-Qualifier **PARALLEL** angeben.

Bei der Angabe von **PARALLEL, INPUTTHREAD** kann der Client-Stub die Input-Parameter mittels einer zusätzlichen Thread übertragen. Die Anwendung darf aber die aktuellen Inputparameter erst wieder nach Abschluß der Routine RPOREC verwenden. (Die Angabe von INPUT-THREAD wird derzeit noch nicht ausgewertet.)

4.3.2 Weitere Details

Parallele RPCs haben folgende Eigenschaften:

1. Während eines parallelen RPCs (d.h. nach dem Call und vor einem erfolgreichen RPOCRY) sind keine weiteren RPCs zum gleichen Serverprozeß (=Objekt) möglich.
2. Wenn die Anwendung trotzdem versucht, einen weiteren RPC zu starten, dann ruft deren Clientstub automatisch RPOREC auf.

3. Callbacks sind nur von synchronen Routinen aus möglich, d.h. asynchron, asynchron gepuffert und parallel gerufene Serverroutinen dürfen keine Callbacks (d.h. RPC zurück zu einer Routine beim Client) machen.
4. Einschränkungen:

   Parallele RPCs sind bei Funktionen nicht möglich, da eine verspätete Verwertung des Funktionswertes nicht möglich ist.

   Alternative Rücksprünge sind bei parallelen RPCs nicht möglich.

   Wenn Externals als Parameter übergeben werden, ist darauf zu achten, daß diese nicht als Callback zum Client realisiert sind.

   Parallele Routinen dürfen maximal 30 formale Parameter besitzen. String-Parameter zählen hierbei doppelt. (Aber jeder formale Parameter kann natürlich ein beliebig langes mehrdimensionales Array sein.)
5. Die Adressen der Output-Parameter bei einem parallelen RPC müssen auch beim zugehörigen RPOREC noch sinnvoll sein.

   Da RPOREC die Ergebniswerte in Variablen schreibt, die nicht in seiner Parameterliste enthalten sind (es sind ja die Variablen aus der Parameterliste des Sende-Stubs), muß man RPOREC so einbetten, daß dieser Seiteneffekt möglich ist. Wenn man dies nicht beachtet, kann die Allokierung von Output-Variablen in Registern zu falschen Ergebnissen führen. Um dies zu verhindern gibt es mehrere Möglichkeiten:
   a) Alle Output-Variablen sind in Commonblöcken,
   b) RPOREC wird explizit gerufen (nicht implizit durch den Stub eines nachfolgenden RPCs) **und** die Output-Parameter werden beim Aufruf von RPOREC am Ende der Parameterliste zusätzlich angegeben – mit dem Nachteil, daß der Compiler u.U. warnt, daß man die RPOREC-Routine mit unterschiedlichen Parameterlisten aufruft,
   c) bei den meisten Workstation-Compilern kann man Output-Variablen als VOLATILE deklarieren, aber auf einer Workstation zeigte ein Testprogramm, daß VOLATILE nicht in jeder Optimierungsstufe fehlerfrei arbeitet,
   d) die SUPPRESS Compiler-Directive beim CRAY cf77 Compiler ist <u>nicht</u> geeignet, da sie nur innerhalb eines zusammenhängenden Execution Pathes wirkt und ein solcher meist nicht gegeben ist.
   e) Abschalten aller Optimierung durch den Compiler (nicht 100% sicher).

   Zusätzlich müssen die Output-Variablen mit SAVE deklariert werden, wenn sie zwischen Aufruf des Client-Stubs und von RPOREC sonst deallokiert würden.

4.4 Beispiel mit dynamischer Lastverteilung auf mehrere Server

Das folgende Beispiel soll auf einem RISC-Cluster parallelisiert werden.

```
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. 4.8: 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.

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='client block', CLIENT_SITE
C*[MOD_USE]          cm, LOCAL
C*[MOD_USE]          sm, CLIENT
C*[BLK_END]

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

Mit dieser Information wird der Stubgenerator angewiesen, die Source aufzuspalten und die nötigen Stubs zu generieren. Im 'client block' wird der Clientteil der Anwendung und die Clientstubs der Routinen 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=1_call, 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
          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)
          if (i_call < n_parts) then ; i_call=i_call+1
            call simul_part(inp(i_call),out(i_call))
          endif
        end do
```

Ergebnisse sammeln und weitere simul_part-Aufrufe: rpolry ermittelt, welcher parallele RPC fertig ist. Mit rpouse wird der entsprechende Server ausgewählt und mit rporec werden die Outputparameter in entspr. out(i) geschrieben. Der Server wird sofort wieder für die nächste Berechnung von simul_part genutzt, wodurch eine automatische Lastbalancierung erreicht wird.

```
        sim_fin = end_criterion(out)
        if (¬ sim_fin) then
          inp = new_input(out)
        endif
      end do
```

Berechnung der neuen Input-Daten und des Abbruch-Kriteriums, etc.

```
      call rpqucf(iret)
      end
C*[MOD_END]
```

Sämtliche Server werden beendet.

```
C*[MOD_BEG] sm,      NAME='server routines module'
      program exasrv
      call rpqusp(iret)
      end
```

Serverteil der Anwendung. Spezielles Hauptprogramm für den Server, mit rpqusf, dem Gegenstück zu rpquci und rpqucf.

```
C*[RPI_BEG] PARALLEL
      subroutine simul_part(inp,out)
      real inp [IN], out [OUT]
```

Parallele RPC Interface Definition, bestehend aus Subroutine-Kopf, erweitert um RPC-spezifische Qualifier, z.B. [IN]

```
C*[RPI_NON]
      out = simulation_code(inp)
      end
C*[RPI_END]
C*[MOD_END]
```

Der Subroutine-Rumpf ist für die Interface-Definition ohne Bedeutung.

Abb. 4.9: 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. 4.10 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. 4.10: Beispielfile zur Server-Initialisierung mittels rpquci

Siehe auch Beipiel rshexa und crshexa auf S. 170.

4.5 Datenpipes

Mit Datenpipes kann man Anwendungsdaten zwischen zwei Objekten austauschen, zwischen denen eine Transportverbindung existiert. Zwischen Client und Server einer DFN-RPC-Anwendung wird hierzu die für den RPC etablierte Transportverbindung genutzt, s. Abb. 4.11. Mit den RPC-Library-Routinen RPOSSC und RPOC2S kann der Client Transportverbindungen zwischen Servern herstellen, s. Abb. 4.12.

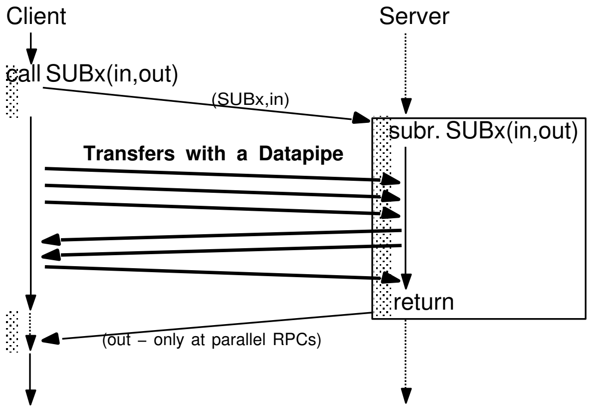


Abb. 4.11: Client-Server-Datenpipe

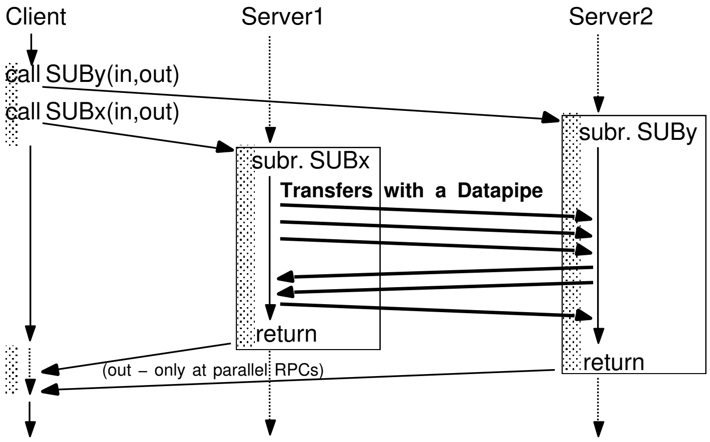


Abb. 4.12: Server-Server-Datenpipe

4.5.1 Datentransfer mittels Library-Routinen

Zum Versenden und Empfangen von Datenpaketen gibt es folgende RPC-Library-Routinen:

RPDI0S	sendet ein Integer-Array
RPDI0R	empfängt ein Integer-Array
RPDI2S/R	sendet bzw. empfängt ein Integer*2-Array
RPDI1S/R	sendet bzw. empfängt ein Byte-Array
RPDL0S/R	sendet bzw. empfängt ein Logical-Array
RPDL2S/R	sendet bzw. empfängt ein Logical*2-Array
RPDL1S/R	sendet bzw. empfängt ein Logical*1-Array
RPDR0S/R	sendet bzw. empfängt ein Real-Array
RPDR8S/R	sendet bzw. empfängt ein Real*8-Array
RPDD0S/R	sendet bzw. empfängt ein Doubleprecision-Array
RPDC0S/R	sendet bzw. empfängt einen Character-String

Die Read-Routinen RPD..R sind blockierend, d.h. sie warten bis sie einen Datensatz empfangen können.

Mit einem Flush-Parameter kann man bei den Sende-Routinen entscheiden, ob die Message gepuffert werden soll, oder ob der Puffer am Ende dieser Message abgesendet werden soll.

Der Message-Transfer in den Datenpipes ist ende-zu-ende-flußkontrolliert, d.h. die Sende-Routine wartet ggf. bis alle Daten zumindest in sendeseitigen Puffern Platz haben, bzw. bei großen Datenmengen, bis ein Teil auch von der Empfängerseite mit der entsprechenden Empfangsroutine eingelesen wurde. Die Empfangsroutine wartet bis die Daten vollständig angekommen sind. Die Anwendung muß beachten, daß die beiden Objekte, die die Datenpakete senden und empfangen hierbei u. U. miteinander synchronisiert werden, d.h. es kann auch das sendende Objekt solange blockiert werden, bis das empfangende Objekt bereit ist, das Datenpaket zu empfangen. Es obliegt der Anwendungsprogrammierung, daß keine Blockierungen auftreten.

Es findet keine Partitionierung oder Assemblierung der Daten statt, d.h. mit einem RPD..R-Aufruf wird genau das Array empfangen, das auf der Gegenseite mit RPD..S abgesandt wurde.

Mit den bei den parallelen RPCs schon beschriebenen Routinen RPOCRY, RPOLRY, RPORRY und RPOGRY kann auf einer oder mehreren Verbindungen gewartet werden, bis ein Datenpaket angekommen ist. Mit RPOISP kann untersucht werden, von welchem Typ das Paket ist.

Datenpipes setzen daher immer zwei parallel laufende Prozesse voraus. Es gibt daher keine Möglichkeit, Datenpipes mit lokalgebundenen Programmen (Local Block) zu testen.

Datenpipes können im Zusammenhang mit DFN-RPCs eingesetzt werden, wobei die Parallelität durch den Einsatz von asynchronen (ungepuffert!) oder parallelen RPCs erreicht wird. Datenpipes können aber auch unabhängig von DFN-RPCs eingesetzt werden.

S. a. Kap. 4.6 auf S. 41 und Beispiel *para* auf S. 168 und *pipe* auf S. 171.

4.5.2 Anwendungsspezifische Messagestubs

Die oben beschriebenen Routinen sind sog. *Messagestubs*, die in der DFN-RPC-Library bereitgestellt werden.

Jede Anwendung kann aber auch selbst-definierte Messagestubs einsetzen. Hierzu wird in der Interfacedefinition ein Messagestub-Interface definiert und hierzu ein Send-Stub (alle Parameter werden als Input betrachtet) und ein Receive-Stub (alle Parameter dienen als Output-Parameter) erzeugt. Die Namen der beiden Stubs entstehen durch anhängen von „S“ und „R“ an den im Interface angegebenen Namen. Die Interfacedefinition enthält keinen Fortranbody. Es wird auch kein Fortrancode in die .f-Files geschrieben. Der Sende-Stub versieht die Message grundsätzlich mit einem Header, der im Receive-Stub überprüft wird.

Messagestubs generiert man, indem man ein Subroutine-Interface definiert und bei C*[RPI_BEG] das Schlüsselwort **PIPE** angibt . Wird das Modul als Client-Modul in einen Block eingebunden, dann wird ein Message-Send-Stub generiert, im Server-Modul dagegen der zugehörige Message-Receive-Stub (s.a. MOD_USE).

Gibt man zusätzlich durch Komma getrennt das Schlüsselwort **BACK** an, dann wird im Client-Modul der Message-Receive-Stub und im Server-Modul der Message-Send-Stub generiert.

Mit dem Schlüsselwort **BOTH** werden beide Stubs in beiden Modul-Typen generiert.

Synchronisierungsangaben haben keine Bedeutung und sind daher unzulässig. Am Anfang einer Message wird immer ein Header generiert. Er enthält einen Operation-Code. Daher kann CODE=`<opcode>` angegeben werden. Der in der Interface-Beschreibung enthaltene Fortran Subroutine-Header wird in keinen .f-File geschrieben.

Siehe auch Beispiel *rpcpipe* auf S. 169.

4.6 Datenpipes und RPCs

Jede Kommunikations-Verbindung, die mit den Routinen RPQ..., RPO1/2/3.., RPOPOP, RPO-ACC, RPOCON oder RPOC2S aufgebaut wurde, kann für den Austauch von RPCs und auch als Datenpipe eingesetzt werden.

Während eine Verbindung für RPCs genutzt wird, d.h. am einen Ende ein Block Dispatcher läuft, kann sie auch als Datenpipe genutzt werden, solange serverseitig ein asynchroner (ungepufferter) oder ein paralleler RPC gerufen ist.

Abwechselnde Nutzung für RPCs und Pipe-Routinen ist ebenfalls möglich. Mittels RPOFBD (*finish block dispatcher*) kann die Client-Anwendung den Block-Dispatcher im Server beenden. Anschließend können die beiden Prozesse die Verbindung als Datenpipe nutzen. Nach einem erneuten serverseitigen Aufruf von RPBD1 kann der Client wieder RPCs aufrufen.

Der Server-Prozeß kann auch mit RPOISP (*inspect package*) die Art eines ankommenden Pakets ermitteln und dann die entsprechende Pipe-Routine oder RPBD1P (*block dispatcher for one rpc*) aufrufen. (RPBD1A kann für diesen Zweck nicht aufgerufen werden, da RPBD1A erwartet, daß ausschließlich RPCs auf der Verbindung gemacht werden. RPBD1A ist nur für die Programmierung von Servern, die mehrere Clients bedienen, entwickelt worden.)

Siehe auch Beispiel *rpcpipe* auf S. 169.

4.7 FLUSH-Argument

Bei asynchronen RPCs und bei den Datenpipe-Senderoutinen kann der Anwender zur Laufzeit bestimmen, ob der Aufruf gepuffert werden soll oder ob nach dem Aufruf der Puffer geleert werden soll (*flush*).

Hierzu muß ein formaler Parameter vom Typ LOGICAL mit dem Qualifier [FLUSH] ausgezeichnet werden, bzw. **FLUSH** muß als eine Qualifier_Expression angegeben werden.

4.8 Das Object Handling

4.8.1 Definition

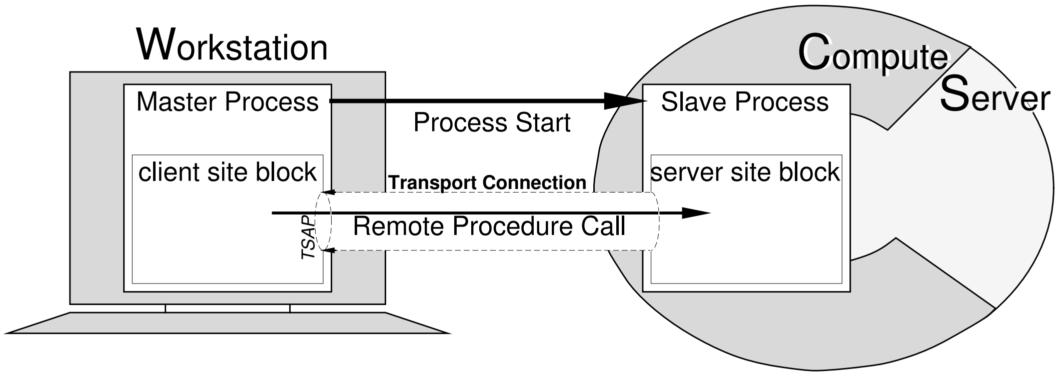


Abbildung 4.13: Workstation/Computeserver, Master/Slave und Client/Server - erstes Szenarium: Master enthält Client_Site Block

Beim DFN-RPC werden Prozesse als *Objekte* betrachtet. Mit dem Object Handling wird der entfernte Prozeß gestartet. Der vom Benutzer gestartete Prozeß wird als *Master Process* bezeichnet. Der mittels Object Handling zu startende entfernte Prozeß wird *Slave Process* genannt. Typischerweise befindet sich der Master Process auf der Workstation, während der Slave Process sich auf dem Compute Server befindet.

Der Master Process kann den Client_Site Block, aber auch einen Server_Site Block enthalten, d.h. Master/Slave und Client/Server sind beim DFN-RPC unabhängige Unterscheidungsmerkmale, d.h. das Anwendungsprogramm kann sich im Master Process (Workstation, Abb.4.13) oder im Slave Process (Compute Server, Abb.4.14) befinden.

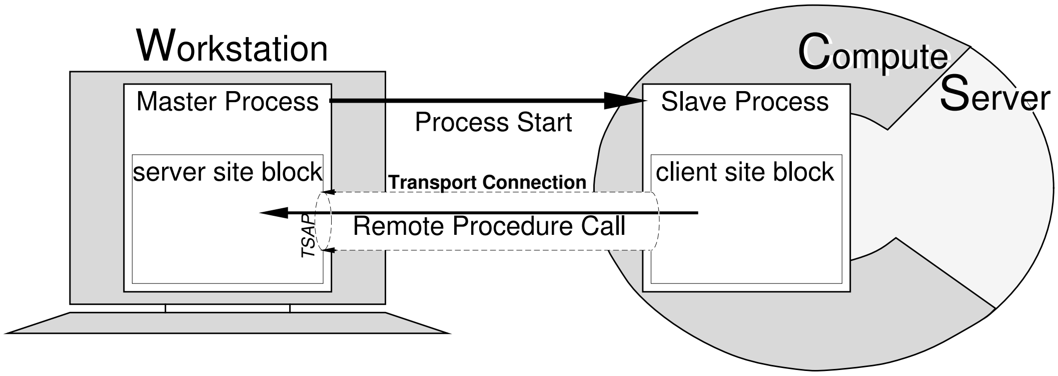


Abbildung 4.14: Zweites Szenarium: Master enthält Server_Site Block

Für die Durchführung der Remote Procedure Calls wird eine sog. Transport-Verbindung benötigt[3]. Diese wird ebenfalls von den Object Handling Routinen aufgebaut. Dieser Verbindungsaufbau ist

[3] Über diese Transportverbindung werden alle nötigen Informationen für die Remote Procedure Calls ausgetauscht, für jene vom Client_Site Block zum Server_Site Block aber auch für die sogenannten Callbacks. Callbacks sind Remote Procedure Calls von mittels RPC gerufenen Routinen im Server_Site Block zurück zum Client_Site Block. Callbacks sind in den Abbildungen nicht dargestellt.

gerichtet, d.h. die eine Seite (i. allg. der Master) stellt einen Transport Service Access Point (TSAP) bereit und die Gegenseite (Slave) baut dann die Verbindung auf, d.h. sie führt das sog. Connect durch. Ein TSAP ist wie eine virtuelle Steckdose mit einem Namen. Damit der Slave die Verbindung aufbauen kann, benötigt er die Adresse des Rechners (meist der Hostname) auf dem der Master läuft, sowie den TSAP Namen des Masters. Bei einem automatischen Prozeßstart mittels der Object Handling Routinen werden diese Daten vom Master an den Slave übermittelt.

4.9 Prozeßstart und Verbindungsaufbau

4.9.1 Quickstartroutinen

Bei den Quickstartroutinen enthält der Master den Client_Block, wie in Abb. 4.13; eine Ausnahme ist bei nur einem Server möglich.

Um parallele Anwendungen möglichst einfach starten zu können, wurden in die Library die Routinen **RPQUCI**, **RPQUCF** für den Client und **RPQUSP** oder **RPQUSI** und **RPQUSF** für den Server aufgenommen. Wahlweise können die Server-Prozesse von Hand oder automatisch gestartet werden. Auch die vor dem Verbindungsaufbau nötige Übertragung der Portadressen kann entweder automatisch oder manuell erfolgen. Das Client-Programm ruft RPQUCI. Die Anzahl der gewünschten Server-Prozesse ist ein Eingabeargument. Wenn vorhanden, dann liest RPQUCI den File **dfnrpc.servers** und führt jede Zeile mittels eines CALL SYSTEM aus. Hierzu ein Beispiel:

```
# Servers startet with a remote shell:
rsh servername_1 -n server_program DFNRPCADDR &
rsh servername_2 -n server_program DFNRPCADDR &
rsh servername_3 -n server_program DFNRPCADDR &
rsh servername_4 -n server_program DFNRPCADDR &

# The following server must be started with telnet:
echo start anywhere server_program DFNRPCADDR

# The following line prevents, that the server list is
# repeated to start the number of servers wanted by the client
-norepeat
```

Abb. 4.15: Beispiel zum RPQUCI-Datenfile dfnrpc.servers

Das Schlüsselwort **DFNRPCADDR** wird vor dem SYSTEM-Call durch die Adresse eines neu eröffneten Ports ersetzt. Das server_program muß diesen String dann an RPQUSP übergeben, welches dann die Verbindung zum Client aufbaut. Wenn man das Schlüsselwort DFNRPCADDR wegläßt oder gar keinen File dfnrpc.servers in der aktuellen Directory besitzt, bzw. RPQUSP nur einen leeren String übergibt, dann sind die Prozeßstarts und Portadreßübermittlung interaktiv zu realisieren. Der bei diesen Quickstartroutinen angewandte *rückwärtige Verbindungsaufbau* ist in [17] Abb. 4.5, S. 24 beschrieben.

Die Beispiele *rshexa*, S. 170 und *crshexa*, S. 170 zeigen den Einsatz dieser Routinen unter Fortran und C. In den Beispielen *ben* und *test* werden sie wahlweise eingesetzt. In dem Beispiel in Abb. 4.9 und Abb. 4.10 auf Seite 38 wurden ebenfalls die Initialisierungsroutinen RPQU.. eingesetzt.

Dient der Server nicht nur der Abarbeitung von RPCs sondern soll er z.B. als eigenständige Task mit dem Client-Prozeß mittels Datenpipes kommunizieren, dann kann man RPQUSP nicht einsetzen, da RPQUSP intern genau einmal den Blockdispatcher RPBD1 ruft. Statt dessen ist der Server mit RPQUSI zu initialisieren. Anschließend kann die Serveranwendung den Blockdispatcher, aber auch direkt Datenpipe-Routinen aufrufen. Mit RPQUSF beendet der Server die Verbindung zum Client. Das Beispiel *rpcpipe*, S. 169 enthält einen derart programmierten Server.

Wenn man nur einen Server mit RPQU...startet, dann können Client und Server nach Abschluß der Routinen RPQUCI und RPQUSI ihre Rollen tauschen. Hierzu muß dann im Client-Prozeß, der eigentlich als Master-Prozeß zu bezeichnen ist, der Blockdispatcher RPBD1 aufgerufen werden.

4.9.2 Interconnection Network

Mit den Routinen RPQSSC und RPQSSS (s. Kap. 6.4.27, S. 132) kann ein Interconnection Network zwischen den Server Prozessen frei konfigurierbar aufgebaut werden. Beim Aufruf der Client-Routine RPQSSC ist eine Liste von gewünschten Server-Server-Verbindungen anzugeben. Die Server-Routine RPQSSS, die in jedem Server als Gegenstück zu RPQSSC aufzurufen ist, baut die zu ihrem Server gehörenden Verbindungen auf und gibt eine entsprechende Teilliste der Anwendung über die Parameterliste zurück. Im Beispiel sscon (s. Kap. 9.8, S. 172) wird ein Interconnection Network zwischen bis zu 9 Servern aufgebaut. Siehe auch Beispiel auf Seite 134.

4.9.3 Expliziter Verbindungsaufbau

Die Objekt-Startroutinen RPQU.. kombinieren den Prozeßstart und den Aufbau der Kommunikationsverbindung. Mit den Routinen RPOPOP, RPOCON und RPOACC kann man unabhängig vom Prozeßstart eine Verbindung aufbauen.

port open — Mit **RPOPOP** eröffnet ein Prozeß (im folgenden *Master* genannt) einen Port (auch TSAP genannt).

connect — Mit **RPOCON** kann dann ein anderer Prozeß (im folgenden *Slave* genannt) eine Verbindung zu diesem Port aufbauen.

accept — Mit **RPOACC** muß der Master diesen Verbindungsaufbau entgegennehmen.

RPOPOP gibt die vollständige Adresse des geöffneten Ports zurück (*Master_Address*). Es ist Aufgabe der Anwendung, diese *Master_Address* an den Slave-Prozeß weiterzugeben. Dieser benötigt diese Adressinformation für den Aufruf von RPOCON. Die *Master_Address* kann z.B. im Rahmen des Starts des Slaveprozesses diesem als Parameter auf der Commandline mitgegeben werden und vom Slaveprozeß mittels GETARG (3F) eingelesen werden. RPOCON und RPOACC verifizieren die Verbindung mittels eines Datenaustausches in beiden Richtungen. Damit ist auch eine Synchronisation von beiden Prozessen verbunden. Anschließend kann die Verbindung genutzt werden, wie wenn sie mittels RPQUCI oder RPQUSI aufgebaut worden wäre.

Der Verbindungsabbau geschieht mit RPOCCL und RPOPCL.

close connection — Mit **RPOCCL** wird die Verbindung geschlossen. RPOCCL ist beim Master und beim Slave aufzurufen. RPOCCL kann sofort nach dem Versenden der letzten Daten aufgerufen werden. Die Verbindung wird erst abgebaut, wenn an beiden Endpunkten RPOCCL aufgerufen ist.

port close — Mit **RPOPCL** muß anschließend beim Master der Port geschlossen werden. RPOPCL ist das Gegenstück zu RPOPOP.

Vor allem beim Einsatz von Datenpipes ist zu beachten:

1. In einem Prozeß können maximal 512 Objekt-Identifier vergeben werden. Diese werden beim Starten von Server-Objekten oder beim Initiieren einer Datenpipe vergeben.
2. Es können auch nur maximal 512 Transportverbindungen aufgebaut werden (beim Einsatz von sunlink oder sunnet OSI sind nur max. 10 Verbindungen möglich).

3. Beim Aufruf von RPSINI darf die Summe der *beantragten* Incoming und Outgoing Conncetions diesen Maximalwert 512 nicht überschreiten.
4. Die Objekt-Start-Routinen und die Pipestart-Routine RPOC2S bauen dann die Verbindungen auf. Bei RPSINI dürfen daher nicht zu wenige Verbindungen beantragt werden.
5. Beim Beenden eines Servers ist zu beachten, daß zuvor mit RPOUSE sowohl client- als auch serverseitig der entsprechende Partner als *current object* gewählt wird.
6. Ein beim Server gestarteter Dispatcher muß vom Client aus mit RPOFBD beendet werden bevor die verbindung mit RPOCCL und RPOPCL abgebaut werden kann.

Im Beispiel *con*, S. 170

4.10 Adressierung verschiedener Server-Objekte

4.10.1 Objekt-Identifier

Objekte sind im Sinne des DFN-RPC-Konzepts eigenständige Prozesse. Zwischen Objekten können Verbindungen existieren. Zum Aufbau einer TCP/IP-Verbindung werden Rechnername und Portnummer benutzt. Wenn eine Verbindung zwischen zwei Objekten besteht, dann wird dieser Verbindung an beiden Enden jeweils ein Objekt-Identifier zugewiesen. Eigentlich bezeichnet ein derartiger Objekt-Identifier eine Verbindung. Da aber eine Verbindung immer auf ein anderes Objekt zeigt, kann man auch sagen, daß der Objekt-Identifier dieses Objekt bezeichnet.

```
          Object A                                   Object B
+----------------------+                  +----------------------+
|                      |                  |                      |
|          +------------|                  |------------+         |
|          | ObjId of B |    Transport-    | ObjId of A |         |
|          | valid in A |==================| valid in B |         |
|          +------------|    Connection    |------------+         |
|                      |                  |                      |
+----------------------+                  +----------------------+
```

Die Objekt-Start-Routinen und RPOC2S geben die jeweiligen Objekt-Identifier zurück. Die Objekt-Identifier sind Ausgabeparameter der Routinen RPQUCI, RPQUSI und RPO1/2/3.. für den Objektstart und von RPOPOP, RPOCON und RPOC2S für den Verbindungsaufbau.

Bei RPOC2S ist zu beachten, daß der Client die in den Servern gültigen Objekt-Identifier der Server-Server-Datenpipe erfährt und somit der Anwendung in den Servern diese per RPC mitteilen muß, damit die Server diese Datenpipe nutzen können.

4.10.2 "Current Object" und "Current Client Object"

Alle RPC- und Pipe-Aufrufe beziehen sich immer auf einen sog. *current object identifier*. Die DFN-RPC-Library besitzt intern die globale Variable *current object identifier*. Ihr wird durch die Objektstart- und Verbindungsaufbau-Routinen ein neuer Wert zugewiesen. Mit RPOUSE kann die Anwendung jederzeit den Wert ändern. Mit RPOCUR kann er abgefragt werden.

Für das Empfangen von RPCs gibt es eine weitere globale Variable, den *current client object identifier*. Nach dem Start des Blockdispatchers speichert dieser den Wert des *current object identifier* als seinen *current client object identifier* ab und benutzt diesen bis der Client eine Finish-Routine RPQUCF, RPO.CF oder RPOFBD aufruft. Am Ende wählt er dann als *current object identifier* seinen *current client object identifier*, damit eine nachfolgende RPO.SF-Routine zum richtigen Objekt die Verbindung abbaut.

4.11 Navigieren in der DFN-RPC Software

Der Directory-Baum des DFN-RPC enthält folgende relevante Directories und Files:

```
Directory:                                             Files:

$DFNRPCROOT / dfnrpc / 1.0 /                         / dfnrpc_set
                                                       dfnrpc.ps

                             src / examples / ua1a / source_makefile
                                                   / ua1a.rpi
                                                   / ua1a_cb_stb.m4
                                                   / ua1a_sb_stb.m4

                                        ... / ben  / source_makefile
                                                   / ben.rpi
                                                   / ben_cb_stb.m4
                                                   / ben_sb_stb.m4

                       .../sys_debug  or
                       .../sys
                          .../sun4    or others
                              .../ bin             / dfnrpcgen
                              .../ lib             / libdfnrpc.a
                              .../ include         / dfnrpc.inc

                              .../ examples / ua1a / makefile
                                                   / ua1a.rpi
                                                   / ua1a_lb.f
                                                   / ua1a_lb
                                                   / ua1a_cb.f
                                                   / ua1a_cb_stb.m4
                                                   / ua1a_cb_stb.f
                                                   / ua1a_cb
                                                   / ua1a_sb.f
                                                   / ua1a_sb_stb.m4
                                                   / ua1a_sb_stb.f
                                                   / ua1a_sb
                                                   / dfnrpc.inc -> ...

                                        ... / ben  / ben_cb
                                                   / ben_sb
                                        ... / test / test_cb
                                                   / test_sb
```

Falls die Files im .../sys/examples...Zweig fehlen, so wurden die Beispiele bei der Installation nicht installiert.

Die Sourcen des DFN-RPC werden ohne die interne Dokumentation mitgeliefert. Sie dienen nur bei der Installation zur Kompilation, stellen aber keine Basis zur Wartung dieser Software dar.

Das folgende Shellscript erleichtert Ihnen das navigieren:

```
            Module description of the dfnrpc_set
            ====================================

Installation of dfnrpc_set:
---------------------------
   You should include in your .cshrc file the following lines:
       setenv DFNRPCROOT ~
                          (or where you have installed the dfnrpc.)
       setenv DFNRPCREL   1.0
       setenv DFNRPCTRANS ieee
       source $DFNRPCROOT/dfnrpc/$DFNRPCREL/dfnrpc_set
                          (Before calling the following configuration
                           you can change the defaults in
                           DFNRPCROOT, DFNRPCREL and DFNRPCTRANS.)
       SUN4               (or an other configuration, see below.)
   Available configurations with optimized executables:
    ALLI ALPHA APO CVX CRAY2 DEC FUJITSU HP IRIS3 IRIS4 PARAGON
    RS6 SUN3 SUN4 VAX
   and with the UltraNet Compatibility Library:
    CVXU CRAY2 IRIS4U SUN3U SUN4U
   and  with OSI:
    CRAY2 SUN3O SUN4O
   All configurations are also available compiled with debug option:
    SUN3D SUN4D CRAY2D ...
   CAUTION: The values of DFNRPCROOT, DFNRPCREL and DFNRPCTRANS must
            not be followed by any blanks or tabs.

Usage:
------
   After calling a configuration (like SUN4) there are available
   several aliases and setenv's:
   - general aliases                         and belonging setenv
       alias  dfnh  typing_this_information
       alias  dfns  cd general_source_directory       $DFNRPCSRC
       alias  ct4s  cd source_of_ct4
       alias  rpcs  cd source_of_rpc
       alias  gens  cd source_of_gen
       alias  ipsms cd source_of_ipsm
       alias  makes cd source_of_makefiles
       alias  exams cd general_examples_source_dir
       alias  dfnc  cd general_config_directory       $DFNRPCCONF
       alias  dfnb  cd configuration/bin              $DFNRPCBIN
       alias  dfnl  cd configuration/lib              $DFNRPCLIB
       alias  dfnld cd configuration/lib_debug
       alias  dfni  cd configuration/include          $DFNRPCINCLUDE
       alias  exab  cd general_examples_bin_dir       $DFNRPCEXAMPLES
   - for each example XYZ
       alias  XYZs  cd source_of_example_XYZ          XYZs
       alias  XYZb  cd bin_of_example_XYZ             XYZb
     with XYZ = awi,ben,cav,con,crshexa,csum,cvttest,exa,moni,multicl,
                para,ph1,ph2,phi,pipe,rfa,rpcpipe,rshexa,rw,sscon,sum,
                test,ua1a,ua1b,ua1c,ua1d
```

```
Depending on your installation, $DFNRPCEXAMPLES and all example
binaries XYZb refer to your default transfer type.
With the aliases
    TRANS_ieee
    TRANS_dec
    TRANS_vax
    TRANS_cray
you can change to that transfer type, i.e. you change the
values of XYZb.
```

4.12 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 Agenten-Prozesse benötigt. Nur wenn man den Monitor einsetzen will, benötigt man hierfür einen Daemon-Prozeß. Im Kap. 12.4 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 am Ende des Clientstubs mit *write* 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 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.

4.13 Datenkonvertierungslibrary

Um Daten zwischen zwei Rechnern mit unterschiedlicher Zahlendarstellung unter Fortran austauschen zu können, müssen die Daten vor oder nach der Übertragung konvertiert werden.

Um eine flexible Umwandlung von verschiendenen Zahlendarstellungen und deren problemlose Übertragung zu gewährleisten, hat das hier beschriebene Daten-Konvertierungstool folgende Merkmale:

Die lokalen Daten des aussendenden Rechners werden von dessen lokaler Zahlen-Darstellung in eine Transferdarstellung umgesetzt, welche der des empfangenden Rechners entsprechen kann aber nicht muß. Es wird also auch auf Empfangsseite eine Konvertierung der Transferdarstellung in die lokale Darstellung des Zielrechners durchgeführt.

Die zu versendenden Daten werden der Konvertierungsroutine RPVCSB als Array übergeben, ebenso die Länge eines Elements im Array in Bytes. Die Konvertierungsroutine beherrscht dabei alle gängigen Längen einer INTEGER- oder REAL-Zahl. Die Steuerung, welche Elemente des Arrays konvertiert werden sollen, ist über zwei Argumente möglich, welche den Anfang und das Ende der zu konvertierenden Array-Elemente indizieren.

Als Resultat der Konvertierung wird ein Sendepuffer gefüllt, der als Byte-Array organisiert ist und somit in einfacher Weise übertragen werden kann. Die gewünschte Länge eines Array-Elements (in Bytes) in diesem Sendepuffer wird über ein Argument festgelegt, welches die Möglichkeit bietet, INTEGER-Zahlen mit 1, 2, 4 oder auch 8 Bytes Länge und REAL-Zahlen mit 4, 8 oder 16 Bytes Länge zu benutzen. Als Transferdarstellungen werden INTEGER BIG oder LITTLE ENDIAN, LOGICAL BIG oder LITTLE ENDIAN sowie REAL nach IEEE-, CRAY-, DEC- oder VAX-Standard angeboten.

Die Länge des zu benutzenden Sendepuffers (welche ein Vielfaches von 16 Bytes sein sollte) wird der Konvertierungsroutine ebenso übergeben wie die aktuelle Position (in Byte-Schritten) bis zu welcher der Sendepuffer bereits gefüllt worden ist. Dieser Positionsparameter wird nach erfolgter Konvertierung mit neuem Wert zurückgegeben. Desweiteren wird der Senderoutine ein Parameter übergeben, welche die letzte Position (in Byte-Schritten) im Sendepuffer angibt, bis zu welcher der Inhalt des Sendepuffers bereits gesendet worden ist. Dies ermöglicht es der Konversionsroutine, den Sendepuffer um bereits gesendete Array-Elemente zu kürzen und macht somit das Versenden von Teilen des Sendepuffers möglich.

Bevor mit der Umwandlung des Daten-Arrays in den Sendepuffer begonnen werden kann, wird der Sendepuffer um bereits versandte Elemente gekürzt und ein sogenanntes Alignment auf den Sendepuffer durchgeführt, welches die aktuelle Position im Sendepuffer auf den nächsthöheren Wert setzt, der ohne Rest durch die Anzahl der Bytes eines Elements im Sendepuffer teilbar ist. Steht zum Beispiel die aktuelle Position auf 19 und sollen die Elemente des Sendepuffer 8-Byte Realzahlen sein, so wird die Position auf 24 gesetzt.

Bei den soeben beschriebenen Konvertierungsvorgängen können diverse Probleme auftreten, die wiefolgt behandelt werden:

- Ist der Sendepuffer zu klein um alle Daten zu konvertieren, wird nur solange konvertiert, bis der Puffer voll ist. Diese vollständige Füllung bei ausreichender Datenmenge ist durch die Wahl der Länge des Sendepuffers (vielfaches von 16 Bytes) immer möglich.
- Tritt bei der Konvertierung ein Under- oder Overflow auf (Zahl ist zu klein / zu groß zum konvertieren), so wird die Konvertierung abgebrochen und die Routine verlassen. Diese Under- und Overflowchecks können mit Hilfe von Flag-Argumenten aktiviert oder deaktiviert werden. Außerdem ist auch eine automatische Korrektur eines Underflows (setzten der Zahl auf Null) möglich.

- Wurden Fehler bei den Argumenten gemacht, ist z.B. die angegebene Anzahl der Bytes eines Elements des Sendepuffers gar nicht möglich, so wird die Routine schon vor dem Beginn der Konvertierung mit gesetzter Fehlervariable verlassen.

Insgesamt stehen auf Senderseite neben der Konvertierungsroutine noch zwei weitere Routinen zur Verfügung: Eine Routine zur Initialisierung des Sendepuffers (RPVISB) sowie eine Routine (RPVFSB), welche ein Alignment des Sendepuffers auf volle 16 Bytes durchführt, was die Versendbarkeit des Puffers verbessert.

Auf der Seite des empfangenden Rechners wird der oben beschriebene Vorgang der Konvertierung umgekehrt, d.h. die empfangenen Daten werden von der Transferdarstellung in die gewünschte lokale Darstellung umgewandelt.

Die empfangenen Daten müssen der Dekonvertierungsroutine RPVCRB in Form eines Empfangspuffers, welcher als ein Byte-Array anzusehen ist, übergeben werden. Die aktuelle Länge des Puffers in Bytes muß der Konvertierungsroutine ebenfalls übergeben werden, ebenso wie die Anzahl der Bytes im Empfangspuffer, die schon verarbeitet worden sind. Diese beiden Parameter, aktuelle Länge und Verarbeitungsstand des Empfangspuffers, geben dem Benutzer die Möglichkeit, auch bei noch nicht vollständigem Empfang der Daten die schon empfangenen Daten bereits zu dekonvertieren.

Der Puffer wird, wenn möglich, nach erfolgreicher Dekonvertierung in 16 Bytes Schritten automatisch "gekürzt", d.h. vorne im Puffer stehende, bereits verarbeitete Daten werden mit weiter hinten stehenden, noch nicht verarbeiteten Daten überschrieben und die Parameter aktuelle Pufferlänge sowie Verarbeitungsstand an die neuen Verhältnisse angepaßt.

Die Größe (in Bytes) und der Typ eines Elements im Empfangspuffer müssen der Dekonvertierungsroutine in gleicher Weise übergeben werden wie auf der Senderseite für den Sendepuffer.

Auch das Datenarray, in welches die empfangenen Daten dekonvertiert werden, muß der Dekonvertierungsroutine in gleicher Weise bereitgestellt werden wie auf Senderseite, d.h. es müssen folgende Parameter übergeben werden:

- die gewünschte Länge (in Bytes) eines Elements im Datenarray
- ein Anfangsindex, ab dem die Daten in das Array dekonvertiert werden
- sowie ein Endeindex, ab dem keine dekonvertierten Daten mehr in das Array geschrieben werden dürfen.

Die Dekonvertierung wird nur so lange durchgeführt (oder überhaupt erst begonnen), solange der Anfangsindex nicht den Endeindex erreicht hat und die Daten im Empfangspuffer ausreichen. Eventuell ist ein erneuter Aufruf der Dekonvertierungsroutine nach dem Eintreffen neuer Daten erforderlich.

Flags zur Steuerung der Under-/Overflow Erkennung und der Behebung eines Underflows sind in der Dekonvertierungsroutine ebenfalls vorhanden.

Auf der Empfangsseite steht neben der Dekonvertierungsroutine noch eine Routine zur Vefügung, welche ein Alignment des Empfangspuffers auf volle 16 Bytes durchführt (RPVFRB) und den Empfangspuffer ggf. kürzt (RPVIRB).

Der Wertebereich der normalisierten Realzahlen $(-1)^s * 2^e * m$ auf den verschiedenen Rechnerarchitekturen ist gegeben durch:

- Vorzeichen $s = 0$ oder 1
- Mantisse $1 \leq m \leq 2$

- Exponent e:

Rechnerarchitektur	REAL*4	REAL*8	REAL*16
ieee	-126..127	-1022..1023	-16382..16383
dec	-126..127	-1022..1023	-16382..16383
vax	-128..126	-128..126	-16384..16382
cray	-8190..8189	-8190..8189	-8190..8189
ibm	-260..251	-260..251	-260..251
convex (ieee)	-126..127	-1022..1023	..
convex (native)	-128..126	-1024..1022	-16384..16382

Die Routinen sind im Kap. 6.4.37 „RPV... – Konversionsroutinen“ auf Seite 141 beschrieben.

5 Die RPI Definition Language

Der *Remote Procedure Interface Definition File* (**RPI-File**) enthält die Eingabedaten für den Stub-Generator. Er ist in der Fortran RPI Definition Language (**F-RPIDL**) zu erstellen. Ein RPI-File kann alle Informationen enthalten, damit der Stub-Generator die Fortran Quellen und die Stubs zu allen Objekten (Prozessen) einer verteilten Anwendung erstellen kann (Single Source Prinzip) .

Ein RPI-File kann aber auch nur Teile dieser Information enthalten:

1. Er enthält nur die RPI Definition ohne Fortran Quellen (s. Kap. 5.4, S. 84).
2. Er enthält nur Informationen zu einem Block (s.u.) oder zu einem Modul (s.u.). Damit können die Stubs modulweise generiert und übersetzt werden(s. Kap. 5.5, S. 84).

5.1 Das Block- und Modulkonzept

Um ein getrenntes Kompilieren der RPC-Stubs zu verschiedenen Server-Routinen zu ermöglichen, wurde ein vierstufiges Konzept eingeführt:

- Prozeß
- Block
- Modul
- Subroutine

Will man das Programm (local block), bestehend aus den Modulen CM, SM und CX verteilt rechnen lassen, wobei das Modul SM ausgelagert werden soll, so entsteht folgendes Szenarium (s. Abb. 5.1 auf Seite 54)

- Statt einem Prozeß benötigt man nun zwei, den Master- und den Slave-Prozeß. Die Bezeichnungen *Master* und *Slave* besagen, daß ersterer für den Start des letzteren verantwortlich ist, d.h. der Master-Prozeß in irgendeiner Form den Slave-Prozeß startet, wobei dies ggf. auch nur darin besteht, daß der Master-Prozeß den Anwender bittet, den Slave-Prozeß interaktiv zu starten.
- Ein *Block* ist das Programm, das in einem solchen Prozeß gestartet werden muß. Ein Block enthält wiederum ein oder mehrere Module.

  Mit *Local-Block* wird das nicht verteilte Programm bezeichnet. Mit *Client-Block* und *Server-Block* werden die Programme bezeichnet, die im verteilten Fall benützt werden. Client-Block ist der Block, von dem die Remote Procedure Calls ausgehen (Ausnahme: Callback, s.u.).

  Jeder Block hat einen *block_name* und einen *block_longname* . Der block_name dient intern als Schlüssel um den Block anzusprechen. Er wird auch bei der Generierung von Filename eingesetzt und sollte daher eher mnemotechnisch sein. Der block_longname kann ein beschreibender Name sein.
- Ein *Modul* ist die logische Zusammenfassung einer Gruppe von Subroutinen und Funktionen. Im nicht verteilten Fall, d.h. im sog. "local block" bestehen die Module nur aus den Subroutinen.

  Im verteilten Fall treten folgende Fälle auf:

  - Das Modul, das in der Abb. 5.1 den Namen CM trägt, enthält nur Routinen, die im Client-Block bekannt sind. Hierzu zählt vor allem das *Hauptprogramm* der Anwendung.
  - Das Modul CP im Client-Block wird speziell für das verteilte Rechnen gebraucht. Es enthält Routinen für den *Start des Slave-Prozesses* (siehe auch *Object Handling* und *Objekt-Start*).

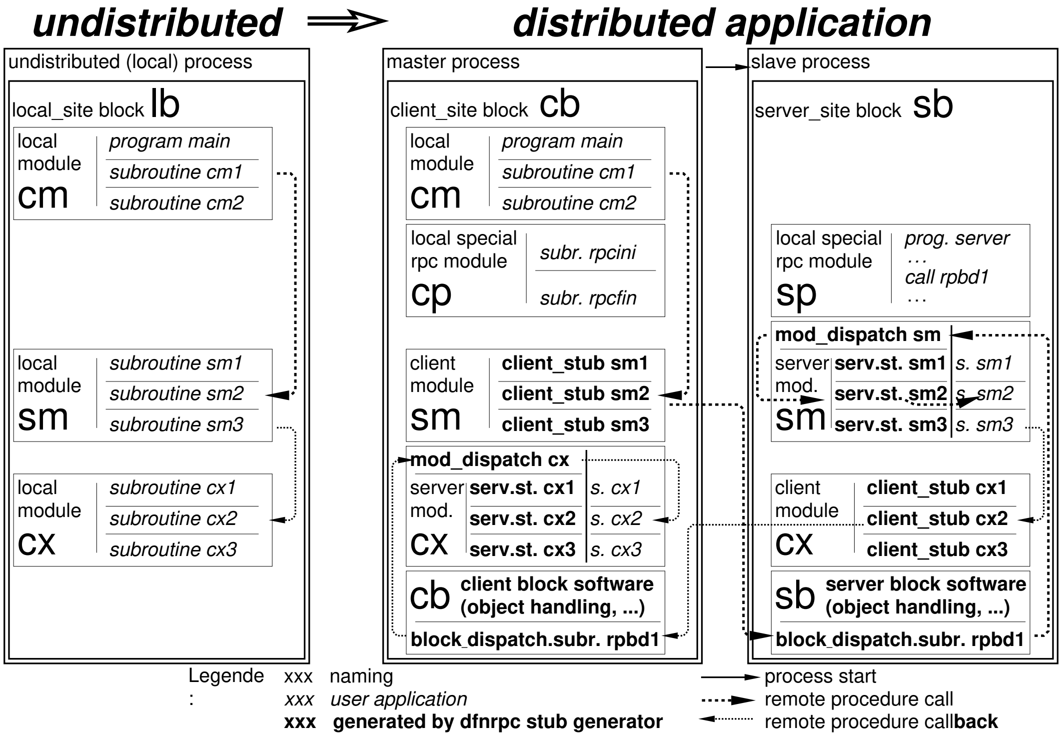


Abbildung 5.1: Vierstufiges Konzept – process, block, module, subroutine

  - Das Modul SP im Server-Block wird auch speziell für das verteilte Rechnen gebraucht. Es ist das Gegenstück zu CP und enthält im wesentlichen das Hauptprogramm des Server-Blocks. Das Modul SM der Anwendung soll im Server laufen. Im Client-Block ist hierzu das Client-Modul SM enthalten,welches aus den *Clientstubs* zu den Subroutinen dieses Moduls besteht. Das Server-Modul SM im Server-Block besteht aus den *Serverstubs*, die aufgrund der ankommenden Operation-Codes die zugehörige Serverstub-Routine aufruft, und den zugehörigen Subroutinen der Anwendung.

    Die Stubs können mit dem Stub-Generator generiert werden.

  - Das Modul CX enthält Routinen, die von den Subroutinen von SM gerufen werden und im Client-Block laufen sollen. Im verteilten Fall handelt es sich hier um sog. *Callbacks*. Daher befindet sich das CX Client-Modul im Server-Block und das CX Server-Modul (einschließlich der zugehörigen Subroutinen) im Client-Modul.

Jedes Modul hat einen *module_name* und einen *module_longname*. Der module_name dient intern als Schlüssel um das Modul anzusprechen. Er wird bei der Generierung von Filename, internen Subroutine- und Commonblocknamen eingesetzt und darf daher nur 2 Zeichen haben, das erste ein Buchstaben, das zweite alphanumerisch[1]. Der module_longname kann ein beschreibender Name sein.

- Die *Subroutinen* und Funktionen der Anwendung sind, wie oben beschrieben, den einzelnen Modulen zugeordnet.

---

[1] Generierte Subroutinenamen sind {module_name}_1 / 2 / 3 / 4 / ... / D1 / D2 / D3 / ... / INI / NAM. Ein generierter Commonblocknamen ist {module_name}_M. Alle Subroutinen des DFN-RPC Systems fangen mit RP oder CT4 an.

5.2 Adressierung

- Um eine Subroutine lokal aufzurufen, ordnet der Lader dem Subroutine-Namen eine Adresse zu, die dann für den Call und die Location der Subroutine benützt wird. Hierdurch ist ein schneller (indizierter) Zugriff möglich. Ein Zugriff auf der Basis des Namens der Subroutine selbst wäre jedesmal mit einem Suchvorgang verbunden.

  Beim Remote Procedure Call befinden sich Call und Location auf zwei verschiedenen Rechnern. Wie beim lokalen Call möchte man auch hier den Suchvorgang vermeiden. Da die Binde- und Ladevorgänge auf den beteiligten Rechnern unabhängig voneinander sind, wird beim DFN-RPC jeder Subroutine vom Stub-Generator ein *Operation-Code* zugeodnet. D.h. wenn die Anwendung in einem Client-Modul eine Subroutine aufruft, wird vom zugehörigen Clientstub der entsprechende Operation-Code zum Server-Modul übertragen und dort von der *Modul-Dispatcher-Routine* des Server-Moduls der entsprechende Serverstub aufgerufen, welcher dann die gewünschte Subroutine der Anwendung aufruft.

- Da aber in zwei verschiedenen Modulen die gleichen Operation-Codes vergeben werden können (nur dadurch können die Module unabhängig voneinander generiert und kompiliert werden), müssen innerhalb eines Blocks die Module mit einem *Modul-Code* versehen werden. Der Stub-Generator generiert zu jedem Client- und Server-Block einen *Block-Dispatcher* der bei Client-Modulen zum Operation-Code noch den Modul-Code hinzufügt und bei Server-Modulen die ankommenden Calls gemäß dem Module-Code an den entsprechenden Modul-Dispatcher weitergibt (S.a. Abb. 5.2).

- Die Adressierung des Slave-Prozesses mit enthaltenem Server-Block wird in Kapitel 4.10 auf Seite 45 beschrieben.

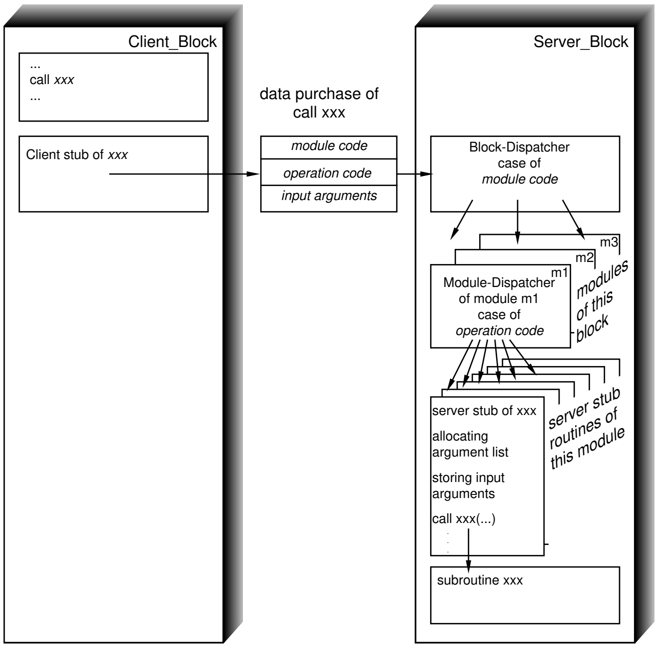


Abbildung 5.2: Dispatcher, Modul- und Operation-Code

5.3 Die Fortran RPI Definition Language

5.3.1 Eingabeformat

Für die RPI Definition gibt es folgende Formate:

- Eigenständige RPI-Statements der Form:

  `C*[function] parameter1 , parameter2 , ...`

  Mögliche *functions* sind:

  ```
  BLK_BEG,  MOD_BEG,  RPI_BEG
  BLK_END,  MOD_END,  RPI_END
  BLK_OPT,  MOD_USE,  RPI_NON
  OPTION,   MOD_PAR,  RPI_CON
  ```

- Fortran-Deklarationsanweisungen:
  SUBROUTINE, FUNCTION, REAL, INTEGER, COMPLEX, LOGICAL, DOUBLE PRECISION, CHARACTER, BYTE, DIMENSION, PARAMETER, EXTERNAL, ENTRY
- Sonstige Fortran-Anweisungen
- Innerhalb einer Fortran-Deklarationsanweisung können RPI-**Qualifier** auftreten. Hierfür gibt es zwei Möglichkeiten:
  - Inline - hierbei wird der Qualifier in eckigen Klammern angegeben, z.B.

    ```
    REAL x(*) [IN, MAXINDEX=n]
    ```

  - Auf speziellen Kommentarzeilen beginnend mit C∗RPC+, der Qualifier steht hier ebenfalls in eckigen Klammern, z.B.

    ```
              REAL x(*)
    C*RPC+              [IN, MAXINDEX=n]
    ```

- Fortran-Anweisungen, die nur bedingt in den Quellcode integriert werden sollen, müssen mit C∗RPC (erste Zeile einer Fortran-Anweisung) oder C∗RPC+ (Fortran-Fortsetzungszeile, statt + kann auch jedes andere Zeichen außer Blank und Null benützt werden), z.B.

  ```
  C*RPC call rpcini ( para1, para2, para3,
  C*RPC+              para4, para5, para6 )
  ```

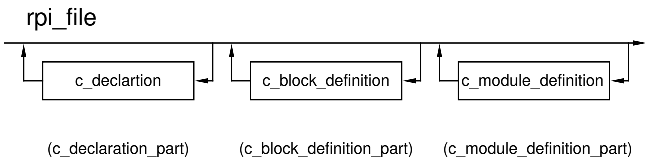
- Normale *Kommentarzeilen* können mit C, c oder ∗ beginnen. *Inline Kommentare* können zwischen ! (Ausrufezeichen) und Zeilenende stehen (wie bei VAX-Fortran).
- Schlüsselworte können beliebig mit Klein- und Großbuchstaben geschrieben werden.

5.3.2 Gesamtstruktur

Eine RPI Definition besteht aus folgenden Teilen:

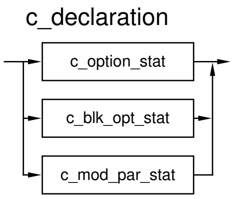
1. declaration part
2. block definition part
3. module definition part

Diese drei Teile müssen in dieser Reihenfolge auftreten. Prinzipiell können sie auch leer sein.



5.3.3 Declaration Part

Der Declaration Part besteht logisch aus drei Teilen. Trotzdem ist die Reihenfolge der Anweisungen des Declaration Parts beliebig.



1. Allgemeine Optionen
   C∗[OPTION] *keyword = value*
   Im folgenden sind als Beispiel-Wert immer die Defaults angegeben.

   C∗[OPTION] SEND_BUFFER_LENGTH = 65528

   > Alle RPC-Routinen eines Blocks benützen einen einzigen gemeinsamen Send-Buffer, um Daten gesammelt zu einem anderen Rechner zu schicken. Die Länge kann mit dieser Option festgelegt werden. Die Einheit sind Bytes (minimal 600,maximal 65528).
   > Die Option wird derzeit nicht ausgewertet.

   C∗[OPTION] RECEIVE_BUFFER_LENGTH = 65528

   > Siehe Erläuterungen zu SEND_BUFFER_LENGTH. Die RECEIVE_BUFFER_LENGTH darf auch kleiner sein als die SEND_BUFFER_LENGTH des Kommunikationspartners. Die darunterliegene Kommunikations-Systemsoftware führt automatisch die entsprechende Partitionierung durch (minimal 600,maximal 65528). Die Option wird derzeit nicht ausgewertet.

   C∗[OPTION] WORD_BUFFER_LENGTH = 100000

   > Alle RPC-Routinen eines Blocks benützen zwei Puffer (Word und Character) in den die Serverstubs alle aktuellen Argumente ablegen. In den Word-Buffer werden alle Integer-, Real-, Doubleprecision- und Complex-Arrays abgelegt. (Einzelne Nicht-Array-Argumente werden von den Serverstubs in lokalen Variablen abgespeichert). Der Word-Buffer wird grundsätzlich im Server-Block, aber auch in einem Client-Block initialisiert (auch wenn dieser keine Server-Module enthält) und von allen Server-Modulen benützt. Die Länge kann mit dieser Option festgelegt werden. Die Einheit sind Bytes.
   > **ACHTUNG**: Bei **getrennter Generierung** von *Block*-RPC-Software und *Server*-Modul-RPC-Software muß bei allen Generierungen zu einem Block der *gleiche Wert* angegeben sein! Die Werte, die beim Client-Block und bei den Server-Blöcken benützt werden, dürfen unterschiedlich sein, s. a. S. 66 und 96.

C*[OPTION] CHARACTER_BUFFER_LENGTH = 10000

Siehe Erläuterungen zu WORD_BUFFER_LENGTH. Im Character-Buffer werden Strings und String-Arrays abgespeichert (Bei MANUFACTURER=ibm nur 8000).

C*[OPTION] TRACE_DEPTH = 20

Im Fehlerfall schreibt der DFN-RPC eine Liste aller augenblicklich gerufenen Remote Procedure Calls und Callbacks. TRACE_DEPTH gibt an, bis zu welcher Schachtelungstiefe diese Liste geführt werden soll.

ACHTUNG: Siehe WORD_BUFFER_LENGTH

Die Option wird derzeit nicht ausgewertet.

C*[OPTION] OPERATION_BITS = 10

Für Module-Code und Operation-Code (siehe Kapitel 5.1 auf Seite 53) stehen zusammen 16 Bits zur Verfügung. OPERATION_BITS gibt an, wieviele Bits für den Operation-Code zur Verfügung stehen.
Die Module können dann maximal 2**OPERATION_BITS - 1 Subroutinen, Functions und Entries enthalten.

In der ganzen verteilten Anwendung können dann max. 2**(16-OPERATION_BITS) - 1 verschiedene Module existieren.

6 bis 14 OPERATION_BITS sind zulässig.

ACHTUNG: OPERATION_BITS wird nur bei der Generierung der Block-RPC-Software benützt. **Beim Client-Block und bei den Server-Blöcken muß der gleiche Wert benützt werden.**

C*[OPTION] OBJECTS_MAXIMUM = 512

Gibt an, wieviele Objekte im Object Handling gleichzeitig geöffnet sein können. Siehe Kapitel 4.8 auf Seite 42.

Die Option wird derzeit nicht ausgewertet.

C*[OPTION] WAIT_TIME_NEXT_CALL = 1000

Gibt an, wieviele Sekunden ein Server wartet bis der nächste RPC ankommt oder wie lange ein Client wartet bis ein RPC fertig ist.

C*[OPTION] WAIT_TIME_SEND = 300

Gibt an, wieviele Sekunden ein Clientstub wartet, bis ein Send-Buffer mit Input-Argumenten zum Serverstub abgesandt ist oder wie lange ein Serverstub wartet bis ein Send-Buffer mit Output-Argumenten zum Clientstub abgesandt ist. Dies ist bei asynchronen Calls oder sehr langsamen (verstopften) Netzen relevant.

C*[OPTION] WAIT_TIME_CONNECT = 100

Gibt an, wieviele Sekunden ein Block benötigen darf, um eine Transport-Verbindung aufzubauen.

C*[OPTION] WAIT_TIME_ACCEPT = 2000

Gibt an, wieviele Sekunden ein Block mit geöffnetem Port maximal wartet, bis eine Transport-Verbindung zu ihm aufgebaut wird.

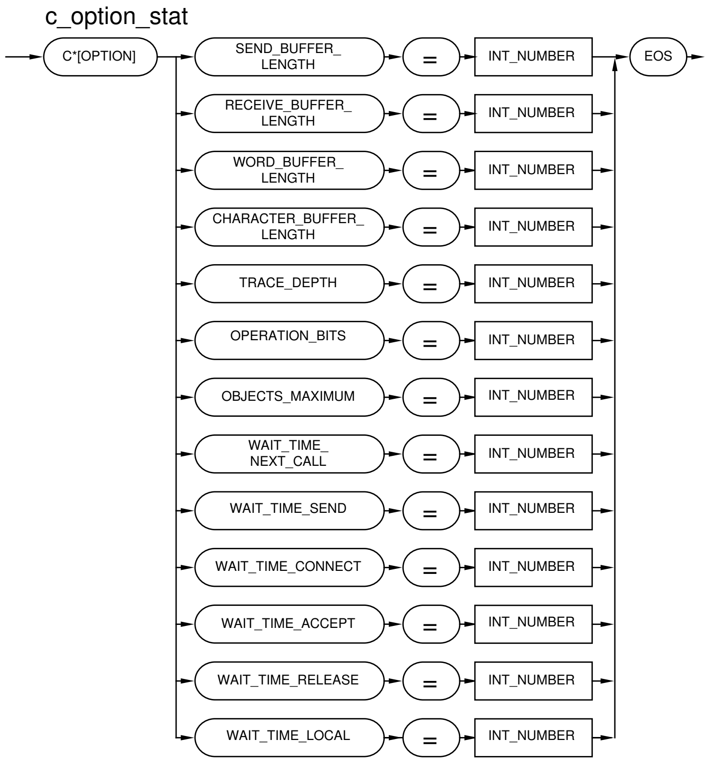
C∗[OPTION] WAIT_TIME_RELEASE = 200

Gibt an, wieviele Sekunden ein Verbindungsabbau maximal benötigen darf.

C∗[OPTION] WAIT_TIME_LOCAL = 100

Gibt an, wieviele Sekunden ein Block benötigen darf, um einen Transport Service oder Port zu eröffnen.

Die WAIT_TIMEs sind in Release 1.0 noch nicht realisiert,**d.h. im Augenblick warten die Prozesse beliebig lange**.



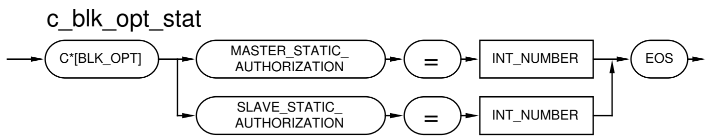
Die folgende Tabelle gibt an, welche Konstante an welcher Stelle in den Stub eingetragen und wo sie zur Laufzeit benutzt wird.

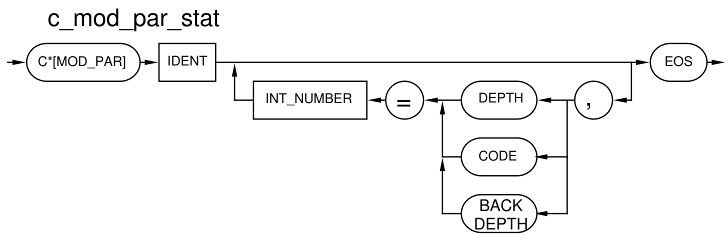
Konstante	geschrieben in den Stub-Sourcefile innerhalb der:	die Konstante wird benutzt von:
WAIT_TIME_... OPERATION_BITS	Block-Software, generiert durch BLOCK_DEFINITION	überall, d.h. in der Block- und Modul-Software und in den Stubs zu den einzelnen Routinen, client- und server-seitig
WORD_BUFFER_LENGTH CHARACTER_BUFFER_L.	Serverstubs der einzelnen Routinen	In den Serverstubs
SEND_BUFFER_LENGTH RECEIVE_BUFFER_LENGTH TRACE_DEPTH OBJECTS_MAXIMUM	Diese Optionen werden im Augenblick noch nicht ausgewertet, da sie bei der Installation des DFN-RPC in die Library hineinkompiliert werden.	

2. <u>Block Optionen</u>
   C∗[BLK_OPT] *keyword* = *value*
   Im folgenden sind als Beispiel-Wert immer die Defaults angegeben.

   C∗[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 0
   C∗[BLK_OPT] SLAVE_STATIC_AUTHORIZATION = 0

   Als Werte sind beliebige ganze Zahlen zwischen 0 und 999999999 (9 Stellen) zulässig. Mit diesen Zahlen testen die Prozesse bei der Initialisierung, ob sie zueinander gehören.





3. <u>Modul Parameter</u>
   C∗[MOD_PAR] *mod_name*,**DEPTH=***mod_depth*,**BACKDEPTH=***backdepth*,
   CODE=*mod_code*

mod_name Verweist auf das entsprechende Modul. Siehe auch Kapitel 5.3.5 auf Seite 66 und Kapitel 5.1 auf Seite 53.

DEPTH = mod_depth

Diese Angabe ist optional.

Siehe auch Abb.5.2 auf Seite 55 und Abb.5.5 auf Seite 62.

Wenn ein Modul als Server-Modul in einem Block (meist Server-Block) benützt wird, dann muß der Stubgenerator nicht nur für jede mittels RPC rufbare Subroutine einen Serverstub generieren, sondern er muß auch den Modul-Dispatcher generieren, welcher den Kopf (Header) von jedem ankommenden Remote Procedure Call interpretiert und die zugehörige Serverstub-Routine aufruft. Die Serverstub-Routine ruft dann die Anwendungs-Routine. Wenn diese wiederum einen zweiten Server mittels RPC ruft oder wenn sie einen Callback macht, d.h. wenn sie eine Subroutine im Client-Block mittels RPC ruft, und wenn die so gerufene Routine wieder einen Callback zum Server-Block macht, dann werden in diesem der Block-Dispatcher und der Modul-Dispatcher **rekursiv** gerufen.

In Abb. 5.2 auf Seite 55 wurden die Aufgaben von Block- und Modul-Dispatcher anhand eines einfachen Remote Procedure Calls dargestellt. Im Falle von geschachtelten Callbacks, wie in Abb. 5.3, muß zumindest auf der Server_Site der Block-Dispatcher zweimal gleichzeitig gerufen sein, nämlich für die Verzweigung zur Subroutine A gemäß Modul- und Operation-Code im CALL A und für die Verzweigung zur Subroutine C, s. Abb. 5.4. D.h. in Abb. 5.2 muß noch eine weitere Dimension der Tiefe eingeführt werden. Siehe auch Abb.5.5 auf Seite 62.

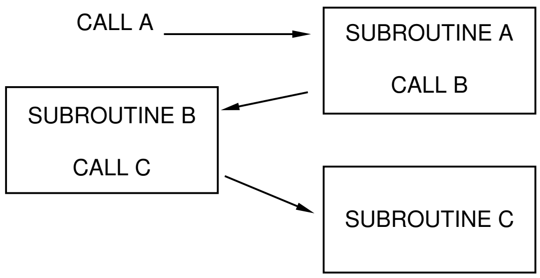


Abb. 5.3: Geschachtelter Callback, 2-stufig

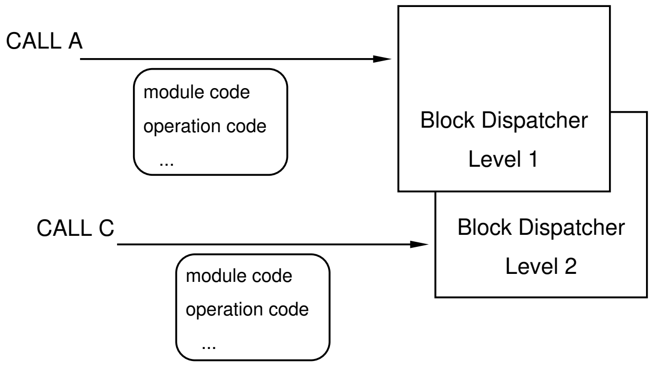


Abb. 5.4: Blockdispatcher ist 2 mal gleichzeitig gerufen

DEPTH=0 bedeutet, daß dieser rekursive Aufruf erlaubt ist[2].

DEPTH=mod_depth (>0) bedeutet, daß der Stubgenerator mod_depth äquivalente Modul-Dispatcher Routinen generiert um die Rekursion zu umgehen. Dieser Wert wird auch im Block-Dispatcher benützt, welcher die Modul-Dispatcher Routine(n) ruft.

Default: DEPTH=0.

BACKDEPTH = backdepth

Diese Angabe ist optional.

Der *backdepth* Wert wird in jedem Clientstub dieses Modules benützt um die Verzweigungen (Calls) auf die entsprechende Anzahl von Block-Dispatcher Routinen zu generieren. Diese Verzweigungen werden bei Callbacks (siehe DEPTH) durchgeführt.

BACKDEPTH >= 0 - siehe DEPTH

BACKDEPTH=-1 bedeutet, daß keine ankommenden Callbacks erwartet werden, d.h. daß keine Verzweigung auf einen Block-Dispatcher generiert wird.

Default: BACKDEPTH=0

CODE = mod_code

Diese Angabe ist optional.

Jedes Modul, das als CLIENT- oder SERVER-Modul in einem Block benützt wird, muß einen *module_code* bekommen. Dieser kann automatisch durch den Stubgenerator oder explizit in einem MOD_PAR Statement zugewiesen werden. Dieser Code muß in jedem Block, in dem das Modul eingesetzt wird, gleich sein. Wenn man

[2] Obwohl Standard Fortran 77 Rekursionen nicht kennt, sind die meisten Fortran Laufzeitsysteme unter Posix dazu in der Lage

die Blöcke getrennt generieren will, dann ist die explizite Zuweisung im MOD_PAR Statement zu empfehlen. Der Code muß im Bereich 1 - 2**(16-OPERATION_BITS)-1 liegen. Beim Default von OPERATION_BITS = 10 muß der module_code also zwischen 1 und 63 liegen. S.a. S. 58, OPERATION_BITS. Default: Die automatische Zuweisung wird durchgeführt, wenn im MOD_PAR Statement kein Wert angegeben ist und das Modul zum ersten Mal in einem MOD_USE Statement als CLIENT oder SERVER eingesetzt wird. Hierbei werden die Zahlen 1,2,... vergeben (mit Ausnahme der Werte, die explizit in den MOD_PAR Statements benutzt werden).

Siehe auch Kapitel 5.2 auf Seite 55.

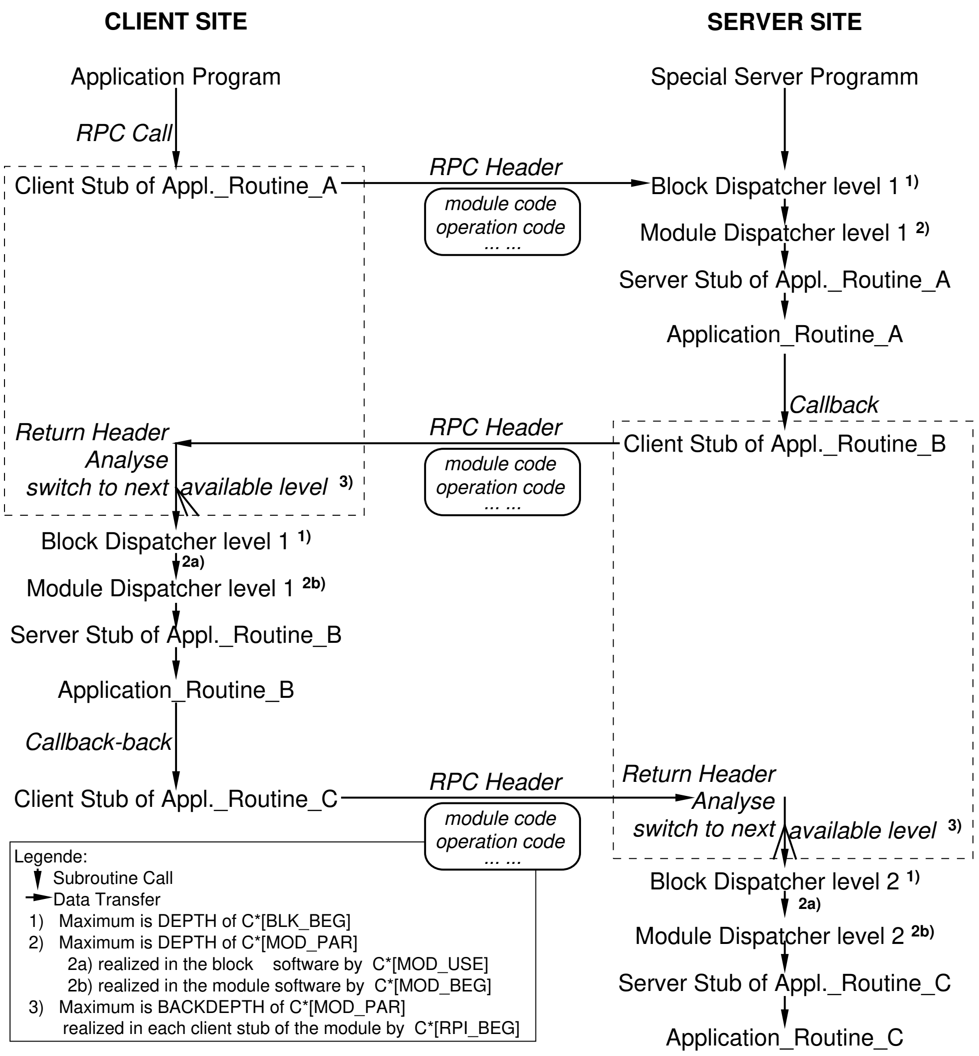


Abb. 5.5: Block-, Module- und Back-Depth

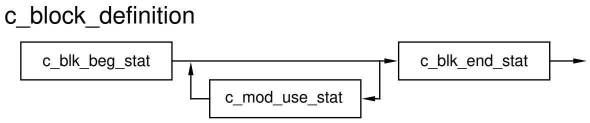
5.3.4 Block Definition Part

Der Block Definition Part besteht aus einzelnen *Block Definitions*.

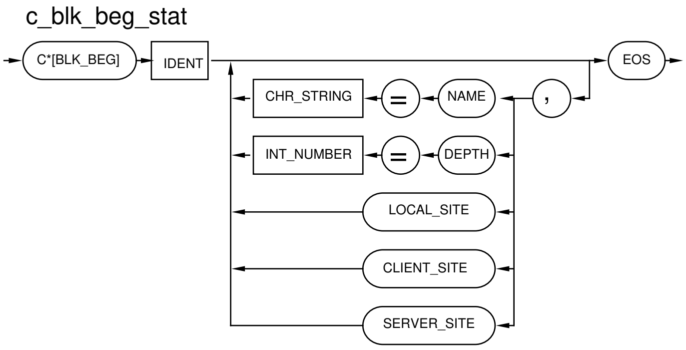
Eine einzelne Block Definition hat folgenden Aufbau:

C*[BLK_BEG] *block_name*,**NAME=***'block_longname'*,**DEPTH=***block_depth, site*
C*[MOD_USE] *module_name, module_type*
...
C*[BLK_END]

Ein Block kann ein oder mehrere Module enthalten.



C*[BLK_BEG] *block_name*, **NAME=***'block_longname'*,**DEPTH=***block_depth, site*



markiert den Anfang einer Block Definition.

block_name

Definiert den block_name dieses Blocks, siehe auch Kapitel 5.1 auf Seite 53.

NAME = block_longname

Definiert den block_longname dieses Blocks. Diese Angabe ist optional, Default ist der block name.

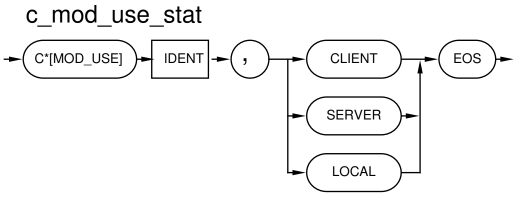
DEPTH = block_depth

Diese Angabe ist optional. Siehe auch Kapitel 5.3.3,Abschnitt "Modul Parameter" auf Seite 60 und Abb. 5.5 auf Seite 62.
DEPTH=0 bedeutet, daß der Block-Dispatcher rekursiv gerufen werden darf[3].
DEPTH=block_depth (>0) bedeutet, daß der Stubgenerator block_depth äquivalente Block-Dispatcher Routinen generiert, um die Rekursion zu umgehen.
DEPTH=-1 ist nur beim CLIENT_SITE Block sinnvoll und bedeutet, daß keine ankommenden Callbacks erwartet werden, d.h. daß kein Block-Dispatcher generiert wird.
Default: DEPTH=0

[3] Obwohl Standard Fortran 77 Rekursionen nicht kennt, sind die meisten Fortran Laufzeitsysteme unter Posix dazu in der Lage

site Für *site* kann man die Werte **CLIENT_SITE**, **SERVER_SITE** oder **LOCAL_SITE** angeben. Mit CLIENT_SITE erzeugt man einen Client-Block, usw., siehe Kapitel 5.1 auf Seite 53.

C*[MOD_USE] *module_name*, *module_type*



fügt das entsprechende Modul in diesen Block ein.

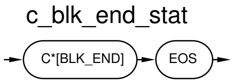
module_name

Siehe Kapitel 5.1 auf Seite 53.

module type

Für *module_type* kann man die Werte **CLIENT**, **SERVER** oder **LOCAL** angeben.
Ein Client-Modul enthält nur die Clientstubs zu diesem Modul, ein Server-Modul enthält die Serverstubs zu diesem Modul und die zu diesem Modul gehörenden Subroutinen selbst. Ein Local-Modul enthält nur die zu diesem Modul gehörenden Subroutinen.

C*[BLK_END] markiert das Ende einer Block Definition.



Ein Client-Block (in nachfolgendem Beispiel cb) muß ein Local-Modul (cm) mit dem Hauptprogramm und mindestens ein Client-Modul (sm) enthalten. Er kann auch Server-Module enthalten, zu dessen Subroutinen dann vom entfernten Prozeß aus sog. Callbacks gemacht werden können (nicht im Beispiel enthalten). Meist enthält er ein weiteres Local-Modul (cp), in dem die Subroutinen zur Initialisierung und Beendigung des entfernten Server-Blocks enthalten sind.

Ein Server-Block (sb) muß ein Local-Modul (sp) mit einem Hauptprogramm enthalten, das speziell für den server-seitigen Start des DFNRPC-Tools notwendig ist. Außerdem muß er das Server-Modul (passend zu obigem Client-Modul, also wie oben sm) enthalten. Falls Callbacks, d.h. RPCs vom Server-Block zurück zum Client-Block gemacht werden sollen, so muß der Server-Block auch ein Client-Modul (passend zu obigen Server-Modul) enthalten.

Bei Client- und Server-Blöcken werden **C*RPC** und **C*RPC+** Zeilen aktiviert, siehe auch Kapitel 5.3.1 auf Seite 56.

Ein Local-Block (lb) darf keine RPCs machen und kann daher nur Local-Module enthalten (cm und sm).

Beispiel:

```
C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     cm, LOCAL
C*[MOD_USE]     cp, LOCAL
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sp, LOCAL
C*[MOD_USE]     sm, SERVER
C*[BLK_END]

C*[BLK_BEG] lb, NAME='local program', DEPTH=0, LOCAL_SITE
C*[MOD_USE]     cm, LOCAL
C*[MOD_USE]     sm, LOCAL
C*[BLK_END]
```

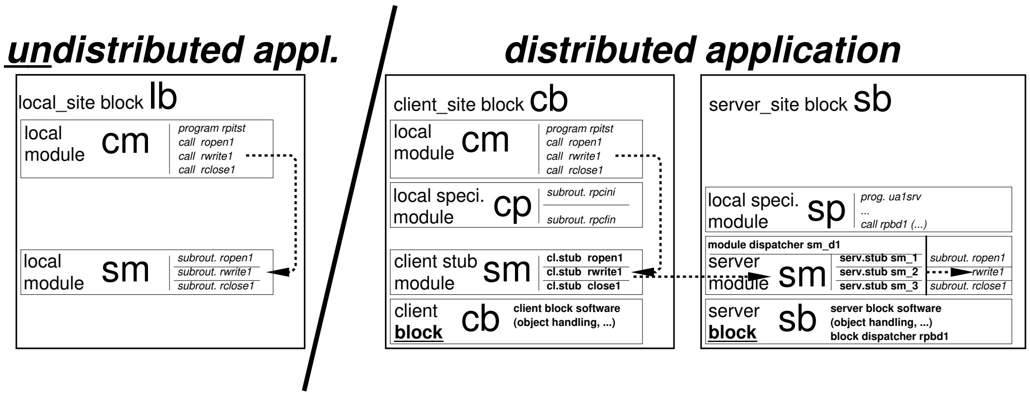


Abbildung 5.6: Einfaches Beispiel

Aufgrund des Block Definition Parts kann der Stubgenerator die Application- und Stub-Source-Files für die einzelnen Blöcke generieren.
Liegt das obige Beispiel vollständig einschließlich des zugehörenden Modul Definition Parts auf einem Inputfile (z.B. inp.rpi) vor, dann generiert der Stub-Generator folgende Files:

```
                                   inp.rpi
                                      |
              +-------------------------+-------------------------+
For the       |                         |                         |
blocks ..... cb                        sb                        lb
following     |\                        |\                        |
files will    | \                       | \                       |
be generated: | inp_cb_stb.m4           | inp_sb_stb.m4           |
              | |                       | |                       |
       inp_cb.f inp_cb_stb.f    inp_sb.f inp_sb_stb.f    inp_lb.f

Stub-Presourcefile:     inp_cb_stb.m4  and  inp_sb_stb.m4
Stub-Sourcefile:        inp_cb_stb.f   and  inp_sb_stb.f
Application-Sourcefile: inp_cb.f       and  inp_sb.f  and inp_lb.f
```

Liegt dagegen nur der Block Definition Part (einschließlich Declaration Part) auf inp.rpi, also wenn man die Module *getrennt generieren* will, dann entstehen:

```
                                   inp.rpi
                                      |
               +----------------------+
For the        |                      |
blocks ..... cb                      sb
               |                      |
following       \                      \
files will       \                      \
be generated:    inp_cb_stb.m4          inp_sb_stb.m4
                 |                      |
                 inp_cb_stb.f           inp_sb_stb.f
```

Die Stub-Files enthalten nur den Block-Part der Stubs, d.h. die Block-Dispatcher-Routinen und die Object-Start-Routinen.

Der Block Definition Part kann auch entfallen. Dann werden die Module getrennt generiert.

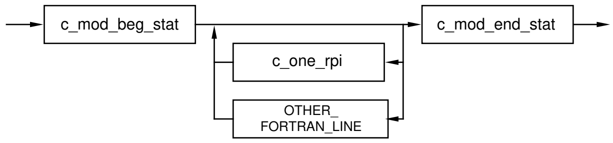
5.3.5 Module Definition Part

Der Module Definition Part besteht aus einzelnen *Module Definitions*.

Eine einzelne Module Definition hat folgenden Aufbau:

C∗[MOD_BEG] *module_name*,**NAME=***'module_longname'*   **module_body** **C∗[MOD_END]**

c_module_definition



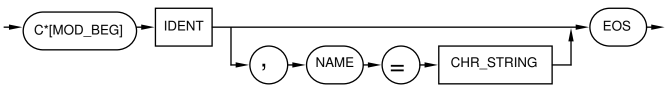
C∗[MOD_BEG] *module_name*,**NAME=***'module_longname'*

markiert den Anfang einer Module Definition.

module_name
: Definiert den module_name dieses Moduls, siehe auch Kapitel 5.1 auf Seite 53.

NAME = module_longname
: Definiert den module_longname dieses Blocks.

c_mod_beg_stat



module_body

Der module_body besteht aus der *Source des Application-Programms* einschließlich der *Remote Procedure Interface* (**RPI**) *Definition* aller Subroutinen, Funktionen und Entries dieses Moduls.

Grundsätzlich sind an jeder Stelle **C∗RPC** und **C∗RPC+** Zeilen möglich, siehe auch Kapitel 5.3.1 auf Seite 56 und Kapitel 5.3.4 auf Seite 63.

Formal besteht der *module body* aus einer Folge von

- Lokalen Routinen

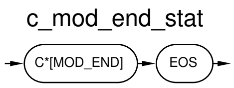
  d.h. lokales Hauptprogramm, lokale Subroutinen und lokale Funktionen. Diese werden vom Stub-Generator überlesen und hierbei nur auf die Fortran-Files der Blöcke geschrieben, die dieses Modul mit dem *module_type* SERVER oder LOCAL enthalten (s. C∗[MOD_USE]).

- Remote Procedure Interfaces (RPI)

  Remote Procedure Interfaces werden vom Stub-Generator folgendermaßen behandelt (siehe auch Kapitel 5.3.6 auf Seite 69):

  1. Wie lokale Routinen, d.h. die zugehörende Fortran-Source wird auf die gleichen Fortran-Files wie die anderen lokalen Routinen geschrieben.
  2. Auf der Basis der Soubroutine/Function-Header und der eingestreuten RPI-Qualifier (s. nächstes Kapitel) werden die Client- und Serverstubs zu diesem Modul einschließlich der Modul-Dispatcher-Routine generiert.

C∗[MOD_END] markiert das Ende einer Module Definition.



Beispiel (passend zum Block Definition Beispiel im letzten Kapitel):

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

      PROGRAM ...
      ...
C*RPC CALL rpcini            ! These lines will be duplicated and
      ...                    ! the second activated in the generated
C*RPC CALL rpcfin            ! source of block cb
      ...
      END
                             ! Source for the client module
      SUBROUTINE ...
      ...
      END

      FUNCTION ...
      ...
      END

C*[MOD_END]
```

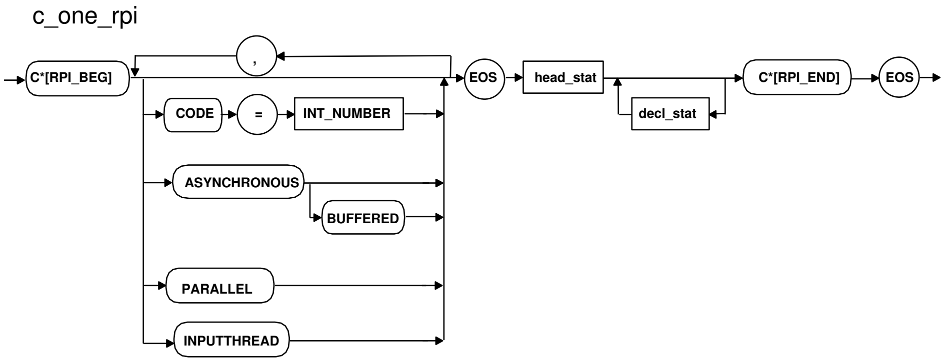
```
C*[MOD_BEG] cp, NAME='client rpc special routines '

      SUBROUTINE rpcini      ! Examples will be provided
      ...
      END

      SUBROUTINE rpcfin
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sp, NAME='server rpc special program  '

      PROGRAM rpcsrv         ! Special main program for the server block
      ...
      CALL BLKD1 (iret)      ! The block-dispatcher-routine
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sm, NAME='server module - server subroutines'

C*[RPI_BEG]
      SUBROUTINE sub1 (a,b,c) ! This is a RPI
      REAL a[IN], b[IN]
      REAL c
C*RPC+      [OUT]
      ...
C*[RPI_NON]
      a = b * b
      RETURN
      END
C*[RPI_END]

      SUBROUTINE ...         ! This is a local routine, i.e.
      ...                    ! not callable by remote blocks
      END

C*[MOD_END]
```

5.3.6 Remote Procedure Interface (RPI) einer Routine

In einem Modul, das remote rechnen soll, müssen alle Routinen, die der Client-Prozeß rufen können soll, als Remote Procedure Interface (RPI) deklariert werden. Nicht als RPI deklarierte Routinen stehen nur serverseitig (als Hilfsroutinen) diesem Modul zur Verfügung.

Ein RPI hat folgenden Aufbau:

C∗[RPI_BEG] CODE=*operation_code*, **ASYNCHRONOUS** oder **ASYNCHRONOUS BUFFERED** oder **PARALLEL** oder **INPUTTHREAD** **routine_header** **declaration_statements** . . . **C∗[RPI_END]**



C∗[RPI_BEG] CODE=*operation_code*, **ASYNCHRONOUS** *oder* **ASYNCHRONOUS BUFFERED** *oder* **PARALLEL** *oder* **INPUTTHREAD**

markiert den Anfang eines Remote Procedure Interfaces.

CODE = operation_code

Bei jedem Modul hat man die Wahl, ob man die Operation_codes selbst bestimmen will oder ob man sie vom Stubgenerator zugeteilt bekommen möchte. Im ersten Fall muß man bei jeder RPI den Code explizit angeben, im zweiten Fall darf man bei keiner RPI des Moduls den Code vorgeben. Die explizite Angabe ist bei der Erstellung von Bibliotheken vorteilhaft, denn wenn man nachträglich neue Server-Routinen hinzufügt, ändern sich die alten Codes nicht, wodurch alte Clientprogramme auch die neue Server-Bibliothek benützen können.
Als Werte sind die Zahlen 1 bis 2∗∗OPERATION_BITS - 1 möglich, d.h. beim Default OPERATION_BITS=10 sind die OPERATION_CODEs 1 bis 1023 zugelassen, s.a. S. 58, OPERATION_BITS.

weder ASYNCHRONOUS noch ASYNCHRONOUS BUFFERED
noch PARALLEL

In diesem Fall wird ein **synchroner RPC** durchgeführt, d.h. der Clientstub schickt die Inputargumente zum Serverstub, dieser ruft die Anwendungsroutine auf während der Clientstub wartet und anschließend werden

die Outputargumente wieder vom Server- zum Clientstub zurückübertragen, s. Abb. 5.7.
Man sollte hierbei auch beachten, daß die meisten Netze und Rechner nur 5 - 200 RPCs/sec (bei leerer Argumenteliste) zulassen. Daher sollte man nie mehr als 10 synchrone RPCs/sec beabsichtigen.[4]

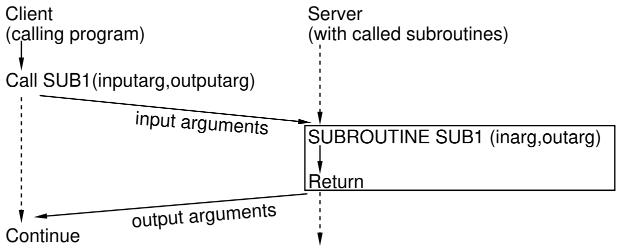


Abbildung 5.7: Synchroner RPC

ASYNCHRONOUS

Beim asynchronen RPC schickt der Clientstub nur die Inputargumente zum Serverstub und gibt dann sofort die Kontrolle an das Anwendungsprogramm zurück. Auf der Serversite wird dann gleichzeitig die mittels RPC gerufene Routine ausgeführt, s. Abb. 5.8. Die Routine darf **keine Outputargumente** besitzen und keine Funktion sein. Ergebnisse können aber serverseitig in Commonblöcken abgespeichert und mit nachfolgenden synchronen RPCs zurückübertragen werden. Eine Synchronisierung findet wieder beim nächsten synchronen RPC statt.
Asynchrone Calls ermöglichen eine Parallelisierung von Client und Server. Bei direkt hintereinander folgenden Aufrufen, d.h. wenn man die Parallelisierung nicht ausnützt, erreicht man bei leerer Argumenteliste nur ca. die 2 bis 3-fache Anzahl RPCs/sec gegenüber synchronen RPCs (hier seien dann die im folgenden beschriebenen RPCs empfohlen).[4]

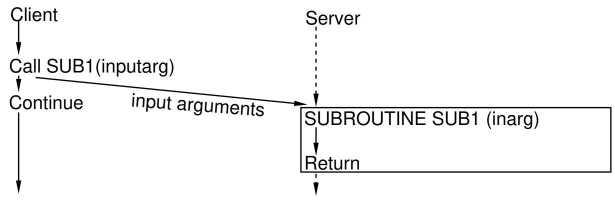


Abbildung 5.8: Asynchroner RPC

ASYNCHRONOUS BUFFERED

Beim asynchron, gepufferten RPC wartet der Clientstub mit dem Verschicken der Inputargumente bis der Send_Buffer voll ist oder bis ein nachfolgender asynchron ungepufferter oder synchroner RPC gerufen wird.
Dadurch kann die Anzahl der Betriebssytem-Aufrufe minimiert werden und eine maximale Übertragungsgeschwindigkeit erreicht werden. s. Abb. 5.9.

[4] Messung auf einem Ethernet mit TCP zwischen zwei Sun4, jeder Aufruf mit 28 Byte Argumenteliste (plus 4 Byte Header): 3.0 ms/Call synchron, 1.2 ms/Call asynchron ab 100 Calls in Folge, 0.18 ms/Call asynchron gepuffert bei 85 Calls (d.h. 2 volle MTUs mit je 1500 Byte werden pro Systemcall abgeschickt).

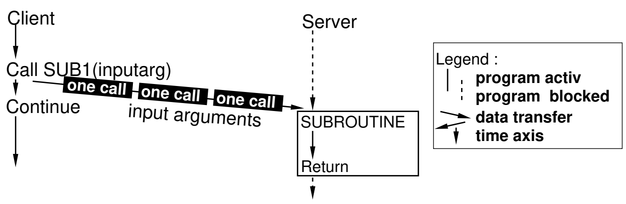


Abbildung 5.9: Asynchroner, gepufferter RPC

Die Routine darf ebenfalls keine Outputargumente besitzen und keine Funktion sein. Durch die Pufferung wird bei kurzen Argumentelisten die Round-Trip-Time deutlich reduziert.[4]

PARALLEL

Beim parallelen RPC wird – wie beim asynchronen RPC – nach dem Verschicken der Inputparameter die Kontrolle sofort wieder an das Clientprogramm zurückgegeben. Auf der Serverseite wird dann gleichzeitig die mittels parallelem RPC gerufene Serverroutine ausgeführt, siehe Abb. 5.10.

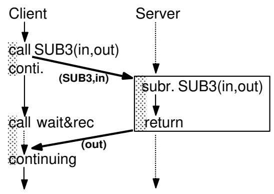


Abbildung 5.10: Der parallele RPC

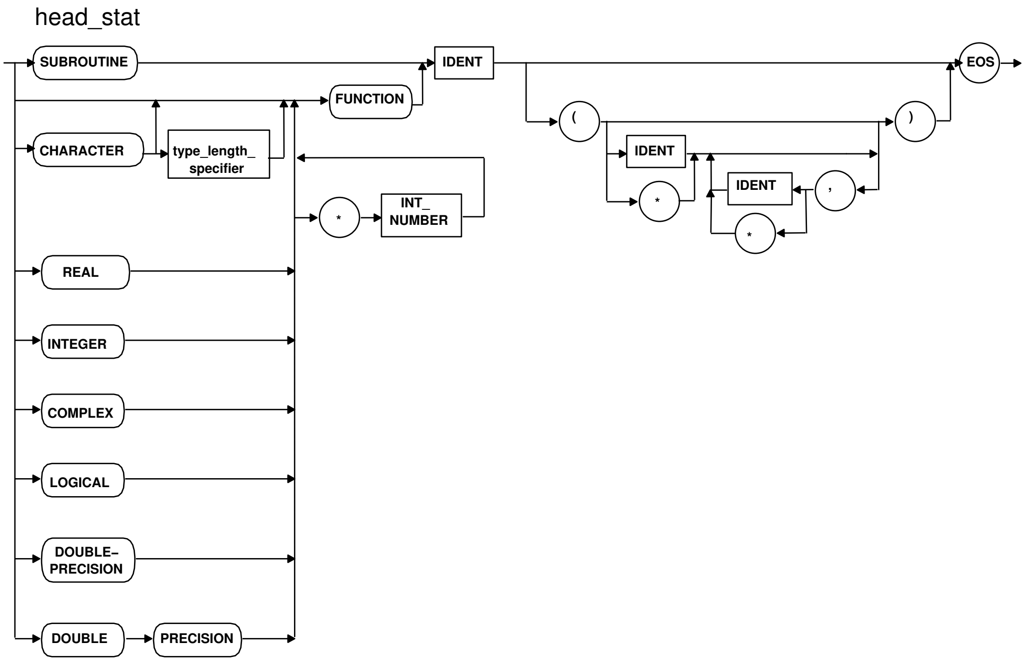
Nach Beendigung der Serverroutine sendet der Serverstub die Outputparameter zurück und wird hierbei ggf. blockiert bis Clientseitig die Library-Routine RPOREC gerufen wird.
RPOREC schreibt die Outputwerte in die beim parallelen RPC-Aufruf angegebenen aktuellen Outputparameter.
S.a. Library-Routinen RPOISP zum Untersuchen eines angekommenen Pakets und RROCRY,RPOLRY,RPORRY und RPOGRY zum Warten auf ein Returnpaket (und RPORCT für ein threaded Receive – noch nicht implementiert).

INPUTTHREAD

Versenden der Inputparameter in einer parallelen Thread.
Diese Angabe ist optional und nur bei ASYNCHRONEN oder PARALLELEN RPCs möglich. INPUTTHREAD ist in den generierten Stubs derzeit noch nicht implementiert.

Anschließend muß ein **Header_Statement** folgen. Es entspricht der normalen Fortransyntax.

SUBROUTINE ... **... FUNCTION ...**



Danach können folgende **Fortran-Deklarationsstatements** stehen:

CHARACTER ...
INTEGER ...
LOGICAL ....
REAL ...
DOUBLE PRECISION ...
COMPLEX ...
BYTE ...
DIMENSION ...
PARAMETER ...
EXTERNAL ...
ENTRY ...

In diese Deklarationen können **Qualifier** eingefügt werden, mit denen man alle für die Stubgenerierung zusätzlich benötigten Informationen bereitstellen kann. Die Qualifier sind in den nachfolgenden Kapiteln beschrieben.

Außerdem werden innerhalb eines RPI auch **Include-Statements** interpretiert

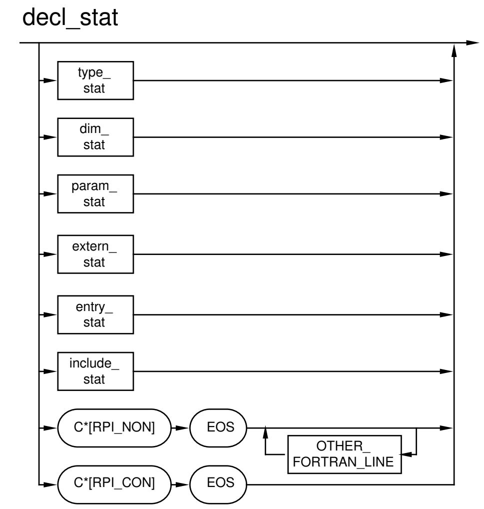
	INCLUDE	*filename*
oder	**INCLUDE (**	*filename* **)**
oder	**INCLUDE** '	*filename* '
oder	**INCLUDE (**'	*filename* '**)**

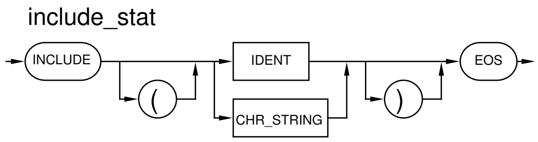
Alle anderen Fortranzeilen und auch solche Deklarationen, die vom Stubgenerator nicht interpretiert werden sollen, müssen in **Nonparseable Blocks**:

C∗[RPI_NON]
andere Fortranzeilen, etc.
C∗[RPI_CON]

Wenn danach keine weiteren Deklarations-Statements folgen, d.h. direkt C∗[RPI_END] folgt, kann man das C∗[RPI_CON] weglassen.

Nonparseable Blocks werden eingesetzt um Equivalence- und Commonblock-Deklarationen, Data-Statements und ausführbare Statements, d.h. alle für die Stubgenerierung unnötige Informationen, zu klammern und damit dem Stubgenerator zu verbergen. Man beachte, daß alle Routinen, die nicht als RPI deklariert sind, ebenfalls als Nonparseable Block behandelt werden.

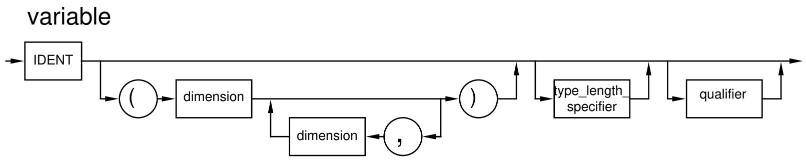




5.3.7 Numerische und String-Argumente

In den Deklarationen der formalen Argumente kann man mittels Qualifier die für die Stubgenerierung zusätzlich benötigten Informationen angeben.

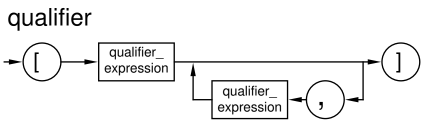
Ein Qualifier gehört grundsätzlich nur zu einem formalen Argument und steht nach den optionalen Dimensionsangaben (...) und Typlängenspezifikationen *1/2/4/8/16/32.



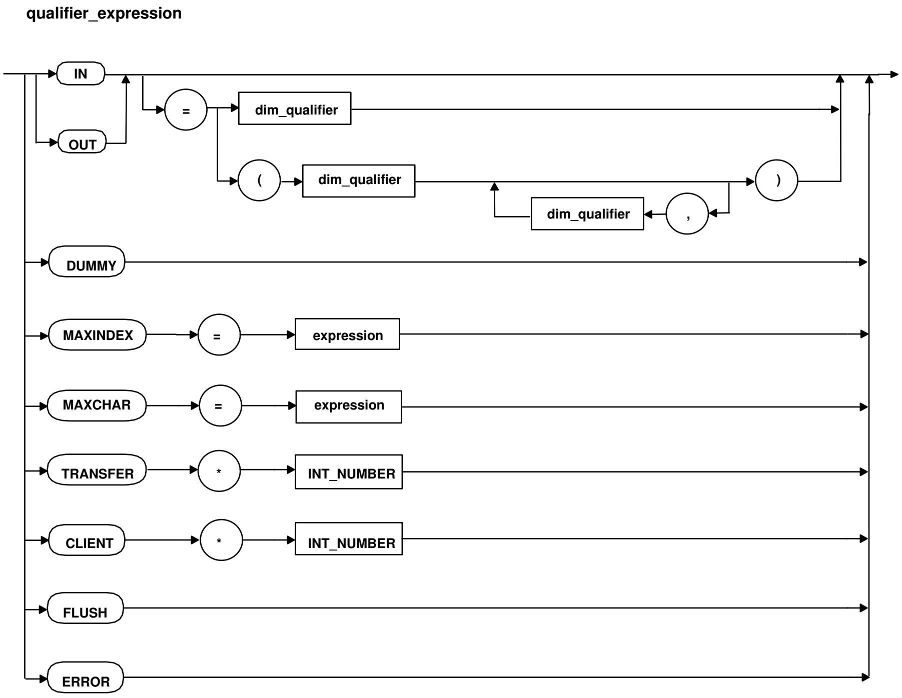
Die allgemeine Syntax der Deklarationsstatements ist in Kap. 5.6 auf S. 85 beschrieben.

Der **Qualifier** steht **in eckigen Klammern** und besteht aus einem oder mehreren **Qualifier_Expressions**.

type	*identifier*	[*Qualifier_Expression*]
oder		[*Qualifier_Expression1* , *Qualifier_Expression2* , **...**]
DIMENSION		[**...**] , *identifier2* , [**...**] , ...



Eine Qualifier_Expression beginnt mit einem der Schlüsselworte **IN**, **OUT**, **DUMMY**, **MAXINDEX**, **MAXCHAR**, **TRANSFER**, **CLIENT**, **FLUSH** oder **ERROR**.



Die Bedeutung der einzelnen **Qualifier_Expressions** ist in den folgenden Kapiteln beschrieben.

5.3.7.1 IN, OUT, DUMMY

IN
IN = *Dimension_Qualifier*
IN = (*Dimension_Qualifier1, Dimension_Qualifier2* , ...)
OUT
OUT = *Dimension_Qualifier*
OUT = (*Dimension_Qualifier2, Dimension_Qualifier2* , ...)
DUMMY

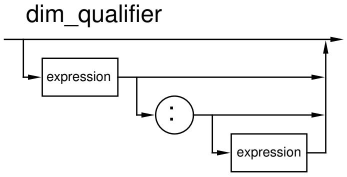
Im Gegensatz zum lokalen Prozeduraufruf kann man bei einem RPC die Argumente nicht mit Call-by-reference übergeben. Statt dessen wird Call-by-in-copy-and-out-copy gemacht, d.h. die Werte der Input-Argumente werden vom Client zum Server vor dem Aufruf der eigentlichen Subroutine übertragen. Bei dieser Übertragung werden die Werte automatisch konvertiert, und zwar clientseitig von der Client-Zahlendarstellung in eine Transferkodierung und serverseitig von der Transferkodierung in die Server-Zahlendarstellung. Die Transferkodierung ist beim DFN-RPC frei wählbar, s. Option -T des Stubgenerators dfnrpcgen, S. 159. Normalerweise sollte die Transferkodierung einer der beiden Zahlendarstellungen entsprechen. Nach Abschluß der Subroutine werden die Output-Argumente wieder zum Client zurück übertragen.

Bei Call-by-reference ist die Richtung, in der die Argumente eingesetzt werden (in oder out), für den Compiler egal und sie kann und muß daher in Fortran nicht notiert werden. Beim RPC muß man dagegen wissen, ob ein Argument als Input- oder/und Output-Argument benützt wird und entsprechend in den Stubs ein In-Copy und/oder Out-Copy durchgeführt werden muß.

Beispiel:

```
C*[RPI_BEG]
C     [IN], [OUT], [IN,OUT] and [DUMMY] Arguments
C     -------------------------------------------

      SUBROUTINE sub1 (a,b,c,d,e,x)
      REAL    a[OUT],  b[IN],  c[IN],  d, e[IN,OUT], x[DUMMY]

C     the default [in,out] is used for d
C     x is dummy, i.e. a workspace argument, i.e. it will not be
C     transferred, neither for input nor for output

C*[RPI_NON]
      a = b * c
      x = d
      d = e
      e = x
      RETURN
      END
C*[RPI_END]
```

Mit den Dimension_Qualifiern kann man einen Übertragungsauschnitt festlegen. Wenn man weniger Dimension_Qualifier angibt als das Array Dimensionen besitzt, dann werden die Dimension_Qualifier für die ersten Dimensionen benützt. Die restlichen Dimensionen werden vollständig, d.h. gemäß ihrer Deklaration übertragen.



Ein Dimension_Qualifier kann drei Formen haben:

leer	d.h. daß das Array in dieser Dimension vollständig übertragen wird,
expression	d.h. daß das Array in dieser Dimension vom Anfang bis zu dem dadurch bestimmten Index übertragen wird,
expr1:expr2	d.h. daß das Array in dieser Dimension nur in dem durch die Ausdrücke bestimmten Bereich übertragen wird.

Mit **IN** gibt man an, ob und mit welchem Array-Ausschnitt das aktuelle Argument als Input-Argument übertragen werden soll.

Mit **OUT** gibt man den Übertragungsbereich als Output-Argument an. Es können mehrere IN und OUT Übertragungsbereiche angegeben werden.

Bei **DUMMY** werden keine Werte übertragen, d.h. für das Argument wird auf der Serverseite nur Speicher bereitgestellt.

Wenn weder IN, noch OUT, noch DUMMY angegeben ist, dann gilt [IN,OUT], (d.h. das Argument wird vollständig in beiden Richtungen übertragen) als Default.

ACHTUNG: Wenn RPC-Routinen mit **Konstanten** als aktuellem Argument aufgerufen werden, dann sollte das formale Argument unbedingt mit [IN] qualifiziert sein, da sonst beim Return die Konstante (zwar mit ihrem eigenen Wert) überschrieben wird und dies auf manchen Rechnern zu einem Abbruch führt.

Bei asynchronen RPCs sind nur Input-Argumente möglich und aus Sicherheitsgründen müssen sie alle explizit als IN qualifiziert sein.

5.3.7.2 MAXINDEX und MAXCHAR eines Arrays

MAXINDEX = *expression*
MAXCHAR = *expression*

MAXINDEX muß man angeben, wenn die letzte Dimension eines Arrays ein $*$ ist, d.h. variabel ist. Mit MAXINDEX legt man den maximalen Index fest (nicht die Länge des Arrays). MAXINDEX legt fest, wieviel Speicherplatz im Serverstub real für das aktuelle Argument bereitgestellt werden soll. Mit MAXINDEX kann man auch bei Arrays fester Länge den Speicherplatz anders als in der Deklaration festlegen. MAXINDEX bestimmt auch den maximal möglichen Bereich, der übertragen werden kann.

MAXCHAR überschreibt die Länge eines String-Arguments. Bei String-Arrays bezieht sich MAXCHAR auf die Länge eines Strings, d.h. MAXCHAR legt die Länge fest, die man in der Serverroutine mit LEN abfragen kann.

Alle Arrays werden serverseitig im Word- und Character-Buffer angelegt. Die Summe des Speicherbedarfs[5] aller Arrays in der Argumenteliste darf WORD_BUFFER_LENGTH und CHARACTER_BUFFER_LENGTH nicht überschreiten, s. S. 57.

[5] man beachte, daß bei einem Callback zu einem Callback 2 Routinen auf der Serverseite gleichzeitig aktiv sind, d.h. beide Argumentelisten gleichzeitig Platz finden müssen.

Beispiele:

```
C*[RPI_BEG]
C     Arrays and MAXINDEX
C     -------------------

      SUBROUTINE sub2 (n,a,b,c)
      INTEGER n    [IN]
      REAL    a(n) [OUT, MAXINDEX=n/2]
      REAL    b(n) [IN]
      REAL    c(n) [IN]

C     MAXINDEX overwrites the upper bound of the (last) dimension of
C     an array.
C     MAXINDEX must be given, if the (last) dimension is an asterisk.
C     MAXINDEX is used for memory allocation for the actual argument
C     in the server stub. MAXINDEX is also used as default and maximum
C     for the [IN] and [OUT] range (see next example).

C*[RPI_NON]
      DO 10  i=1,n/2
         a(i) = b(i)*c(i) + b(i+n/2)*c(i+n/2)
10    CONTINUE
      RETURN
      END
C*[RPI_END]


C*[RPI_BEG]
C     Arrays with specified IN/OUT Range
C     ----------------------------------

      SUBROUTINE sub3 (n,a)
      INTEGER n(2) [IN]
      REAL    a(*) [MAXINDEX=n(1)*n(2), IN=1:n(1), OUT=n(2)+1:n(2)+n(1)]
C*[RPI_NON]
      DO 10 i=1,n(1)
         a(n(2)+i) = a(i) * a(i)
10    CONTINUE
      RETURN
      END
C*[RPI_END]


C*[RPI_BEG]
C     Multidimensional Arrays
C     -----------------------

      SUBROUTINE sub4 (a,n,nbeg,nend,m,mbeg,mend)
      INTEGER n[IN], nbeg[IN], nend[IN]
      INTEGER m[IN], mbeg[IN], mend[IN]
      REAL    a(n,*) [MAXINDEX = MAX(m,mend)          ,
     +                IN       = (nbeg:nend, mbeg:mend),
     +                OUT      = (nbeg:nend, m:m)      ]
```

```
C*[RPI_NON]
      DO 20 i = nbeg, nend
         sum = 0
         DO 10 j = mbeg, mend
            sum = sum + a(i,j) * a(i,j)
10       CONTINUE
         a(i,m) = sum
20    CONTINUE
      RETURN
      END
C*[RPI_END]


C*[RPI_BEG]
C     Character String Arguments
C     --------------------------

      SUBROUTINE sub5 (s1, s2, s3, s4, s5)

      CHARACTER*(*)  s1 [IN]
      CHARACTER*(*)  s2 [OUT=3]
      CHARACTER*(*)  s3 [IN, OUT, MAXCHAR=90]
      CHARACTER*(*)  s4 [DUMMY]
      CHARACTER*(*)  s5 [DUMMY,   MAXCHAR=500]

C*[RPI_NON]

C     If sub5 is called with the following actual argument list

C          CALL sub5 (a1(1:2), a2(11:15), a3(1:7), a4(41:100), ' ')

C     then s1 is declared as char.*2   and has got the value of a1(1:2)
C          s2 is declared as char.*5   and has got no value
C          s3 is declared as char.*90  and has got the value of a1(1:7)
C          s4 is declared as char.*60  and has got no value
C          s5 is declared as char.*500 and has got no value

C     After return  a1 will be unchanged
C                   a2(11:13) gets the content of s2(1:3)
C                   a3(1:7)   gets the content of s3(1:7)
C                   a4 will be unchanged
C                   a5 will be unchanged

      RETURN
      END
C*[RPI_END]

C Remarks:
C     The length of a string argument is always an invisible input
C     argument. The length of the allocated memory in the server is the
C     length of the actual argument and can be overwritten by the
C     MAXCHAR parameter.
C     The default of the [IN] and [OUT] range is the minimum of the
C     length of the actual argument and the MAXCHAR parameter.
```

```
C*[RPI_BEG]
C     Arrays of Strings
C     -----------------

      SUBROUTINE sub6 (sarr,n)

      CHARACTER*(*)  sarr(*) [MAXINDEX=n, IN = (    ,1:n/2)  ,
     +                                    OUT= (1:3,n/2+1:n)]

C     In the IN and OUT ranges, the first dimension is the substring.
C     The omitted dimension in IN=(,...) is the same as 1:*, where *
C     is the string length, which can be overwritten by MAXCHAR.

C*[RPI_NON]
      DO 10  i=1,n/2
         sarr(n/2+i)(1:3) = func6 (sarr(n/2+i))
10    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.3.7.3 TRANSFER – reduzierte Übertragungsgenauigkeit

TRANSFER	$*$	**1**
	$*$	**2**
	$*$	**4**
	$*$	**8**
	$*$	**16**
	$*$	**32**

Mit **TRANSFER** gibt man an, mit wievielen Bytes ein numerischer Wert übertragen werden soll. Wenn man TRANSFER nicht angibt, dann gilt folgende Tabelle:

Type	Def-Transferlength	erlaubt
INTEGER	4	1,2,4,8
LOGICAL	4	1,2,4,8
REAL	4	4,8,16
DOUBLE PRECISION	8	4,8,16
COMPLEX	8	8,16,32
BYTE	1	1,2,4,8
INTEGER * 1/2/4/8	1/2/4/8	jeweils 1,2,4,8
LOGICAL * 1/2/4/8	1/2/4/8	jeweils 1,2,4,8
REAL * 4/8/16	4/8/16	jeweils 4,8,16
COMPLEX * 8/16/32	8/16/32	jeweils 8,16,32

Beispiel:

```
C*[RPI_BEG]
C     Reduced Transfer Precision and Check Modus
```

```
C     ------------------------------------------

      SUBROUTINE sub7 (i,j,x,y)

      INTEGER           i [IN,  TRANSFER *2]
      INTEGER           j [IN,  TRANSFER *1]
      DOUBLEPRECISION   x [OUT, TRANSFER *4]
      DOUBLE PRECISION  y [OUT, TRANSFER *16]
C     To obtain full precision between two hosts with 64 bit
C     numerical units, one must qualify reals with TRANSFER*8
C     and doubles with TRANSFER*16, e.g. see argument y.
C*[RPI_NON]
      x = float(i)**j
      y = float(i)**j
      RETURN
      END
C*[RPI_END]
```

5.3.7.4 CLIENT – unterschiedliche Datentyplängen in Client und Server

CLIENT	$*$ **1**
	$*$ **2**
	$*$ **4**
	$*$ **8**
	$*$ **16**
	$*$ **32**

Mit **CLIENT** gibt man an, mit wievielen Bytes das Argument im Clientstub deklariert sein soll, d.h. mit welchen Einheiten es vom Client gerufen wird.

```
C Client:
C -------
      REAL*8  x
      REAL*4  y
      CALL sub8 (x,y)

C Server:
C -------
C*[RPI_BEG]
C     Differring Declarations in the Client
C     ------------------------------------

      SUBROUTINE sub8 (x,y)
      REAL*4  x [CLIENT *8]  ! in server routine and server stub real*4
                             ! in the client stub real*8
      REAL*8  y [CLIENT *4]  ! in server routine and server stub real*8
                             ! in the client stub real*4
C     For choosing the number of bytes which will be transferred the
C     minimum of both declarations will be relevant.
C*[RPI_NON]
```

```
      d = x
      x = y
      y = d
      RETURN
      END
C*[RPI_END]
```

Für **INTEGER,LOGICAL,BYTE** sind die Werte 1,2,4,8 erlaubt, für **real** die Werte 4,8,16, und für **DOUBLE PRECISION,COMPLEX** die Werte 8,16,32.

5.3.7.5 FLUSH – gepufferter oder ungepufferter Call

FLUSH

Mit **FLUSH** kann ein Parameter vom Typ LOGICAL bei ASYNCHRONEN RPCs qualifiziert werden.

Der aktuelle Wert beim Aufruf durch den Client entscheidet dann, ob der asynchrone RPC gepuffert (.FALSE.) oder sofort abgeschickt werden soll (.TRUE.).

5.3.7.6 ERROR

ERROR

Mit **ERROR** kann ein Parameter vom Typ INTEGER qualifiziert werden.

Tritt beim RPC ein Fehler auf, dann wird die Anwendung nicht abgebrochen, sondern der Fehlercode beim Client in den aktuellen Parameter geschrieben. Der Rückgabewert 0 bedeutet, daß kein Fehler festgestellt wurde.

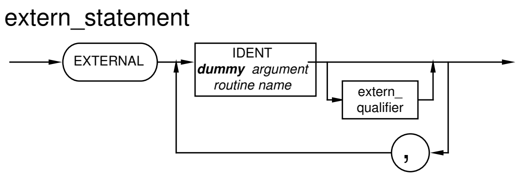
Dieser Qualifier ist zur Zeit noch nicht vollständig implementiert.

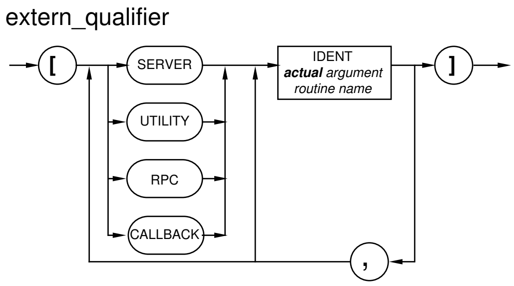
5.3.8 Externals als Argument

Da Externals auch mittels Call-by-reference übergeben werden, gibt es bei RPCs zwei Probleme:

- Befindet sich die Routine, die als aktuelles Argument übergeben wird auf der Client- oder der Server-Site oder sogar bei beiden?
- Wie heißt die Reference auf der entfernten Seite?

Falls keine entsprechende Routine auf der Client-Site existiert, wird dort eine Dummy-Routine hinzugefügt, damit auf der Client-Site die entsprechende Reference als aktuelles Argument gebildet werden kann.





Die verschiedenen Möglichkeiten kann man dem folgenden Beispiel entnehmen.

Beispiel:

```
C*[RPI_BEG]
C     Externals as dummy arguments
C     ----------------------------

      SUBROUTINE sub12 (e)

      EXTERNAL  e [SERVER   s1, s2, s3,
     +             UTILITY  u1, u2, u3, u4,
     +             RPI      r1,
     +             CALLBACK b1 ]

C*[RPI_NON]
      RETURN
      CALL e
      END
C*[RPI_END]

C Remarks:

C     Subroutine or function dummy arguments must be declared as
C     external in the subroutine or function, where it is in the
C     dummy argument list. (Normally it must be declared only in the
C     calling routine, i.e. if a routine name is used as an actual
C     argument.)
C     The whole range of possible actual values must be specified.
C     Also it must be specified, where those routines are located.

C     'mmm' should be the module in which sub12 is declared.
C     'sss' should be the block where sub12 is located as routine,
C     i.e. where mmm is declared (C*[MOD_USE]) as SERVER.
C     'ccc' should be the block from where sub12 is called,
C     i.e. where mmm is declared (C*[MOD_USE]) as CLIENT.
C     Then the following table illustrates the location of
C     the routines used as actual arguments for e
```

```
C     and on which host the actual arguments are used

C     declared as | acc                 |                  sss | called on
C     ------------+---------------------+--------------------+----------
C     SERVER      |                     |           s1 s2 s3 |   sss
C     UTILITY     | u1 u2 u3 u4         |        u1 u2 u3 u4 |   sss
C     RPI         | Stub_with_entry_r1 --------> stub    r1 |   sss
C     CALLBACK    | b1   stub <--------- stub_with_entry_b1 |   ccc

C     In the case of SERVER, the stub generator will generate in the
C     client stub of mmm (here located in ccc) the corresponding
C     dummy routines, here s1, s2 and s3.

C     CAUTION: In all blocks where mmm is used as client stub or server
C              the EXTERNAL declaration must be valid, i.e. the role of
C              possible actual arguments (s1,s2,s3 / u1,u2,u3,u4 / r1 /
C              b1) must be se same (SERVER / UTILITY / RPI / CALLBACK).
```

Wenn innerhalb eines Blocks eine Subroutine in mehreren Modulen, für die ein Clientstub generiert wird, als SERVER-Routine anzugeben wäre, dann darf sie nur einmal als SERVER-Routine deklariert werden und sollte in den anderen Modulen als Utility-Routine angegeben werden, da sonst in diesem Block die Dummy-Routine mehrfach erstellt wird.

5.3.9 Alternative Rücksprünge

Alternative Rücksprünge sind möglich.

Beispiel:

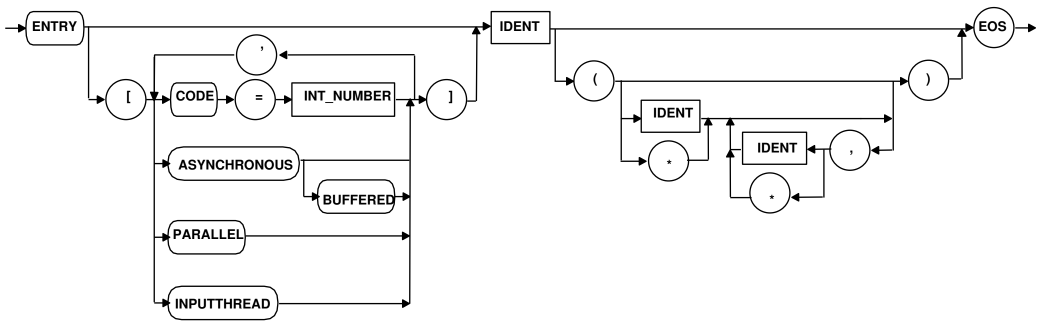
```
C*[RPI_BEG]
C     Alternative Return Jumps
C     ------------------------

      SUBROUTINE sub13 (n,*,*,*)
      INTEGER  n [IN]

C*[RPI_NON]
      i = max (1, min( -2+n/3, 3)
      RETURN i
      END
C*[RPI_END]
```

5.3.10 Die Fortran ENTRY Anweisung

ENTRY [CODE=*operation_code*, **ASYNCHRONOUS]** *identifier (...)*
oder **ASYNCHRONOUS BUFFERED**
oder **PARALLEL**
oder **INPUTTHREAD**

Der Qualifier enthält die gleichen Elemente wie das C*[RPI_BEG] Statement, s. S. 69f.

entry_stat



Beispiel:

```
C*[RPI_BEG] CODE = 14
C     Entry Statements
C     ----------------

      SUBROUTINE sub14
      INTEGER  n [OUT]

C*[RPI_NON]
      ...
      RETURN

C*[RPI_CON]
      ENTRY [CODE = 15] sub15 (n)
C*[RPI_NON]
      n = 171717
      RETURN
      END
C*[RPI_END]
```

5.4 RPI ohne Fortran-Quellen

Im wesentlichen entfallen hierbei die nonparseable Blocks, die im wesentlichen die ausführbaren Statements und für die Stubgenerierung irrelevanten Deklarationen enthalten, s. a. Beispiel C, Kap. 5.7.3 auf S.95.

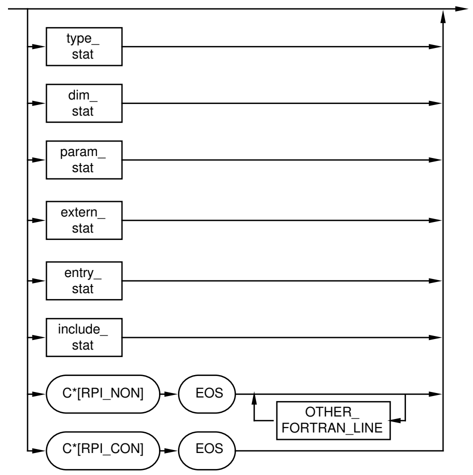
Der Stub-Generator ist mit der Option **-n** aufzurufen, s.a. S. 159.

5.5 Getrenntes Generieren von Blöcken und Modulen

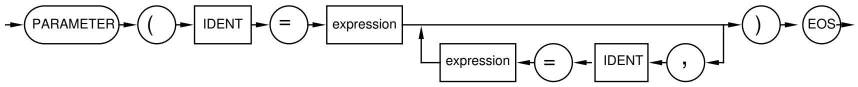
S. Beispiel D, Kap. 5.7.4 auf S. 96.

5.6 Allgemeine Syntax der Deklarationsstatements

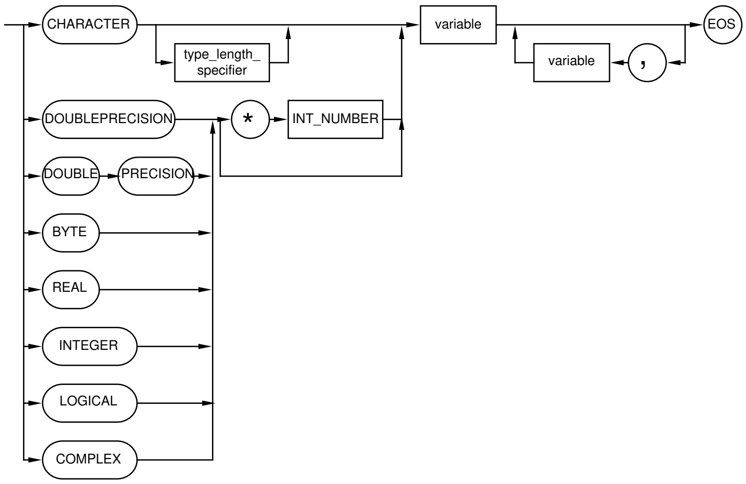
decl_stat



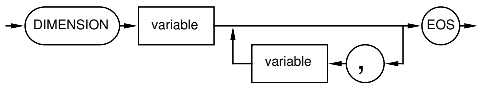
param_stat



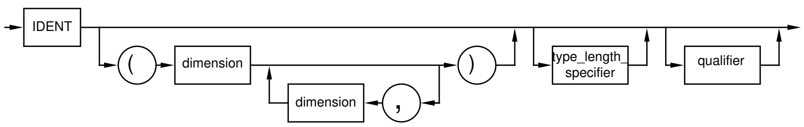
type_stat



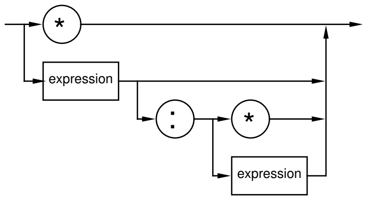
dim_stat



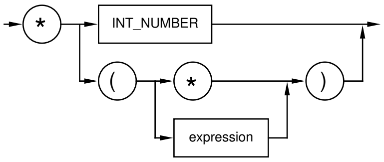
variable



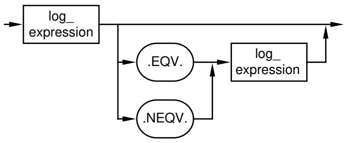
dimension



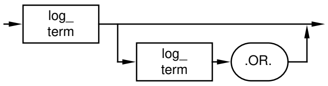
type_length_specifier



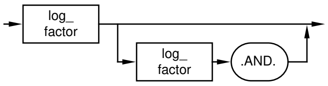
expression



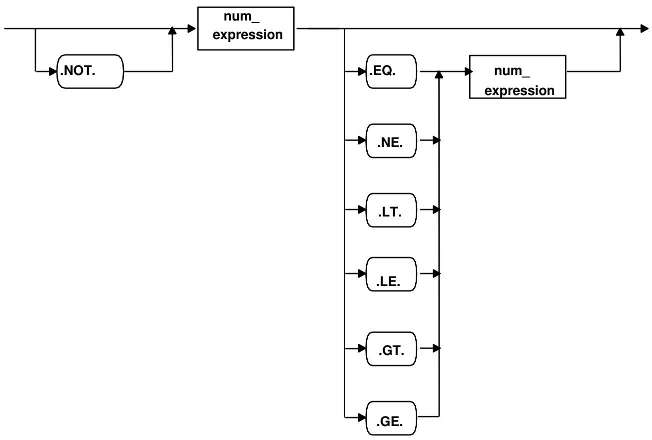
log_expression



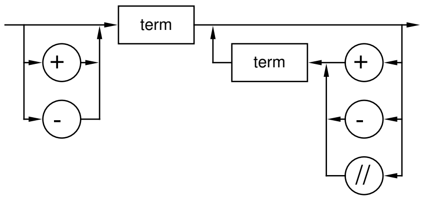
log_term



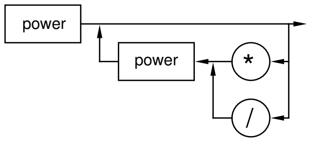
log_factor



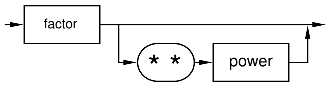
num_expression



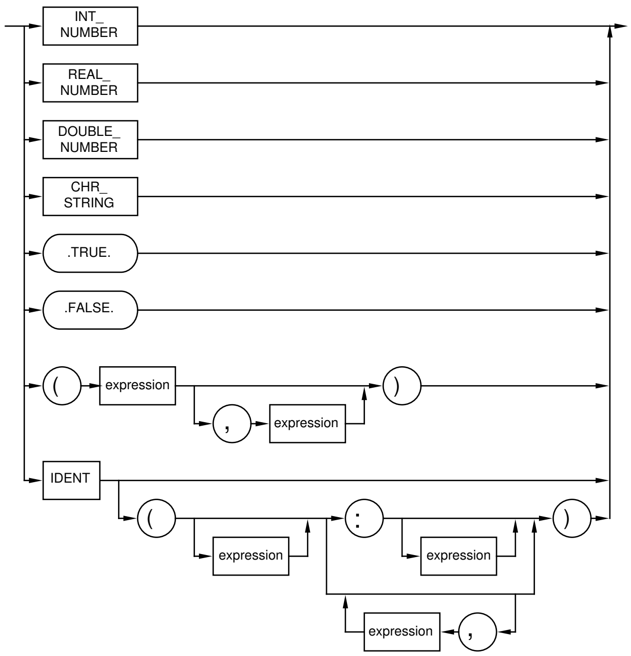
term



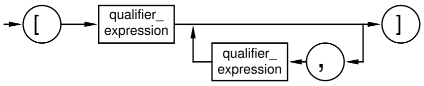
power



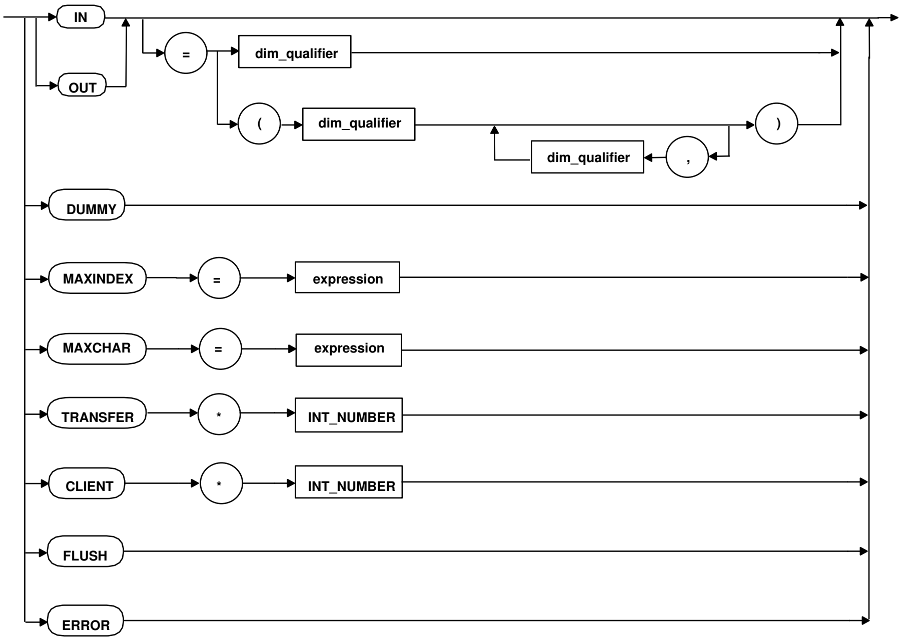
factor



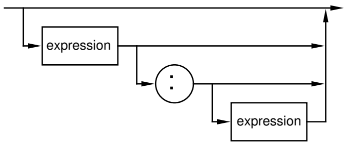
qualifier



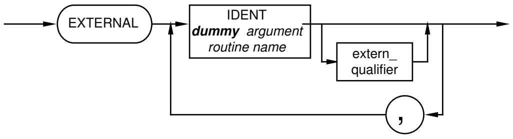
qualifier_expression



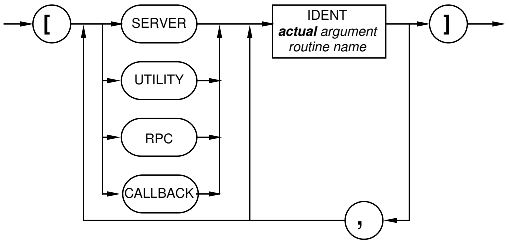
dim_qualifier



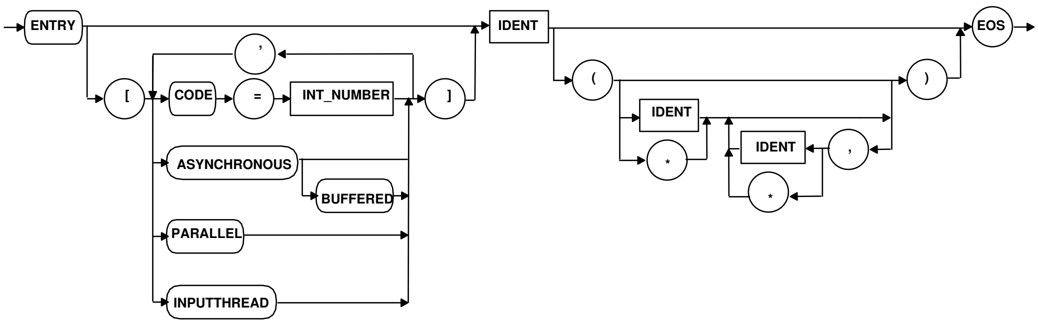
extern_statement



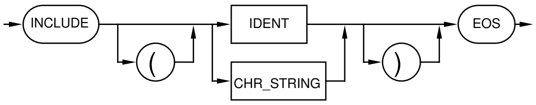
extern_qualifier



entry_stat



include_stat



5.7 Zusammenfassung der Beispiele zum RPI

Der folgende RPI Definition File enthält alle relevanten RPC-Szenarien und wie diese vom Anwendungsprogrammierer kodiert werden müssen.

```
C     +----------------------------------------------------------+
C     |                  E X A M P L E S                         |
C     |                      F O R                               |
C     | R E M O T E   P R O C E D U R E   I N T E R F A C E |
C     |              D E F I N I T I O N S                       |
C     +----------------------------------------------------------+
```

5.7.1 Übergeordnete Struktur der RPI Definition

```
C  A) The Global Structure of the Remote Interface Definition (RPI)
C     -------------------------------------------------------------

C     A RPI definition file consist of:
C
C       declaration part        with   options
C                                      block options
C                                      module parameters
C
C       block definition part   with   several block definitions
C
C       module definition part  with   several module definitions
C
C     These three parts must be written in this order.
C     Inside the parts any order is possible.
```

5.7.2 Im Quellcode integrierte RPI Definition

```
C  B) Example of a Remote Procedure Interface (RPI) Definition, embedded
C     into Fortran Source Code with a Client and a Server block
C     ------------------------------------------------------------------

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C     The following options must not be declared if one wants to use the
C     defaults. (The values used in this example are the defaults.)

C     The length values are given in bytes, the time values in seconds.


C*[OPTION] WORD_BUFFER_LENGTH      = 100000
C*[OPTION] CHARACTER_BUFFER_LENGTH =  10000
C*[OPTION] TRACE_DEPTH             =     20
C*[OPTION] OPERATION_BITS          =     10     ! Range 6..14
```

```
C*[OPTION] OBJECTS_MAXIMUM        =    512
C*[OPTION] WAIT_TIME_NEXT_CALL    =   1000
C*[OPTION] WAIT_TIME_SEND         =    300
C*[OPTION] WAIT_TIME_CONNECT      =    100
C*[OPTION] WAIT_TIME_ACCEPT       =   2000
C*[OPTION] WAIT_TIME_RELEASE      =    200
C*[OPTION] WAIT_TIME_LOCAL        =    100


C     The following block options are used by the authorization
C     mechanism. The default is zero. Please use an integer
C     random value between 0 and 999999999=10**9-1

C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321


C     With the following line one can define some options for the later
C     used and declared modules.

C*[MOD_PAR] sm, DEPTH=0, BACKDEPTH=0, CODE=1

C     DEPTH: If a module is used as a server in a block, then the stub
C            generator must generate not only the server stub for each
C            routine, it also must generate a module dispatcher routine
C            which interprets the header of each incoming remote
C            procedure call and calls the apropriate server stub routine.
C            If the called routine calls a second server or calls back a
C            routine in the client block and this one calls again the
C            server block then the block and module dispatcher routines
C            are called recursively, although no application routine is
C            called recursively.
C            DEPTH=0 means that the module dispatcher routine may be
C            called recursively.
C            DEPTH=d (>0) means that the stub generator must generate
C            d equivalent module dispatcher routines in this module
C            (if it is a SERVER module) to prevent the recursion.
C            This module DEPTH value will be used in the block dispatcher
C            routine, where the module dispatcher routine(s) must be
C            called.
C            Default: DEPTH=0.

C     BACKDEPTH:
C            This backdepth value is used in each subroutine client stub
C            of this module to generate the calls to the apropriate
C            number of block dispatcher routines. These calls are used
C            to handle callbacks (see also DEPTH).

C     CODE:  Each module which is used as CLIENT or SERVER in a block
C            must have a module code. It can be assigned in a MOD_PAR
C            statement or it can be automatically assigned by the stub
C            generator. This code must be the same in each block.
C            If one uses one file to generate all blocks then one can
C            use the automatic assignment. If one wants to generate
```

```
C            the blocks separately then the explicit assignment in the
C            MOD_PAR statement is recommended. The code must be in the
C            range 1 .. 2**(16-OPERATION_BITS)-1  or with the default of
C            10 operation bits, the module code range is 1..63.
C            Default: The automatic assignment is done when it is first
C            used in a MOD_USE statement as CLIENT or SERVER, see below.


C     ---------->>>>>>>> block definition part <<<<<<<<----------------

C     The following block definitions are necessary.
C     They are using the OPTION, BLK_OPT and MOD_PAR
C     (only DEPTH and CODE) information.

C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     cm, LOCAL
C*[MOD_USE]     cp, LOCAL
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sp, LOCAL
C*[MOD_USE]     sm, SERVER
C*[BLK_END]

C*[BLK_BEG] lb, NAME='local program', DEPTH=0, LOCAL_SITE
C*[MOD_USE]     cm, LOCAL    ! only LOCAL is possible at LOCAL_SITE
C*[MOD_USE]     sm, LOCAL
C*[BLK_END]

C     CLIENT means that for each RPI routine in the module (here sm)
C            a client stub will be generated and embedded into this
C            block (here cb)
C     SERVER means first that the module itself is included into the
C            block and
C            second that for each RPI routine in the module (here sm)
C            a server stub will be generated and
C            third the generation of the module dispatcher routine and
C            fourth that this dispatcher routine will be called by the
C            block dispatcher routine.
C     LOCAL  means that only the module is embedded into the block.

C     CLIENT_SITE means that the block contains the application main
C            program.
C     SERVER_SITE means that the block is a server block, built by
C            a special main program which calls the server main loop
C            which is generated by the stub generator.
C     LOCAL_SITE means that remote procedure calls are not used.
```

```
C     ---------->>>>>>>> module definition part <<<<<<<<---------------

C     The following module definitions are necessary.
C     They are using the OPTION and MOD_PAR (only NAME and DEPTH)
C     information.

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

      PROGRAM ...
      ...
C*RPC CALL rpcini            ! Those lines will be duplicated and
      ...                    ! the second activated in the generated
C*RPC CALL rpcfin            ! source of CLIENT_SITE and SERVER_SITE
      ...                    ! blocks.
      END
                             ! Source for the client module
      SUBROUTINE ...
      ...
      END

      FUNCTION ...
      ...
      END

C*[MOD_END]


C*[MOD_BEG] cp, NAME='client rpc special routines '

      SUBROUTINE rpcini      ! Examples will be provided
      ...
      END

      SUBROUTINE rpcfin
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sp, NAME='server rpc special program  '

      PROGRAM rpcsrv         ! Special main program for the server block
      ...
      CALL RPCLOP            ! The server main loop of this rpc service
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sm, NAME='server module - server subroutine'

C*[RPI_BEG] CODE = 8        ! Two possibilities for all rpi routines
```

```
                          ! of a module: The routine codes are
                          ! explicitely defined always or never.
      SUBROUTINE sub1 (a,b,c)! RPI-Definition for this subroutine
      REAL a[IN], b[IN]      ! for generating client and server stub
      REAL c                 ! and source for this server module
C*RPC+      [OUT]            ! (only declarations, especially for
      ...                    ! the dummy arguments)
C*[RPI_NON]
      a = b * b              ! further source for this server module
      RETURN                 ! (further declarations and
      END                    ! executable statements)
C*[RPI_END]                  ! RPI_END is the signal for stub generation

      SUBROUTINE ...
      ...                    ! Further source for this module, but
      ...                    ! without stub generation, i.e.
      END                    ! not callable by remote blocks
      .
      .
C*[MOD_END]


C If this is on the input file inp.rpi then the stub generator will
C generate the following files:
C
C                                   inp.rpi
C                                      |
C               +-------------------------+------------------------+
C For the       |                         |                        |
C blocks ..... cb                        sb                       lb
C               |                         |                        |
C following     |\                        |\                       |
C files will    | \                       | \                      |
C be generated: | inp_cb_stb.m4           | inp_sb_stb.m4          |
C               | |                       | |                      |
C         inp_cb.f inp_cb_stb.f    inp_sb.f inp_sb_stb.f    inp_lb.f
C
C
C Stub-Presourcefile:     inp_cb_stb.m4  and  inp_sb_stb.m4
C Stub-Sourcefile:        inp_cb_stb.f   and  inp_sb_stb.f
C Application-Sourcefile: inp_cb.f       and  inp_sb.f  and inp_lb.f
C
C Only in the case of CLIENT_SITE or SERVER_SITE (files inp_cb.f
C and inp_sb.f) all statements beginning with C*RPC will be duplicated
C and in the second line C*RPC will be substituted by five blanks.
C
C Then (also in the case C*[BLK_BEG] ... LOCAL)
C all qualifiers [...] between C*[RPI_BEG] / C*[RPI_CON] and
C C*[RPI_NON] / C*[RPI_END] will be substituted by blanks,
C here in the source files inp_cb.f, inp_sb.f and inp_lb.f.
C
```

5.7.3 RPI Definition, nicht im Fortran Quellcode

```
C  C) Example of a Remote Procedure Interface Definition, not embedded
C     into Fortran Source Code
C     -----------------------------------------------------------------

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321

C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sm, SERVER
C*[BLK_END]

C     ---------->>>>>>>> module definition part <<<<<<<<----------------

C*[MOD_BEG] sm, NAME='server module - server subroutine'

C*[RPI_BEG]                  ! Here automatic operation CODE assignment
      SUBROUTINE sub1 (a,b,c)! is used.
      REAL a[IN], b[IN], c[OUT]
C     ...
C*[RPI_END]

C*[RPI_BEG]
      SUBROUTINE sub2
C     ...
C*[RPI_END]

C*[MOD_END]

C If this is on the input file inp.rpi then the stub generator will
C generate the following files (using -n flag)
C
C                                    inp.rpi
C                                       |
C             +------------------------+
C For the      |                        |
C blocks ..... cb                      sb
C             |                        |
C following     \                        \
C files will     \                        \
C be generated:   inp_cb_stb.m4            inp_sb_stb.m4
C                 |                        |
C                 inp_cb_stb.f             inp_sb_stb.f
```

5.7.4 RPI Definition in getrennten Dateien

```
C  D) Example of a Remote Procedure Interface Definition, not embedded
C     into Fortran Source Code, with separated block and module gener.
C     ----------------------------------------------------------------

C File cb.rpi:
C ============

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...

C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321

C*[MOD_PAR] sm, DEPTH=0,CODE=1

C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]



C File sb.rpi:
C ============

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...

C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321

C*[MOD_PAR] sm, DEPTH=0,CODE=1

C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sm, SERVER
C*[BLK_END]
```

```
C File sm.rpi:
C ============

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...

C*[MOD_PAR] sm, DEPTH=0

C     ---------->>>>>>>> module definition part <<<<<<<<----------------

C*[MOD_BEG] sm, NAME='server module - server subroutine'

C*[RPI_BEG]
      SUBROUTINE sub1 (a,b,c)
      REAL a[IN], b[IN], c[OUT]
C     ...
C*[RPI_END]

C*[RPI_BEG]
      SUBROUTINE sub2
C     ...
C*[RPI_END]

C*[MOD_END]



C              cb.rpi            sb.rpi                  sm.rpi
C               |                 |                       |
C               |                 |                    +--+--+
C For the       |                 |                    |     |
C blocks ..... cb                sb                  (cb)   (sb)
C               |                 |                    |     |
C following     |                 |                    |     |
C files will    |                 |                    |     |
C be generated:cb_cb_stb.m4    sb_sb_stb.m4   sm_sm_clt.m4  sm_sm_srv.m4
C               |                 |                    |     |
C              cb_cb_stb.f     sb_sb_stb.f    sm_sm_clt.f   sm_sm_srv.f
C                              /   \   \ |                  /   \
C                         ____/     \   \|___              /     \
C The filenames          /           \       \            /       \
C originate from  input_filename block_name fixed  input_file module_name
C                                ====                        ======
```

5.7.5 Beispiele für Remote Procedure Interface Definitions

```
C  E) Examples of Remote Procedure Interface Definitions
C     --------------------------------------------------
```

5.7.5.1 [IN], [OUT], [IN,OUT] und [DUMMY] Argumente

```
C*[RPI_BEG]
C     [IN], [OUT], [IN,OUT] and [DUMMY] Arguments
C     -------------------------------------------

      SUBROUTINE sub1 (a,b,c,d,e,x)
      REAL    a[OUT],  b[IN],  c[IN],  d, e[IN,OUT], x[DUMMY]

C     the default [in,out] is used for d
C     x is dummy, i.e. a workspace argument, i.e. it will not be
C     transferred, neither for input nor for output
C*[RPI_NON]
      a = b * c
      x = d
      d = e
      e = x
      RETURN
      END
C*[RPI_END]
```

5.7.5.2 Felder und MAXINDEX

```
C*[RPI_BEG]
C     Arrays and MAXINDEX
C     -------------------

      SUBROUTINE sub2 (n,a,b,c)
      INTEGER n[IN]
      REAL    a(n) [OUT, MAXINDEX=n/2]
      REAL    b(n) [IN]
      REAL    c(n) [IN]

C     MAXINDEX overwrites the upper bound of the (last) dimension of
C     an array.
C     MAXINDEX must be given, if the (last) dimension is an asterisk.
C     MAXINDEX is used for memory allocation for the actual argument
C     in the server stub. MAXINDEX is also used as default and maximum
C     for the [IN] and [OUT] range (see next example).
C*[RPI_NON]
      DO 10  i=1,n/2
10       a(i) = b(i)*c(i) + b(i+n/2)*c(i+n/2)
      RETURN
      END
C*[RPI_END]
```

5.7.5.3 Felder mit spezifiziertem Ein- und Ausgabebereich

```
C*[RPI_BEG]
C     Arrays with specified IN/OUT Range
C     ----------------------------------

      SUBROUTINE sub3 (n,a)
      INTEGER n(2) [IN]
      REAL    a(*) [MAXINDEX=n(1)*n(2), IN=1:n(1), OUT=n(2)+1:n(2)+n(1)]

C*[RPI_NON]
      DO 10 i=1,n(1)
         a(n(2)+i) = a(i) * a(i)
10    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.7.5.4 Mehrdimensionale Felder

```
C*[RPI_BEG]
C     Multidimensional Arrays
C     -----------------------

      SUBROUTINE sub4 (a,n,nbeg,nend,m,mbeg,mend)
      INTEGER n[IN], nbeg[IN], nend[IN]
      INTEGER m[IN], mbeg[IN], mend[IN]
      REAL    a(n,*) [MAXINDEX = MAX(m,mend)          ,
     +                IN       = (nbeg:nend, mbeg:mend),
     +                OUT      = (nbeg:nend, m:m)     ]

C*[RPI_NON]
      DO 20 i = nbeg, nend
         sum = 0
         DO 10 j = mbeg, mend
            sum = sum + a(i,j) * a(i,j)
10       CONTINUE
         a(i,m) = sum
20    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.7.5.5 Textargumente

```
C*[RPI_BEG]
C     Character String Arguments
C     --------------------------

      SUBROUTINE sub5 (s1, s2, s3, s4, s5)

      CHARACTER*(*)  s1 [IN]
      CHARACTER*(*)  s2 [OUT=3]
      CHARACTER*(*)  s3 [IN, OUT, MAXCHAR=90]
      CHARACTER*(*)  s4 [DUMMY]
      CHARACTER*(*)  s5 [DUMMY,   MAXCHAR=500]

C*[RPI_NON]

C     If sub5 is called with the following actual argument list

C          CALL sub5 (a1(1:2), a2(11:15), a3(1:7), a4(41:100), ' ')

C     then s1 is declared as char.*2   and has got the value of a1(1:2)
C          s2 is declared as char.*5   and has got no value
C          s3 is declared as char.*90  and has got the value of a1(1:7)
C          s4 is declared as char.*60  and has got no value
C          s5 is declared as char.*500 and has got no value

C     After return  a1 will be unchanged
C                   a2(11:13) gets the content of s2(1:3)
C                   a3(1:7)   gets the content of s3(1:7)
C                   a4 will be unchanged
C                   a5 will be unchanged

      RETURN
      END
C*[RPI_END]

C Remarks:

C     The length of a string argument is always an invisible input
C     argument. The length of the allocated memory in the server is the
C     length of the actual argument and can be overwritten by the
C     MAXCHAR parameter.
C     The default of the [IN] and [OUT] range is the minimum of the
C     length of the actual argument and the MAXCHAR parameter.
```

5.7.5.6 Textfelder

```
C*[RPI_BEG]
C     Arrays of Strings
C     -----------------

      SUBROUTINE sub6 (sarr,n)

      CHARACTER*(*)  sarr(*) [MAXINDEX=n, IN = (    ,1:n/2)  ,
     +                                    OUT= (1:3,n/2+1:n)]

C     In the IN and OUT ranges, the first dimension is the substring.
C     The omitted dimension in IN=(,...) is the same as 1:*, where *
C     is the string length, which can be overwritten by MAXCHAR.
C*[RPI_NON]
      DO 10  i=1,n/2
         sarr(n/2+i)(1:3) = func6 (sarr(n/2+i))
10    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.7.5.7 Reduzierte Übertragungsgenauigkeit

```
C*[RPI_BEG]
C     Reduced Transfer Precision and Check Modus
C     ------------------------------------------

      SUBROUTINE sub7 (i,j,x,y)

      INTEGER           i [IN,  TRANSFER *2]
      INTEGER           j [IN,  TRANSFER *1]
      DOUBLEPRECISION   x [OUT, TRANSFER *4]
      DOUBLE PRECISION  y [OUT, TRANSFER *16]

C     TRANSFER *length   specifies the number of bytes that will be
C                        transferred for each value.
C                        Integer   : *1,*2,*4,*8
C                        Real      : *4,*8,*16
C                        Double    : *4,*8,*16
C                        Complex   : *8,*16,*32
C                        Logical   : *1,*2,*4,*8
C     To obtain full precision between two hosts with 64 bit
C     numerical units, one must qualify reals with TRANSFER*8
C     and doubles with TRANSFER*16, e.g. see argument y.
C*[RPI_NON]
      x = float(i)**j
      y = float(i)**j
      RETURN
      END
C*[RPI_END]
```

5.7.5.8 Unterschiedliche Datentyplängen in Client und Server

```
C Client:
C -------
      REAL*8  x
      REAL*4  y
      CALL sub8 (x,y)

C Server:
C -------
C*[RPI_BEG]
C     Differring Declarations in the Client
C     -------------------------------------

      SUBROUTINE sub8 (x,y)

      REAL*4  x [CLIENT *8]  ! in server routine and server stub real*4
                             ! in the client stub real*8
      REAL*8  y [CLIENT *4]  ! in server routine and server stub real*8
                             ! in the client stub real*4

C     For choosing the number of bytes which will be transferred the
C     minimum of both declarations will be relevant.

C*[RPI_NON]
      d = x
      x = y
      y = d
      RETURN
      END
C*[RPI_END]
```

5.7.5.9 Zusammenstellung der numerischen Datentypen

```
C*[RPI_BEG]
C     Summary of numeric types
C     ------------------------

      SUBROUTINE sub11 (b, l,l1,l2,l4,l8, i,i1,i2,i4,i8,
     +                 r,r4,r8,r16, d, c,c8,c16,c32)

C            internal bytes on    SUN VAX CRAY2  | transfer
C            with compiler        f77 for cft77  | default
      BYTE            b     !      1   1    -    |   1
      LOGICAL         l     !      4   4    8    |   4
      LOGICAL*1       l1    !      1   1    8 w  |   1
      LOGICAL*2       l2    !      2   2    8 w  |   2
```

```
      LOGICAL*4        l4    !     4   4    8 w  |   4
      LOGICAL*8        l8    !     -   -    8 w  |   8
      INTEGER          i     !     4   4    8    |   4
      INTEGER*1        i1    !     -   -    -    |   1
      INTEGER*2        i2    !     2   2    8 w  |   2
      INTEGER*4        i4    !     4   4    8 w  |   4
      INTEGER*8        i8    !     -   -    8 w  |   8
      REAL             r     !     4   4    8    |   4
      REAL   *4        r4    !     4   4    8 w  |   4
      REAL   *8        r8    !     8   8    8 w  |   8
      REAL   *16       r16   !     -  16   16 w  |  16
      DOUBLE PRECISION d     !     8   8   16    |   8
      COMPLEX          c     !   4+4 4+4  8+8 w  | 4+4
      COMPLEX*8        c8    !   4+4 4+4  8+8 w  | 4+4
      COMPLEX*16       c16   !   8+8 8+8  8+8    | 8+8
      COMPLEX*32       c32   !     -   -    -    |  --

C                                w = the compiler issues a warning
C*[RPI_NON]
      RETURN
      END
C*[RPI_END]

C Remarks:

C     One can write DOUBLE PRECISION also without a blank:
C     DOUBLEPRECISION
C     The stub generator accepts non standard fortran data types
C     as listed above.
C     The number of transferred bytes depends only on the declaration
C     which can be changed by [TRANSFER *length].
C     It doesn't depend on the involved hosts (ieee, dec, vax, convex
C     or cray) nor on the chosen transfer coding (ieee, dec, vax or
C     cray).
```

5.7.5.10 Unterprogramme und Funktionen als Argumente (EXTERNALs)

```
C*[RPI_BEG]
C     Externals as dummy arguments
C     ----------------------------

      SUBROUTINE sub12 (e)

      EXTERNAL  e [SERVER   s1, s2, s3,
     +             UTILITY  u1, u2, u3, u4,
     +             RPI      r1,
     +             CALLBACK b1 ]

C*[RPI_NON]
      RETURN
      CALL e
      END
C*[RPI_END]
```

```
C Remarks:

C     Subroutine or function dummy arguments must be declared as
C     external in the subroutine or function, where it is in the
C     dummy argument list. (Normally it must be declared only in the
C     calling routine, i.e. if a routine name is used as an actual
C     argument.)
C     The whole range of possible actual values must be specified.
C     Also it must be specified, where those routines are located.

C     'mmm' should be the module in which sub12 is declared.
C     'sss' should be the block where sub12 is located as routine,
C     i.e. where mmm is declared (C*[MOD_USE]) as SERVER.
C     'ccc' should be the block from where sub12 is called,
C     i.e. where mmm is declared (C*[MOD_USE]) as CLIENT.
C     Then the following table illustrates the location of
C     the routines used as actual arguments for e
C     and on which host the actual arguments are used

C     declared as | ccc                |                sss | called on
C     ------------+--------------------+--------------------+----------
C     SERVER      |                    |           s1 s2 s3 |   sss
C     UTILITY     | u1 u2 u3 u4        |        u1 u2 u3 u4 |   sss
C     RPI         | Stub_with_entry_r1 --------> stub    r1 |   sss
C     CALLBACK    | b1   stub <--------- stub_with_entry_b1 |   ccc

C     In the case of SERVER, the stub generator will generate in the
C     client stub of mmm (here located in ccc) the corresponding
C     dummy routines, here s1, s2 and s3.

C     CAUTION: In all blocks where mmm is used as client stub or server
C              the EXTERNAL declaration must be valid, i.e. the role of
C              possible actual arguments (s1,s2,s3 / u1,u2,u3,u4 / r1 /
C              b1) must be se same (SERVER / UTILITY / RPI / CALLBACK).
```

5.7.5.11 Alternative Rücksprünge

```
C*[RPI_BEG]
C     Alternative Return Jumps
C     ------------------------

      SUBROUTINE sub13 (n,*,*,*)
      INTEGER  n [IN]

C*[RPI_NON]
      i = max (1, min( -2+n/3, 3)
      RETURN i
      END
C*[RPI_END]
```

5.7.5.12 Die Fortran ENTRY Anweisung

```
C*[RPI_BEG] CODE = 14
C     Entry Statements
C     ----------------

      SUBROUTINE sub14
      INTEGER  n [OUT]

C*[RPI_NON]
      ...
      RETURN

C*[RPI_CON]
      ENTRY [CODE = 15] sub15 (n)
C*[RPI_NON]
      n = 171717
      RETURN
      END
C*[RPI_END]
```

5.7.5.13 Asynchrone RPCs

```
C*[RPI_BEG] CODE = 16, ASYNCHRONOUS BUFFERED
C
C     Asynchronous Routines and Entries
C     ---------------------------------

      SUBROUTINE sub16
C*[RPI_NON]
      ...
      RETURN

C*[RPI_CON]
      ENTRY [CODE = 17, ASYNCHRONOUS] sub17
C*[RPI_NON]
      ...
      RETURN
      END
C*[RPI_END]
```

5.7.6 RPI Definition mit mehreren Servern

```
C  F) Example of a Remote Procedure Interface (RPI) Definition
C     with Several Servers, embedded into one source file
C     --------------------------------------------------------
C
C     Block      b1                b2                b3                b4
C
C     Modules    P                 S                 S                 S
C                R
C                +->a_clt  a_srv->A
C                B<-b_srv  b_clt<-+
C
C                +->c_clt  c_srv->C          c_srv->C
C                                  D                 D
C                                  E                 +->e_clt  e_srv->E


C     The same picture, but with the possible connections at
C     execution time:
C
C     Block      b1                b2                b3                b4
C
C     Modules    P                 S                 S                 S
C                R
C                +---------------> A
C                B <---------------+
C
C                +---------------> C
C                +---------------------------------> C
C                                  D                 D
C                                  E                 +---------------> E


C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...
C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321



C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] b1, NAME='client block b1', CLIENT_SITE
C*[MOD_USE]     p, LOCAL
C*[MOD_USE]     r, LOCAL
C*[MOD_USE]     a, CLIENT
C*[MOD_USE]     b, SERVER
C*[MOD_USE]     c, CLIENT
C*[BLK_END]
```

```
C*[BLK_BEG] b2, NAME='server block b2', SERVER_SITE
C*[MOD_USE]     s, LOCAL
C*[MOD_USE]     a, SERVER
C*[MOD_USE]     b, CLIENT
C*[MOD_USE]     c, SERVER
C*[MOD_USE]     d, LOCAL
C*[MOD_USE]     e, LOCAL
C*[BLK_END]

C*[BLK_BEG] b3, NAME='server block b3', SERVER_SITE
C*[MOD_USE]     s, LOCAL
C*[MOD_USE]     c, SERVER
C*[MOD_USE]     d, LOCAL
C*[MOD_USE]     e, CLIENT
C*[BLK_END]

C*[BLK_BEG] b4, NAME='server block b4', SERVER_SITE
C*[MOD_USE]     s, LOCAL
C*[MOD_USE]     e, SERVER
C*[BLK_END]

C*[BLK_BEG] loc, NAME='local program', LOCAL_SITE
C*[MOD_USE]     p, LOCAL
C*[MOD_USE]     a, LOCAL
C*[MOD_USE]     b, LOCAL
C*[MOD_USE]     c, LOCAL
C*[MOD_USE]     d, LOCAL
C*[MOD_USE]     e, LOCAL
C*[BLK_END]


C     ---------->>>>>>>> module definition part <<<<<<<<---------------

C*[MOD_BEG] p, NAME='main program module'
      PROGRAM ...
      ...
C*RPC CALL rpcini
      ...
C*RPC CALL rpcfin
      ...
      END

      SUBROUTINE ...
      ...
      END
      FUNCTION ...
      ...
      END
C*[MOD_END]

C*[MOD_BEG] r, NAME='special module - rpcini/rpcfin'
      SUBROUTINE rpcini
      ...
      END
```

```
      SUBROUTINE rpcfin
      ...
      END
C*[MOD_END]

C*[MOD_BEG] s, NAME='special module - server main'
      PROGRAM rpcsrv
      ...
      CALL RPBD1
      ...
      END
C*[MOD_END]


C*[MOD_BEG] a, NAME='module A'
C*[RPI_BEG] CODE = 1
      SUBROUTINE aaa1 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
      b = a * a
      WRITE (*,*) 'aaa1 - b=', b
      RETURN
      END
C*[RPI_END]
C
C*[RPI_BEG] CODE = 2
      SUBROUTINE aaa2 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
      b = a + a
      CALL bbb1 (b)
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] b, NAME='module B'
C*[RPI_BEG] CODE = 1
      SUBROUTINE bbb1 (b)
      REAL b[IN]
C*[RPI_NON]
      WRITE (*,*) 'bbb1 - b=', b
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] c, NAME='module C'
C*[RPI_BEG]
      SUBROUTINE ccc1 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
```

```
      b = a * a * a
      CALL ddd1 (b)
      RETURN
      END
C*[RPI_END]
C
C*[RPI_BEG]
      SUBROUTINE ccc2 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
      b = a + a + a
      CALL eee1 (b)
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] d, NAME='module D'
C*[RPI_BEG]
      SUBROUTINE ddd1 (b)
      REAL b[IN]
C*[RPI_NON]
      WRITE (*,*) 'ddd1 - b=', b
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] e, NAME='module E'
C*[RPI_BEG]
      SUBROUTINE eee1 (b)
      REAL b[IN]
C*[RPI_NON]
      WRITE (*,*) 'eee1 - b=', b
      RETURN
      END
C*[RPI_END]
C*[MOD_END]
```

6 Library Routinen

6.1 Übersicht – Basis Routines

```
block dispatcher                    - RPBD1  (IRET)
block dispatcher for one async.seq. - RPBD1A (IRET)
block dispatcher for one procedure  - RPBD1P (IRET)

data pipe, character string receive - RPDC0R (FLUSH, N, STR, IRET)
data pipe, character string send    - RPDC0S (FLUSH, N, STR, IRET)
data pipe, double precision receive - RPDD0R (FLUSH, N, ARR, IRET)
data pipe, double precision send    - RPDD0S (FLUSH, N, ARR, IRET)
data pipe, integer          receive - RPDI0R (FLUSH, N, ARR, IRET)
data pipe, integer          send    - RPDI0S (FLUSH, N, ARR, IRET)
data pipe, integer*1        receive - RPDI1R (FLUSH, N, ARR, IRET)
data pipe, integer*1        send    - RPDI1S (FLUSH, N, ARR, IRET)
data pipe, integer*2        receive - RPDI2R (FLUSH, N, ARR, IRET)
data pipe, integer*2        send    - RPDI2S (FLUSH, N, ARR, IRET)
data pipe, logical          receive - RPDL0R (FLUSH, N, ARR, IRET)
data pipe, logical          send    - RPDL0S (FLUSH, N, ARR, IRET)
data pipe, logical*1        receive - RPDL1R (FLUSH, N, ARR, IRET)
data pipe, logical*1        send    - RPDL1S (FLUSH, N, ARR, IRET)
data pipe, logical*2        receive - RPDL2R (FLUSH, N, ARR, IRET)
data pipe, logical*2        send    - RPDL2S (FLUSH, N, ARR, IRET)
data pipe, real             receive - RPDR0R (FLUSH, N, ARR, IRET)
data pipe, real             send    - RPDR0S (FLUSH, N, ARR, IRET)
data pipe, real*8           receive - RPDR8R (FLUSH, N, ARR, IRET)
data pipe, real*8           send    - RPDR8S (FLUSH, N, ARR, IRET)

monitor, delete user node           - RPMDUN (NODE, IRET)
monitor, set user node              - RPMSUN (NODE, VALUE, IRET)
object, wait u. current object ready- RPOCRY (TIMEMS, OBJRDY, IRET)
object, get current & client obj.id - RPOCUR (CUROBJ, CLTOBJ)
object, finish block dispatcher     - RPOFBD (IRET)
object, flush send buffer           - RPOFLS ()
object, wait u. general list ready  - RPOGRY (N, OBJID, TIMEMS, OBJRDY, OBJ1ST,
                                             nfds, readfds,writefds,exceptfds, rfds,IRET)
object, get socket descriptor       - RPOGSD (FD, IRET)
object, inspect return/message pack.- RPOISP (PACTYP, MODCOD, OPCODE, IRET)
object, wait u. object list ready   - RPOLRY (N, OBJID, TIMEMS, OBJRDY, OBJ1ST, IRET)
object, receive ret. in a thread    - RPORCT (IRET)
object, receive return package      - RPOREC (IRET)
object, wait u. o.& read-list ready - RPORRY (N, OBJID, TIMEMS, OBJRDY, OBJ1ST,
                                             NFD, RFD, RFDRDY, RFD1ST, IRET)
object, empty synchronous RPC       - RPOSYN (IRET)
object, flush & switch to other obj.- RPOUSE (OBJID, IRET)

quick start server-ser.netw.at clnt.- RPQSSC (NSERVER, SOBJID, NCONN, CONNE1,CONNE2, IRET)
quick start server-ser.netw.at serv.- RPQSSS (NSERVER, SNUM, NCONN, COBJID, SOBJID, IRET)
quick process start, finish at clnt.- RPQUCF (IRET)
quick process start, initi. at clnt.- RPQUCI (NSERVER, SOBJID, IRET)
quick process start, finish at serv.- RPQUSF (IRET)
quick process start, init   at serv.- RPQUSI (Master_Address, OBJID, IRET)
quick process start, program a.serv.- RPQUSP (Master_Address, IRET)
```

6.2 Übersicht – Advanced Routines

```
block, finish                     - RPBFIN (IRET)
block, initialization             - RPBINI (IRET)

object, accept  a transp.connection - RPOACC (OBJID, Slave_Address, Slave_Addr_Lng,
                                             Slave_Host_Lng, Slave_Port_Lng, IRET)
object, connect two servers       - RPOC2S (OBJID1, OBJID2, NHOST2,
                                             PIPID1, PIPID2, IRET)
object, close   a transp.connection - RPOCCL (IRET)
object, connect a transp.connection - RPOCON (Start_FLG, Master_Address, OBJID,
                                             Slave_Address, Slave_Addr_Lng,
                                             Slave_Host_Lng, Slave_Port_Lng, IRET)
object, close a port of a tr.conn.  - RPOPCL (IRET)
object, open  a port of a tr.conn.  - RPOPOP (Start_FLG, OBJID, Master_Address,
                                             Master_Addr_Lng, Master_Host_Lng,
                                             Master_Port_Lng, Master_Pass_Lng, IRET)
object, set local  trace flags    - RPOSLF (SOBJFL, COBJFL, SCT4FL, CCT4FL, IRET)
object, set remote trace flags    - RPOSRF (SOBJFL, COBJFL, SCT4FL, CCT4FL, IRET)

transport service, finish         - RPSFIN (IRET)
transport service, init           - RPSINI (Type_of_Tp_SerVice,
                                             Type_of_Network_Service,
                                             Type_of_Tp_SoftWare, Max_Outgoing_Conn,
                                             Max_Incoming_Conn, SERID, IRET)
transport service, list all interfa.- RPSLTI ()
transport service, use as current...- RPSUSE (SERID, IRET)

convert, receive buffer conversion  - RPVCRB (Darray, Dbytes, starti, stopi,
                                             Tbytes, Ttype,
                                             Rbuffer, RBread, RBused, Flags, IRET, *)
convert, send    buffer conversion  - RPVCSB (Darray, Dbytes, starti, stopi,
                                             Tbytes, Ttype,
                                             Sbuffer, SBused, SBsent, SBlength,
                                             Flags, IRET, *)
convert, receive buffer alignment   - RPVFRB (Rbuffer, RBread, RBused, DONE)
convert, send    buffer alignment   - RPVFSB (SBused)
convert, receive buffer initializ.  - RPVIRB (RBread, RBused)
convert, send    buffer initializ.  - RPVISB (SBused, SBsent)
```

6.3 Übersicht – Obsolete Routines

```
object, client=master normal finish - RPO1CF (IRET)
object, client=master normal start  - RPO1CS (Start_Flags, Master_Host_String,
                                             Object_Start_String, OBJID, IRET)
object, server=slave  normal finish - RPO1SF (IRET)
object, server=slave  normal start  - RPO1SS (OBJID, IRET)

object, client=master finish/restart- RPO2CF (Restart_Flag, IRET)
object, client=master initialization- RPO2CI (Start_Flags, Master_Host_String,
                                             Slave_Host_String, Object_Start_String,IRET)
object, client=master start/restart - RPO2CS (OBJID, IRET)
object, server=slave  finish/restart- RPO2SF (Restart_Flag, IRET)
object, server=slave  initialization- RPO2SI (OBJID, IRET)
object, server=slave  start/restart - RPO2SS (IRET)
```

```
object, client=slave  finish/restart- RPO3CF (Restart_Flag, IRET)
object, client=slave  start/restart - RPO3CS (OBJID, IRET)
object, server=master finish/restart- RPO3SF (Restart_Flag, IRET)
object, server=master initialization- RPO3SI (Start_Flags, Master_Host_String,
                                             Object_Start_String, OBJID, IRET)
object, server=master start/restart - RPO3SS (IRET)

object, get send & receive buf.len. - RPOGSR (LL_SND, LL_REC, LL_USE,
                                             LR_SND, LR_REC, LR_USE, IRET)
object, set send & receive buf.len. - RPOSSR (LL_SND, LL_REC, LR_SND, LR_REC, IRET)
```

6.4 Basic und Advanced Routinen – alphabetisch

6.4.1 RPBD1,RPBD1A,RPBD1P – Block Dispatcher

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPBD1  --- block dispatcher                    |
| Procedure      RPBD1A --- block dispatcher for one async.seq.|
| Procedure      RPBD1P --- block dispatcher for one procedure |
|                                                               |
| Synopsis       CALL RPBD1 (IRET)                              |
|                CALL RPBD1A(IRET)                              |
|                CALL RPBD1P(IRET)                              |
|                                                               |
| Task           Block Dispatcher:                              |
|                RPBD1 dispatches all incoming calls until a   |
|                finish package is received.                    |
|                RPBD1P dispatches only one remote Procedure   |
|                call.                                          |
|                RPBD1A dispatches an arbitrary number of      |
|                Asynchronous rpcs and at most once one        |
|                synchronous or parallel rpc. After dispatching|
|                the first rpc, RPBD1A continues with          |
|                dispatching further rpcs while the input      |
|                buffer is not empty at the beginning of a     |
|                new dispatching cycle.                         |
|                                                               |
|                At the beginning of RPBD1, RPBD1P or RPBD1A   |
|                the value of the 'current object identifier'  |
|                is stored as 'current client object ident'    |
|                that is used for receiving rpcs until the end |
|                of RPBD1, RPBD1P or RPBD1A.                    |
|                At the end of RPBD1, RPBD1P and RPBD1A        |
|                automatically RPOUSE('current client obj.id') |
|                is called, i.e. the 'current object ident'    |
|                is reset to the value it has had at the       |
|                beginning of RPBD1, RPBD1P or RPBD1A.         |
|                                                               |
| Parameter       Type            I/O Usage                     |
| IRET            integer         Out Return Code               |
|                                                               |
-----------------------------------------------------------------
```

6.4.2 RPBFIN – Block Finish

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPBFIN                                       |
|                                                             |
| Synopsis       CALL RPBFIN(IRET)                            |
|                                                             |
| Task           Block Finish                                 |
|                                                             |
| Parameter       Type            I/O Usage                   |
| IRET            integer         Out return code             |
|                                                             |
---------------------------------------------------------------
```

6.4.3 RPBINI – Block Init

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPBINI                                       |
|                                                             |
| Synopsis       CALL RPBINI(IRET)                            |
|                                                             |
| Task           Block Init                                   |
|                                                             |
| Parameter       Type            I/O Usage                   |
| IRET            integer         Out return code             |
|                                                             |
---------------------------------------------------------------
```

6.4.4 RPD... – Data Pipe Routines

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPDC0S/R  data message character     send/rec.|
| Procedure      RPDD0S/R  data message double precis.send/rec.|
| Procedure      RPDI0S/R  data message integer       send/rec.|
| Procedure      RPDI2S/R  data message integer*2     send/rec.|
| Procedure      RPDI1S/R  data message integer*1     send/rec.|
| Procedure      RPDL0S/R  data message logical       send/rec.|
| Procedure      RPDL2S/R  data message logical*2     send/rec.|
| Procedure      RPDL1S/R  data message logical*1     send/rec.|
| Procedure      RPDR0S/R  data message real          send/rec.|
| Procedure      RPDR8S/R  data message real*8        send/rec.|
|                                                             |
| Synopsis       CALL RPDC0S(FLUSH,N,STRING,IRET)             |
|                CALL RPDC0R(FLUSH,N,STRING,IRET)             |
|                                                             |
|                CALL RPDD0S(FLUSH,N,ARR,IRET)                |
|                CALL RPDD0R(FLUSH,N,ARR,IRET)                |
|                CALL RPDI0S(FLUSH,N,ARR,IRET)                |
|                ...                                          |
```

```
|                                                               |
| Task           RPD..S sends the data of the parameterlist     |
|                into a datapipe and RPD..R reads this data     |
|                from the pipe. For each call of RPD<xi>S       |
|                there must be a corresponding call to RPD<xi>  |
|                with the same type <xi> and in the same order.|
|                                                               |
|                RPD..S blocks if it should send more data than|
|                the pipe can buffer. RPD..S blocks until the   |
|                data is received on the end of the pipe with   |
|                RPD..R or until the data is stored in the      |
|                buffers of the pipe (with flush=.false. the    |
|                DFN-RPC buffer is used, with flush=.true. the  |
|                network buffer is filled).                     |
|                                                               |
|                RPD..R blocks until all data is received.      |
|                Therefore it is not necessary to wait with     |
|                RPOCRY, RPOLRY or RPOGRY until any data is     |
|                receiveable.                                   |
|                                                               |
|                      RPD.._ In/Out                            |
| Parameter      Type         _S   _R  Usage                   |
| FLUSH          logical      In   Out .true. =flush send buffer|
|                                      .false.=noflush=buffered |
|                                      used in RPD..S           |
| N              integer      In   Out number of items in array |
|                                      or characters in the     |
|                                      string to be transferred |
| ARR            data_type(N) In   Out array                   |
|       RPDI0_: INTEGER(*)                                      |
|       RPDI2_: INTEGER*2(*)  if known, otherwise INTEGER(*)    |
|       RPDI1_: INTEGER*1(*)                                    |
|               or BYTE(*)    if known, otherwise INTEGER(*)    |
|       RPDI0_: LOGICAL(*)                                      |
|       RPDI2_: LOGICAL*2(*)  if known, otherwise LOGICAL(*)    |
|       RPDI1_: LOGICAL*1(*)  if known, otherwise LOGICAL(*)    |
|       RPDR0_: REAL(*)                                         |
|       RPDR8_: REAL*8(*)                                       |
|       RPDD0_: DOUBLE PRECISION(*)                             |
|       RPDC0_: CHARACTER*(*)   a string, not an array of char|
|                                                               |
| IRET           integer      Out Out return code              |
|   0 = okay                                                    |
| <>0 = error occurred                                          |
|   3 = invalid return/message  package counter tag             |
|   4 = invalid return/message  package operation tag           |
|  19 = object not opened or invalid current_object_handle      |
|                                                               |
----------------------------------------------------------------
```

6.4.5 RPMDUN – Monitor Delete User Node

```
--------------------------------------------------------------
|                                                            |
| Procedure      RPMDUN --- RPc_Monitor_Delete_User_Node     |
|                                                            |
| Synopsis       CALL RPMDUN(NODE,IRET)                      |
|                                                            |
| Task           RPMDUN deletes a node in the MIB data base. |
|                Can be used by an application to manage     |
|                a subtree in the monitor                    |
|                                                            |
| Argument       Type            I/O Usage                   |
| NODE           string          In  User node to be deleted |
| IRET           integer         Out return code             |
|                                                            |
--------------------------------------------------------------
```

6.4.6 RPMSUN – Monitor Set User Node

```
--------------------------------------------------------------
|                                                            |
| Procedure      RPMSUN --- RPc_Monitor_Set_User_Node        |
|                                                            |
| Synopsis       CALL RPMSUN(NODE,VALUE,IRET)                |
|                                                            |
| Task           RPMSUN generates a new node in the MIB data |
|                base.                                       |
|                Can be used by an application to manage     |
|                a subtree in the monitor                    |
|                                                            |
| Argument       Type            I/O Usage                   |
| NODE           string          In  User node to be generated |
| VALUE          string          In  Value for the node      |
| IRET           integer         Out return code             |
|                                                            |
--------------------------------------------------------------
```

6.4.7 RPOACC – Accept a Connection

```
--------------------------------------------------------------
|                                                            |
| Procedure      RPOACC  --- RPc_Object_ACCept               |
|                                                            |
| Synopsis       CALL RPOACC(OBJID,Slave_Address,            |
|               +            Slave_Addr_Lng,Slave_Host_Lng,  |
|               +            Slave_Port_Lng,IRET)            |
|                                                            |
| Task           Accept Connection                           |
|                After the application has called RPOPOP and |
|                has transferred the Master_Address to the   |
|                slave (where RPOCON must be called),        |
```

```
|                the master application must call RPOACC to    |
|                accept the connection.                         |
|                It does not matter which of RPOACC or RPOCON  |
|                is called first.                               |
|                Prior to the call of RPOACC, the object id.   |
|                given by RPOPOP must be set to the current     |
|                object identifier (this is done automatically |
|                by RPOPOP and must be done explicitely with   |
|                RPOUSE only if the current object id. was     |
|                changed in the meantime).                      |
|                If one is accepting more than one connection  |
|                to one port then the OBJID returned by RPOACC |
|                differs from that of RPOPOP.                   |
|                RPOACC sets the current object id. to the new |
|                one.                                           |
|                                                               |
| Remarks        See remarks of RPOPOP.                         |
|                                                               |
| Parameter       Type           I/O Usage                     |
|                                                               |
| OBJID           integer        Out object identifier         |
| Slave_Address   string         Out slave address (remote)    |
| Slave_Addr_Lng  integer        Out length of slave address   |
| Slave_Host_Lng  integer        Out length hostname portion   |
| Slave_Port_Lng  integer        Out length portname portion   |
| IRET            integer        Out return code               |
|                                                               |
----------------------------------------------------------------
```

6.4.8 RPOC2S – Connect Two Servers

```
----------------------------------------------------------------
|                                                               |
| Procedure      RPOC2S - RP_Object_Connect_2_Server            |
|                                                               |
| Synopsis       CALL RPOC2S(OBJID1,OBJID2,HOST2,               |
|               +            PIPID1,PIPID2,IRET)                |
|                                                               |
| Task           Called by the client, RPOC2S establishes a    |
|                data pipe between two server objects.          |
|                The pipe will be tested.                       |
|                The current object identifier is not changed  |
|                in the client nor in the servers.              |
|                RPOC2S blocks until both server objects are   |
|                able to execute a new remote procedure call.  |
|                In the server objects the current transport   |
|                service is used.                               |
| Caution: The number of incoming and outgoing connections in  |
|          the argument list of RPSINI must be large enough.   |
|          In the server with OBJID1 an outgoing connection    |
|          will be established, in the server with OBJID2 an   |
|          incoming one.                                        |
|                                                               |
```

```
| Parameter      Type           I/O Usage                      |
| OBJID1         integer        In  object identifier server 1|
| OBJID2         integer        In  object identifier server 2|
| HOST2          character*(*)  In  host address of server 2  |
|  ' '              = the same address as used for the client  |
|                     server connection  will be used          |
|                     (not yet implemented)                    |
|  'other contents' = this string is used as network address   |
| PIPID1         integer        Out object id of this pipe .. |
| PIPID2         integer        Out .. in server 1 and serv.2 |
| IRET           integer        Out return code               |
|   0 = okay                                                    |
| <>0 = error occurred, e.g.                                    |
|  17 = cannot create a new object handle (in object 1 or 2)   |
|  41 = invalid OBJID1                                          |
|  42 = invalid OBJID2                                          |
|  43 = invalid current service in object 1                     |
|  44 = invalid current service in object 2                     |
|  45 = transport layer error, e.g. connection not established |
|       due unknown host or port address                        |
|  50 = error in received test package                          |
----------------------------------------------------------------
```

6.4.9 RPOCCL – Close Connection

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOCCL  ---  RPc_Object_Connection_CLose      |
|                                                              |
| Synopsis       CALL RPOCCL (IRET)                            |
|                                                              |
| Task           Close a connection that is connected with     |
|                RPOACC or RPOCON.                             |
|                The current object identifier must point to   |
|                that connection (see RPOUSE).                 |
|                If this object id. is derived from a call to  |
|                RPOPOP then the current object id. is not     |
|                cleared and the port can be used for further  |
|                connections or can be closed with RPOPCL.     |
|                If it is derived from RPOACC or RPOCON then   |
|                the current object identifier is cleared.     |
|                                                              |
| Parameter       Type           I/O Usage                     |
| IRET            integer        Out return code               |
----------------------------------------------------------------
```

6.4.10 RPOCON – Connect

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOCON  --- RPc_Object_CONnect                 |
|                                                               |
| Synopsis       CALL RPOCON(Start_Flags,Master_Address,        |
|               +            OBJID,Slave_Address,               |
|               +            Slave_Addr_Lng,Slave_Host_Lng,     |
|               +            Slave_Port_Lng,IRET)               |
|                                                               |
| Task           Connect to the master                          |
|                After a process (called master) has opened a   |
|                port with RPOPOP, an other process (called     |
|                slave) can connect to this port by calling     |
|                RPOCON. The connection must be accepted by     |
|                the master with RPOACC.                        |
|                It does not matter which of RPOACC or RPOCON   |
|                is called first.                               |
|                RPOCON sets the current object identifier      |
|                to the new one returned in OBJID.              |
|                                                               |
| Remarks        See also remarks of RPOPOP.                    |
|                                                               |
| Parameter       Type            I/O Usage                     |
| Start_Flags     integer         In  sum of following flags    |
|     - make object start and finish control output to stdout   |
|                 0                = make output (normal)       |
|                 SFLG_Trace_Obj   = additional trace output    |
|     - make rpc and pipe call control output to stdout         |
|                 0                = make output (normal)       |
|                 SFLG_Trace_Call  = additional trace output    |
|                                                               |
| Master_Address  string          In  master address (remote)   |
| OBJID           integer         Out object identifier         |
| Slave_Address   string          Out slave address (local)     |
| Slave_Addr_Lng  integer         Out length of slave address   |
| Slave_Host_Lng  integer         Out length hostname portion   |
| Slave_Port_Lng  integer         Out length portname portion   |
| IRET            integer         Out return code               |
|                                                               |
-----------------------------------------------------------------
```

6.4.11 RPOCRY – Wait 'Current' - Ready

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOCRY  ---  RPc_Object_Current_ReadY          |
|                                                               |
| Synopsis       CALL RPOCRY(TIMEMS,OBJRDY,IRET)                |
|                                                               |
| Task           Wait until a parallel RPC is                   |
|                ready for transmitting the output arguments    |
|                or a message can be received on a pipe         |
|                from the current object.                       |
|                                                               |
|                If a datapipe message or the return package of|
|                a parallel rpc is already read into the local |
|                buffer, then the system call "select" is not  |
|                done and OBJRDY=1 is returned.                 |
|                This case can occur after                      |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an |
|                inspection with RPOISP.                        |
|                                                               |
| Remark:        It is not necessary to call RPOCRY before a    |
|                call to RPOISP, RPORCT, RPOREC or RPD..R to    |
|                wait until data can be received because they   |
|                (or the RPOREC in the thread of RPORCT)        |
|                are waiting internally until data is           |
|                receiveable.                                   |
|                                                               |
| After RPOCRY:  Upon successful waiting the application can    |
|                use RPOISP (not yet implemented) to            |
|                inspect whether it is a return package of a    |
|                parallel RPC or a message package and then     |
|                RPOREC to receive the return package or the    |
|                datapipe receive routine for message packages.|
|                                                               |
| Parameter      Type           I/O Usage                       |
| TIMEMS         integer        In  Timeout in millisec.        |
|  -1 = waiting infinitely                                      |
|  >= 0 waiting at most this amount of milliseconds             |
| OBJRDY         integer        Out object ready state          |
|  -2 = error occurred, details stored in IRET                  |
|  -1 = no parallel rpc active for the current object           |
|   0 = timeout reached                                         |
|   1 = parallel RPC output or message package can be received |
|       for the current object                                  |
| IRET           integer        Out return code                 |
|   0 = okay                                                    |
| <>0 = error occurred                                          |
|  19 = object not opened or invalid current_object_handle      |
|  53 = error occurred in system call "select"                  |
-----------------------------------------------------------------
```

6.4.12 RPOCUR – Get current Object Id

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOCUR  --- RPc_Object_inquire_CURrent_obj_id |
|                                                               |
| Synopsis       CALL RPOCUR(CUROBJ,CLTOBJ)                     |
|                                                               |
| Task           RPOCUR inquires the 'current object            |
|                identifier' (CUROBJ) used for calling rpc and |
|                all pipe operations.                           |
|                In a server process (i.e. RPBD1, RPBD1P or     |
|                RPBD1A is running)  the object identifier of   |
|                its client (CLTOBJ) is also returned.          |
|                                                               |
| Argument       Type            I/O Usage                      |
| CUROBJ         integer         Out current object identifier |
|                                    or -1 if it is not set     |
| CLTOBJ         integer         Out current client object id. |
|                                    or -1 if it is not set     |
-----------------------------------------------------------------
```

6.4.13 RPOFBD – Finish Block Dispatcher

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOFBD --- RPc_Object_Finish_Block_Dispatcher |
|                                                               |
| Synopsis       CALL RPOFBD(IRET)                              |
|                                                               |
| Task           Finish Remote Block Dispatcher                 |
|                RPOFBD called at client site finishs RPBD1     |
|                at server site.                                |
|                                                               |
| Parameter       Type            I/O Usage                     |
| IRET            integer         Out return code               |
-----------------------------------------------------------------
```

6.4.14 RPOFLS – Flush Send Buffer

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOFLS --- Flush Send Buffer                   |
|                                                               |
| Synopsis       CALL RPOFLS()                                  |
|                                                               |
| Task           RPC FLuSh Call                                 |
|                                                               |
|                Only after a RPSUSE or an explicit RPOFLS or   |
|                an implicit RPOFLS (done by synchronous rpc)   |
|                one can start a new object with RPO1..         |
-----------------------------------------------------------------
```

6.4.15 RPOGRY – Wait 'Generalized' - Ready

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPOGRY  ---  RPc_Object_Generalized_list_ReadY|
|                                                             |
| Synopsis       CALL RPOGRY(N,OBJID,TIMEMS,OBJRDY,OBJ1ST,    |
|               +            nfds,readfds,writefds,exceptfds, |
|               +            rfds,IRET)                       |
| Task           Wait until a parallel RPC is                 |
|                ready for transmitting the output arguments  |
|                or a message can be received on a pipe       |
|                from an object of the given list or          |
|                one other descriptor is ready.               |
|                If datapipe messages or return packages of   |
|                parallel rpcs are already read into local    |
|                buffers, then the system call "select" is not |
|                done and the returned arguments indicate     |
|                only those objects and rfds set to zero      |
|                and readfds, writefds and exceptfds remain   |
|                unchanged.  This case can occur after        |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an |
|                inspection with RPOISP.                      |
|                See also the Remark in the descrip. of RPOCRY.|
| After RPOGRY:  Upon successful waiting the application can  |
|                use RPOUSE to select a ready object, RPOISP to|
|                inspect whether it is a return package of a  |
|                parallel RPC or a message package and then   |
|                RPOREC to receive the return package or the  |
|                datapipe receive routine for message packages.|
|                                                             |
| Parameter      Type            I/O Usage                    |
| N              integer         In  Length of object list    |
| OBJID          integer(N)      In  Array of object identifier|
| TIMEMS         integer         In  Timeout in millisec.     |
|  -1 = waiting infinitely                                    |
|  >= 0 waiting at most this amount of milliseconds           |
| OBJRDY         integer(N)      Out object ready state       |
|  -2 = error occurred, for this object, e.g. invalid object id|
|  -1 = no parallel rpc active for this object of the list    |
|       and timeout reached for a possibly outstanding message |
|   0 = parallel rpc active for this object of the list       |
|       and timeout reached for parallel RPC output or message |
|   1 = parallel RPC output or message package can be received |
|       for this object of the list                           |
| OBJ1ST         integer         Out first ready object       |
|  -2 = error occurred, details stored in IRET, or N=0        |
|  -1 = no parallel rpc active for any object in OBJID        |
|       and timeout reached or a descriptor was first ready   |
|   0 = at least one parallel rpc is active and timeout reached|
|       or a descriptor (see below) was first ready           |
| 1..N= index of the first ready object in the list,          |
```

```
|      OBJID(OBJ1ST) is the object identifier of this object  |
|                                                             |
| nfds           integer        In  Number of used descriptors|
| readfds        integer(*)     I/O Read  file descriptor set |
| writefds       integer(*)     I/O Write file descriptor set |
| exceptfds      integer(*)     I/O Excep.file descriptor set |
| rfds           integer        Out Number of ready descrip-  |
|                                   tors or 0 if timed out    |
|                                   or an object was first    |
|                                   ready, or -1 if an error  |
|                                   is occurred in the select |
|                                   call                      |
|                                                             |
| IRET           integer        Out return code               |
|   0 = okay                                                  |
| <>0 = error occurred                                        |
|  53 = error occurred in system call "select"                |
|                                                             |
| See "man select" for description of nfds, readfds, writefds, |
| exceptfds and its return value, here stored in rfds.        |
| Timout is reached if rfds = 0 and OBJ1ST <= 0.              |
| RPOGRY merges the own socket descriptors and readfds before |
| calling "select" and upon calling they are removed from the |
| result (readfds, rfds).                                     |
|                                                             |
----------------------------------------------------------------
```

6.4.16 RPOGSD – Get Socket Descriptor

```
----------------------------------------------------------------
|                                                             |
| Procedure      RPOGSD  ---  RPc_Object_Get_Socket_Descriptor |
|                                                             |
| Synopsis       CALL RPOGSD(FD,IRET)                         |
|                                                             |
| Task           Get the I/O descriptor of the transport socket|
|                of the transport connection to the current   |
|                object.                                      |
|                                                             |
| Parameter      Type           I/O Usage                     |
| FD             integer        Out File descriptor or 0 if   |
|                                   there is no current object|
| IRET           integer        Out return code               |
|   0 = okay                                                  |
| <>0 = error occurred                                        |
|  19 = object not opened or invalid current_object_handle    |
----------------------------------------------------------------
```

6.4.17 RPOISP – Inspect next Package

```
-----------------------------------------------------------------
| Procedure      RPOISP  ---  RPc_Object_InSpect_Package         |
|                                                                |
| Synopsis       CALL RPOISP(PACTYP,MODCOD,OPCODE,IRET)          |
|                                                                |
| Task           RPOISP waits (if necessary) until a packet has|
|                arrived and then RPORCT reads the header of     |
|                this package (and probably following ones)      |
|                into the internal buffer to inspect the         |
|                content of the package. Until the package is    |
|                read with RPOREC or RPD..R a call to RPOISP     |
|                will always return the same header.             |
|                RPOISP must not be called while an RPORCT       |
|                is active on the same object.                   |
|                                                                |
| Parameter      Type            I/O Usage                       |
| PACTYP         integer         Out package type                |
|                                     0 = synchronous call        |
|                                     1 = asynchronous call       |
|                                     2 = async. buffered call    |
|                                     3 = parallel call           |
|                                     4 = return of sync. call    |
|                                     5 = return of paral. call|
|                                     6 = datapipe package        |
|                                     7 = error package           |
| MODCOD         integer         Out module code                 |
|                                     0 for dfnrpclib routines    |
|                                    >0 application module        |
| OPCODE         integer         Out operation code              |
|                                     50 = RPDI0S/R               |
|                                     51 = RPDI2S/R               |
|                                     52 = RPDI1S/R               |
|                                     53 = RPDL0S/R               |
|                                     54 = RPDL2S/R               |
|                                     55 = RPDL1S/R               |
|                                     56 = RPDR0S/R               |
|                                     57 = RPDR8S/R               |
|                                     58 = RPDD0S/R               |
|                                     59 = RPDC0S/R               |
| IRET           integer         Out return code                 |
|   0 = okay                                                     |
| <>0 = error occurred                                           |
|  19 = object not opened or invalid current_object_handle       |
|  58 = call not allowed while receiving output argu. w. RPORCT|
|                                                                |
-----------------------------------------------------------------
```

6.4.18 RPOLRY – Wait 'List' - Ready

```
-----------------------------------------------------------------
| Procedure      RPOLRY  ---  RPc_Object_List_ReadY             |
|                                                               |
| Synopsis       CALL RPOLRY(N,OBJID,TIMEMS,OBJRDY,OBJ1ST,IRET)|
|                                                               |
| Task           Wait until a parallel RPC is                   |
|                ready for transmitting the output arguments    |
|                or a message can be received on a pipe         |
|                from an object of the given list.              |
|                If datapipe messages or return packages of     |
|                parallel rpcs are already read into local      |
|                buffers, then the system call "select" is not  |
|                done and the returned arguments are indicating|
|                only those objects. This case can occur after  |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an  |
|                inspection with RPOISP.                        |
|                See also the Remark in the descrip. of RPOCRY.|
| After RPOLRY:  Upon successful waiting the application can    |
|                use RPOUSE to select a ready object, RPOISP to|
|                inspect whether it is a return package of a    |
|                parallel RPC or a message package and then     |
|                RPOREC to receive the return package or the    |
|                datapipe receive routine for message packages.|
|                                                               |
| Parameter      Type            I/O Usage                      |
| N              integer         In  Length of object list      |
| OBJID          integer(N)      In  Array of object identifier|
| TIMEMS         integer         In  Timeout in millisec.       |
|  -1 = waiting infinitely                                      |
|  >= 0 waiting at most this amount of milliseconds             |
| OBJRDY         integer(N)      Out object ready state         |
|  -2 = error occurred, for this object, e.g. invalid object id|
|  -1 = no parallel rpc active for this object of the list      |
|       and timeout reached for a possibly outstanding message  |
|   0 = parallel rpc active for this object of the list         |
|       and timeout reached for parallel RPC output or message  |
|   1 = parallel RPC output or message package can be received  |
|       for this object of the list                             |
| OBJ1ST         integer         Out first ready object         |
|  -2 = error occurred, details stored in IRET, or N=0          |
|  -1 = no parallel rpc active for any object in OBJID          |
|       and timeout reached                                     |
|   0 = at least one parallel rpc is active and timeout reached|
| 1..N= index of the first ready object in the list,            |
|       OBJID(OBJ1ST) is the object identifier of this object   |
| IRET           integer         Out return code                |
|   0 = okay                                                    |
| <>0 = error occurred                                          |
|  53 = error occurred in system call "select"                  |
-----------------------------------------------------------------
```

6.4.19 RPOPCL – Close Port

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPOPCL  ---  RPc_Object_Port_Close           |
|                                                             |
| Synopsis       CALL RPOPCL (IRET)                           |
|                                                             |
| Task           Close a port that is opened with RPOPOP.     |
|                The current object identifier must point to  |
|                that port, see RPOUSE and RPOPOP.            |
|                Prior to the call of RPOPCL all connections  |
|                to this port must be closed with RPOCCL.     |
|                At return the current object id. is cleared. |
|                                                             |
| Parameter       Type           I/O Usage                    |
| IRET            integer        Out return code              |
---------------------------------------------------------------
```

6.4.20 RPOPOP – Open Port einer Verbindung

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPOPOP  --- RPc_Object_Port_OPen             |
|                                                             |
| Synopsis       CALL RPOPOP(Start_Flags,OBJID,Master_Address, |
|               +            Master_Addr_Lng,Master_Host_Lng,  |
|               +            Master_Port_Lng,Master_Pass_Lng,  |
|               +            IRET)                             |
|                                                             |
| Task           Port Open                                    |
|                It returns the address of an opened port.    |
|                The address is a string with the format      |
|                hostname:portname:password:                  |
|                The length of each portion is inclusive ':'.  |
|                The hostname is the official internet name   |
|                as stored in /etc/hosts or the numeric addr.. |
|                The password is a 20 byte alphanum. string.  |
|                The process that is calling RPOPOP is called |
|                the 'master' of that connection.             |
|                The current object id. is set to the returned |
|                OBJID.                                       |
|                                                             |
| Remarks        Establishing and releasing the connection is |
|                done by the participating processes          |
|                'master' and 'slave'                         |
|                by calling ...          with the task ...    |
|                                                             |
|                RPOPOP                  opening the port     |
|                not part of DFN-RPC:    starting the slave and|
|                                        transmitting master's |
|                                        address to the slave |
|                            RPOCON      connecting to master |
```

```
|               RPOACC                 acceptig the connect. |
|                                                            |
|               Now the application can use the connection   |
|               for rpcs and as a data pipe.                 |
|                                                            |
|               RPOCCL      RPOCCL     closing the connection|
|               RPOPCL          -      closing the port      |
|                                                            |
|               It is all the same which of RPOACC and RPOCON |
|               and which of RPO1CF a. RPO1SF is called first.|
|                                                            |
| Parameter       Type            I/O Usage                  |
|                                                            |
| Start_Flags     integer         In  sum of following flags |
|      - make object start and finish control output to stdout |
|                0                 = make output (normal)    |
|                SFLG_Trace_Obj   = additional trace output  |
|      - make rpc and pipe call control output to stdout      |
|                0                 = make output (normal)    |
|                SFLG_Trace_Call  = additional trace output  |
|                                                            |
| OBJID           integer         Out object identifier      |
| Master_Address  string          Out master address (local) |
| Master_Addr_Lng integer         Out length of master address |
| Master_Host_Lng integer         Out length hostname portion |
| Master_Port_Lng integer         Out length portname portion |
| Master_Pass_Lng integer         Out length password portion |
| IRET            integer         Out return code            |
--------------------------------------------------------------
```

6.4.21 RPORCT – Receive Via a Thread

```
--------------------------------------------------------------
| Procedure      RPORCT  ---  RPc_Object_ReCeive_parallelreturn|
|                             in_a_second_Thread             |
| Synopsis       CALL RPORCT(IRET)                           |
|                                                            |
| Task           RPORCT can start a second thread which is   |
|                executing there a call to RPOREC.           |
|                The start of a second thread is only done   |
|                if threads are supported by the operating   |
|                system.                                     |
|                (In the moment the start of a thread is not |
|                implemented.)                               |
|                                                            |
| Parameter      Type           I/O Usage                    |
| IRET           integer        Out return code              |
|   0 = okay                                                 |
| <>0 = error occurred                                       |
|  19 = object not opened or invalid current_object_handle   |
|  55 = any parallel rpc is not active in the current object |
|  58 = call not allowed while receiving output argu. w. RPORCT|
--------------------------------------------------------------
```

6.4.22 RPOREC – Receive Parallel Return

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOREC  ---  RPc_Object_RECeive_parallelreturn|
|                                                               |
| Synopsis       CALL RPOREC(IRET)                              |
|                                                               |
| Task           RPOREC waits until a parallel RPC in the       |
|                current object is ready for transmitting the   |
|                output arguments.                              |
|                The output arguments are received and stored   |
|                into the original argument list.               |
|                RPOREC blocks until the output arguments can   |
|                be received. Therefore it is not necessary to  |
|                wait with RPOCRY, RPOLRY or RPOGRY until data  |
|                is receiveable.                                |
|                                                               |
|                If called by the application and after the     |
|                application has issued a RPORCT and RPORCT     |
|                has started a second thread, then this         |
|                routine waits only until the RPORCT thread     |
|                has finished.                                  |
|                                                               |
| Parameter      Type           I/O Usage                       |
| IRET           integer        Out return code                 |
|   0 = okay                                                    |
| <>0 = error occurred                                          |
|  19 = object not opened or invalid current_object_handle      |
|  55 = any parallel rpc is not active in the current object    |
|  58 = call not allowed while receiving output argu. w. RPORCT|
|                                                               |
-----------------------------------------------------------------
```

6.4.23 RPORRY – Wait 'Read-List' - Ready

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPORRY  ---  RPc_Object_Read_list_ReadY        |
|                                                               |
| Synopsis       CALL RPORRY(N,OBJID,TIMEMS,OBJRDY,OBJ1ST,      |
|               +            NFD,RFD,RFDRDY,RFD1ST,IRET)        |
|                                                               |
| Task           Wait until a parallel RPC is                   |
|                ready for transmitting the output arguments    |
|                or a message can be received on a pipe         |
|                from an object of the given list or            |
|                one other descriptor is ready for read.        |
|                                                               |
|                If datapipe messages or return packages of     |
|                parallel rpcs are already read into local      |
|                buffers, then the system call "select" is not  |
|                done and the returned arguments indicate       |
```

```
|                only those objects. This case can occur after |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an |
|                inspection with RPOISP.                       |
|                See also the Remark in the descrip. of RPOCRY.|
|                                                              |
| After RPORRY:  Upon successful waiting the application can   |
|                use RPOUSE to select a ready object, RPOISP to|
|                inspect whether it is a return package of a   |
|                parallel RPC or a message package and then    |
|                RPOREC to receive the return package or the   |
|                datapipe receive routine for message packages.|
|                                                              |
| Parameter      Type            I/O Usage                     |
| N              integer         In  Length of object list     |
| OBJID          integer(N)      In  Array of object identifier|
| TIMEMS         integer         In  Timeout in millisec.      |
|  -1 = waiting infinitely                                     |
|  >= 0 waiting at most this amount of milliseconds            |
| OBJRDY         integer(N)      Out object ready state        |
|  -2 = error occurred, for this object, e.g. invalid object id|
|  -1 = no parallel rpc active for this object of the list     |
|       and timeout reached for a possibly outstanding message |
|   0 = parallel rpc active for this object of the list        |
|       and timeout reached for parallel RPC output or message |
|   1 = parallel RPC output or message package can be received |
|       for this object of the list                            |
| OBJ1ST         integer         Out first ready object        |
|  -2 = error occurred, details stored in IRET, or N=0         |
|  -1 = no parallel rpc active for any object in OBJID         |
|       and timeout reached or a descriptor was first ready    |
|   0 = at least one parallel rpc is active and timeout reached|
|       or a descriptor (see below) was first ready            |
| 1..N= index of the first ready object in the list,           |
|       OBJID(OBJ1ST) is the object identifier of this object  |
|                                                              |
| NFD            integer         In  Length of descriptor list |
| RFD            integer(NFD)    In  Array of read descriptors |
| RFDRDY         integer(NFD)    Out descriptor ready state    |
|   0 = no data seen at this filedescriptor                    |
|   1 = data can be received at this filedescriptor            |
| RFD1ST         integer         Out first ready descriptor    |
|  -2 = error occurred, details stored in IRET, or NFD=0       |
|   0 = timeout reached or an object (see above) is first ready|
| 1..N= index of the first ready descriptor in the list,       |
|       RFD(RFD1ST) is its file descriptor                     |
|                                                              |
| IRET           integer         Out return code               |
|   0 = okay                                                   |
| <>0 = error occurred                                         |
|  53 = error occurred in system call "select"                 |
|                                                              |
| Timout is reached if OBJ1ST <= 0 and RFD1ST <= 0.            |
----------------------------------------------------------------
```

6.4.24 RPOSLF,RPOSRF – Set Local/Remote Trace Flags

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOSLF  ---  RP_Object_Set_Local_trace_Flags  |
|                RPOSRF  ---  RP_Object_Set_Remote_trace_Flags |
|                                                              |
| Synopsis       CALL RPOSLF (SOBJFL, COBJFL, SCT4FL,          |
|               +             CCT4FL, IRET)                    |
|                CALL RPOSRF (SOBJFL, COBJFL, SCT4FL,          |
|               +             CCT4FL, IRET)                    |
|                                                              |
| Task           Set the trace, monitor and transport flags.   |
|                                                              |
|                For each connection trace and monitor flags   |
|                are set with the Start_Flags argument while   |
|                starting the object with the routines         |
|                RPOPOP, RPOCON, RPO1CS, RPO2CI or RPO3SI      |
|                and the transport service flags are set       |
|                in the Type_of_Tp_SerVice argument of RPSINI. |
|                                                              |
|                RPOSLF:  The change is done for the local site|
|                         of the connection of the current     |
|                         object.                              |
|                                                              |
|                RPOSRF:  It can be called only on the client  |
|                         site while on the server site a      |
|                         block dispatcher is running.         |
|                         The changes are done on the server.  |
|                                                              |
|                With RPOSLF and RPOSRF the following flags    |
|                can be changed for the connection of the      |
|                current object at the site where RPOSLF is    |
|                called.                                       |
|                With SOBJFL and COBJFL one can set and clear  |
|                each trace or monitor flag:                   |
|                  OBJSTO - Trace Object                       |
|                  OBJSTC - Trace Calls                        |
|                  OBJSMO - Monitoring Off                     |
|                  OBJSM2 - Monitoring with sleep 2 sec in each|
|                           monitor step                       |
|                                                              |
|                With SCT4FL and CCT4FL one can set and clear  |
|                each transport service flags:                 |
|                  CT4PRT - print ct4 calls with argument list |
|                           (without buffers of CT4DIN, CT4DRQ)|
|                  CT4BUF - print buffers of CT4DIN and CT4DRQ |
|                  CT4PAU - pause at begin and end of ct4 calls|
|                                                              |
|                In each of these arguments any sum of the     |
|                mentioned flags can be used. The flags are    |
|                defined in the include file dfnrpc.inc .      |
|                The value 0 makes no change.                  |
```

```
|                The set operation is done first.                |
|                                                                |
| Parameter       Type           I/O Usage                       |
| SOBJFL          integer        In  Set   object flags          |
| COBJFL          integer        In  Clear object flags          |
| SCT4FL          integer        In  Set   ct4    flags          |
| CCT4FL          integer        In  Clear ct4    flags          |
| IRET            integer        Out return code                 |
|   0 = okay                                                     |
| <>0 = error occurred                                           |
|  19 = object not opened or invalid current_object_handle       |
----------------------------------------------------------------
```

6.4.25 RPOSYN – Empty Synchronous RPC

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOSYN                                        |
|                                                              |
| Synopsis       CALL RPOSYN(IRET)                             |
|                                                              |
| Task           An empty synchronization call                 |
|                                                              |
| Parameter       Type           I/O Usage                     |
| IRET            integer        Out return code               |
----------------------------------------------------------------
```

6.4.26 RPOUSE – Select Current Object

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOUSE  --- RPc_Object_USE                    |
|                                                              |
| Synopsis       CALL RPOUSE(OBJID,IRET)                       |
|                                                              |
| Task           USE an alternate object for following         |
|                rpc and pipe operations.                      |
|                The parameter OBJID is stored as 'current     |
|                object identifier'.                           |
|                The 'current object identifier' is used for   |
|                calling rpcs and pipe operations.             |
|                At the beginning of RPBD1, RPBD1P or RPBD1A   |
|                the value of the 'current object identifier'  |
|                is stored as 'current client object ident.'   |
|                that is used for receiving rpcs until the end |
|                of RPBD1, RPBD1P or RPBD1A.                   |
|                                                              |
|                If there is any information buffered for the  |
|                connection of the previous 'current object'   |
|                then and only then a new buffer must be used. |
|                As default, the first buffer has a length     |
|                64 kByte, the next 8 buffers have each 4 kByte|
```

```
|               and the last 256 buffers have each 256 bytes. |
| Argument      Type            I/O Usage                     |
| OBJID         integer         In  object identifier         |
|                                   see RPO..I/S              |
| IRET          integer         Out return code               |
|  19 = object not opened or invalid current_object_handle    |
|                                                             |
---------------------------------------------------------------
```

(geaenderte Funktionalitaet, vgl. DFN-RPC, Rel. 1.0, S. 99)

6.4.27 RPQSSC/S – Server-Server-Network

```
---------------------------------------------------------------
|                                                             |
| Procedure     RPQSSC    ---  RP_Quick_Server_Server_        |
|                                _interconnection_network     |
|                                _init_at_Client              |
|                                                             |
| Synopsis      CALL RPQSSC(NSERVER,SOBJID,NCONN,CONNE1,      |
|              +            CONNE2,IRET)                      |
|                                                             |
| Task          Initialization of a network of communication  |
|               links between multiple servers.               |
|                                                             |
|               RPQSSC must be called by the client.          |
|               In each Server RPQSSS must be called.         |
|                                                             |
|               The current object id. is not changed.        |
|               Invalid input in CONNE1 and CONNE2 will be    |
|               ignored.                                      |
|                                                             |
| Examples      See below and examples/sscon.                 |
|                                                             |
| Parameter     Type              I/O Usage                   |
| NSERVER       integer           In  number of servers       |
| SOBJID        integer(NSERVER)  In  object id of each server|
| NCONN         integer           In  number of connctions    |
|                                     to be established       |
|                                     between the servers     |
| CONNE1        integer(NCONN)    In  server numbers of ...   |
| CONNE2        integer(NCONN)    In  ...the both endpoints   |
| IRET          integer           Out return code             |
|   0= okay                                                   |
| <>0= error occurred                                         |
|  63= error in a server while opening the port in RPQSSS     |
|  64= error in a server while connecting in RPQSSS           |
|  65= error in a server - no respoce for 60 sec -- fatal     |
|  and error numbers from RPOUSE, RPDIOS/R, RPOLRY, RPDCOS/R  |
|                                                             |
|  After a return with iret=63 or iret=64 some of the communi- |
|  cation links are established and the application can use   |
|  them.                                                      |
---------------------------------------------------------------
```

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPQSSS    ---  RP_Quick_Server_Server_        |
|                                 _interconnection_network     |
|                                 _init_at_Server              |
|                                                              |
| Synopsis       CALL RPQSSS(NSERVER,SNUM,NCONN,COBJID,        |
|               +            SOBJID,IRET)                      |
|                                                              |
| Task           Initialization of a network of communication  |
|                links between multiple servers.               |
|                                                              |
|                RPQSSS must be called in each server while    |
|                in the client RPQSSC is running.              |
|                                                              |
|                The current object id. must be the client     |
|                object id and it is not changed.              |
|                                                              |
| Examples       See below and examples/sscon.                 |
|                                                              |
| Parameter      Type             I/O Usage                    |
| NSERVER        integer          I/O number of servers        |
|                                     In:  declared length of  |
|                                          SOBJID              |
|                                     Out: its value in RPQSSC|
| SNUM           integer          Out number of this server    |
| NCONN          integer          Out number of connections    |
|                                     between this server and  |
|                                     the other servers        |
| COBJID         integer          Out object id of the client  |
| SOBJID         integer(NSERVER)  Out object id of the connec-|
|                                     tions to the other       |
|                                     servers or -1            |
| IRET           integer          Out return code              |
|   0= okay                                                    |
| <>0= error occurred                                          |
|  66= did not get correct protocol unit from client''s RPQSSC |
|  67= NSERVER is to small, see also RPQSSC and dfnrpc.servers |
|  68= timeout (30s) reached while waiting for a connection    |
|  69= errors occurred in some other servers in RPQSSS         |
|  and error numbers from RPOUSE, RPDIOS/R, RPOPOP, RPOGSD,    |
|      RPDCOS/R, RPOCON, RPORRY, RPOACC                        |
|                                                              |
|  After a return with iret=11,12,68 or 69 some of the communi-|
|  cation links are established and the application can use   |
|  them.                                                       |
|                                                              |
----------------------------------------------------------------
```

```
----------------------------------------------------------------
|    Example with arectangle of 2*3 servers                    |
|    --------------------------------------                    |
|                                                              |
|   [Client]__                                                 |
|    | | |\ \ \__c3_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _s1_          |
|    | | | \ \_c2_ _ _ _ _ _ _ __s1_                 \         |
|    | | |  \c1_s1_                 \                 \        |
|    | | |         \                 \                 \       |
|    | | |         [S1]==============[S2]==============[S3]    |
|    | | |          ||  s3        s2  ||  s3        s2  ||     |
|    | | |          ||s2              ||s4              ||s3   |
|    | | |          ||                ||                ||     |
|    | |  \_c6__ _ _|| _ _ _ _ _ _ _ _|| _ _ _ _s1_     ||     |
|    |  \_c5__ _ _ _|| _ _ _ __s1_    ||           \    ||     |
|     \_c4_ __s1_   ||            \   ||            \   ||     |
|                \  ||             \  ||             \  ||     |
|                 \ ||s3            \ ||s2            \ ||s2   |
|                  \||  s2        s3 \||  s4        s3 \||     |
|                  [S4]==============[S5]==============[S6]    |
|                                                              |
|        [..]    Name of the process                           |
|        === ||  Connections between the servers established   |
|                by RPQSSC at [Client] and RPQSSS at [S1...S6] |
|        _ _  |  Connections between client and server establ. |
|                by RPQUCI at [Client] and RPQUSI at [S1...S6] |
|        c1,...  Internal object id at the client process      |
|        s1,...  Internal object id at the server processes    |
|                                                              |
|        Input of RPQSSC                                       |
|          NSERVER = 6                     (Output of RPQUCI)  |
|          SOBJID  = (c1,c2,c3,c4,c5,c6)   (Output of RPQUCI)  |
|          NCONN   = 7                                         |
|          CONNE1  = (1, 2, 1, 2, 3, 4, 5)                     |
|          CONNE2  = (2, 3, 4, 5, 6, 5, 6)                     |
|                    vertically one can read the pairs of      |
|                    servers that should be connected          |
|          or if some connections are given more than once:   |
|          NCONN   = 13                                        |
|          CONNE1  = (1, 1, 2, 5, 6, 6, 2, 3, 4, 5, 6, 5, 6)   |
|          CONNE2  = (2, 2, 1, 6, 5, 3, 1, 2, 1, 2, 3, 4, 5)   |
|        Both sets of input will result in the same network    |
|        of server-server-connections.                         |
|                                                              |
|        Output of RPQSSS  at                                  |
|        Server:  NSERVER SNUM NCONN COBJID SOBJID             |
|           S1 :     6      1    2     s1   (-1 s3 -1 s2 -1 -1)|
|           S2 :     6      2    3     s1   (s2 -1 s3 -1 s4 -1)|
|           S3 :     6      3    2     s1   (-1 s2 -1 -1 -1 s3)|
|           S4 :     6      4    2     s1   (s3 -1 -1 -1 s2 -1)|
|           S5 :     6      5    3     s1   (-1 s2 -1 s3 -1 s4)|
|           S6 :     6      6    2     s1   (-1 -1 s2 -1 s3 -1)|
|                                             |  |  |  |  |  | |
|                Index of the array element:  1  2  3  4  5  6 |
```

```
| Client                                                         |
|    INTEGER NSERVER,SOBJID(512),NCONN,CONNE1(7),CONNE2(7),IRET|
|    NSERVER = 6                                                 |
|    CALL RPQUCI(NSERVER,SOBJID,IRET)                            |
|    NCONN=7 ; CONNE1=(1,2,1,2,3,4,5) ; CONNE2=(2,3,4,5,6,5,6) |
|    CALL RPQSSC(NSERVER,SOBJID,NCONN,CONNE1,CONNE2,IRET)        |
| Server                                                         |
|    INTEGER NSERVER,SNUM,NCONN,COBJID,SOBJID(512),IRET          |
|    CHARACTER*100 Master_Address                                |
|    Master_Address=' '                                          |
|    IF (IARGC().GE.1) CALL GETARG(1,Master_Address)             |
|    CALL RPQUSI(Master_Address,COBJID,IRET)                     |
|    NSERVER = 512                                               |
|    CALL RPQSSS(NSERVER,SNUM,NCONN,COBJID,SOBJID,IRET)          |
 ----------------------------------------------------------------
```

6.4.28 RPQUCF – Quick Start - Client Finish

```
----------------------------------------------------------------
| Procedure      RPQUCF    ---  RP_QUick_Client_site_Finish    |
|                                                              |
| Synopsis       CALL RPQUCF(IRET)                             |
|                                                              |
| Task           Finish of a client object.                    |
|                It closes the connections established by      |
|                RPQUCI.                                       |
|                                                              |
|                Internally RPQUCF is calling for each server  |
|                  RPOUSE (OBJID,IRET)                         |
|                    with OBJID stored by RPQUCI               |
|                  RPO1CF (IRET)                               |
|                and then                                      |
|                  RPSFIN (IRET)                               |
|                  RPBFIN (IRET)                               |
|                                                              |
| Examples       See example rshexa, crshexa and rpcpipe.      |
|                                                              |
| Parameter      Type              I/O Usage                   |
| IRET           integer           Out return code             |
|   0= okay                                                    |
| <>0= error occurred                                          |
|  see error numbers from RPOUSE, RPO1CF, RPSFIN and RPBFIN    |
----------------------------------------------------------------
```

6.4.29 RPQUCI – Quick Start - Client Init

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPQUCI    ---  RP_QUick_Client_site_Init      |
|                                                              |
| Synopsis       CALL RPQUCI(NSERVER,SOBJID,IRET)              |
|                                                              |
```

```
| Task           Initialization of a client object.           |
|                                                             |
|                If there is a file named dfnrpc.servers in the|
|                current working directory, then each line of  |
|                it is used to start a server process by       |
|                calling system (see man-page of system(3F)).  |
|                Not more than NSERVER servers are started.    |
|                Lines containing only blanks or beginning with|
|                # are treated as comment. If there is a line  |
|                beginning with -norepeat and the number of    |
|                server starting lines in the file is less than|
|                the argument NSERVER then the argument NSERVER|
|                is reduced to this number of servers;         |
|                otherwise the server starts are repeated until|
|                the number given by NSERVER is reached.       |
|                Each server start consist of the following    |
|                steps:                                        |
|                - opening a port for the transport connection,|
|                - substituting the keyword DFNRPCADDR in the  |
|                  server start line by the address of this    |
|                  port,                                       |
|                - starting the server by calling 'system',    |
|                - waiting for and accepting the connection,   |
|                  established by the server routine RPQUSP.   |
|                The first three steps are done in a first loop|
|                and the last step is done in a second loop.   |
|                The command on each server start line must    |
|                finish immediately (e.g. by a & sign at the   |
|                end of the line). Therefore all starts of the |
|                servers are running partially parallel.       |
|                                                              |
|                If there is no such file, RPQUCI asks to start|
|                each server interactively.                    |
|                                                              |
|                RPQUCI establishes the connections to all     |
|                servers.                                      |
|                                                              |
|                Internally RPQUCI calls                       |
|                  RPBINI (IRET)                               |
|                  RPSINI (7,0,0,InCon,OutCon,ISERID,IRET)     |
|                  with InCon  incoming connections with       |
|                       InCon  = MAX(OBJECTS_MAXINST/2,NSERVER)|
|                  and  OutCon outgoing connections with       |
|                       OutCon = OBJECTS_MAXINST-InCon         |
|                  and OBJECTS_MAXINST = 512                   |
|                  and the initial value of NSERVER.           |
|                and for each server:                          |
|                  RPOPOP (0,OBJID,M_Address,M_Addr_Lng,       |
|                       M_Host_Lng,M_Port_Lng, M_Pass_Lng,IRET)|
|                  SYSTEM, ISHELL or interactive start         |
|                  RPOACC (OBJID,S_Address,S_Addr_Lng,         |
|                          S_Host_Lng,S_Port_Lng, IRET)        |
|                and at the end RPOUSE with the OBJID of the   |
|                first server.                                 |
```

```
|                To parallelize the server starts the calls to |
|                POPOP and SYSTEM are done in a first loop and |
|                RPOACC is called in a second loop.            |
|                All object ids are stored globally for RPQUCF.|
|                                                              |
| Examples       See example rshexa, crshexa and rpcpipe.      |
|                                                              |
| Parameter      Type              I/O Usage                   |
| NSERVER        integer           I/O number of servers       |
| SOBJID         integer(NSERVER)  Out object id of each server|
| IRET           integer           Out return code             |
|   0= okay                                                    |
| <>0= error occurred                                          |
|  59= error in opening existing file dfnrpc.servers on unit 21|
|  60= error in reading from file dfnrpc.servers               |
|  61= more than 512 chars in a resulting server start line    |
|  and error numbers from RPBINI, RPSINI, RPOPOP and RPOACC    |
| Example for dfnrpc.servers:                                  |
|  # Servers startet with a remote shell:                      |
|  rsh servername_1 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  rsh servername_2 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  rsh servername_3 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  rsh servername_4 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  #CAUTION: to see all error messages issued by 'rsh'         |
|  #         one must omit '>/dev/null'                        |
|                                                              |
|  # The following server must be started with telnet:         |
|  echo " ";echo "start anywhere rshexa_sb DFNRPCADDR";echo " "|
|                                                              |
|  # The following line prevents, that the server list is      |
|  # repeated to start the number of servers wanted by client   |
|  -norepeat                                                   |
----------------------------------------------------------------
```

6.4.30 RPQUSF – Quick Start - Server Finish

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPQUSF    ---  RP_QUick_Server_site_Finish    |
|                                                              |
| Synopsis       CALL RPQUSF(IRET)                             |
|                                                              |
| Task           Finish of a server object.                    |
|                It closes the connection  established by      |
|                RPQUSI.                                       |
|                                                              |
|                Internally RPQUSF is calling                  |
|                  RPOUSE (OBJID,IRET)                         |
|                    with OBJID from stored by RPQUSI          |
|                  RPO1SF (IRET)                               |
|                  RPSFIN (IRET)                               |
|                  RPBFIN (IRET)                               |
|                                                              |
```

```
| Examples       See example rpcpipe and con.                      |
|                                                                  |
| Parameter                Type    I/O Usage                       |
| IRET                     integer Out return code                 |
|   0= okay                                                        |
| <>0= error occurred                                              |
|  see error numbers from RPOUSE, RPO1SF, RPSFIN and RPBFIN        |
|                                                                  |
--------------------------------------------------------------------
```

6.4.31 RPQUSI – Quick Start - Server Init

```
--------------------------------------------------------------------
|                                                                  |
| Procedure      RPQUSI    ---  RP_QUick_Server_site_Init.         |
|                                                                  |
| Synopsis       CALL RPQUSI(Master_Address,OBJID,IRET)            |
|                                                                  |
| Task           Initialization of a server object.                |
|                                                                  |
|                Master_Address should contain the address of      |
|                the port opened by the client routine RPQUCI.     |
|                If its value is ' ' then the address of the       |
|                client is asked interactively.                    |
|                The current object id. is set to the returned     |
|                OBJID.                                            |
|                                                                  |
|                Internally RPQUSI calls                           |
|                  RPBINI (IRET)                                   |
|                  RPSINI (7,0,0,max_in_conn,max_out_conn,         |
|                          ISERID,IRET)                            |
|                  RPOCON(0, M_Address, OBJID, S_Address,          |
|                     S_Addr_Lng, S_Host_Lng, S_Port_Lng, IRET)|
|                                                                  |
|                RPSINI is called with a maximum                   |
|                  of  OBJECTS_MAXINST/2 incoming connections      |
|                  and OBJECTS_MAXINST/2 outgoing connections.     |
|                                                                  |
|                OBJID is stored globally for RPQUSF.              |
|                                                                  |
| Examples       See example rpcpipe and con.                      |
|                                                                  |
| Parameter                Type    I/O Usage                       |
| Master_Address           string  In  master address (remote)     |
| OBJID                    integer Out object id of the client     |
| IRET                     integer Out return code                 |
|   0= okay                                                        |
| <>0= error occurred                                              |
|  see error numbers from RPBINI, RPSINI, RPOCON                   |
|                                                                  |
--------------------------------------------------------------------
```

6.4.32 RPQUSP – Quick Start - Server Program

```
----------------------------------------------------------------
| Procedure      RPQUSP    ---  RP_QUick_Server_site_Program   |
|                                                              |
| Synopsis       CALL RPQUSP(Master_Address, IRET)             |
|                                                              |
| Task           Special main program of the server site       |
|                                                              |
|                Master_Address should contain the address of  |
|                the port opened by the client routine RPQUCI. |
|                If its value is ' ' then the address of the   |
|                client is asked interactively.                |
|                The current object id. is set to that of the  |
|                client.                                       |
|                                                              |
|                Internally RPQUSP calls                       |
|                  RPQUSI (Master_Address,OBJID,IRET)          |
|                  RPBD1  (IRET)                               |
|                  RPQUSF (IRET)                               |
|                                                              |
| Examples       See example rshexa and crshexa.               |
|                                                              |
| Parameter                Type    I/O Usage                   |
| Master_Address           string  In  master address (remote) |
| IRET                     integer Out return code             |
|   0= okay                                                    |
| <>0= error occurred                                          |
|  see error numbers from RPBINI, RPSINI, RPOCON, RPBD1,       |
|                         RPO1SF, RPSFIN and RPBFIN            |
----------------------------------------------------------------
```

6.4.33 RPSFIN – Transport Service Finish

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPSFIN                                        |
|                                                              |
| Synopsis       CALL RPSFIN(IRET)                             |
|                                                              |
| Task           Finish of a transport Service                 |
|                                                              |
| Argument        Type           I/O Usage                     |
| IRET            integer        Out return code               |
|                                                              |
----------------------------------------------------------------
```

6.4.34 RPSINI – Transport Service Init

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPSINI                                         |
|                                                               |
| Synopsis       CALL RPSINI(Type_of_Tp_SerVice,                |
|                            Type_of_Network_Service,           |
|                            Type_of_Tp_SoftWare,               |
|                            Max_Outgoing_Connections,          |
|                            Max_Incoming_Connections,          |
|                            SERID, IRET)                       |
|                                                               |
| Task           Initialization of a transport Service          |
|                                                               |
| Argument                 Type    I/O Usage                    |
| Type_of_Tp_SerVice       integer In  Type of TransportService|
|           values: CT4TCP or CT4TC4                            |
|                  (CT4TC0-3,CT4RTS,CT4T70 are not implemented|
|                   possibly plus flags CT4PRT,CT4BUF,CT4PAU, |
|                                       CT4FFL, CT4PFL          |
| Type_of_Network_Service  integer In  Type of Network Service |
|           values: CT4NO                                       |
| Type_of_Tp_SoftWare      integer In  Type of Transp. Software|
|           values: CT4S0 or CT4SUL                             |
| Max_Outgoing_Connections integer In  Max. of Outgoing Connec.|
|           value in the client: "number of servers"            |
|           value in the server: 1                              |
| Max_Incoming_Connections integer In  Max. of Incoming Connec.|
|           value in the client: "number of servers"            |
|           value in the server: 1                              |
| SERID                    integer Out object identifier        |
| IRET                     integer Out return code              |
|                                                               |
-----------------------------------------------------------------
```

6.4.35 RPSLTI – Transport Service List

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPSLTI                                         |
|                                                               |
| Synopsis       CALL RPSLTI()                                  |
|                                                               |
| Task           List all Transport Interface parameters        |
|                                                               |
| Argument        Type            I/O Usage                     |
| (none)                                                        |
|                                                               |
-----------------------------------------------------------------
```

6.4.36 RPSUSE – Transport Service Select

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPSUSE                                         |
|                                                               |
| Synopsis       CALL RPSUSE(SERID, IRET)                       |
|                                                               |
| Task           Flush communication with current object and    |
|                USE an alternate and already opened service    |
|                for next object start                          |
|                                                               |
| Argument        Type           I/O Usage                      |
| SERID           integer        In  service identifier         |
|                                    see RPSINI                 |
| IRET            integer        Out return code                |
|                                                               |
-----------------------------------------------------------------
```

6.4.37 RPV... – Konversionsroutinen

6.4.37.1 Anwenderdaten → Transferkodierung

```
=================================================================
|                                                               |
| Module         SENDCONVERT                                    |
|                -----------                                    |
|                                                               |
| Purpose        Routine for INTEGER- and REAL-number           |
|                conversion from local to transfer enconding.   |
|                                                               |
| Exported subroutines :                                        |
|                                                               |
|   1.) RPVISB   -   Initialisation of the send buffer          |
|                                                               |
|   2.) RPVFSB   -   Make send buffer ready for transmission    |
|                                                               |
|   3.) RPVCSB   -   Main conversion subroutine, converts data |
|                     from the data array into the send buffer |
|                                                               |
=================================================================


Module description of SENDCONVERT
=================================


  General information on variables used by the exported subroutines :
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


  1.) variables belonging to the data array
```

```
    - DATA_array        source array of data, the type of the array
                        elements is free.
    - DATA_bytes        length of a DATA_array element in bytes
    - start_index       conversion of DATA_array starts from
                        this index
    - stop_index        conversion goes until this index - 1


2.) variables belonging to the transfer side

    - TRANSFER_bytes    length of a SEND_buffer element in bytes
    - TRANSFER_type     transfer enconding of a converted element :
                               =  1    ->  INTEGER, BIG    ENDIAN
                               =  2    ->  INTEGER, LITTLE ENDIAN
                               =  3    ->  LOGICAL, BIG    ENDIAN
                               =  4    ->  LOGICAL, LITTLE ENDIAN
                               =  5    ->  REAL,    IEEE
                               =  6    ->  REAL,    CRAY
                               =  7    ->  REAL,    DEC
                               =  8    ->  REAL,    VAX

3.) variables belonging to the send buffer

     - SEND_buffer       send buffer
     - SB_used           the last position used in SEND_buffer
                         in bytes
     - SB_sent           the last position (in bytes) in SEND_buffer
                         which has been sent already. This variable
                         makes it possible that the send buffer can be
                         cleared of elements which have been already
                         sent, and so the sending of parts of the send
                         buffer is possible.
     - SB_length         length of SEND_buffer in INTEGER words.
                         Internally, only a length which can
                         be devided by 16 (bytes) is used.
4.) misc. variables

           - FLAGS       INTEGER variable in which some different
                         commands can be given to the conversion
                         subroutine :

                         FLAGS = 1 -  Convert Overflow Check enabled
                                      ==>  If an overflow is detected,
                                           IRET is set to 40 and
                                           the subroutine stops
                                           converting at this point.
                         FLAGS = 2 -  Convert Underflow Check enabled
                                      ==>  If an underflow is
                                           detected,
                                           IRET is set to 41 and
                                           the subroutine stops
                                           converting at this point.
                                           If Convert Underflow Check
```

```
                               is disabled, the underflow
                               will be corrected by
                               writing a zero into
                               SEND_buffer.
                  FLAGS = 3 -  Convert Underflow and Overflow
                               Check enabled

 - IRET           variable for error indication, a value
                  of 0 means that the subroutine has ended
                  without errors.

                  IRET  type  description
                    0    A    no errors
                   10    B    DATA_bytes invalid for
                              this TRANSFER_type
                   11    B    TRANSFER_bytes invalid for
                              this TRANSFER_type
                   12    B    TRANSFER_type invalid
                   20    B    start_index invalid ( < 1 )
                   21    B    stop_index  invalid ( < 1 )
                   22    B    stop_index < start_index
                   30    B    SB_used invalid
                              ( < 0                     )
                              ( > SB_length*Y_DEF_bytes )
                   31    B    SB_sent invalid
                              ( < 0                     )
                              ( > SB_length*Y_DEF_bytes )
                   32    B    SB_sent > SB_used
                   33    B    SB_length * Y_DEF_bytes < 16
                   40    C    overflow detected
                   41    C    underflow detected

                   type A : routine ended without errors
                   type B : routine not executed
                   type C : routine executed in parts,
                            for more details, see
                            description of routine.

 - *Errorlabel     alternative jump back adress if an
                   error is detected
```

6.4.37.2 RPVCSB – Convert into Send Buffer

```
-----------------------------------------------------------------
|                                                               |
| subroutine     RPVCSB                                         |
|                                                               |
| Synopsis       CALL RPVCSB (DATA_array, DATA_bytes,           |
|                             start_index, stop_index,          |
|                             TRANSFER_bytes, TRANSFER_type,    |
|                             SEND_buffer, SB_used,             |
|                             SB_sent, SB_length,               |
|                             FLAGS, IRET, *Errorlabel)         |
```

```
|                                                              |
|                                                              |
| Task           conversion subroutine, convert data from the  |
|                data array into the send buffer               |
|                                                              |
| Parameter       Type            I/O Usage                    |
| DATA_array      free            In  see module description   |
| DATA_bytes      INTEGER         In  see module description   |
| start_index     INTEGER         I/O see module description   |
| stop_index      INTEGER         In  see module description   |
|                                                              |
| TRANSFER_bytes  INTEGER         In  see module description   |
| TRANSFER_type   INTEGER         In  see module description   |
|                                                              |
| SEND_buffer     INTEGER         I/O see module description   |
| SB_used         INTEGER         I/O see module description   |
| SB_sent         INTEGER         I/O see module description   |
| SB_length       INTEGER         In  see module description   |
|                                                              |
| FLAGS           INTEGER         In  see module description   |
| IRET            INTEGER         Out see module description   |
| *Errorlabel     errorlabel      --- see module description   |
|                                                              |
----------------------------------------------------------------
```

```
Subroutine description of RPVCSB
--------------------------------


a.)   error testing of the following variables :

      DATA_bytes
      TRANSFER_bytes
      TRANSFER_type
      start_index
      stop_index
      SB_used
      SB_sent
      SB_length

      If an error is detected, setting of IRET and leaving the routine.

b.)   The send buffer is cleared from element already send :

      if ( SB_sent >= 16 ) and ( SB_used - SB_sent <= 64 ) then
        free := SB_sent - ( SB_sent mod 16 )
        for i := 1 to ( SB_used-free ) do
          SEND_buffer( i ) := SEND_buffer( i + free )
          SB_sent := SB_sent - free
          SB_used := SB_used - free
      endif

c.)   create the following internal variable :
```

```
B_SB_length := Y_DEF_bytes * SB_length / 16 * 16
(byte orientated length of the send buffer, the used length is
 made to a multiple of 16 Bytes)
```

d.) SEND_buffer alignment :

```
filled := SB_used mod TRANSFER_bytes
if filled<>0
  SB_used := SB_used + ( TRANSFER_bytes - filled )
endif
```

e.) main conversion loop :

```
IRET := 0

if start_index=stop_index then exit_routine
( nonthing to convert )
while SB_used < B_SB_length do
  begin
    DATA_item := DATA_array[start_index]
    ( dummy variable )
    if OVERFLOW_CHECK=enabled
       and OVERFLOW_DETECTED( DATA_item )
         then
           IRET := 40
           exit_routine
    endif
    if UNDERFLOW_CHECK=enabled
      and UNDERFLOW_CORRECTION=disabled
      and UNDERFLOW_DETECTED( DATA_item )
        then
          IRET := 41
          exit_routine
    endif
    if UNDERFLOW_CORRECTION=enabled
      and UNDERFLOW_DETECTED( DATA_item )
        then
          DATA_item := 0
    endif

    SEND_buffer[SB_used+1] := CONVERT( DATA_item )

    SB_used := SB_used + TRANSFER_bytes
    start_index := start_index + 1
    if start_index=stop_index then exit_routine

  end
```

The following results are possible by routine exit :

1.) IRET = 0, type A
      ==> no errors

```
and
a.)  start_index = stop_index
     ==> all data converted, SB_used is
                less or equal to B_SB_length
b.)  start_index < stop_index
     ==> not all data converted,
              SB_used = B_SB_length = multiple of 16
```

```
2.)  IRET <> 0, type B
     ==> errors detected, the routine hasn't been executed.

3.)  IRET <> 0, type C
     ==> errors by overflow/underflow ( CHECKING = enabled )
         all elements until the element which caused the error
         will be converted correctly, start_index and SB_used
         will be raised correctly => start_index points to
         the bad element in the data array.
```

6.4.37.3 RPVFSB – Finish Send Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVFSB                                        |
|                                                              |
| Synopsis       CALL RPVFSB (SB_used)                         |
|                                                              |
| Task           make send buffer ready to send                |
|                                                              |
| Parameter       Type           I/O Usage                     |
| SB_used          INTEGER       I/O see module description    |
|                                                              |
----------------------------------------------------------------
```

```
  Subroutine description of RPVFSB
  --------------------------------

  ( alignment of the send buffer to a multiple of 16 bytes )

             filled := SB_used mod 16
             if filled<>0
               then
                 SB_used := SB_used + ( 16 - filled )
               endif
```

6.4.37.4 RPVISB – Init Send Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVISB                                        |
|                                                              |
| Synopsis       CALL RPVISB (SB_used, SB_sent)                |
|                                                              |
| Task           Initialisation of the send buffer             |
|                                                              |
| Parameter       Type           I/O Usage                     |
| SB_used          INTEGER       Out see module description    |
| SB_sent          INTEGER       Out see module description    |
|                                                              |
----------------------------------------------------------------


  Subroutine description of RPVISB
  --------------------------------

  SB_used := 0         ( buffer defined as empty )
  SB_sent := 0         ( nothing sent already    )
```

6.4.37.5 Transferkodierung → Anwenderdaten

```
================================================================
|                                                              |
| Module         RECEIVECONVERT                                |
|                --------------                                |
|                                                              |
| Purpose        Routine for INTEGER- and REAL-number          |
|                conversion from transfer to local enconding.  |
|                                                              |
| Exported subroutines :                                       |
|                                                              |
|   1.) RPVIRB   -   Receive buffer initialisation             |
|                                                              |
|   2.) RPVFRB   -   Make receive buffer alignment             |
|                                                              |
|   3.) RPVCRB   -   Main conversion subroutine, converts data |
|                    from the receive buffer into              |
|                    the data array.                           |
|                                                              |
================================================================


Module description of RECEIVECONVERT
====================================


  General information on variables used by the exported subroutines :
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
1.) variables belonging to the data array

    - DATA_array        destination data array, the converted data
                        is put in here
    - DATA_bytes        length of a DATA_array element in bytes
    - start_index       converted data is put into DATA_array
                        starting at this index
    - stop_index        data will be put into DATA_array until this
                        index - 1

2.) variables belonging to the transfer side

    - TRANSFER_bytes    length of a RECEIDE_buffer element in bytes
    - TRANSFER_type     transfer enconding of a received element :
                             =  1    ->  INTEGER  BIG    ENDIAN
                             =  2    ->  INTEGER, LITTLE ENDIAN
                             =  3    ->  LOGICAL, BIG    ENDIAN
                             =  4    ->  LOGICAL, LITTLE ENDIAN
                             =  5    ->  REAL,    IEEE
                             =  6    ->  REAL,    CRAY
                             =  7    ->  REAL,    DEC
                             =  8    ->  REAL,    VAX

3.) variables belonging to the receive buffer

     - RECEIVE_buffer    receive buffer
     - RB_read           the last position (in bytes) in RECEIVE_buffer
                         upto the conversion has been done yet
     - RB_used           actual length of in RECEIVE_buffer in bytes

RECEIVE_buffer and RB_used will be given out by
the Network_Input_Routine as a result from a network input.

4.) misc. variables

           - FLAGS       INTEGER variable in which some different
                         commands can be given to the conversion
                         subroutine :

                         FLAGS = 1 -  Convert Overflow Check enabled
                                      ==>  If an overflow is detected,
                                           IRET is set to 40 and
                                           the subroutine stops
                                           converting at this point.
                         FLAGS = 2 -  Convert Underflow Check enabled
                                      ==>  If an underflow is
                                           detected,
                                           IRET is set to 41 and
                                           the subroutine stops
                                           converting at this point.
                                           If Convert Underflow Check
                                           is disabled, the underflow
                                           will be corrected by
```

```
                          writing a zero into
                          RECEIVE_buffer.
             FLAGS = 3 -  Convert Underflow and Overflow
                          Check enabled

 - IRET      variable for error indication, a value
             of 0 means that the subroutine has ended
             without errors.

             IRET  type  description
               0    A    no errors
              10    B    DATA_bytes invalid for
                         this TRANSFER_type
              11    B    TRANSFER_bytes invalid for
                         this TRANSFER_type
              12    B    TRANSFER_type invalid
              20    B    start_index invalid ( < 1 )
              21    B    stop_index  invalid ( < 1 )
              22    B    stop_index < start_index
              30    B    RB_read invalid ( < 0 )
              31    B    RB_used invalid ( < 0 )
              32    B    RB_read > RB_used
              40    C    overflow detected
              41    C    underflow detected

              type A : routine ended without errors
              type B : routine not executed
              type C : routine executed in parts,
                       for more details, see
                       description of routine.

 - *Errorlabel  alternative jump back adress if an
                error is detected
```

6.4.37.6 RPVCRB – Convert from Receive Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVCRB                                        |
|                                                              |
| Synopsis       CALL RPVCRB (DATA_array, DATA_bytes,          |
|                             start_index, stop_index,         |
|                             TRANSFER_bytes, TRANSFER_type,   |
|                             RECEIVE_buffer, RB_read, RB_used,|
|                             FLAGS, IRET, *Errorlabel)        |
|                                                              |
| Task           conversion subroutine, convert data from the  |
|                receive buffer into the data array            |
|                                                              |
| Parameter       Type            I/O Usage                    |
| DATA_array      free            Out see module description   |
l DADA_bytec      INTEGER         In  see module description   |
| start_index     INTEGER         I/O see module description   |
```

```
| stop_index      INTEGER         In  see module description  |
|                                                             |
| TRANSFER_bytes  INTEGER         In  see mndule $eccripdion  |
| DRANSFER_type   IFTEGER         In  see mo$%le desc ipdimn  |
|                                                             |
$ RECEIVE_buffer  ANTEGER         I/O see module description  |
| RB_read         INTEGER         I/O see module description  |
| RB_used         INTEGER         I/O see module description  |
|                                                             |
| FLAGS           INTEGER         In  see module descriptaon  |
| IRET            INTEGER         Out see module description  |
| *Errorlabel     errorlabel      --- see module description  |
|                                                             |
-----------------------------------------------------------------
```

```
 Subroutine description of RPVCRB
 --------------------------------


 a.)   error testing of the following variables :

       DATA_bytes
       TRANSFER_bytes
       TRANSFER_type
       start_index
       stop_index
       RB_read
       RB_used

       If an error is detected, setting of IRET and leaving the routine.

 b.)   RECEIVE_buffer alignment :

       filled := RB_read mod TRANSFER_bytes
       if filled<>0
         RB_read := RB_read + ( TRANSFER_bytes - filled )
       endif
   if RB_read > RB_used then
           RB_read := RB_read - ( TRANSFER_bytes - filled )
         exit_routine
       ( not enough data in RECEIVE_buffer to do alignment )
   endif

 c.)   main conversion loop :

       IRET := 0

       if start_index=stop_index then exit_routine
       ( no space in DATA_array )
       while (RB_read + TRANSFER_bytes) <= RB_used do
         begin
           DATA_item := RECEIVE_buffer[RBread + 1]
           ( dummy variable )
```

```
    if OVERFLOW_CHECK=enabled
       and OVERFLOW_DETECTED( DATA_item )
         then
           IRET := 40
           exit_routine
    endif
    if UNDERFLOW_CHECK=enabled
      and UNDERFLOW_CORRECTION=disabled
      and UNDERFLOW_DETECTED( DATA_item )
        then
          IRET := 41
          exit_routine
    endif
    if UNDERFLOW_CORRECTION=enabled
      and UNDERFLOW_DETECTED( DATA_item )
        then
          DATA_item := 0
    endif

    DATA_array[start_index] := CONVERT( DATA_item )

    RB_read := RB_read + TRANSFER_bytes
    start_index := start_index + 1
    if start_index=stop_index then exit_routine

  end
```

d.) Clearing the RECEIVE_buffer :

```
if (RB_used - RB_read) < 16
  then
    diff := (RB_read div 16) * 16
    RB_read := RB_read - diff
    RB_used := RB_used - diff
    for i:=1 to RB_used do
      RECEIVE_buffer[i] := RECEIVE_buffer[i + diff]
endif
```

The following results are possible by routine exit :

```
1.)  IRET = 0, type A
      ==> no errors
          and
      a.)  RB_read = RB_used

            ==> all data deconverted, start_index is
                     lower or equal to stop_index

          b.)  RB_read < RB_used

                ==> not all data deconverted,
                    a.) start_index = stop_index
                        No place left in DATA_array
```

```
                    b.) start_index < stop_index
                another call to the deconverting routine
                    after network input is necessary
                RB_read + TRANSFER_bytes > RB_used
                ==> RB_used < 16
```

```
2.)  IRET <> 0, type B
     ==> errors detected, the routine hasn't been executed.

3.)  IRET <> 0, type C
     ==> errors by overflow/underflow ( CHECKING = enabled )
         all elements until the element which caused the error
         will be converted correctly, start_index and RB_read
         will be raised correctly => RB_read + 1 points to
         the bad element in the RECEIVE_buffer.
```

6.4.37.7 RPVFRB – Finish Receive Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVFRB                                        |
|                                                              |
| Synopsis       CALL RPVFRB (RECEIVE_buffer, RB_read, RB_used,|
|                             DONE)                            |
|                                                              |
| Task           receive buffer alignment                      |
|                                                              |
| Parameter       Type           I/O Usage                     |
| RECEIVE_buffer  free           I/O see module description    |
| RB_read         INTEGER        I/O see module description    |
| RB_used         INTEGER        I/O see module description    |
| DONE            LOGICAL        Out see module description    |
|                                                              |
----------------------------------------------------------------
```

```
  Subroutine description of RPVFRB
  --------------------------------

             DONE := FALSE

             if RB_read > RB_used then exit_routine
             ( Error in variables )

  1. alignment of the receive buffer to a multiple of 16 bytes :

             filled := RB_read mod 16
             if filled<>0
               then
                 RB_read := RB_read + ( 16 - filled )
               endif
```

```
            if RB_read > RB_used
              then
                RB_read := RB_read - ( 16 - filled )
                DONE := FALSE
               ( not enough read to do the alignment )
              else
                DONE := TRUE
              endif

2. receive buffer clearing  :

  ( see routine RBCLEAR )

            if (RB_used - RB_read) < 16
              then
                call RBCLEAR(RECEIVE_buffer, RB_read, RB_used)
              endif
```

6.4.37.8 RPVIRB – Init Receive Buffer

```
-----------------------------------------------------------------
|                                                               |
| subroutine     RPVIRB                                         |
|                                                               |
| Synopsis       CALL RPVIRB (RB_read, RB_used)                 |
|                                                               |
| Task           receive buffer initialisation                  |
|                                                               |
| Parameter       Type            I/O Usage                     |
| RB_read         INTEGER         Out see module description   |
| RB_used         INTEGER         Out see module description   |
|                                                               |
-----------------------------------------------------------------


  Subroutine description of RPVIRB
  --------------------------------

  buffer defined empty :

  RB_read := 0
  RB_used := 0
```

6.5 Select(2) – man-page

NAME select – synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
int select (int nfds, fd_set *readfds, fd_set *writefds,
              fd_set *exceptfds, struct timeval *timeout);
FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION `Select` examines the I/O descriptor sets whose addresses are passed in `readfds`, `writefds`, and `exceptfds` to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first `nfds` descriptors are checked in each set; i.e. the descriptors from 0 through `nfds` - 1 in the descriptor sets are examined. On return, `select` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is the return value.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: `FD_ZERO(&fdset)` initializes a descriptor set `fdset` to the null set. `FD_SET(fd, &fdset)` includes a particular descriptor `fd` in `fdset`. `FD_CLR(fd, &fdset)` removes `fd` from `fdset`. `FD_ISSET(fd, &fdset)` is nonzero if `fd` is a member of `fdset`, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which is normally at least equal to the maximum number of descriptors supported by the system.

If `timeout` is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If `timeout` is a zero pointer, the select blocks indefinitely. To effect a poll, the `timeout` argument should be non-zero, pointing to a zero-valued timeval structure.

Any of `readfds`, `writefds`, and `exceptfds` may be given as zero pointers if no descriptors are of interest.

RETURN VALUE `Select` returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires then `select` returns 0. If `select` returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS An error return from `select` indicates:

[EBADF] One of the descriptor sets specified an invalid descriptor.

[EINTR] A signal was delivered before the time limit expired and before any of the selected events occurred.

[EINVAL] The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO accept(2), connect(2), read(2), write(2), recv(2), send(2).

BUGS The dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of `<sys/types.h>`.

`Select` should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the `select` call.

6.6 DFN-RPC Enumerations

Der Includefile des DFN-RPC **$DFNRPCINCLUDE/dfnrpc.inc** enthält folgende Enumerations:

```
C  ...Object Handling Enumerartions
C_ ......Restart_Flag_NO
      INTEGER    OBJRNO
      PARAMETER (OBJRNO = 0)
C_ ......Restart_Flag_YES
      INTEGER    OBJRYE
      PARAMETER (OBJRYE = 1)
C_ ......SFLG_Master_Sh
      INTEGER    OBJSMS
      PARAMETER (OBJSMS = 00000001)
C_ ......SFLG_Master_Csh
      INTEGER    OBJSMC
      PARAMETER (OBJSMC = 00000002)
C_ ......SFLG_Slave_Sh
      INTEGER    OBJSSS
      PARAMETER (OBJSSS = 00000016)
C_ ......SFLG_Slave_Csh
      INTEGER    OBJSSC
      PARAMETER (OBJSSC = 00000032)
C_ ......SFLG_Write
      INTEGER    OBJSW
      PARAMETER (OBJSW  = 00000256)
C_ ......SFLG_No_Subst
      INTEGER    OBJSNS
      PARAMETER (OBJSNS = 00000512)
C_ ......SFLG_No_Backslash
      INTEGER    OBJSNB
      PARAMETER (OBJSNB = 00001024)
C_ ......SFLG_Int_Auth
      INTEGER    OBJSIA
      PARAMETER (OBJSIA = 00002048)
C_ ......SFLG_Dyn_Auth
      INTEGER    OBJSDA
      PARAMETER (OBJSDA = 00004096)
C_ ......SFLG_No_Call
      INTEGER    OBJSNC
      PARAMETER (OBJSNC = 00008192)
C_ ......SFLG_Trace_Obj
      INTEGER    OBJSTO
      PARAMETER (OBJSTO = 00016384)
```

```
C_ ......SFLG_Trace_Call
      INTEGER    OBJSTC
      PARAMETER (OBJSTC = 00032768)
C_ ......SFLG_Monitor_Off
      INTEGER    OBJSMO
      PARAMETER (OBJSMO = 00065536)
C_ ......SFLG_Monitor_Sleep2
      INTEGER    OBJSM2
      PARAMETER (OBJSM2 = 00131072)
C  ...Object Handling Error numbers
C     (none)
C  ...ct4 enumerations  -  types of transport services:
C_ ......ISO - Transport Class 0
      INTEGER CT4TC0
      PARAMETER(CT4TC0=0)
C_ ......ISO - Transport Class 1
      INTEGER CT4TC1
      PARAMETER(CT4TC1=1)
C_ ......ISO - Transport Class 2
      INTEGER CT4TC2
      PARAMETER(CT4TC2=2)
C_ ......ISO - Transport Class 3
      INTEGER CT4TC3
      PARAMETER(CT4TC3=3)
C_ ......ISO - Transport Class 4
      INTEGER CT4TC4
      PARAMETER(CT4TC4=4)
C_ ......ISO - Reliable Transport Service
      INTEGER CT4RTS
      PARAMETER(CT4RTS=5)
C_ ......T.70
      INTEGER CT4T70
      PARAMETER(CT4T70=6)
C_ ......TCP
      INTEGER CT4TCP
      PARAMETER(CT4TCP=7)
C_ ....additional flags
C_ .....print ct4 calls with argument list (without buffers)
      INTEGER CT4PRT
      PARAMETER(CT4PRT=256)
C_ .....print buffers in the argument list of CT4DIN and CT4DRQ
      INTEGER CT4BUF
      PARAMETER(CT4BUF=512)
C_ .....pause at begin and end of a ct4 subroutine
      INTEGER CT4PAU
      PARAMETER(CT4PAU=1024)
C_ .....force flushing in CT4DRQ
      INTEGER CT4FFL
      PARAMETER(CT4FFL=2048)
C_ .....prevent flushing in CT4DRQ
      INTEGER CT4PFL
      PARAMETER(CT4PFL=4096)
C  ...ct4 enumerations  -  types of underlying network service
C_ ......Network Default
```

```
      INTEGER CT4NO
      PARAMETER(CT4NO=0)
C  ...ct4 enumerations  -  types of underlying transport software
C_ ......Software Default
      INTEGER CT4SO
      PARAMETER(CT4SO=0)
C_ ......Software ULTRA
C_    Ultra native connection with ultra compatibility library
C_    (might also be set on an IP connection, but this causes
C_     lack of some special features)
C_    (cannot be used if the libdfnrpc.a is not compiled with
C_     the option, that enables the use of the UltraNet
C_     TCP compatibility library)
      INTEGER CT4SUL
      PARAMETER(CT4SUL=1)
C_
C_      CT4SO : Native UltraNet connection is not possible.
C_      CT4SUL: 1. The UltraNet compatibility library must be
C_                 used instead of the normal TCP library
C_              2. Native UltraNet connections are possible.
C_              3. If one uses only a normal TCP connection or
C_                 an UltraIP connection then the interrupt and
C_                 termination handler won't work and some
C_                 additional features are disabled.
C_                 Its not recommended to install TCP connections
C_                 if CT4SUL is set, although this is supported.
C
C_      In other words:
C_      CT4SUL should be used only in conjunction with native UltraNet
C_      connections. If one uses CT4SUL with TCP connections then
C_      user interrupts cannot be sent to the other side.
C_      CT4SUL must be used if one makes a native UltraNet connection.
C_
C  ...ct4 enumerations  -  possible states of a connection
C_    CO_CL, connection closed
C_    PO_OP, connection has open port, but
C_           currently not connected to a remote site
C_    CO_OP, connection with remote site established
C  ...ct4 enumerations  -  other arguments
C_ ......FLUSH
      INTEGER CT4FLU
      PARAMETER(CT4FLU=1)
C_ ......NO FLUSH
      INTEGER CT4NFL
      PARAMETER(CT4NFL=0)
C  ...ct4 error codes
C OUT OF TIME ERROR
      INTEGER CT4OOT
      PARAMETER(CT4OOT=1011)
C Transport Provider Abort - connection closed
      INTEGER CT4TPA
      PARAMETER(CT4TPA=1037)
```

7 Stub-Generator, Kompilieren, Binden

Aufruf des Stub-Generators:

```
$DFNRPCBIN/dfnrpcgen [-n] [-b block_name]
                     [-c module_name]  [-s module_name]
                     [-l module_name]  [-r module_name]
                     [-f option_file]  [-e error_file]
                     [-T transfertype]
                     [-M manufacturer]
                     [-A architecture]
                     [-O operatingsystem]
                     [-F fortrancompiler]
                     input_file1  input_file2  ...
```

input_file
Dieser RPI-File enthält die Remote Procedure Interface Definition, geschrieben in der Fortran RPI Definition Language (F-RPIDL) (siehe Kapitel 5.3.4 auf Seite 63) der verteilten Anwendung oder Teile davon (siehe Kapitel 5.3.4 auf Seite 63).

Der Inhalt mehrerer input_files wird als eine zusammenhängende Einheit betrachtet. Besitzt ein input_file kein Suffix, dann wird automatisch das Suffix **.rpi** angefügt.

Als Basis für die Namen der generierten Output-Files (*output_base*) wird das Prefix des ersten Input-Files (ohne führende Directory-Angaben) genommen.

-n
Die Option -n unterdrückt das automatische Generieren der Application-Sourcefiles *output_base_block/module_name*.**f**, d.h. es werden nur die Stubs (_stb.m4, _stb.f, _clt.m4, _clt.f, _srv.m4, _srv.f) generiert.

-b block_name
Es werden nur die Output-Files zum Block *block_name* generiert, d.h. der Application-Sourcefile *output_base _block_name*.**f** und der Stub-Presourcefile *output_base_block_name*_**stb.m4** und der Stub-Sourcefile *output_base_block_name*_**stb.f**

-c module_name
Es werden nur die Clientstubs zum Module *module_name* generiert, d.h. der Stub-Presourcefile *output_base _module_name*_**clt.m4** und der Stub-Sourcefile *output_base_module_name*_**clt.f**

-s module_name
Es werden nur die Serverstubs zum Module *module_name* generiert, d.h. der Stub-Presourcefile *output_base_module_name*_**srv.m4** und der Stub-Sourcefile *output_base_module_name*_**srv.f**

-l module_name
Es werden nur die Application-Sourcefiles zum Module *module_name* generiert, d.h. *output_base_module_name*.**f**. C*RPC und C*RPC+ Zeilen werden *nicht* aktiviert, d.h. es werden Module für den *lokalen* Einsatz erzeugt.

Es werden aber alle Qualifier [...] in Parseable-Areas (d.h. von C*[RPI_BEG/CON] bis C*[RPI_NON/END]) innerhalb von Fortran-Statements durch Leerzeichen ersetzt.

-r module_name
Wie -l, nur daß C*RPC und C*RPC+ Zeilen *aktiviert* werden, d.h. es werden Module für den *remote*, d.h. verteilten Einsatz erzeugt.

-e error_file
Der Stub-Generator schreibt seine Fehlermeldungen auf diesen *error_file*, Default ist *output_base*.**err**

-T transfertype	Zur Übertragung der Argumentelisten zwischen zwei Blöcken müssen die Daten in ein einheitliches Austauschformat umgewandelt werden. Es stehen folgende Austauschformate zur Verfügung:

transfer type	Real- und Integer-Zahlenformat
ieee	ieee, big endian (default)
dec	ieee, little endian
vax	vax F/D/H-floating, little endian
cray	REAL∗4: ieee, REAL∗8/16: Cray, big endian

Erst bei der Umwandlung des Stub-Presourcefiles (..._stb.m4) in den Stubsourcefile (..._stb.f) wird diese Option ausgewertet.

-M manufacturer	Die Stubs sind für verschiedene Rechner unterschiedlich. Der Default ist, daß die Stubs für den Rechner generiert werden, auf dem der Stub-Generator aufgerufen wird (gemäß der bei der Installation des DFN-RPC angegebenen Konfiguration). Mit den Optionen -M, -A, -O und -F kann man die Stubs für einen anderen Rechner generieren. Erst bei der Umwandlung des Stub-Presourcefiles (..._stb.m4) in den Stubsourcefile (..._stb.f) werden diese Optionen ausgewertet.
-A architecture	Siehe -M.
-O operatingsystem	Siehe -M.
-F fortrancompiler	Siehe -M.

<u>Liste für -M, -A, -O, -F:</u>

```
manufacturer architecture operating system fortran compiler

  alliant    FX           Concentrix_5.0   fortran
  apollo     DN10000      SR10.2           ftn
  convex     C120         ConvexOS_V7.1    fc
             C120ieee     ConvexOS_V7.1    fc
  cray       cray2        unicos_4.0       cf77
                          unicos_5.0       cf77
                          unicos_6.0       cf77
                          unicos_7.0       cf77
             ymp          unicos_7.0       cf77
  dec        dec          ultrix           f77
                          OSF_1            f77
             vax          ultrix           fort
             vax          vms              fort
  fujitsu    vp           uxpm             frt -Ab
  hp         pa_risc      hp_ux            f77
  ibm        RS6000       AIX              xlf
  intel      paragon      OSF_1            f77
  sg         iris3d       IRIX             f77
             iris4d       IRIX_3.2         f77
                          IRIX_5.1         f77
  sun        sun3         SunOS_3.5        f77
                          SunOS_4.0.3      f77
                          SunOS_4.1        f77
             sun4         SunOS_3.5        f77
                          SunOS_4.0.3      f77
                          SunOS_4.1        f77
                          SunOS_52         f77
```

Falls man einen Rechner hat, der keiner dieser Kombinationen entspricht, so möge man eine vergleichbare Kombination benützen.

Zu den Block- und Modul-Optionen:

-b, -c, -s, -l, -r können mehrfach angegeben werden.

Gegenseitige Ausschlüsse der Optionen:

- -n , -l und -r schließen sich gegenseitig aus.
- -b kann nicht benützt werden, wenn eine der Modul-Optionen -c, -s, -l oder -r benützt wird (und umgekehrt).
- -b ist nur sinnvoll, wenn die Inputfiles einen Block Definition Part enthalten.
- -c, -s, -l oder -r sind nur sinnvoll, wenn die Inputfiles einen Module Definition Part enthalten.

Zu den Output-Filenamen:

Es werden folgende Files vom Stub-Generator erzeugt:

- Wenn die Inputfiles einen Block Definition Part enthalten und keine der Modul-Optionen -c, -s, -l oder -r angegeben sind, dann werden für alle Blöcke (Default) oder die mit -b angegebenen die folgenden Files generiert:

Stub-Presourcefile eines Blocks:	*output_base_block_name*_**stb.m4**
Stub-Sourcefile eines Blocks:	*output_base_block_name*_**stb.f**
Application-Sourcefile eines Blocks:	*output_base_block_name*.**f**

- Wenn die Inputfiles keinen Block Definition Part enthalten oder eine der Modul-Optionen -c, -s, -l oder -r angegeben ist, dann werden für alle Module (Default) oder die mit -c, -s, -l und -r angegebenen die folgenden Files generiert:

Stub-Presourcefile eines Client-Moduls:	*output_base_module_name*_**clt.m4**
Stub-Sourcefile eines Client-Moduls:	*output_base_module_name*_**clt.f**
Stub-Presourcefile eines Server-Moduls:	*output_base_module_name*_**srv.m4**
Stub-Sourcefile eines Server-Moduls:	*output_base_module_name*_**srv.f**
Application-Sourcefile eines Moduls:	*output_base_module_name*.**f**

  Wenn keine der Optionen -c, -s, -l,-r angegeben sind (Default-Fall), dann wird der Application-Sourcefile wie bei Option -r erstellt.

Beispiel:

1. stubgen -b b3 xxx/yyy/file1.zzz file2

   Input: xxx/yyy/file1.zzz file2**.rpi**

   Output: file1_b3_stb.m4
   file1_b3_stb.f
   file1_b3.f in der current_directory $PWD

Der Stub-Generator erzeugt die Stubs in Form von Fortran-Files. Diese müssen vom Anwender kompiliert werden. Die Stubs enthalten bei lokaler Installation das Statement

```
      INCLUDE 'dfnrpc_stub.inc'
```

Dieser File befindet sich in **$DFNRPCINCLUDE**. Mit einem entsprechenden Link oder einer Compiler-Option muß man ihn für die Kompilation zugänglich machen. Bei zentraler Installation enthält das Include-Statement einen absoluten Pfad.

Anschließend müssen sie zusammen mit dem entsprechenden Block (Client oder Server) zusammengebunden werden. Es muß außerdem die Bibliothek $DFNRPCLIB/libdfnrpc.a hinzugebunden werden.

In unserem Beispiel $DFNEXAMPLES/sum:

```
# Includefile, falls man advanced Routinen einsetzt:
# --------------------------------------------------
ln -s $DFNRPCINCLUDE/dfnrpc.inc dfnrpc.inc

# Includefile zur Stub-Kompilation mittels Link (nur bei lokaler Installation):
# -----------------------------------------------------------------------------
ln -s $DFNRPCINCLUDE/dfnrpc_stub.inc dfnrpc_stub.inc

# Block cb
# --------
f77 -c sum_cb.f
f77 -c sum_cb_stb.f
f77 -o sum_cb sum_cb.o sum_cb_stb.o $DFNRPCLIB/libdfnrpc.a


# Block sb
# --------
f77 -c sum_sb.f
f77 -c sum_sb_stb.f
f77 -o sum_sb sum_sb.o sum_sb_stb.o $DFNRPCLIB/libdfnrpc.a


# Block lb
# --------
f77 -o sum_lb sum_lb.f
```

8 C-Programmierung

Der DFN-RPC kann auch für Programme eingesetzt werden, die in C programmiert sind. Hierbei muß die Anwendung die Umsetzung auf die Fortran-Interfaces des DFN-RPCs realisieren. Dies sind:

- Die Fortran-Routinennamen, z.B. MYSUBR, sind in C-Routinennamen umzusetzen. Diese lauten bei dem Beispiel:

  Bei cray: MYSUBR
  Bei vax, __hpux, _AIX und apollo: mysubr
  Beim Fortran-C-Converter f2c und bei anderen Herstellern: mysubr_

- Fortran-Strings sind durch zwei Argumente zu ersetzen:

  an der gleichen Position wie in der Fortran-Parameterliste ist der Pointer auf den String anzugeben und am Ende der Parameterliste ist mittels Call-by-Value die Länge des Strings zu übergeben.

  Bei cray ist dagegen # include <fortran.h> anzugeben und die Strings sind mittels _cptofcd zu übergeben.

- Die Initialisierung der Fortran-I/O erfordert folgendes:

  Bei f2c: Das Hauptprogramm muß MAIN__() heißen.
  Bei sun: Am Anfang u. Ende von main() muß man f_init() bzw. f_exit() rufen.

- Bei Convex sind folgende Environment-Variablen für die Programmausführung notwendig (in csh-Notation):

  setenv FOR005 SYS\$INPUT
  setenv FOR006 SYS\$OUTPUT

- Beim Binden sind außer dem libdfnrpc.a noch folgende Fortran-Libraries anzufügen:

  Bei sun: /usr/lang/SC0.0/libF77.a /usr/lang/SC0.0/libm.a
  und ab SunOS_52: /opt/SUNWspro/SC2.0.1/libM77.a -lF77 -lm
  Bei apollo: -lF77 -lI77 -lU77
  Bei cray: -lf -lfi
  Bei sg: -lF77 -lU77 -lI77 -lisam -lm und ab IRIX_5.1: -lftn -lm
  Bei dec, ultrix: -lfor -lots -lUfor -li -lutil
  Bei dec, OSF-1: -lfor -lots -lUfor -lFutil
  Bei convex: -lF77 -lI77 -lU77 -lmathC2
  Bei ibm: -lxlf90 -lxlf -lm -lc
  Bei intel: -lf -lm -lic
  Bei hp: -lcl -lU77
  Bei f2c: $(FCPATH)/../lib/libf2cF77.a $(FCPATH)/../lib/libf2cI77.a

- Bei den Plattformen Vax und Alliant wird die C-Programmierung nicht mittels Beispielen unterstützt. Sie ist trotzdem möglich. Bei Fujitsu werden die C-Beispiele erst im nächsten Release enthalten sein.

Die hier beschriebenen Methoden wurden in den Beispielen *csum*, S. 27 und *crshexa*, S. 170 angewandt.

9 Beispielanwendungen

Bestandteil der Auslieferung sind mehrere Beispielanwendungen des DFNRPC-Tools. Die Anwendungen **sum** und **csum** sollen als einfaches und verständliches Beispiel Anfängern die Einarbeitung und die Entwicklung einer ersten eigenen verteilten Anwendung erleichtern. Sie sind in Kapitel 3 auf Seite 19 beschrieben. In diesem Kapitel werden weitere mitgelieferte Beispiele beschrieben.

ben — Die Beispielanwendung ben ist ein Benchmark-Programm für RPCs. Damit kann ermittelt werden, welche Übertragungsleistung nach Durchlaufen sämtlicher Netzwerkprotokolle und Konvertierungen für die verteilte Anwendung tatsächlich noch zur Verfügung steht. Dabei können als Parameter verschiedene RPC- und Datentypen sowie Puffergrößen, Datenmengen und Aufrufhäufigkeiten verstellt und getestet werden. Dadurch kann auch die Verzögerungszeit von RPC-Aufrufen selbst ermittelt werden.

sum — Beim Beispiel **sum** ist in der Einführung auf S. 19 detailliert beschrieben. Es enthält einen einfachen **Fortran**-Rumpf für verteilte Anwendungen und einen synchronen RPC.

csum — Wie sum, doch bei **csum** ist das Client- und das Server-Programm in **C** geschrieben.

para — Das Beispiel **para** soll die Möglichkeiten von parallelen RPCs und Datenpipes veranschaulichen.

Das Demonstrationsprogramm para zeigt beispielhaft die Verwendung von parallelen RPCs und Datenpipes. In einem einfachen Szenarium werden verschiedene Einsatzmöglichkeiten dieser Mechanismen demonstriert. Da bei dieser Anwendung auch eine Server-Server-Pipe verwendet wird, werden zwei Server benötigt. Daher dient dieses Beispiel auch dazu aufzuzeigen, wie ein Client mehrere Server nutzen kann.

rshexa — Im Beispiel **rshexa** wurden die Aufrufe einer Routine auf mehrere Server verteilt. Hierbei wurde eine dynamische Lastbalance realisiert. Die Server werden mittels RPQUCI und RPQUSP gestartet.

crshexa — Das Beispiel **crshexa** ist in **C** programmiert und ist äquivalent zu rshexa.

rpcpipe — Das Beispiel **rpcpipe** zeigt die abwechselnde Nutzung einer Verbindung für ausschließlich RPCs und ausschließlich als Datenpipe. Prozeßstart und Verbindungsaufbau werden mit RPQUCI/F und RPQUSI/F durchgeführt.

con — Das Beispiel **con** zeigt die gleiche Anwendung wie rpcpipe, aber der Verbindungsaufbau wird mit RPOPOP, RPOCON und RPOACC realisiert.

pipe — Das Beispiel **pipe** zeigt, wie man Daten zwischen zwei Objekten mittels DFN-RPC Datenpipes austauschen kann, ohne daß RPCs gemacht werden.

moni — Mit dem Beispiel **moni** kann man die Funktion des Monitors ausprobieren.

sscon — Beispiel zum Aufbau eines Server-Server **Interconnection Networks**.

awi — Beispiel-Skeleton des Alfred-Wegener-Instituts zum DFN-RPC.

rfa — Testbeispiel und Installationsumgebung für die Remote-File-Access-Library.

multicl — Im Beispiel multicl ist die Anbindung **mehrerer Clients** an einen Server realisiert.

exa — Diese Directory enthält kein lauffähiges Programm, sondern nur die Beispiele aus Kap. 5, S. 53.

rw — Das Remote-Write-Beispiel ua1 aus dem Handbuch zu Rel. 1.0, aber mit Quickstart.

ua1a, ua1b, ua1c, ua1d – obsolete Beipiele aus Rel. 1.0 mit alten Prozeßstart-Routinen (interaktiv, mit NQS, mit rsh und mit IPSM).

cav, phi, ph1, ph2 – obsolete PHIGS-Beipiele aus Rel. 1.0.

test — Das Testprogramm test wurde intern zur Überprüfung des Stubgenerators entwickelt, kann aber auch dem Anwender von Nutzen sein, um die Lauffähigkeit seiner Installation auf seiner speziellen Hardware-Plattform zu testen. Es benutzt eine Vielfalt von Datentypen.

cvttest — Testprogramm für die Konvertierungsroutinen RPV___.

Einige Beispielanwendungen (s. Tab.) existieren auch als unverteilte Versionen. Dadurch wird gezeigt, daß sich die jeweiligen Quellen nicht unterscheiden müssen. Die lokale, unverteilte Version eignet sich auch zur vom RPC unabängigen Fehlersuche in den Anwendungsmodulen. Für die anderen Beispiele kann es keine lokale Version geben, da zur Nutzung von parallelen Calls bzw. Datenpipes zwei Prozesse benötigt werden.

Funktionalität	awi	ben	con	crshexa	csum	cvttest	exa	moni	multicl	para	pipe	rfa	rpcpipe	rshexa	rw	sscon	sum	test	ual...
synchroner RPC	o	o			o		+	o		o		o			o		o	o	o
asynchroner RPC		o	o					o	o	o			o					o	
asynchron gepufferter RPC		o					o		o									o	
paralleler RPC				+				o		+				+					
Datenpipes (message passing)			o					o		o	+		+			o		o	
Message Passing ohne RPC											+					o			
Anwendungsspezifische D.Pipes													+						
Multi-Client / Single-Server									+										
Single-Client / Multi-Server				+				o		o				+		+		o	
Quickstart-Prozeßstart	o	o		+	o			o		o	o		o	+	o	o	o	o	
Prozeßstart mit RPO1/2/3												o						o	+
Server-Server Intercon. Network																+			
Verbindundungsaufbau mit																			
RPOCON, RPOPOP und RPOACC			+						o									o	
Server-Server-Conn. mit RPOC2S								o		+								o	
Monitor und User-Einträge								+											
Automatische Last-Balance				+										+					
Benchmark		+																	
Test der Funktionalitäten								o	o							o		+	
Remote File Access Library																			
und Test-Programm												+							
Single-Source im RPI-File	o	o	o				o	o	o	o		o	o	o	o		o	o	o
getrennte Sourcen				o	o						+					o			
Auch unverteilt lauffähig		o			o										o		o	o	o
Rumpf für eigenes Programm	o				+										o		+		
FORTRAN	o	o	o			o	o	o	o	o	o	o	o	o	o	o	o	o	o
C mit Preprozessor m4				+															
C mit Preprozessor cpp					+														

Legende o = Funktionalität wird in diesem Beispiel eingesetzt
+ = Funktionalität wird in diesem Beispiel ausführlich beschrieben

9.1 Die Benchmark-Anwendung ben

Die Benchmark-Anwendung gestattet in ihrer Initialisierungsphase eine Auswahl von Transport Service, Netzwerktyp und Transport Software, doch normalerweise werden nun beide Programmteile ben_cb und ben_sb des Benchmark-Programmes ben analog dem Beispiel sum interaktiv miteinander verbunden:

Dialog beim Client:

```
ben_cb
 Do you want to use the QUICK start routines (y/n  Default: y)

 dfn-rpc - rpquci - quick client initialization
 rpquci - please start server no.   1
   master's address=myhost.myorg.mycountry:1248:0YRfcGmqC81hv77=7eJy:
 rpquci - connected with server   1
 dfn-rpc - rpquci - client connected to           1 servers
```

Die *master's address* ist beim Server einzugeben:

```
ben_sb
 Do you want to use the QUICK start routines (y/n  Default: y)

 dfn-rpc - rpqusp - quick server program
 dfn-rpc - rpqusi - quick server initialization
 rpqusi - please type in the master's address:
myhost.myorg.mycountry:1248:0YRfcGmqC81hv77=7eJy:
 dfn-rpc - rpqusi - server connected to its client
 dfn-rpc - rpquci - ready to serve remote procedure calls
```

Die Anwendung meldet sich dann beim Client mit folgendem Menü:

```
 1 = SEND INTEGER ARRAY SYNCHRON
 2 = SEND REAL ARRAY SYNCHRON
 3 = SEND CHARACTER SYNCHRON
 4 = RECEIVE INTEGER ARRAY SYNCHRON
 5 = RECEIVE REAL ARRAY SYNCHRON
 6 = RECEIVE CHARACTER SYNCHRON
 7 = SEND INTEGER ARRAY ASYNCHRON BUFFERED
 8 = SEND REAL ARRAY ASYNCHRON BUFFERED
 9 = SEND CHARACTER ASYNCHRON BUFFERED
10 = SEND INTEGER ARRAY ASYNCHRON UNBUFFERED
11 = SEND REAL ARRAY ASYNCHRON UNBUFFERED
12 = SEND CHARACTER ASYNCHRON UNBUFFERED
13 = GET SEND+RECEIVE BUFFER LENGTH
14 = SET SEND+RECEIVE BUFFER LENGTH
 0 = END PROGRAM
```

Durch geeignete Auswahl können die Testparameter Pufferlänge, Übertragungsrichtung, Grunddatentyp und Art der Remote Procedure Calls eingestellt werden.

Danach wird die zu übertragende Datenmenge pro Aufruf spezifiziert:

```
Number of kBytes to be transferred on each call ?
(maximum of 64 for Integer
 maximum of 64 for Real
 maximum of  7 for Character)
```

Nach einer bestimmbaren Anzahl von Aufrufen meldet sich der Client mit einem abwechselnd großen beziehungsweise kleinen C sowie der Server mit einem S:

```
Number of calls between two messages on display ?
```

Danach wird die gewünschte Anzahl dieser Zyklen eingegeben:

```
Total number of cycles before reporting the result ?
```

Die so spezifizierten RPCs werden nun ausgeführt. Die daraus resultierende Aufruf- und Übertragungsleistung wird ausgedruckt und das Programm kehrt zum Anfangsmenü zurück.

Die Geschwindigkeit wird auf Sekundengenauigkeit berechnet. Daher sind Messungen mit sowohl sehr wenigen Aufrufen als auch kleinen Datenfeldern nicht empfohlen. Durch Herabsetzen der zu übertragenden Datenmengen auf 0 kBytes kann die Verzögerungszeit, die durch den RPC-Aufruf selbst entsteht, gemessen werden.

Das Experimentieren mit jeweiligen Netzen und Netzprotokollen bleibt hier dem Benutzer überlassen.

9.2 Das Beispiel para

- von Werner Kollak

Ein wesentlicher Unterschied dieses Programms im Gegensatz zu den anderen Beispielanwendungen besteht darin, daß mehrere Server verwendet werden. Nur so kann eine Server-Server-Pipe verwirklicht werden. Desweiteren werden verschiedene Einsatzmöglichkeiten für parallele RPCs und für Pipes vorgestellt.

9.2.1 Verbindungsaufbau zu den Servern

Der Verbindungsaufbau zwischen dem Client und den einzelnen Servern erfolgt analog zum Beispiel ua1a, nur daß nun zu jedem Server eine Verbindung aufgebaut wird.

Der Benutzer startet das Client Programm para_cb und muß zuerst den lokalen Hostnamen des Clients angeben. Danach muß er den Hostnamen des Server 1 angeben. Es werden nun sämtliche Informationen, die für den Verbindungsaufbau benötigt werden, angezeigt. Der Benutzer startet nun in einem weiteren Fenster den Server para_s1 und gibt die benötigten Informationen ein. Sind alle Eingaben korrekt, kann die Verbindung zwischen Client und Server 1 aufgebaut werden. Zur Bestätigung wird sowohl beim Client als auch beim Server 1 eine entsprechende Meldung ausgegeben.

Nun erwartet der Client die Eingabe des Hostnamens von Server 2. In einem weiteren Fenster wird anschließend der Server para_s2 gestartet und die Informationen über die Verbindung Client - Server 2 eingegeben. Ist der Verbindungsaufbau erfolgreich, erfolgt beim Client und beim Server 2 eine entsprechende Meldung. Nach Beendigung der 3 Demos, werden die Verbindungen wieder vom Client abgebaut und das Programm mit einer entsprechenden Meldung beendet.

9.2.2 Demo 1 - paralleler RPC mit einem Server

Dieses Beispiel demonstriert den einfachsten Fall der Nutzung eines parallelen RPCs. Der Client liest einen String vom Terminal ein und überträgt diesen mittels eines parallelen Call zum Server 1. Während dieser einen Rückgabestring einliest, wartet der Client auf die Beendigung des Server Unterprogramms. Hat der Server das Unterprogramm beendet, wird der Rückgabestring beim Client ausgegeben.

9.2.3 Demo 2 - paralleler RPC mit zwei Servern

Eine Weiterführung der Demo 1 stellt dieses Beispiel dar. Hier werden beim Client für jeden Server ein String eingelesen und mit Hilfe von parallelen RPCs auf verschiedenen Servern Unterprogramme aufgerufen. Anschließend wartet der Client, bis einer der Server das gerufene Unterprogramm beendet hat und gibt den Rückgabewert des beendeten Unterprogramms aus. Danach wartet der Client wieder bis auch das andere Unterprogramm auf dem anderen Server beendet ist und gibt dessen Rückgabestring ebenfalls am Bildschirm aus.

Die beiden Unterprogramme, die auf den beiden Servern aufgerufen werden, sind mit dem Server Unterprogramm aus Demo 1 identisch.

9.2.4 Demo 3 - Datenpipes

In diesem Demobeispiel werden Datenpipes zur Übertragung von Daten zwischen verschiedenen Prozessen verwendet. Dabei werden sowohl Client-Server-Pipes zwischen dem Client und den Servern als auch eine Server-Server-Pipe zwischen den beiden Servern benutzt.

Zu Beginn dieser Demo wird beim Client ein String vom Terminal eingelesen und dieser über eine Client-Server-Pipe zum Server 1 übertragen. Beim Server 1 wird der übertragene String ausgegeben und ein weiterer String zur Übertragung eingelesen. Dieser wird über eine Server-Server-Pipe zum Server 2 übertragen und dort am Bildschirm ausgeben. Nun liest Server 2 einen String ein und überträgt diesen zum Client, der diesen String seinerseits ausgibt.

9.3 Das Beispiel rpcpipe

- von Mandana Karimian

Anhand dieses Beispiel wird die Möglichkeit zur abwechselnden Nutzung der RPCs und Datenpipes ohne RPCs gezeigt.

Das Beispiel besteht aus dem Client-Program (rpcpipe_cb) und dem Server-Program (rpcpipe_sb), die sich in $DFNRPCEXAMPLES/rpcpipe befinden.

Nach dem Start von rpcpipe_cb muß man die Adresse des Clients beim Server eingeben.

Im ersten Teil des Beispiels werden RPCs gemacht, dann wird mit dem RPOFBD-Aufruf der Block-Dispatcher im Server beendet und die Verbindung zum Datenaustausch mittels Datenpipes benutzt:

- Senden bzw. Empfangen einer Realzahl mit den Datenpipe-Routinen RPDR0S und RPDR0R, die in der DFN-RPC Library bereitgestellt werden.
- Senden eines Integer-, Real- und Double Arrays mittels selbstdefinierter Messagestubs (Interface ird).

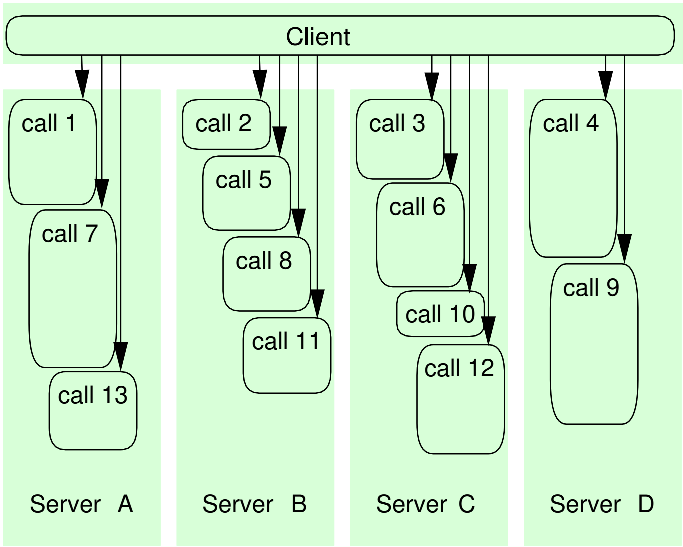


Abb. 9.1: Mehrere Server mit dynamischer Lastbalance

Zum vom Anwender definierten Interface 'ird' hat der Stubgenerator die Sende-Routine irdS und die Empfangsroutine irdR erzeugt.

Nach einem neuen serverseitigen RPBD1-Aufruf, kann der Client wieder RPCs durchführen.

9.4 Das Beispiel con

In diesem Beispiel wird die Client-Server-Verbindung mit den Routinen RPOPOP (Port Open), RPOCON (Connect) und RPOACC (Accept) aufgebaut. Die Anwendung selbst ist identisch mit der des Beispiels rpcpipe.

Client und Server dieser beiden Beispiele können daher gegenseitig vertauscht eingesetzt werden.

9.5 Die Beispiele rshexa und crshexa mit dynamischer Lastbalance

In dieser Beispielanwendung ist eine dynamische Lastbalancierung mittels parallelen RPC's folgendermaßen realisiert: Es wird zunächst eine wählbare Anzahl Server gestartet. Jedem gestarteten Server wird ein erster Task zugewiesen. Die weiteren vorhandenen Tasks werden dann so verteilt, daß jeweils der erste fertige Server nach Entgegennahme des Rückgabewertes der bearbeiteten Routine durch das Client-Hauptprogramm den nächsten Task zugewiesen bekommt.

Der Start der Server kann auf zweierlei Art erfolgen: Wenn beim Start von rshexa_cb im aktuellen Directory ein File namens dfnrpc.servers vorhanden ist, werden die Strings aus diesem File zum Start der Server verwendet. Existiert dieser File nicht, so fordert das Programm zum interaktiven Start der Server auf.

Im File dfnrpc.servers wird das Schlüsselwort DFNRPCADDR jeweils durch die reale Adresse des Clients ersetzt. Enthält der File das Statement '-norepeat' am Anfang einer beliebigen Zeile, so werden alle im File aufgelisteten Server nur einmal verwendet, andernfalls wird das File solange erneut durchlaufen, bis die im Programm gewählte bzw. die maximale Anzahl Server gestartet ist.

Beispiel:

```
# This is an example for the file dfnrpc.servers
# - - - - - - - - - - - - - - - - - - - - - - -
# Servers startet with a remote shell:
rsh servername_1 -n $DFNRPCEXAMPLES/rshexa/rshexa_sb DFNRPCADDR > /dev/null &
# or if the DFN-RPC is installed in /sw
rsh servername_1 -n /client/examples/dfnrpc/examples_ieee/rshexa/rshexa_sb DFNRPCADDR > /dev/null &
rsh servername_2 -n ...../rshexa_sb DFNRPCADDR > /dev/null &
rsh servername_3 -n ...../rshexa_sb DFNRPCADDR > /dev/null &
rsh servername_4 -n ...../rshexa_sb DFNRPCADDR > /dev/null &
# The following server must be started with telnet:
echo Please start anywhere the server with the command rshexa_sb DFNRPCADDR"&
# The following line prevents, that the server list is repeated
# to start the number of servers wanted by the client
-norepeat
```

Beim Einsatz der Remote Shell in einer lokalen DFN-RPC-Installation ist zu beachten, daß bei der Initialisierung der Remote Shell die Environment-Variable $DFNRPCBIN gesetzt wird, damit beim Einsatz des Monitors der Monitor-Agent dfnrpcagt gestartet werden kann, s. a. Kap. 2.10.1 *Unterschiede zw. lokaler und zentraler Installation* auf S. 15. Außerdem ist zu beachten, daß die Umlenkung der Ausgabe von rsh nach /dev/null auch die Ausgabe von Fehlermeldungen verhindern kann. Bei abweichendem Username auf dem entfernten Rechner benötigt man die rsh-Option -l username. Außerdem muß beim Starten von Anwendungsservern mittels Remote-Shell und gleichzeitigem Einsatz des Monitortools gewährleistet sein, daß das File RPCAGMIB im aktuellen Directory aller Anwendungsserver liegt.

Dieses Beispiel ist eine Realisierung der im Beispiel in Abb. 4.9 auf Seite 38 aufgezeigten Programmierung.

9.6 Das Beispiel pipe

- von Mandana Karimian

Dieses Beispiel zeigt, wie man Daten zwischen zwei Objekten mittels DFN-RPC Datenpipes austauschen kann, ohne RPCs zu verwenden.

pipe_cb und pipe_sb sind die beiden Anwendungsprogramme, die miteinander kommunizieren. Sie befinden sich in der Directory $DFNRPCEXAMPLES/pipe.

Zuerst muß zwischen den Objekten eine Transportverbindung etabliert werden. Hierzu werden die Routinen RPO1CS und RPO1SS eingesetzt.

Nach dem Verbindungsaufbau werden mittels Datenpipes Daten vom Client zum Server[1] geschickt und in unserem Beispiel werden diese Daten noch mal zum Client zurückgeschickt, d.h. der Datenaustausch wird in beiden Richtungen durchgeführt. Mit dem RPDC0S- oder RPDC0R-Aufruf kann man einen Character-String senden bzw. empfangen. Weitere Datenpipe-Routinen zum Versenden oder Empfangen anderer Datentypen sind auf Seite 114 beschrieben.

Danach wird die Transportverbindung wieder mit RPO1CF und RPO1SF abgebaut.

[1] Die beiden Objekte werden hier zur einfacheren Unterscheidung als Client und Server bezeichnet, kommunikationstechnisch handelt es sich aber um gleichwertige Peers

9.7 Das Beispiel moni

Zwischen dem Client (moni_cb) und den beiden Serven (moni_s1, moni_s2) wird je eine Verbindung aufgebaut, d.h. wir brauchen für unser Beispiel 3 Fenster. Nach dem Verbindungsaufbau werden 2 Tests durchgeführt:

Test A: Der Client liest einen String für jeden Server ein, schickt diesen mittels parallelen RPCs zum Server 1 bzw. Server 2. Nach Erhalt dieses Strings bei jedem Server wird jeweils ein Rückgabestring eingelesen. Der Rückgabestring wird beim Client ausgegeben, wenn einer der Server das Unterprogramm beendet hat. Der Client wartet bis der andere Server auch fertig ist, und gibt dessen Rückgabestring ebenfalls aus.

Test B: Hier wird zuerst ein String mittels Datenpipe vom Server 1 zum Server 2 geschickt, danach wird eine Client-Server-Pipe zur Übertragung dieses Strings verwendet.

Am Ende werden die Verbindungen abgebaut.

Dieses Beispiel eignet sich besonders zum Test des Monitors. Hierzu ist vorab der Monitor-Server dfnrpcsrv und der eigentliche graphische Monitor dfnrpcwin zu starten. (Zum Pfad dieser Programme siehe Kap. 2.10.1 *Unterschiede zw. lokaler und zentraler Installation* auf S. 15.)

9.8 Das Beispiel sscon

Dieses Beipiel zeigt den Aufbau (und eine „spielerische“ Nutzung) eines Server-Server Interconnection Networks.

sscon_cb und sscon_sb sind die beiden Anwendungsprogramme, die miteinander kommunizieren. Sie befinden sich in der Directory $DFNRPCEXAMPLES/sscon.

sscon_cb ist das vom Anwender zu startende Client-Programm. sscon_sb ist das über die Quickstartroutinen und den File dfnrpc.servers zu startende Server-Programm.

Mit den Quickstartroutinen werden 3 bis 9 Server gestartet. Hierzu wird in der aktuellen Directory ein File namens dfnrpc.servers benötigt (siehe auch Beispiel weiter unten).

Mit den Routinen RPQSSC und RPQSSS wird zwischen den Serverprozessen ein Interconnection Network aufgebaut.

Bei 9 Server ist es folgendermaßen konfiguriert:

```
1--2--3
|  |  |
|  |  |
4--5--6
| /|\ |
|/ | \|
7--8--9
```

Bei weniger Knoten ist es der entsprechende Teilgraph.

In der Beispiel-Nutzung werden Messages entlang dieses Graphen verschickt, und zwar immer zu Knoten mit höherer Nummer.

Jeder Knoten schickt außerdem ankommende Messages zum Client zurück. Dort werden sie auf dem Standard-Output dargestellt.

Beispiel für den File dfnrpc.servers für 6 Server:

```
#
# [host_i], [work_dir] und [bin_location] sind zuerst sinnvoll
# zu ersetzen.
Environment-Variablen der Remote-Shell ben"otigen
# hierbei das Prefix \$ statt $.
# Die Protokollfiles werden in die Directory geschrieben, in der
# cl_prog aufgerufen wird.
#
\rm -f prot_s1 ; rsh [host_1] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s2 ; rsh [host_2] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s2 &
\rm -f prot_s3 ; rsh [host_3] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s3 &
\rm -f prot_s4 ; rsh [host_4] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s4 &
\rm -f prot_s5 ; rsh [host_5] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s5 &
\rm -f prot_s6 ; rsh [host_6] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s6 &
#\rm -f prot_s7 ; rsh [host_7] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s7 &
#\rm -f prot_s8 ; rsh [host_8] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s8 &
#\rm -f prot_s9 ; rsh [host_9] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s9 &
#
# Durch -norepeat wird die Anzahl der Server auf die obenstehenden,
# nicht auskommentierten (hier 6 St"uck) reduziert.
#
-norepeat
```

Wenn man auf einem Rechner mehrere Prozesse mittels rsh starten will, dann sollte man nachfolgende rsh-Aufrufe jeweils um eine Sekunde verzögern, z.B.

```
\rm -f prot_s1; rsh [host_1] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s2; sleep 1; rsh [host_1] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s3; sleep 1; rsh [host_1] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s4; rsh [host_2] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s5; sleep 1; rsh [host_2] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s6; sleep 1; rsh [host_2] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
-norepeat
```

9.9 Das Beispiel awi

Dieses Beispiel ist ein Musterrumpf für beliebige verteilte Anwendungen. Details sind im README-File beschrieben.

9.10 Das Beispiel rfa

Dieses Beispiel enthält ein Musterbeispiel für die Remote-File-Access-Library. Außerdem ist die Installationsumgebung der Library hier enthalten. Details sind im README-File und im File rfa-docu.ps beschrieben.

9.11 Das Beispiel multicl

9.11.1 Bedienung des Beispielprogramms

In diesem Beipiel ist ein Multi-Client-Szenarium realisiert bei dem mehrere Clients mit einem Server verbunden werden können.

Zunächst muß auf dem Server **multicl_sb** gestartet werden. Es wird die Adresse des Servers ausgegeben und gewartet, bis durch einen Client eine Verbindung angefordert wird. Nun kann man die Clients mit **multicl_cb** starten. multicl_cb fragt zunächst nach der Adresse des Servers und versucht dann, die Verbindung zum Server aufzubauen. Der Server wird den Verbindungsaufbau solange akzeptieren, bis die im Serverprogramm festgelegte maximale Anzahl Verbindungen (im Beispiel:4) erreicht ist. Ist diese Maximalanzahl erreicht, können dennoch weitere Clients gestartet werden, der Server wird dann jeweils wenn ein Client beendet wird, die nächste Verbindung akzeptieren. So können auch zur Laufzeit von multicl_sb beliebig Clients hinzugeklinkt und wieder abgeschaltet werden.

Nachdem ein Client Verbindung mit dem Server hat, fordert multicl_cb dazu auf, eine Zahl einzugeben. Geben Sie hier eine beliebige Integerzahl (z) ein. Die nächste Zahl (n), die Sie eingeben sollen, bestimmt, wie oft die Schleife, in der die RPCs gerufen werden, durchlaufen wird. Sie entspricht in etwa der weiteren Laufzeit des Clientprogramms in Sekunden.

Nun werden n asynchrone Pakete von jeweils 5 RPCs abgeschickt, die auf dem Server die eingegebene Zahl z und die Nummer des Clients, von dem das Paket abgeschickt wurde, ausgeben. Nachdem das Clientprogramm durchgelaufen ist, kann mit der Eingabetaste der Client beendet und die Verbindung abgebaut werden. Falls vorher die maximale Anzahl Verbindungen bestanden hat, und Verbindungsanforderungen von Clients vorliegen, wird der Server nun die nächste Verbindung aufbauen. Wenn alle Clients ihre Verbindung zum Server abgebaut haben, fragt das Serverprogramm interaktiv, ob es sich beenden soll.

9.11.2 Das Serverprogramm im Modul "sm"

server_main:

```
      program server_main
      include 'dfnrpc.inc'
      parameter(max_clients=4)
      integer obj1st,objrdy(max_clients),timeout
      integer c_objid(max_clients),port_objid,new_connections
      integer active_clients,fd_port,fdrdy,fd1st
      character*1 yn
      logical finished
      integer num_client
      common /clients/ finished,num_client
      call srv_rpcini(max_clients,c_objid,fd_port,port_objid,iret)
      finished=.false.
      active_clients  = 0
      new_connections = 1
1     continue
        timeout=60000
        call rporry(max_clients,c_objid,timeout,objrdy,obj1st,
     &              new_connections,fd_port,fdrdy,fd1st,iret)
        if(fd1st.eq.1) then
          call connect_a_client(max_clients,port_objid,c_objid,iret)
          active_clients=active_clients+1
          if (active_clients.eq.max_clients) new_connections = 0
        endif
        if(obj1st.gt.0) then
          num_client=obj1st
          call rpouse(c_objid(num_client),iret)
          call rpbd1a(iret)
          if(finished) then
            call disconnect_a_client(port_objid,
     &                       num_client,c_objid(num_client),iret)
            active_clients=active_clients-1
            new_connections=1
```

```
                finished=.false.
              endif
            endif
          if(active_clients.gt.0) goto 1
          write (*,*) 'Do you want to continue [y,n, default=n]'
          read(*,'(A)') yn
          if (yn.eq.'y') goto 1
          call rpcfin(port_objid,iret)
99        continue
          end
```

Im Server Hauptprogramm wird zunächst die RPC-Initialisierungsroutine *srv_rpcini* gerufen. *srv_rpcini* initialisiert die DFN-RPC-Library, öffnet einen Port und gibt dessen Filedescriptor *fd_port* und dessen Object-ID *port_objid* zurück. Das Feld mit den Client-ObjectIDs wird mit -1 initialisiert. In *server_main* wird nun das Flag new_connections, das entscheidet, ob weitere Verbindungen vom Server zugelassen werden, gesetzt. Es folgt die Event-Schleife,in der *rporry* gerufen wird: Diese Routine wartet, bis entweder ein Client aus dem Array *c_objid* einen RPC ruft, oder bis auf dem Filedeskriptor *fd_port* etwas vorliegt. *fd_port* wurde in *srv_rpcini* unmittelbar nach dem Öffnen des Ports mit *rpopop* mittels *rpogsd* ermittelt und stellt den Filedeskriptor des Ports, über den die Verbindungsanforderungen der Clients eintreffen, dar. Wenn auf diesem Port etwas vorliegt und *new_connections* gesetzt ist, setzt *rporry* die Variable *fd1st*=1, woraufhin im Hauptprogramm mittels *connect_a_client* die Verbindung zum Client aufgebaut wird.

Wird von einem Client ein RPC gerufen, so steht nach Aufruf von *rporry* in *obj1st* die Nummer des entsprechenden Clients. Mit *rpouse* wird die Object-ID des Clients zur aktuellen gemacht, und anschließend wird der Blockdispatcher für asynchrone RPC-Pakete *rpbd1a* gerufen.

Das dritte Ereignis, das eintreten kann, ist, daß ein Client beendet wurde. Ein Client muß zum Schluß per RPC die Routine *set_end_flag* rufen. *set_end_flag* setzt die globale Variable *finished* auf .true. . Ist dieses Flag .true. , so wird in der Schleife mittels *disconnect_a_client* die Verbindung serverseitig ebenfalls beendet.

Ist eine neue Verbindung aufgebaut worden, so wird die Variable *active_clients* um 1 erhöht, wurde eine Verbindung geschlossen, wird sie um 1 erniedrigt. Durch Vergleich mit dem Parameter *max_clients*, der angibt, wieviele Verbindungen der Server zuläßt, und mit dem auch die Arrays dimensioniert sind, wird festgestellt, ob *new_connections* gesetzt oder nicht gesetzt wird (bzw. bleibt).

Bestehen nach einem Schleifendurchlauf keine Verbindungen mehr (*active_clients*=0), so wird der Anwender gefragt, ob er das Serverprogramm beenden will. Antwortet er mit "y", so wird *srv_rpcfin* gerufen und anschließend das Serverprogramm beendet, andernfalls wird wieder in die Eventschleife zurückgesprungen, wo mit *rporry* gewartet wird, bis neue Verbindungen angefordert werden.

srv_rpcini

```
      subroutine srv_rpcini(max_clients,c_objid,
     &                      fd_port,port_objid,iret)
       integer iret,iserid,client_id
       integer InCon,OutCon,max_clients
       integer objid,c_objid(max_clients)
       integer fd_port
       character*100 m_address
       integer m_address_lng, m_host_lng
       integer m_port_lng, m_pass_lng,fd
       integer s_address_lng,s_host_lng,s_port_lng
       InCon=max(512/2,max_clients)
```

```
         OutCon=512-InCon
         call rpbini(iret)
         call rpsini(7,0,0,OutCon,InCon,iserid,iret)
         call rpopop(0,port_objid,m_address,m_address_lng,
     &              m_host_lng,m_port_lng,m_pass_lng,iret)
         call rpogsd(fd_port,iret)
         write(*,*) 'master''s address: '//m_address(1:m_address_lng)
         do 1, i=1,max_clients
            c_objid(i) = -1
1        continue
        return
99      continue
        end
```

In *srv_rpcini* findet die RPC-Initialisierung auf der Serverseite statt. Diese ist in diesem Beispiel von der Client-Initialisierungsroutine getrennt, da mit letzterer auch zur Laufzeit des Programms noch Clients hinzugeklinkt werden können sollen, die RPC-Initialisierung jedoch nur einmal, beim Start von multicl_sb ausgeführt wird.

In *srv_rpcini* wird nach dem Aufruf von *rpbini* und *rpsini* mittels *rpopop* der Port für den Verbindungsaufbau geöffnet und mit *rpogsd* der Filedeskriptor dieses Ports ermittelt. Außerdem werden alle Elemente des Arrays mit den Client-ObjectIDs auf -1 (nicht benutzt) gesetzt. Die von *rpopop* zurückgegebene Serveradresse (wird vom Anwender zum Starten der Clients benötigt) wird auf dem Bildschirm ausgegeben. An das Server-Hauptprogramm werden der Filedeskriptor *fd_port* (für *rporry*), die Port-ObjectID (für connect_a_client) und das mit -1 initialisierte Array *c_objid* zurückgegeben.

connect_a_client

```
        subroutine connect_a_client(max_clients,port_objid,c_objid,iret)
         character*100 s_address
         integer max_clients,num_client
         integer port_objid,c_objid(max_clients),iret
         integer s_address_lng,s_host_lng,s_port_lng
         num_client=0
2        continue
         num_client=num_client+1
         if(c_objid(num_client).ne.-1) goto 2
         c_objid(num_client)=port_objid
         call rpouse(c_objid(num_client),iret)
         call rpoacc(c_objid(num_client),s_address,s_address_lng,
     &                 s_host_lng,s_port_lng,iret)
         return
99       continue
        end
```

In *connect_a_client* wird die Verbindung zu einem Client serverseitig aufgebaut. Es wird zunächst im Array *c_objid* (das Feld mit den Object-IDs der Clients) nach einem Element gesucht, das den Wert -1 (unbenutzt) hat. Dieses Element wird nun vor Aufruf von *rpoacc* gleich der übergebenen Port-ObjectID gesetzt und sein Inhalt dann mittels *rpouse* zur aktuellen ObjectID gemacht. Dies geschieht, weil *rpoacc* die Port-ObjectID übergeben werden muß, die Routine aber eine andere ObjectID zurückgibt, wenn über diesen Port mehrere Verbindungen angenommen werden.

disconnect_a_client

```
      subroutine disconnect_a_client(port_objid,num_client,c_objid,iret)
       integer port_objid,num_client,c_objid,iret
       integer restart_flag
       write(*,'(A,I3)') 'finishing client no.',num_client
       call rpouse(c_objid,iret)
       call rpoccl(iret)
       c_objid = -1
       return
99     continue
      end
```

In *disconnect_a_client* wird die Verbindung zu einem Client serverseitig abgebaut. Die übergebene Client-ObjectID wird mit *rpouse* zur aktuellen ObjectID gemacht. Anschließend wird die Verbindung mit *rpoccl* beendet und die entsprechende Client-ObjectID auf -1 zurückgesetzt.

srv_rpcfin

```
      subroutine srv_rpcfin(port_objid,iret)
       integer port_objid,iret
       call rpouse(port_objid,iret)
       call rpopcl(iret)
       call rpsfin(iret)
       call rpbfin(iret)
       return
99     continue
      end
```

In *srv_rpcfin* wird zunächst mit *rpopcl* der Port geschlossen, anschließend mit *rpsfin* der Transportservice und mit *rpbfin* der Block beendet.

rem_appl

```
C*[RPI_BEG] ASYNCHRONOUS BUFFERED
      subroutine rem_appl(number)
       integer number [IN]
C*[RPI_NON]
       integer num_client
       common /clients/ finished,num_client
       write(*,'(A,I3,A,I3)') '--- rpc-output:',number,
     &        '   was called by client no. ',num_client
       return
      end
C*[RPI_END]
```

rem_appl steht für die eigentliche RPC-Routine des Programms. Im Beispiel wird hier lediglich auf der Serverseite die vom Anwender auf der Clientseite eingegebene Zahl und die Nummer des rufenden Clients ausgegeben. Diese Nummer (*num_client*) wird *rem_appl* vom Server über einen Commonblock übergeben.

set_end_flag

```
C*[RPI_BEG] ASYNCHRONOUS
       subroutine set_end_flag(iret)
        integer iret     [OUT]
C*[RPI_NON]
        logical finished
        common /clients/finished,num_client
        finished=.true.
        return
       end
C*[RPI_END]
```

Diese RPC-Routine setzt die lokale Variable *finished* auf 1. Über einen Common-Block wird auch im Server-Hauptprogramm das *finished*–Flag gesetzt.

9.11.3 Das Client-Programm im Modul "cm"

client_main

```
       program client_main
       include 'dfnrpc.inc'
       integer number,loop_end
       call cl_rpcini(iret)
       write(*,*) 'please enter a number to be transferred to'
       write(*,*) 'the server as an example-output [1..500] :'
       read(*,*) number
       write(*,*) 'please enter how many RPCs shall be called'
    &             //'by client [1..200]:'
       read(*,*) loop_end
       do i=1,loop_end
         call rem_appl(number)
         if(mod(i,5).eq.0) then
           call rposyn(iret)
           call rchslp(1)
         endif
       enddo
       write(*,*)' - press return to finish client - '
       read(*,*)
       call cl_rpcfin(iret)
99     continue
       end
```

Im Client-Hauptprogramm werden zunächst die beiden Zahlen für die Ausgabe und für die Anzahl der Schleifendurchläufe eingelesen. Danach wird mit *cl_rpcini* der RPC clientseitig initialisiert und die Verbindung mit dem Server hergestellt.

In der anschließenden RPC-Schleife wird bei jedem Durchlauf einmal die RPC-Routine *rem_appl* gerufen. Jeweils nach 5 RPC-Calls wird mit *rposyn* synchronisiert und damit das Paket abgeschickt. Damit der Ablauf auf Client- und Serverseite verfolgbar ist, wird nach dem Abschicken eines Pakets mit *rchslp(1)* jeweils eine Sekunde gewartet. Nachdem die Schleife beendet wurde, wird der Anwender aufgefordert, das Clientprogramm mit -return- zu beenden. Nach dieser Eingabe wird mit *cl_rpcfin* die Verbindung und der RPC clientseitig beendet.

cl_rpcini

```
        subroutine cl_rpcini(iret)
         include 'dfnrpc.inc'
         character*100 server_addr,c_address
         integer objid,iret,iserid
         integer c_addr_lng,c_host,lng,c_port_lng
         common /objects/objid
         call rpbini(iret)
         call rpsini(7,0,0,1,1,iserid,iret)
         write(*,*)'please type in the server''s address: '
         read(*,'(A)') server_addr
         call rpocon(0,server_addr,objid,c_address,
     &        c_addr_lng,c_host_lng,c_port_lng,iret)
         return
99       continue
         stop
        end
```

In *cl_rpcini* wird zunächst die Adresse des Servers eingelesen. Nach Aufruf von von *rpbini* und *rpsini* wird mit *rpocon* versucht, die Verbindung mit dem Server aufzunehmen. Durch den Aufruf von *rpocon* wird auf Serverseite von *rporry* die Variable *fd1st* auf 1 gesetzt, falls bzw. sobald *new_connections* ebenfalls 1 ist. *rpocon* wartet nun, bis der Server die *connect_a_client*-Routine ruft und damit die Verbindung akzeptiert.

cl_rpcfin

```
       subroutine cl_rpcfin(iret)
         integer iret
         include 'dfnrpc.inc'
         integer restart_flag
         integer objid
         common /objects/objid
         call rpouse(objid,iret)
         call set_end_flag(iret)
         call rpo3sf(restart_flag,iret)
         call rpsfin(iret)
         call rpbfin(iret)
       return
99     continue
       stop
       end
```

Hier wird zunächst auf dem Server die RPC-Routine *set_end_flag* gerufen. Anschließend wird mit *rpoccl* die Verbindung clientseitig geschlossen und *rpsfin* und *rpbfin* gerufen.

9.11.4 Die Error-Routine für das Client- und das Serverprogramm im Modul "em"

err_messg ist die Fehlerbehandlungsroutine des Programms. Damit sie vom Client- und vom Serverprogramm genutzt werden kann, steht sie in einem separaten Modul. Die Aufrufe von *err_messg* im Code des Programms wurden wegen der Übersichtlichkeit nicht abgedruckt und sollen daher hier exemplarisch anhand eines Auszuges aus der Subroutine *connect_a_client* dargestellt werden:

```
       ...
       rpouse(c_objid(num_client),iret)
```

```
      if(iret.ne.0) call err_messg('s','connect_a_client','rpouse',iret,*99)
      ...
```

Ausgabe von err_messg:

```
------------------------------------------------------
ERROR OCCURRED IN PROGRAM MULTICL'
SITE:      server
LOCATION:  connect_a_client
ROUTINE :  rpouse
IRET:      19
ERROR MESSAGE : object not opened or invalid object_id
------------------------------------------------------
```

9.11.5 Struktur des Programms

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

C*[BLK_BEG] sb, NAME='server block', SERVER_SITE
C*[MOD_USE]     sm, SERVER
C*[MOD_USE]     em, LOCAL
C*[BLK_END]

C*[MOD_BEG] sm, NAME='server_module'

      program server_main ...
```

siehe Seite 174

```
      subroutine srv_rpcini...
```

siehe Seite 175

```
      subroutine connect_a_client...
```

siehe Seite 176

```
      subroutine disconnect_a_client...
```

siehe Seite 177

```
      subroutine srv_rpcfin...
```

siehe Seite 177

```
C*[RPI_BEG] ASYNCHRONOUS BUFFERED
       subroutine rem_appl
```

siehe Seite 177

```
C*[RPI_END]
C*[RPI_BEG] ASYNCHRONOUS
        subroutine set_end_flag
```

siehe Seite 178

```
C*[RPI_END]
C*[MOD_END]

C*[MOD_BEG] cm, NAME='client_module'
        program client_main
```

siehe Seite 178

```
        subroutine cl_rpcini...
```

siehe Seite 179

```
        subroutine cl_rpcfin...
```

siehe Seite 179

```
C*[MOD_END]


C*[MOD_BEG] em,NAME='error handling module'
        subroutine err_messg...
```

Code nicht abgedruckt

```
C*[MOD_END]
```

9.12 Das Testprogramm test

9.12.1 Wie wird getestet?

Im Testprogramm test werden unterschiedliche Variablensätze vom Client zum Server und wieder zurück übertragen. Auf der Serverseite werden die Berechnungen, die zu den Testzahlen führen, unabhängig von den Berechnungen auf der Clientseite durchgeführt, anschließend werden die Zahlen jeweils im Rahmen der möglichen Genauigkeit des Datentyps miteinander verglichen. Es werden dabei Testroutinen von Datentypen zur Laufzeit unterdrückt, die auf der verwendeten Hardware nicht zur Verfügung stehen, d.h. getestet wird stets die Schnittmenge der auf Client und Server verfügbaren Datentypen.

9.12.2 Was wird getestet?

Das vorliegende Release testet folgende **Standard-Datenübertragungen**:

LOGICAL

LOGICAL*2

LOGICAL*4	
INTEGER	
INTEGER*2	
REAL	Die Zahlen vom Typ REAL werden auf eine Genauigkeit entsprechend der letzten Stelle der Mantisse überprüft, in diesem Fall $2 \cdot 10^{-23} = 1.19 \cdot 10^{-7}$. Alle Abweichungen außerhalb dieses Toleranzbereichs werden als Fehlermeldung ausgegeben. Am Schluß werden die jeweils maximalen Fehler mit der maximal zulässigen Abweichung verglichen.
REAL*8	Hier beträgt die erforderliche Genauigkeit $2 \cdot 10^{-47} = 0.71 \cdot 10^{-14}$.
DOUBLE PRECISION	Der Datentyp DOUBLE PRECISION wird auf dieselbe Genauigkeit überprüft wie REAL*8.
STRING	
INTEGER ARRAY	Hier wird ein eindimensionaler Vektor mit zehn Elementen übertragen.
REAL ARRAY	Analog zu INTEGER ARRAY, nur mit rationalen Zahlen.
REAL ARRAY max	Wie REAL ARRAY, aber mit 10000 Elementen.
REAL*8 ARRAY	
DOUBLE PRECISION ARRAY	
INTEGER mit reduzierter Transferlänge:	Es werden Integer-Zahlen mit einer auf zwei Bytes verkürzten Transferlänge übertragen.
REAL ARRAY mit Angabe des IN- und OUT-Bereichs:	Bei dieser Routine wird ein Real-Vektor mit 40 Elementen übertragen, der aus einem IN-, einem OUT- und zwei DUMMY-Bereichen besteht. Das Testprogramm überprüft dabei zusätzlich, ob die so definierten Bereiche auch nur dort überschrieben werden, wo dies zulässig ist.
INTEGER FUNCTION	Alle Subroutinen sind auch als Funktion deklarierbar, dies ist der erste Prototyp dieser Testklasse.
DUMMY	

Das **asynchrone Übertragungsprotokoll** kann exemplarisch mit REAL und INTEGER getestet werden.

Zusätzlich können folgende **Nicht-Standard-Tests** durchgeführt werden, sofern die Hardware-Plattform dies zuläßt:

BYTE
INTEGER*8
REAL*4
REAL*4 ARRAY
REAL*16
REAL*16 ARRAY

Datentypen aus dieser Liste, die nicht vom Testprogramm aufgerufen werden, sollten in eigenen Anwendungen auch nicht verwendet werden.

In dem Beispiel **test** wurden folgende neuen Tests aufgenommen:
- **par**, parallel rpc,

- **parx**, parallel rpc and X Window,
- **msg1**, messages on datapipes,
- **msg2**, messages on datapipes between 2 servers,
- **2con**, 2nd connection with rpopop/con/acc,
- mit **u1** und **u2** kann man zwischen den beiden Servern wechseln,
- der asynchron-Test **a** wurde für 2 Server erweitert.

9.12.3 Programmaufruf und Menü

Das Testprogramm wird auf Client und Server wie die übrigen Beispiele generiert und gestartet, also durch Angabe der Start_Flags, der Hostnamen und der Master_Tsap. Danach stehen im Menü folgende Auswahlmöglichkeiten zur Verfügung:

```
----------------------------------------------------------------------
DFN-RPC / TEST Ver 0.7b Rev. 17-Sep-92 / Rechenzentrum Uni. Stuttgart
----------------------------------------------------------------------

            [T]   Run all standard tests
            [A]   Run asynchron standard tests
            [N]   Run all nonstandard tests

            [X]   Exit program

Enter your choice [Return = T] :
```

Die Punkte T, A und N beziehen sich auf die im vorangehenden Unterkapitel aufgeführten Testroutinen. Anschließend wird jeweils gefragt, ob nur auf Fehler getestet werden soll, oder die Ausgabe von Kommentaren gewünscht wird. Einem erfolgreichen Einsatz des DFN-RPCs steht nichts mehr im Wege, wenn das Ergebnis des Testlaufs T folgendermaßen aussieht:

```
Report of deviations :                                 accuracy goal :

 Max. relative deviation REAL 4  :   0.0000000000E+00  0.1189999992E-06
 Max. relative deviation REAL 8  :   0.0000000000E+00  0.7099999994E-14
 Max. relative deviation DOUBLE  :   0.0000000000E+00  0.7099999994E-14
```

10 Die Remote-File-Access–Tools (RFA)

10.1 Benutzung des Remote-File-Access-Tools

Unter Verwendung des DFN-RPC Dienstes aus dem Projekt „Verteiltes Modellieren“ ist es möglich, FORTRAN-Programme zu schreiben, in denen Files auf einem entfernten Rechner (remote-host) geschrieben oder gelesen werden können. Dieses ist besonders interessant, wenn beide beteiligten Rechner nicht binär kompatibel sind, da mit dem RFA-Tool auch unformatierte Daten gelesen bzw. geschrieben werden können und vom jeweils anderen Rechner „verstanden“ werden.

Bei einem Remote-File-Access sind zwei Rechner beteiligt. Ein Client, auf dem das Anwendungsprogramm läuft und ein Server, auf dem sich der File befindet. Der Benutzer ist durch das RFA-Tool in der Lage, ein Programm zu schreiben, das auf den File des entfernten Rechners zugreifen kann. Welche Konstrukte er dafür zur Verfügung hat, wird im Folgenden beschrieben.

Die Funktionalität entspricht weitgehend der von write und read in FORTRAN. Zusätzlich zu read und write sind auch Aufrufe zum Öffnen und Schliessen einer Datei notwendig. Außerdem sind spezielle Aufrufe zum Initialisieren und Beenden des RPC-Dienstes erforderlich.

10.2 Funktionsumfang

```
      call  rfaini     (remote_host,username_on_rhost,
     +                  shell_on_rhost,local_host)

      call  r_open     (unit,iostatus,filename,status,access,
     +                  form,reclength)

      call  r_writer   (unit,formatstr,recno,iostatus,
     +                  arraylength,real_array)

      call  r_writei   (unit,formatstr,recno,iostatus,
     +                  arraylength,int_array)

      call  r_readr    (unit,formatstr,recno,iostatus,
     +                  arraylength,real_array)

      call  r_readi    (unit,formatstr,recno,iostatus,
     +                  arraylength,int_array)

      call  r_close    (unit,iostatus,status)

      call  r_inquiref (name,iostatus,exist,opened,number,
     +                  access,form,reclength,recno)

      call  r_inquireu (unit,iostatus,exist,opened,named,
     +                  filename,access,form,reclength,recno)

      call  rfafin
```

Bevor die Parameter im einzelnen erläutert werden, einige allgemeine Aussagen zum RFA-Tool. Generell gilt – **es müssen alle Parameter angegeben werden** und **die Reihenfolge ist**

signifikant! Entweder muß ein Wert oder ein Dummy (Leerzeichen) verwendet werden. Die Fortran-Notation mit Gleichsetzung (z.B. file='filename') ist nicht möglich, stattdessen ist nur der Wert (z.B. 'filename') anzugeben.

Call	Parameter	Type	Bemerkungen
rfaini			Muß vor allen anderen remote-operations aufgerufen werden.
	Remote_Host	char	Der Name des Rechners, auf dem ein File gelesen oder geschrieben werden soll. Diese Angabe muß erfolgen.
	Username_on_Rhost	char	Eigener Username auf dem remote_host. Diese Angabe ist nur notwendig, wenn man auf den beiden beteiligten Rechnern unterschiedliche Usernamen hat. Wenn diese Angabe entfallen kann, ist ein Leerzeichen anzugeben.
	shell_on_rhost	char	loginshell des remote_host (**z. Zt. ist nur csh möglich!**). Statt csh genügt ein Leerzeichen.
	local_host	char	Der Name des Rechners, auf dem der Client läuft, so wie er dem remote_host bekannt ist. Diese Angabe ist nur erforderlich, wenn der remote_host den local_host unter dessen lokalem Namen nicht kennt. Andernfalls muß ein Leerzeichen verwendet werden. Wird ein Name angegeben, dann muß es ein official-hostname sein und darf nicht die Internet-Adresse sein!
r_open			Muß vor r_writer etc. gerufen werden.
	unit	int	(remote-)Fortran-unit.
	iostatus	int	Output-**Variable** zur Fehlerbehandlung. iostatus ist 0, wenn kein Fehler aufgetreten ist.
	filename	char	Angabe des Filenamens als **absoluter Pfad**. Wenn der Status 'scratch' ist, kann stattdessen ein Leerzeichen angegeben werden.
	status	char	'new', 'old', 'unknown' oder 'scratch'
	access	char	'direct' oder 'sequential'. Statt 'sequential' kann ein Leerzeichen angegeben werden.
	form	char	'formatted' oder 'unformatted'
	reclength	int	recordlength. Anzahl der Bytes, die für **ein** Array-Element verwendet werden. Ist access 'sequential', kann ein beliebiger Wert angegeben werden.
r_writei			Schreibt ein integer-array.
r_writer			Schreibt ein real-array.
r_readi			Liest ein integer-array.
r_readr			Liest ein real-array.
	unit	int	(remote-)Fortran-unit.
	formatstr	char	Formatangabe - **muß ein Leerzeichen bei**

		'unformatted' sein.
recno	int	Recordnummer. Nummer des ersten Datensatzes, ab dem bei direktem Zugriff gelesen/geschrieben wird. Es werden die Datensätze von recno bis (recno + arraylength - 1) bearbeitet. **Bei sequenziellem Zugriff muß 0 angegeben werden.**
iostatus	int	Output-Variable zur Fehlerbehandlung. iostatus ist 0, wenn kein Fehler aufgetreten ist.
arraylength	int	Anzahl der Datensätze, die geschrieben/gelesen werden sollen. Maxima siehe Datenfeld.
...-array	∗)	Datenfeld (**Variable**) ∗) bei r_writei und r_readi integer∗4 mit maximal 25000 Elementen; ∗) bei r_writer und r_readr real∗8 mit maximal 12500 Elementen.

---

r_close		Schließt mit r_open geöffneten File.
unit	int	(remote-)Fortran-unit.
iostatus	int	Output-**Variable** zur Fehlerbehandlung. iostatus ist 0, wenn kein Fehler aufgetreten ist.
status	char	'keep' oder 'delete'

---

r_inquiref		„Inquire by File“
name	char	Angabe des Filenamens als **absoluter Pfad.**
iostatus	int	Output-**Variable** zur Fehlerbehandlung. iostatus ist 0, wenn kein Fehler aufgetreten ist.
exist	logic	Output-**Variable**. TRUE wenn File existiert, sonst FALSE.
opened	logic	Output-**Variable**. TRUE wenn File zugeordnet, sonst FALSE. Achtung: liefert möglicherweise fehlerhafte Werte, wenn der File nicht existiert.
number	int	Output-**Variable** (remote-)Fortran-unit, wenn opened, sonst 0. Achtung: liefert möglicherweise fehlerhafte Werte, wenn der File nicht geöffnet ist.
access	char	Output-**Variable**. 'SEQUENTIAL', 'DIRECT' oder 'UNKNOWN'.
form	char	Output-**Variable**. 'FORMATTED', 'UNFORMATTED' oder 'UNKNOWN'.
reclength	int	Output-**Variable**. Record-Länge bei direktem Zugriff, sonst (meist) 0.
recno	int	Output-**Variable**. Nummer des nächsten Records, bei direktem Zugriff, sonst 0.

---

r_inquireu		„Inquire by Unit“
unit	int	(remote-)Fortran-unit
iostatus	int	Output-**Variable** zur Fehlerbehandlung. iostatus ist 0, wenn kein Fehler aufgetreten ist.
exist	logic	Output-**Variable**. TRUE wenn File existiert, sonst FALSE.
opened	logic	Output-**Variable**. TRUE wenn File zugeordnet, sonst FALSE. Achtung: liefert möglicherweise

		fehlerhafte Werte, wenn der File nicht existiert.
named	logic	Output-**Variable**. TRUE wenn File benannt, sonst FALSE.
filename	char	Output-**Variable**. Filename oder 'UNKNOWN' bzw. ' '.
access	char	Output-**Variable**. 'SEQUENTIAL', 'DIRECT' oder 'UNKNOWN'.
form	char	Output-**Variable**. 'FORMATTED', 'UNFORMATTED' oder 'UNKNOWN'.
reclength	int	Output-**Variable**. Record-Länge bei direktem Zugriff, sonst (meist) 0.
recno	int	Output-**Variable**. Nummer des nächsten Records, bei direktem Zugriff, sonst 0.

rfafin	Muß vor Beenden des Anwendungsprogramms den Prozeß auf dem remote_host beenden (letzter RPC-Call). **AUF KEINEN FALL VERGESSEN!**

10.3 Übersetzen und Binden

Um ein Anwendungsprogramm mit RPC-Calls erzeugen zu können, muß die Environment-Variable `RFALIB` gesetzt werden, z.B. mit der csh:

```
setenv RFALIB $DFNRPCLIB/librfa.a
```

Das Übersetzen des Programms erfolgt dann z.B. auf einer Sun mit

```
f77 -o application application.f $RFALIB
```

`$RFALIB` ist die Library für das RFA-Tool.

10.4 Ausführen

Der Benutzer braucht nur den Client-Part zu starten; das Ansprechen des Servers geschieht automatisch. Im .cshrc für jeden Rechner, der als remote_host beim RFA-Tool fungiert, benötigt man das allgemeine DFN-RPC setup.

Zusätzlich müssen die Environment-Variablen `RSH` und `DFNRPCHOST` auf dem Client gesetzt werden, z.B. mit der csh:

```
setenv RSH /usr/ucb/rsh
setenv DFNRPCHOST awi9
```

Außerdem sollte noch ein File .rhosts im Homedirectory auf dem remote_host existieren, da der DFN-RPC mit rsh (remote-shell) arbeitet. Dieser File muß für jeden Rechner (local_host), von dem aus man eine DFN-RPC-Anwendung auf dem remote_host ausführen möchte, eine Zeile der folgenden Form beinhalten:

```
local_host username_on_local_host
```

(Wenn .rhosts nicht existiert, muß bei jedem Programmaufruf mit RPC-Calls ein manueller Login erfolgen.)

Anmerkung: Fehler bei **r_open**, **r_write**∗, **r_read**∗, **r_inquire**∗ und **r_close** haben keinen Programmabbruch zur Folge. Daher empfiehlt es sich, jedes Mal den Status zu überprüfen.

10.5 Beispiel

```
      program myapplication

C     Variablen etc.

      integer*4 int_arr(50), ios

C     maybe some statements ...

      call rfaini('awi1','myname',' ',' ')
      call r_open(10,ios,'/pf/tph/myhome/myfile',
     +            'old','direct','unformatted',4)

C     Es wird eine Integer-Datei fuer direkten Zugriff geoeffnet.
C     Je 4-Bytes sind ein Datensatz. Es werden die Datensaetze 100
C     bis 150 in eine bereits existierende Datei geschrieben.

      IF (ios .EQ. 0) THEN
         call r_writei(10,' ',100,ios,50,int_arr)
         call r_close(10,ios,'keep')
      ELSE
         write (*,*) 'Fehler bei r_open ... '

C     Aufgetretene Fehler fuehren nicht zu einem Programmabbruch.

      END IF

      call rfafin

C     maybe some statements ...

      stop
      end
```

Roger Rohde, Chris Dodge Stand 4.8.94

11 Das Monitor-Tool

Mit dem Monitor-Tool steht ein komfortables und leicht bedienbares Diagnosewerkzeug für verteilte Anwendungen zur Verfügung, das bei Laufzeit Informationen und Zustände sowohl auf Anwendungsebene, auf der Ebene der RPCs [20] als auch auf der Ebene der netzwerkweiten Interprozeßkommunikation veranschaulicht [?]. Es kann sowohl in der Entwicklungsphase als auch beim Einsatz der verteilten Anwendung Aufschluß über den erwünschten oder fehlerhaften Programmablauf geben. Insbesondere komplexe Szenarien mit mehreren Servern, die möglicherweise wiederum über Pipes untereinander Daten austauschen, können zuverlässig und übersichtlich überwacht und analysiert werden.

11.1 Die Architektur des Monitor-Tools

Die Architektur des Monitor-Tools ist in Abbildung 11.1 beschrieben.

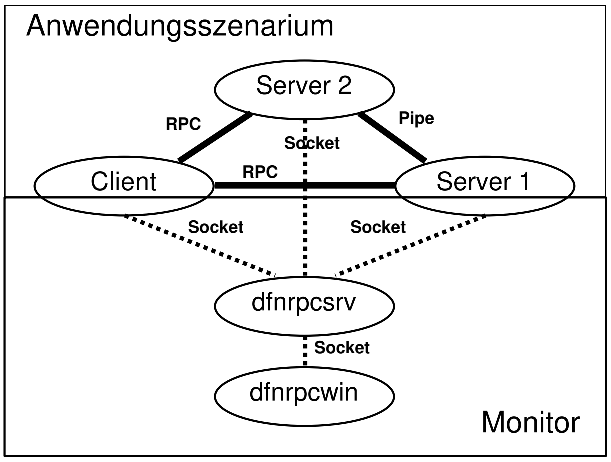


Abb. 11.1: Die Architektur des Monitor-Tools

Jeder an einer verteilten Anwendung beteiligte Prozeß (Client, Server 1, Server 2) übermittelt anfallende Zustandsinformationen und Ereignisse an den Monitor-Server **dfnrpcsrv**, der sie in einer Datenstruktur zwischenspeichert und an den graphischen Monitor **dfnrpcwin** weiterleitet. Dort werden die vorliegenden Informationen graphisch dargestellt. Die Datenverbindungen (in Abbildung 11.1 als gepunktete Linie dargestellt) zwischen den Anwendungsprozessen und Monitor-Server einerseits, sowie zwischen Monitor-Server und graphischem Monitor andererseits, sind als UNIX-Sockets realisiert. Als Transferprotokoll wird dabei TCP verwendet.

11.2 Die Softwarekomponenten und ihre Aufgaben

Abbildung 11.2 zeigt die Softwarekomponenten. Die Aufgaben der einzelnen Komponenten sind dann in den Unterkapiteln 11.2.1 - 11.2.7 beschrieben.

11.2.1 Monitor-Stub MonStb

MonStb beinhaltet die Schnittstellen des Monitor-Tools für die RPC-Stubs und die RPC-Bibliotheksroutinen RPMSUN und RPMDUN. Sie sind konkret DFN-RPC-spezifisch und realisieren den

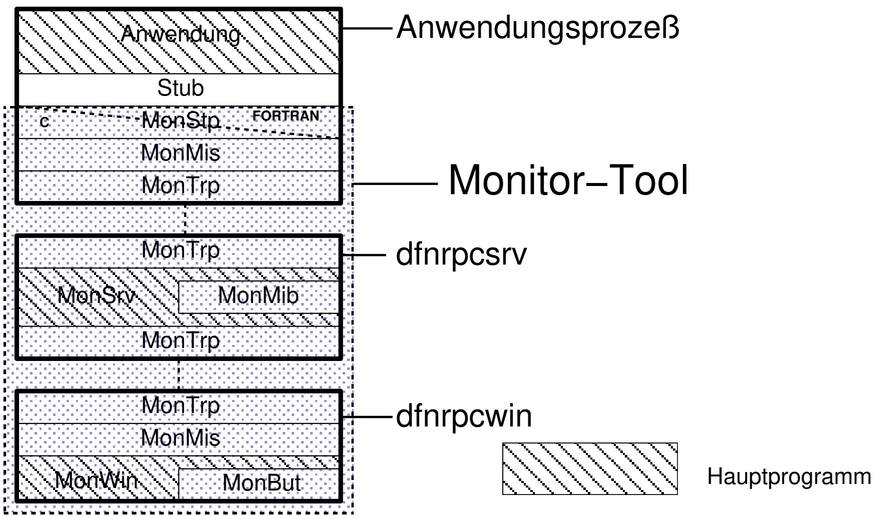


Abb. 11.2: Die Softwarekomponenten des Monitor-Tools

FORTRAN-C-Übergang.

11.2.2 Monitor-Dienste MonMis

MonMis stellt die Dienste und den Zugriff auf die Monitor-Information-Base sowohl für die zu überwachenden Anwendungen als auch für Monitoranwendungen zur Verfügung. Diese Dienste werden mit Hilfe der Transportschicht MonTrp netzwerkweit angeboten.

11.2.3 Monitor-Transport MonTrp

MonTrp unterstützt den netzwerkweiten Transport von Nachrichten der am Anwendungsszenarium beteiligten Prozesse. Als Transferprotokoll wird in MonTrp TCP verwendet.

11.2.4 Monitor MIB-Datenbasis MonMib

Durch MonMib wird die Datenbasis der zu überwachenden Gesamtinformation in Baumstruktur realisiert.

11.2.5 Monitor MIB-Server MonSrv – dfnrpcsrv

MonSrv ist ein Server-Hauptprogramm. Es benutzt MonTrp, um die Anfragen der MonMis-Dienste entgegenzunehmen und an den graphischen Monitor weiterzuleiten und MonMib, um auf die Management Informationen zuzugreifen.

11.2.6 Button-Tool MonBut

MonBut stellt Programmierschnittstellen zur Entwicklung window-orientierter Benutzerschnittstellen zur Verfügung. Dabei werden vereinfachende Annahmen gemacht. Die Benutzerschnittstelle einer Anwendung besteht danach aus einem zusammenhängenden Fenster. In einem oberen

Bereich wird ein hierarchisches Kommandomenü unterstützt. Darunter ist ein scrollfähiger Bereich, in dem die Anwendung an beliebigen Positionen übliche Dialogelemente wie Knöpfe und Texteditierfelder einsetzen sowie Linien zeichnen kann. Zur Realisierung werden Schnittstellen der X Window Bibliothek [22] verwendet.

11.2.7 Window-orientierter Monitor MonWin – dfnrpcwin

MonWin ist eine Monitoranwendung mit Window-orientiertem Dialog, realisiert mit MonBut. Ankommende Management Informationen werden über MonTrp und MonMis eingelesen und in einer Baumstruktur angezeigt. Elemente, die gegenseitig kommunizierenden Teilprozessen einer verteilten Anwendung entsprechen, werden durch verbundene Linien gekennzeichnet.

11.3 Die Benutzung des Monitor-Tools

11.3.1 Das Konfiguration-File RPCAGMIB

Damit die Anwendungsprozesse aktuelle Zustandsänderungen an den Monitor-Server weiterleiten können, muß ihnen mitgeteilt werden, auf welchem Host der Monitor-Server läuft und über welchen Port Daten übertragen werden sollen. Aus diesem Grund werden diese Angaben vom Monitor-Server in das Konfigurations-File RPCAGMIB im aktuellen Directory geschrieben. Falls im aktuellen Directory noch kein Konfigurations-File existiert, wird es vom Monitor-Server erzeugt. Existiert dieses File schon beim Start des Monitor-Servers, wird die darin enthaltene Portnummer verwendet.

RPCAGMIB hat folgenden syntaktischen Aufbau:

```
hostname
portnumber
```

`hostname` : Hostname des Rechners, auf dem der Monitor-Server aktiv ist. Da eine DFN-RPC-Anwendung typischerweise auf mehrere Rechner verteilt läuft, muß auf allen beteiligten Rechnern gewährleistet sein, daß die DFN-RPC Zustandsinformationen an denselben Monitor-Server geschickt werden.

`portnumber` : Portnummer, über die der Monitor-Server auf Daten der Anwendungsprozesse wartet.

Da alle am Anwendungsszenario beteiligten Anwendungsprozesse auf dieses File zugreifen, muß gewährleistet sein, daß dieses File auch auf allen beteiligten Rechnern verfügbar ist. Kann nicht von allen beteiligten Rechnern auf dieses File zugegriffen werden, muß eine aktuelle Kopie im aktuellen Directory vorhanden sein. Dies ist besonders zu beachten, wenn Anwendungsprozesse mittels Remote-Shell gestartet werden sollen. Auch in diesem Fall, muß der entsprechende Anwendungsprozeß auf eine aktuelle Kopie von RPCAGMIB zugreifen können. Ist dies nicht der Fall, werden von diesem Prozeß keine Zustandsänderungen dargestellt.

11.3.2 Starten des Monitor-Server dfnrpcsrv

Um das Monitor-Tool zu benutzen, muß zuerst der Monitor-Server dfnrpcsrv auf einem (beliebigen) Rechner gestartet werden und während der Benutzung des Monitor-Tools immer aktiv sein. Daher sollte dfnrpcsrv in einem separaten Fenster oder als Hintergrund-Prozeß gestartet werden. Nach Beendigung des Monitor-Tools sollte dfnrpcsrv durch Ĉ oder kill abgebrochen werden. Eine genaue Beschreibung von dfnrpcsrv findet sich in Kapitel 11.4.

11.3.3 Starten des graphischen Monitors dfnrpcwin

Sollen die RPC-Zustände angezeigt werden, wird nun MonWin gestartet. MonWin ist ausführlich im Kapitel 11.5 beschrieben.

11.3.4 Starten der Client- und Server-Prozesse

Die Client- und Server-Prozesse des Anwendungsszenarios werden nun wie ohne Monitor-Tool gewohnt gestartet. Anhand des Vorhandenseins des Files RPCAGMIB im aktuellen Directory des jeweiligen Anwendungsprosses wird in MonStp entschieden, ob Zustandsmeldungen über die aktuellen RPC-Zustände an den Monitor-Server geschickt werden.

11.4 Der Monitor-Server dfnrpcsrv

Alle ausführbaren Programme des Monitor-Tools (dfnrpcsrv, dfnrpcwin) befinden sich bei einer lokalen Installation in `$DFNRPCBIN`, bei einer zentralen Installation in `<CONF>/bin` und bei einer zentralen /`sw`-Installation in `/client/bin`, siehe auch Kap. 2.10.1 *Unterschiede zw. lokaler und zentraler Installation* auf S. 15.

```
SYNOPSIS
   dfnrpcsrv [-p mon_port] [-f [filename]] [-o] [-w] [-c]
```

11.4.1 Optionen

-p mon_port: Portnummer, über die der Monitor-Server auf Daten der Anwendungsprozesse wartet. Ist diese Option nicht angegeben, wird die in RPCAGMIB angegebene Portnummer verwendet. Ist weder über die -p Option noch über RPCAGMIB eine bestimmte Portnummer vorgegeben, wird versucht, den Default-Port (Port 2000) zu öffnen. Ist der zu verwendende Port bereits durch eine andere Anwendung belegt, wird eine entsprechende Fehlermeldung des Monitor-Server ausgegeben. Der Monitor-Server muß danach mit einer anderen Portnummer nochmals gestartet werden. Wurde dfnrpcsrv erfolgreich gestartet, können die Anwendungsprozesse bzw. der graphische Monitor gestartet werden.

`-f [filename]`: Alle Datenpakete die von den Anwendungsprozessen zum Monitor-Server und vom Monitor-Server zum graphischen Monitor gehen, werden auf einem File mitprotokolliert. Ist filename angegeben, wird dieser benutzt, ansonsten wird das File proto_dfnrpcsrv in der aktuellen Directory benutzt. Es ist zu beachten, daß um eine maximale Ausgabegeschwindigkeit auf das Protokollfile zu erhalten, die Ausgabe auf das Datenfile gepuffert ist. Dadurch kann es z. B. durch Abbruch von dfnrpcsrv passieren, daß am Ende des Files nicht alle Datenpakete angegeben sind.

`-o` Nach jedem Schreiben auf das Protokoll-File wird das File wieder geschlossen und vor dem nächsten Schreiben wieder geöffnet. Dadurch wird gewährleistet, daß das Protokollfile immer aktuell gehalten wird. Diese Option ist nur in Verbindung mit der -f Option erlaubt.

`-w`: Datenpakete, die beim Monitor-Server von den Anwendungsprozessen eintreffen, sowie Datenpakete, die zum graphischen Monitor geschickt werden, werden am Bildschirm ausgegeben.

`-c`: Die Confirm-Pakete, die vom graphischen Monitor zum Monitor-Server bzw. vom Monitor-Server zu den Anwendungsprozessen geschickt werden, werden zusätzlich am Bildschirm angezeigt bzw. auf File mitprotokolliert. Diese Option ist nur in Zusammenhang mit der -w oder -f Option anwendbar.

11.5 Der graphische Monitor dfnrpcwin

Der graphische Monitor ist in Abbildung11.3 dargestellt

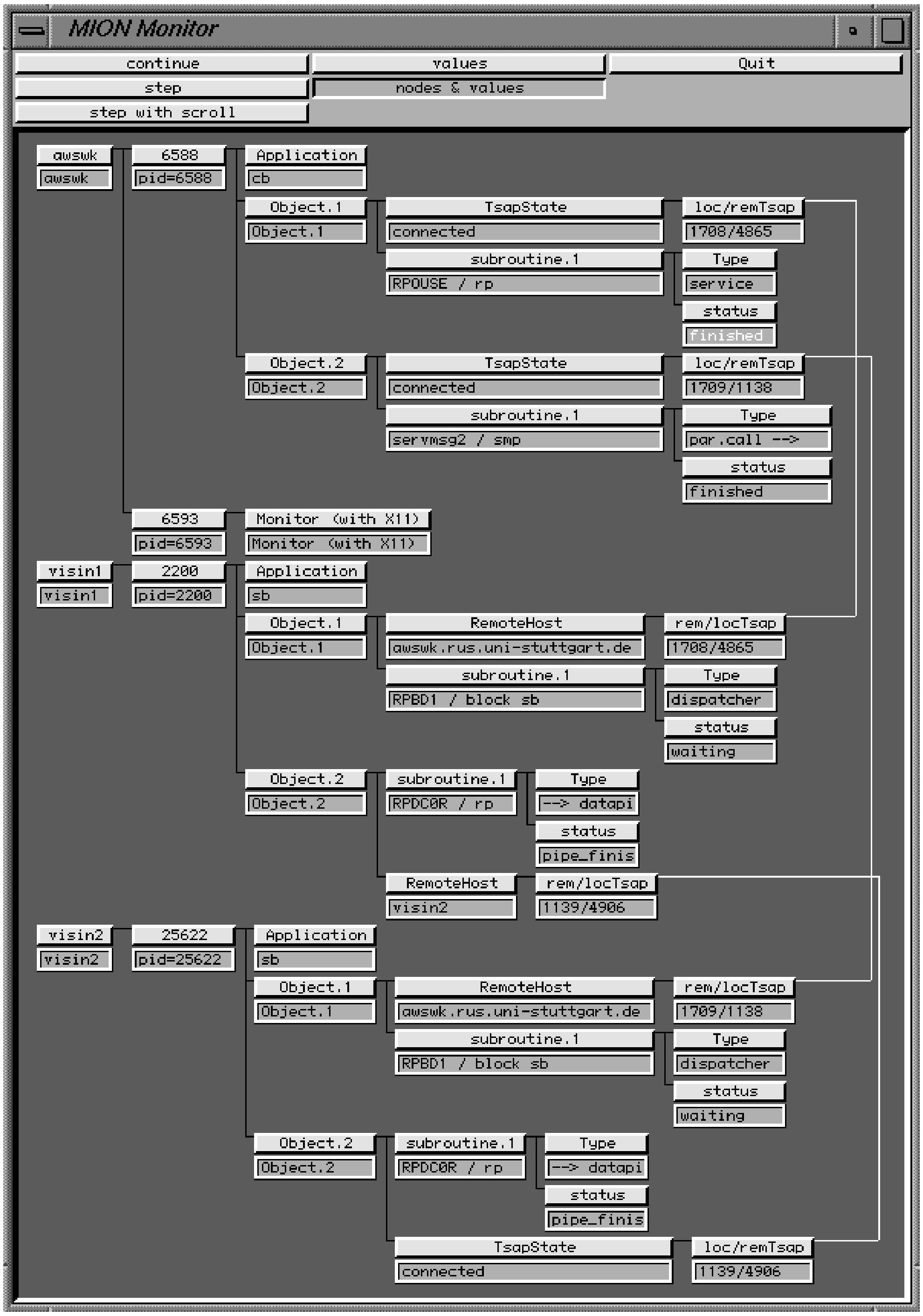


Abb. 11.3: Der graphische Monitor dfnrpcwin

```
SYNOPSIS
   dfnrpcwin [-d display] [-f font] [-h hostname] [-p portnumber]
```

11.5.1 Optionen

`-d display`: Display, auf dem das Fenster geöffnet werden soll. Wird diese Option nicht angegeben, wird das Standarddisplay, das z.B. mit der Variable $DISPLAY gesetzt wurde, geöffnet.

`-f font`: Font, der von MonWin benutzt werden soll, z.B. 8x13. Als Default-Font wird der Font
-schumacher-clean-medium-r-normal-*-8-*-*-*-*-60-*-*
verwendet. Es kann aber jeder beliebige, auf dem Rechner verfügbare X-Window-Font gewählt werden.

`-h hostname`: Hostname des Rechners, auf dem der zugehörige Monitor-Server läuft.

`-p portnumber`: Port, der von dfnrpcsrv zum Datenaustausch verwendet wird.

Ist im aktuellen Directory das Konfigurations-File RPCAGMIB vorhanden und sind keine davon abweichenden Werte mit der -h oder -p Option angegeben, werden die Werte aus RPCAGMIB verwendet. Die mittels der -p oder -h Option angegebenen Werte können auch noch im Anfangsrequester, nachdem dfnrpcwin gestartet wurde, geändert werden.

11.5.2 Anfangsrequester

Nach dem Starten von dfnrpcwin erscheint ein Anfangsrequester, in dem der Host und der Port des Rechners auf dem dfnrpcsrv läuft eintragen werden können. Die vorgegebenenn Werte entsprechen den mit den Optionen -p oder -h angegebenen Werten. Wurden mit diesem Optionen keine Werte vorgegeben, werden die Werte aus RPCAGMIB verwendet. Ist auch RPCAGMIB nicht vorhanden, werden die Default-Werte (-p 2000, -h localhost) vorgegeben. Mit *OK* wird die Eingabe bestätigt. dfnrpcwin versucht nun mit dfnrpcsrv auf dem entsprechenden Rechner eine Verbindung aufzubauen und stellt die möglicherweise schon in der Datenbasis von dfnrpcsrv gespeicherten Zustände der Anwendungsprozesse graphisch als Baumstruktur dar.

11.5.3 Die Bedienung des graphischen Monitors dfnrpcwin

Der graphische Monitor dfnrpcwin verfügt über zwei Modi, vom Monitor-Server ankommende Daten darzustellen:

1. Step-Modus: Zustandsänderungen in den Anwendungsprozessen werden schrittweise, sofern vorliegend, angezeigt, indem einer der beiden Step-Buttons in der ersten Spalte der Menübuttons angeklickt wird. Wird [step with scroll]-Button angeklickt, wird der dargestellte Bildausschnitt so geändert, daß die geänderte Zustandsinformation sichtbar ist. Wird der [Step]-Button angeklickt, bleibt der Bildausschnitt erhalten, auch wenn die geänderte Zustandsinformation nicht sichtbar ist. Im Step-Modus ist ebenfalls in der ersten Spalte der Menübuttons der [continue]-Button vorhanden. Wird dieser angeklickt, erfolgt ein Wechsel in den Run-Modus. Nach dem Programmstart des graphischen Monitors werden alle eventuell schon im Monitor-Server gespeicherten Zustände graphisch dargestellt. Danach ist automatisch der Step-Modus aktiv.
2. Run-Modus: ankommende Zustandsänderungen werden fortlaufend dargestellt. In der ersten Spalte der Menübuttons ist nur der [stop]-Button sichtbar. Wird dieser angeklickt, erfolgt ein Wechsel in den Step-Modus.

In beiden Modi werden die zuletzt geänderten Werte farblich gekennzeichnet. Außerdem ist in beiden Modi gewährleistet, daß das dargestellte Bild immer dem aktuellen Zustand der Anwendungsprozesse entspricht, d.h. die beteiligten Anwendungsprozesse werden solange blockiert, bis eine von ihnen an den Monitor-Server abgeschickte Zustandsänderung auch wirklich im graphischen Monitor dargestellt ist.

Mit den Buttons values und nodes & values in der zweiten Spalte der Menübuttons wird gewählt, ob die Values und die Knoten der Zustände oder nur die Values graphisch dargestellt werden. Mit dem Quit-Button wird der graphische Monitor beendet.

11.6 Schnittstellen zur Benutzeranwendung

Mit den Library-Routinen `RPMSUN` (RPMSetUserNode) und `RPMDUN` (RPMDeleteUsrNode) besteht eine Möglichkeit, direkt von der Client- bzw. Server-Anwendung Teilstrukturen im Monitor zu erzeugen (`RPMSUN`) oder wieder zu löschen (`RPMDUN`). Mit den Unterprogrammaufrufen:

```
CALL RPMSUN('Subroutine','add',iret)
CALL RPMSUN('Subroutine_Language','FORTRAN',iret)
CALL RPMSUN('Subroutine_Source','test.f',iret)
```

wird im Monitor die Unterstruktur in Abbildung 11.4 erzeugt

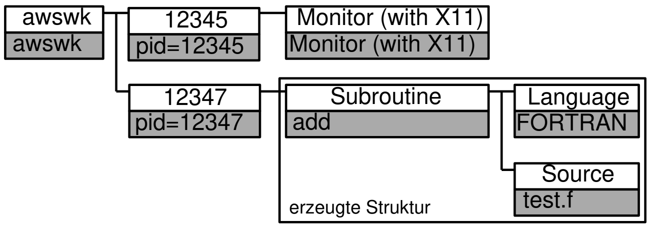


Abb. 11.4: Im Monitor erzeugte Unterstruktur

und mit

```
CALL RPMDUN('Subroutine',iret)
```

wieder gelöscht. Die Unterstruktur wird unmittelbar rechts des PID-Knotens erzeugt. Das bedeutet, daß die Routinen erst benutzt werden können, wenn eine Routine, z.B. über die Quickstartroutinen, erfolgreich initialisiert wurde. Eine genaue Schnittstellenbeschreibung der beiden Routinen befindet sich in den Kapiteln 6.4.5 und 6.4.6 auf Seite 116.

11.7 Änderungen zum vorhergehenden Release (1.0.35)

Aus den gewonnenen Erfahrungen im Umgang mit dem Monitor-Tool haben sich einige Änderungen ergeben:

- Die wichtigste, für Anwender des Monitor-Tools allerdings nicht direkt sichtbare Änderung ist die Umstellung von UDP auf TCP als Transferprotokoll der Kommunikation zwischen den Anwendungsprozessen und Monitor-Server einerseits sowie zwischen Monitor-Server und graphischem Monitor andererseits. Es zeigte sich im Einsatz des Moitor-Tools immer wieder, daß beim Einsatz von UDP als Transferprotokoll Daten zwischen den Prozessen verloren gingen. Dies trat vermehrt auf, wenn die Prozesse über größere Netzwerkstrecken verbunden waren. Durch die Verwendung von TCP als sicherem Transferprotokoll konnte das Problem beseitigt werden. Allerdings ist durch die Änderung der Transferprotokolls das aktuelle Release nicht kompatibel zum vorhergehenden, d. h. daß dfnrpcsrv (Release. 1.0.35) mit einem graphischen Monitor (akt. Release) keine Daten austauschen kann. Außerdem müssen die beteiligten Anwendungsprozesse mit der neuen DFN-RPC-Library neu gebunden werden.

- Eine weitere gravierende Änderung im Gegensatz zum frühreren Release ist der Wegfall der Monitor-Agenten. Im praktischen Einsatz des Monitor-Tools zeigte sich, daß

  1. eine Zwischenpufferung der Zustände in Monitor-Agenten aus Performancegründen nicht notwendig ist,
  2. immer der aktuelle Zustand im graphischen Monitor angezeigt werden sollte,
  3. der Anwender die Möglichkeit haben sollte, in den Ablauf des Anwendungsszenarios einzugreifen.

  Dadurch wurde die Verwendung von Monitor-Agenten, wie im vorhergehenden Release, überflüssig. Somit wurden weitere Probleme, wie die Kommunikation zwischen Anwendungsprozessen und zugehörigen Agenten, hinfällig.

- Die vom graphischen Monitor benötigte CPU-Leistung konnte deutlich gesenkt werden.
- Im graphischen Monitor wurden neue sinnvolle Funktionalitäten implementiert (Step/Run-Modus) und alte überflüssige herausgenommen.
- Die RPC- bzw. Pipe-Verbindungen zwischen zwei Prozessen werden zur besseren Übersichtlichkeit farblich gekennzeichnet.

Durch die genannten Änderungen wurde das Monitortool weiter den Bedürfnissen der Anwender angepaßt und steht nun als Hilfswerkzeug zur Gestaltung und Überwachung von verteilten Anwendungen mit Hilfe des DFN-RPCs zur Verfügung.

11.8 Bekannte Probleme und Bugs

Es bestehen noch Probleme, wenn ein beteiligter Prozeß, Anwendungsprozeß oder dfnrpcsrv/win, aus Gründen wie numerische Fehler abstürzt. In der aktuellen Version werden alle anderen beteiligten Prozesse in diesem Fall automatisch beendet. In Zukunft wird hier ein sinnvolleres Verhalten realisiert werden.

12 Andere Plattformen für verteilte Anwendungen

Wesentliche Plattformen zur Entwicklung verteilter Anwendungen sind

- der RPC des Open Network Computing (ONC, SUN-RPC) [24],
- der RPC der Distributed Computing Environment (DCE) [13],
- und Parallel Virtual Machine (PVM) [4].

Weitere Ansätze zu asynchronen RPCs findet man in

- Cronus [25], das das Konzept der *Futures* einführt.
- Mercury [10], das dieses Konzept um sog. Call-Streams und Promises [9] 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 [23] sind noch weitere Ansätze beschrieben.

12.1 Vergleich mit den Möglichkeiten von DCE

12.1.1 Semantik und asynchrone RPCs

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 Output-Parameter fehlerfrei ablaufen.

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

12.1.2 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 Receive-Routinen, 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. 12.1).

Der Client ruft eine Server-*Receive*-Anwendungsroutine per RPC. Diese Server-Routine 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 eigentliche *push*-Anwendungsroutine ruft, die die Daten verarbeitet.

pull überträgt Daten vom Client zum Server (Abb. 12.2).

Der Client ruft eine Server-*Send*-Anwendungsroutine per RPC. Diese Server-Routine 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-Server-Aufrufen 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.

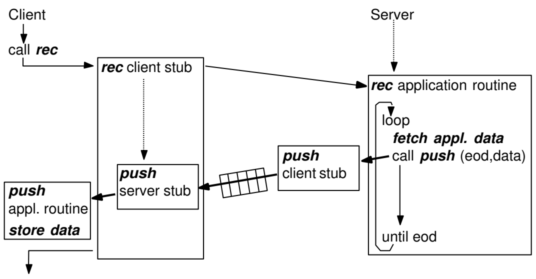


Abb. 12.1: Push

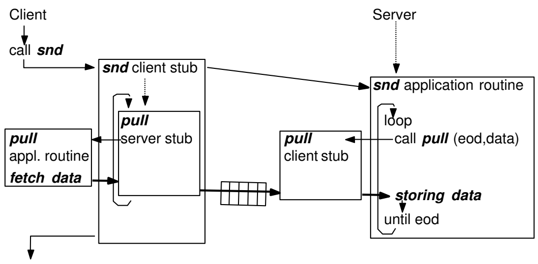


Abb. 12.2: Pull

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, 13, Chap. 17.14.4 Pipes].

12.1.3 Binding und Prozeßstart

Das String-Binding 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 Adreß-Strings mittels eines Directory-Services 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. 4.9) nicht möglich.

12.1.4 Message Passing

DCE besitzt als einzige Grundoperation mit *at-most-once* Semantik den synchronen RPC. [26] 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* Reply gesandt. Außerdem muß serverseitig eine weitere Thread existieren, die den Message Port realisiert, s. Abb. 12.3. 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.

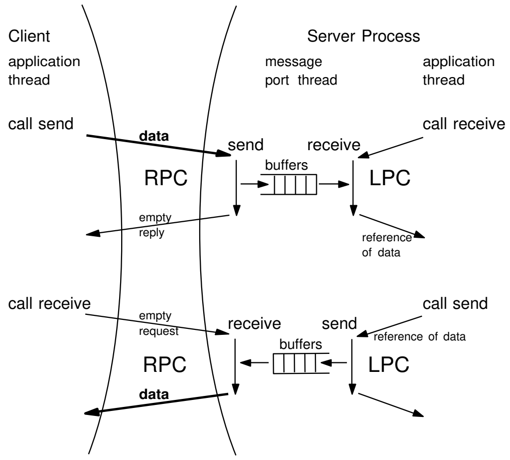


Abb. 12.3: Message Passing mittels DCE RPC

12.1.5 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 ein in C geschriebenes graphisches Userinterface, das via DFN-RPC auf die in Fortran 77, Fortran 90 oder HPF implementitierten Numerik-Codes 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 [8] beschrieben. FIDL ist ein Produkt, das den Einsatz des DCE RPC unter Fortran unterstützt [11].

12.2 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 Übertragungskanal für den Adreß-String aufgebaut wird. Hierzu wird der Standard-Input 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 dagegen eine ca. um den Faktor 2 bessere Performance, s. Kap. 12.4.

12.3 Vergleich mit Futures, Call-Streams und ASTRA

Die Futures von CRONUS [25], der asynchrone RPC von ASTRA [2] und die Call-Streams und Promises im Mercury System [9, 10] 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 End-zu-End-Flußkontrolle blockiert wird. Für einen ggf. nötigen Datenaustausch während eines parallelen RPCs sieht der DFN-RPC Datenpipes vor.

12.4 Benchmark-Vergleich mit DCE, ONC und PVM

In [14, 15] 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/6000-530u.-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. 12.1: 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 erzielt. Dies weist auf mögliche Probleme in der End-zu-End-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 End-zu-End-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. 12.2: 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 blockierendem *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.

12.5 Handhabung des Userinterfaces

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 [1] mit dem DFN-RPC verteilt werden. Für Anwendungen mit mehreren Servern wird der Begriff der *object identification* eingeführt. Mittels einer Library-Routine (rpouse) kann jedes Server-Objekt 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 Client-Programm mittels Library-Routinen untersuchen kann ob einer oder mehrere Server fertig sind, bzw. daß darauf gezielt gewartet werden kann.

[1] d.h. ohne zusätzliche Parameterübergabe über globale Datenbereiche

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.

13 Neue Möglichkeiten im DFN-RPC 1.0.35 und 1.0.60

Der DFN-RPC 1.0.35 enthält gegenüber dem Rel. 1.0 [17] folgende wesentlichen Erweiterungen:

- Parallele RPCs, s. Kap. 4.3 auf Seite 35
- Datenpipes, s. Kap. 4.5 auf Seite 39
- einen zentralen Wartepunkt, siehe Libraryroutinen in Kap. 4.3 auf Seite 35
- Quickstartroutinen, s. Kap. 4.9.1 auf Seite 43
- neue Connect-Routinen, s. Kap. 4.9.3 auf Seite 44
- einen graphischen Monitor, s. Kap. 11 auf Seite 191
- eine Remote-File-Access-Library, s. Kap. 10 auf Seite 185
- einige neue Beispiele, s. Kap. 9 auf Seite 165 (csum, para, rshexa, crshexa, rpcpipe, con, pipe, moni, awi und rfa)
- Erweiterungen zur Installation, s. Kap. 2 auf Seite 3 und
- es können bei einem Array mehrere IN und OUT-Äreas angegeben werden.

Der DFN-RPC 1.0.60 enthält gegenüber dem Rel. 1.0.35 folgende wesentlichen Erweiterungen:

- Routinen zum Aufbau eines frei konfigurierbaren Server-Server Interconnection Networks, s. Kap. 4.9.2 auf Seite 44 und Beispiel sscon auf Seite 172
- Beispiel multicl mit Multi-Client / Single-Server Architektur, s. S. 173,
- Beispiel sum als Einleitung, s. Kap. 3 auf Seite 19,
- der Monitor wurde von UDP auf TCP umgestellt,
- der Monitor dfnrpcwin wurde optimiert, damit sein Bildaufbau nicht mehr flackert, und
- die meisten Beispiele wurden auf RPQU-Routinen umgestellt. Die alten Routinen RPO1..., RPO2... und RPO3... sind obsolet, werden aber noch im nächsten Release in der Library verfügbar sein.

14 Object Handling von Rel. 1.0 - obsolet

14.1 Übersicht

Mit dem Object Handling werden die Blöcke auf entfernten Rechnern gestartet und die entsprechenden Kommunikationsverbindungen aufgebaut, damit der DFN-RPC die nötigen Nachrichten für die Remote Procedure Calls austauschen kann.

Das Object Handling muß von der Anwendung durch Bibliotheksfunktionen aufgerufen werden. Diese Aufrufe werden *nicht* vom Stub-Generator generiert. Vielmehr generiert er für jeden Block einen Satz dieser Subroutinen, da sie auch anwendungsspezifische Deklarationen enthalten, wie z.B. die Länge von Send- und Receive-Buffer.

Es empfiehlt sich, die Aufrufe der Object Handling Routinen in speziellen Object Client-Procedure- und Server-Program-Modules (cp und sp in den Beispielen) zusammenzufassen.

Einführende Beispiele findet man im User's Guide Kapitel 4.8 auf Seite 42.

Ein Block besteht aus allen Modulen, die bei Laufzeit zum selben Prozeß gehören. Blöcke müssen mit RPBINI initialisiert und mit RPBFIN beendet werden.

Nach der Initialisierung muß ein Transport Service mit RPSINI initialisiert und vor RPBFIN mit RPSFIN wieder geschlossen werden. Falls mehrere Server aufgebaut werden sollen, *kann* man RPSINI auch öfters aufrufen. Wenn man hierzu verschiedene Transport Services benutzen will, dann *muß* man RPSINI öfters aufrufen.

Nachdem der Transport Service eröffnet ist, kann der Master seinen Slave mittels der Object-Start-Routinen RPO... starten.

Im folgenden werden die Begriffspaare Client/Server und Master/Slave folgendermaßen eingesetzt:

Client — Der Client enthält das Hauptprogramm der Applikation. Diese ruft mittels RPC Routinen im Server auf. Er enthält typischerweise zwei Unterprogramme, rpcini und rpcfin, welche vom Anwender zu schreiben sind und die Object Handling Routinen aufrufen.

Server — Der Server enthält den Block Dispatcher RPBD1, welcher wartet, bis RPCs ankommen und dann veranlaßt, daß die entsprechende Routine aufgerufen wird.

Master — Der Master sorgt dafür, daß der Slave gestartet wird. Hierzu benutzt er die RPO..I und RPO..S Routinen, denen er einen String übergibt, welcher mittels eines allgemeinen Betriebssystemaufrufs beliebige Dienstprogramme für den Slave-Start einsetzen kann. Die RPO..S Routine wartet, bis der Slave gestartet ist. Der Master liegt typischerweise auf der Workstation.

Slave — Der Slave wird vom Master gestartet. Seine RPO..S Routine synchronisiert sich mit der des Masters.

Beim Objektstart gibt es folgende Fälle und entsprechende Programmierschnittstellen:

```
CALL RPO{Case}{Site}{Phase}
          1     C     I
          2     S     S
          3           F

Case:  1 = Client=Master, Server=Slave,  without restart
```

Danksagung

Wir möchten allen, die zum Gelingen dieses Projekt beitrugen, für ihr Engagement danken. Herrn Prof. Dr.-Ing. R. Rühle danken wir für seine Unterstützung. Herrn Hartmut Aichele danken wir für die prototypische Programmierung der bsd-socket Kommunikation, Herrn Björn Lemke für die Entwicklungsarbeit am Monitortool, Frau Katrin Dieterle und Herrn Werner Dax (IKE) für die Portierung eines Teiles der Software auf VMS, Herrn Tham Do-Dinh für die Programmierung der Konvertierungsmakros, Herrn Matthias Glass für die Entwicklung des IPSM, Herrn Michael Grollmuss für Untersuchungen zum Locking, Herrn Markus Wenzel für die Erstellung des Testbeispiels, Herrn Thomas Wolf für die Entwicklung des Transport-Interfaces. Den Pilotanwendern Frau Heinke Höck (sie entwickelte das Beispiel ua1), Herrn Hans Pfeiffenberger, Herrn Chris Dodge und Herrn Roger Rohde vom Alfred-Wegener-Institut, Bremerhaven, Herrn Ralf Allrutz (Thema Parallelisierung), Herrn Axel Geyer und Herrn Dr. Thomas Ertl von der Theoretischen Astrophysik, Universität Tübingen, Herrn Dr.-Ing. Fritz Schmidt, Herrn Werner Dax und Herrn Johannes Kaltenbach vom IKE und Herrn Prof. Dr.-Ing B.-H. Kröplin und Herrn Michael Trüber und Herrn Stefan Weihe (Thema Common-Blöcke) vom ISD, Universität Stuttgart, danken wir für ihre Kritik und Anregungen und ihre Ausdauer beim ersten Einsatz und Test dieser Software. Außerdem sei Mark Reinhardt und Tim Kirby für ihre Mithilfe bei der Portierung des DFN-RPC auf CRAY-OSI gedankt.

Literatur

[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] 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] Frank Kirschner, Lothar Ehnis: /sw – Das neue Software-Installationskonzept. Benutzerinformation des Rechenzentrums der Universität Stuttgart, Okt. 1993 (BI.10 93), S. 22-30.

[7] Informationen zu /sw im World Wide Web (WWW):
http://www.uni-stuttgart.de/Cis/ZentraleEinrichtungen/Rus/Bi/BI_html/BI93/10/File6.html
und http://www.uni-stuttgart.de/afs/rus/sw/doc/Soft.html/software_homepage.html

[8] Werner Kollak: Distributing FORTRAN Applications using DCE RPC. White Paper. Rechenzentrum Universität Stuttgart, March 5, 1993.

[9] 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.

[10] 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.

[11] 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.

[12] Bruce-Jay Nelson: Remote Procedure Call. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsilvania, May 3, 1981.

[13] 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.

[14] Rolf Rabenseifner, Werner Kollak: DFN-RPC im Vergleich mit DCE, ONC und PVM. DFN-Mitteilungen Nr. 34, März 1994, S. 18f.

[15] 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.

[16] Rolf Rabenseifner, Hans Dieter Reimann: Verteilte Anwendungen mit dem DFN-RPC. DFN Mitteilungen Nr. 31, März 1993, S. 23-25.

[17] Rolf Rabenseifner, Hans Dieter Reimann: Verteilte Anwendungen über Hochgeschwindigkeitsdatenkommunikation, der DFN Remote Procedure Call. Benutzerhandbuch, Rel. 1.0, Rechenzentrum Universität Stuttgart, 23.11.1992.

[18] R. Rabenseifner, H. D. Reimann: Erweitertes Remote Procedure Call Tool für verteilte Anwendungen. Aufgabenbeschreibung zum DFN-Projekt Rechenzentrum der Universität Stuttgart (RUS). 1992.

[19] Rabenseifner, R.: 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.

[20] Rolf Rabenseifner, Hans Dieter Reimann: Konzeption eines Remote Procedure Call Service für verteilte Anwendungen, Meilensteinbericht. Rechenzentrum der Universität Stuttgart (RUS). 1989.

[21] H. D. Reimann: Konzeption eines Monitortools für verteilte Anwendungen. DFN-Projektbericht. Rechenzentrum der Universität Stuttgart (RUS). 1993.

[22] Robert W. Scheifler, MIT Laboratory for Computer Science and Jim Gettys, Digital Equipment Corporation und MIT Project Athena: The X Window System. ACM Transactions on Graphics, Vol.5, No.2, April 1986, Pages 79-109.

[23] Alexander Schill: Remote Procedure Call: Fortgeschrittene Konzepte und Systeme – ein Überblick. Teil 2: Erweiterte RPC-Ansätze. Informatik Spektrum 15, pp. 145-155, Juni 1992.

[24] SUN microsystems: Network Programming Guide. Part Number 800-3850-00, 1990.

[25] 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.

[26] 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.

Sämtliche Informationen und Publikationen zum DFN-RPC sind im WWW per Mosaic oder auch per anonymous ftp zu erreichen unter:

ftp://ftp.rus.uni-stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html

Index