1. DCE Remote Procedure Call

The DCE Remote Procedure Call (RPC) facility is a network protocol used in distributed systems. RPC is modeled after the local procedure call found in most programming languages, but the called procedure is executed in a different process from that of the caller, and is usually executed on another machine. The RPC facility makes the construction of distributed systems easier because developers can focus on the fundamentals of building distributed applications, instead of the underlying communication mechanisms.

Making a remote procedure call involves five different bodies of code:

- the client application
- the client stub
- the RPC runtime library
- the server stub
- the server application

The client and server stubs are created by compiling a description of the remote interface with the DCE Interface Definition Language (IDL) compiler. The client application, the client stub, and one instance of the RPC runtime library all execute in the caller machine; the server application, the server stub, and another instance of the RPC runtime library execute in the called (server) machine.

1.1 Outline of a Remote Procedure Call

When a client application makes a remote procedure call, it actually invokes a local procedure in the client stub. The client stub places a specification of the called procedure and its arguments into one or more packets and asks the RPC runtime library to transmit them to the machine that actually executes the procedure. The process by which a stub converts local application data into network data and packages the network data into packets for transmission is called **marshalling**.

When the RPC runtime library for the server receives these packets, it passes them to the server stub. The server stub extracts the procedure arguments from these packets and makes a local call to the indicated procedure. The process by which a stub disassembles incoming network data and converts it into application data is called **unmarshalling**.

When this local call returns to the server stub, the server stub marshals the data. It places the results (the return code and output parameters) into one or more packets and asks the RPC runtime library to transmit them back to the client.

When the client RPC runtime library receives these packets, it passes them on to the client stub for unmarshalling. The client stub extracts the results and returns them to the client.

In addition to handling all communications between client and server applications, the RPC runtime library provides the following utilities:

- An interface that lets applications access various name servers (which can be used to locate various network resources).
- Management services such as monitoring servers, monitoring runtime operations, and stopping servers.

1.2 Considerations and Dependencies

DCE RPC internally uses a vendor-provided threading facility (POSIX Pthreads). There is wide variation in the completeness and transparency of the various Pthread implementations provided by vendors. The limitations of a given Pthread implementation are inherited by any application that uses DCE RPC, including applications that unknowingly use libraries that internally happen to use DCE RPC.

The DCE RPC runtime has internal threads that need to run in a timely fashion for the runtime to operate correctly. Therefore, the application or Pthreads implementation must neither perform nor permit operations that block the entire process. This restriction is relevant only if you are using a threads implementation other than DCE Threads. Refer to the platform's or vendor's Pthread release notes to determine what limitations the implementation has.

You should also instruct users of any library you develop that uses RPC to refer to the vendor's Pthread release notes. Limitations on the use of threads may include (but are not necessarily limited to): the need for thread-safe libraries; compliance with POSIX; non-process-blocking call behavior; and so on.

1.3 RPC File Locations

The following table lists the locations of libraries and executables built for RPC.

Note: A subcomponent may consist of multiple source files.

TABLE 1. Locations of RPC Subcomponent Files

RPC Subcomponent	Function	Location of Source Files[1]	Location of Installed Files[2]
libnck.a	RPC routines available to applications.	**runtime**	**usr/lib** as part of **libdce.a**
libidl.a	RPC routines for IDL.	**idl/lib**	**usr/lib** as part of **libdce.a**
rpcd	RPC daemon.	**rpcd**	**bin**
idl	IDL compiler.	**idl/idl_compiler**	**bin**
uuidgen	Tool that generates UUIDs for IDL.	**idl/uuidgen**	**bin**

[1] All paths are relative from *dce-root-dir*/**dce/src/rpc**. The path indicates the directory in which the **Makefile** attempts to build the component.

[2] All paths are relative from *dce-root-dir*/**dce/install/***machine_name*/**opt/dce1.1**. The path indicates the directory in which the subcomponent is installed.

2. Porting

OSF^{TM} DCE Version 1.1 contains DCE RPC code ported to the reference platforms listed in Chapter 1 of this guide. As you port RPC to a different platform, you can use this code as a basic structure and basis for comparison. In particular, you will need to consider the information in the following sections.

2.1 Porting the IDL Compiler

The

dce-root-dir/**dce/src/rpc/idl/idl_compiler**

directory contains IDL compiler code ported to the reference platforms listed in Chapter 1 of this guide. If you are porting to a different platform, you may need to modify the following files:

- **sysdep.h**

  An interface definition file can include other interface definition files. To read such an interface definition file, a parser is called recursively. The parser uses global variables to maintain the state of the file being parsed. In order to process an included file, the global variables must be saved, then restored once the included file has been processed.

  sysdep.h defines the **AIX_LEX_YACC**, **APOLLO_LEX_YACC**, **OSF_LEX_YACC**, and **ULTRIX_LEX_YACC** macros. These macros are used to save and restore the global state variables used by output files generated by **lex** and **yacc**. In order to support your platform, enable one of these macros or add an additional set of macros in **sysdep.h**.

  This file also defines the **YACC_VAR** and **YACC_INT** macros to permit sharing of **lex** or **yacc** macros across different implementations. **YACC_VAR** is used to declare a variable as local or external. **YACC_INT** macro is used to declare a variable as integer or short integer.

  sysdep.h also defines the national language versions of the **sprintf** and **fprint** routines. These are called by **NL_SPRINTF** and **NL_VFPRINTF**. If the national language routines on your platform have different names, use a **#define** statement at the top of

  dce-root-dir/**dce/src/rpc/idl/idl_compiler/message.c**

  to rename them. For example, add

  ```
  #if defined(__PLATFORM__)
     #      define NL_SPRINTF platform_sprintf
     #      define NL_VFPRINTF platform_fprintf
     #endif
  ```

  where *PLATFORM* identifies your platform and *platform_sprintf* and *platform_fprintf* are the names of the national language routines on your platform.

  Finally, IDL-generated code in the DCE serviceability component contains a **TRY** - **ENDTRY** block in which **CATCH** and **FINALLY** are both used. If you are porting IDL to a platform that does not support the threads macro sequence **TRY** - **CATCH** - **FINALLY**, you should define the **NO_TRY_CATCH_FINALLY** macro as 1 in **sysdep.h**.

- **acf.h**

  An interface definition file can have a corresponding Attribute Configuration file (*filename*.**acf**). This file renames the global variables used by **lex** and **yacc** when parsing **acf** files. Renaming global variables allows multiple lexical analyzers and parsers to be present in the IDL compiler. If additional state variables are required for save and restore logic, they must be redefined in **acf.h**.

- **idlparse.c**

  Contains a **lex** and **yacc** dependency for state save and restore during recursive parsing.

- **message.c**

  This file contains a layer of message catalog routines specific to the IDL compiler. If you do not have a message catalog system, you must modify this file.

- **sysdep.c**

  Contains functions used only for particular systems. If your system handles such functions differently, make the appropriate additions or changes to this file.

2.1.1 System-Dependent IDL Preprocessor Variables The following system-dependent preprocessor variables are used in building the IDL compiler. They are all defined in:

dce-root-dir/**dce/src/rpc/idl/idl_compiler/sysdep.h**

AUTO_HEAP_STACK_THRESHOLD

> **AUTO_HEAP_STACK_THRESHOLD** defines an estimate for the maximum size of a stack in a server stub. If the IDL compiler estimates that this amount will be exceeded, objects will be allocated via **malloc** instead of on the stack.

AUTO_IMPORT_FILE

> Default input file pathname.

CC_OPT_OBJECT

> Compiler option string to generate object file.

CC_IDIR

> Current directory string for **#include**s.

CD_IDIR

> You must define **CD_IDIR** with the ‘‘current directory’’ symbol for your system. For example, on a UNIX platform, **CD_IDIR** is defined as ‘‘.’’.

CPP

> Default C preprocessor command.

CSTUB_SUFFIX

> Default suffix for client stubs file.

DEFAULT_H_IDIR

> **DEFAULT_H_IDIR** defines default directory for system include (**.h**) files.

DEFAULT_IDIR

> **DEFAULT_IDIR** defines the default directory IDL imports files from.

HASDIRTREE

> **#define HASDIRTREE** if your file system supports directory trees. If you define **HASDIRTREE**, you must also define the tree separator characters (e.g. slash for Unix):
>
> - **BRANCHCHAR**
> - **BRANCHSTRING**

HASINODES

> **#define HASINODES** if your system returns meaningful inode numbers from the **stat( )** system call.

HASPOPEN

#define HASPOPEN if your system supports the **popen**( ) call.

IDL_PROTOTYPES

Defined if IDL should use prototypes.

IDL_VERSION_TEXT

Version string for IDL compiler.

INCLUDE_TEMPLATE

INCLUDE_TEMPLATE tells the IDL compiler how to construct an include statement for DCE include files. For example, when built on a UNIX platform, **INCLUDE_TEMPLATE** is defined as:

```
#include <dce/%s>\n
```

LEX_YACC_STATE_BUFFER

RESTORE_LEX_YACC_STATE

SAVE_LEX_YACC_STATE

The above three variables (actually macros) control the way that **lex** and **yacc** are used by IDL.

Due to differences between implementations of the **lex** and **yacc** tools, different state variables must be saved in order to perform multiple parses within a single program execution. You should either enable one of the **LEX_YACC** sets in

dce-root-dir/**rpc/idl/idl_compiler/sysdep.h**

for your architecture, or add an additional set of macros to save or restore the variables used by **lex** and **yacc**. This is done via inspection of the generated **lex/yacc** output files for any non-automatic state variables. You may also need to make additions to the

dce-root-dir/**rpc/idl/idl_compiler/acf.h**

file, depending on your implementations of **lex/yacc**. See the comments in **acf.h** for more information.

OBJ_FILETYPE

OBJ_FILETYPE is defined as the filename extension on your system for object files. For example, under Unix, **OBJ_FILETYPE** is defined as **''.o''**.

PATH_MAX

Used as filename buffer size if the operating system does not define it.

RESTORE_LEX_YACC_STATE

See **LEX_YACC_STATE_BUFFER**.

S_IFREQ

SAVE_LEX_YACC_STATE

See **LEX_YACC_STATE_BUFFER**.

USER_INCLUDE_TEMPLATE

USER_INCLUDE_TEMPLATE tells the IDL compiler how to construct an include statement for user include files. For example, when built on a UNIX platform,

USER_INCLUDE_TEMPLATE is defined as:

```
#include <%s>
```

2.1.2 Definitions for lex and yacc The IDL compiler uses **lex** and **yacc** to parse interface definitions. Most implementations of **lex** and **yacc** maintain state with global variables. In order to support the *import* statement and the parsing of ACF files, the IDL compiler invokes the parser recursively. In order to make these recursive invocations work with non-reentrant implementations of **lex** and **yacc**, IDL has code to save and restore the global state of the parser.

The following source files depend upon the implementation of **lex** and **yacc**:

dce-root-dir/**dce/src/rpc/idl/idl_compiler/sysdep.h**

dce-root-dir/**dce/src/rpc/idl/idl_compiler/acf.h**

dce-root-dir/**dce/src/rpc/idl/idl_compiler/idlparse.c**

2.2 Porting System IDL Files

The

dce-root-dir/**dce/src/rpc/sys_idl**

directory contains IDL files that clients may need to interact with RPC. You may need to modify the following files when porting to your platform:

- **stubbase.h**

  This file defines macros for marshalling and unmarshalling data. If the default macros do not work on your platform, you may have to redefine them; the *TARGET_MACHINE*/**marshall.h** file (see below) is the place to do this.

- **idlbase.h**

  The following are defined in this file:

  - **HAS_GLOBALDEFS** If this symbol is *not* defined, then the following definitions take effect:

    ```
    #define globaldef
    #define globalref extern
    ```

    You may choose to globally define **HAS_GLOBALDEFS**, or to incorporate the correct definitions from the appropriate header files, or on the command line.

  - **CONST_NOT_SUPPORTED**

  - **VOLATILE_NOT_SUPPORTED**

    If these macros are defined, the C keywords **const** and **volatile** are defined as null strings.

  - **IDL_CHAR_IS_CHAR**

    If this macro is defined, the **idl_char** type will be defined as **char** when client code is being compiled. The base type for the IDL character type is **unsigned char**. However, passing a native character string to a function that requires an **idl_char *** will cause a type mismatch compile error if the native character type of the machine is **signed char**.

- *TARGET_MACHINE*/**ndrtypes.h**

  You may need to edit **ndrtypes.h** (and **idlbase.h**) to add **#define** statements to map IDL types to platform-specific types. A default set of **#define**s will work for most systems. The following table shows which **#define**s are needed:

TABLE 2. idlbase.h and <TARGET_MACHINE>/ndrtypes.h Defines

C define	IDL type	Number of bits
ndr_boolean	boolean	N/A
ndr_byte	byte	8
ndr_char	char OR unsigned char	8
ndr_false	false constant	N/A
ndr_hyper_int	hyper int	64
ndr_long_float	double	64
ndr_long_int	long int	32
ndr_short_float	float	32
ndr_short_int	short int	16
ndr_small_int	small int	8
ndr_true	true constant	N/A
ndr_uhyper_int	unsigned hyper int	64
ndr_ulong_int	unsigned long int	32
ndr_ushort_int	unsigned short int	16
ndr_usmall_int	unsigned small int	8

- *TARGET_MACHINE*/**marshall.h**

  This file contains local definitions (if any) of the **rpc_marshall_*** macros. If platform-specific macros are required, the macro **USE_DEFAULT_MACROS** should be undefined in this file.

  Platforms that are able to use the standard macros defined in **stubbase.h** (see above) can simply leave this file empty.

  The following variables are defined (and undefined) within

  dce-root-dir/**dce/src/rpc/sys_idl/stubbase.h**

  or

  dce-root-dir/**dce/src/rpc/sys_idl/***TARGET_MACHINE*/**marshall.h**

  to control the definition of macros which are emitted into stub files by the IDL compiler. For each variable there is a set of default definitions which is used, unless a target system specific section **#undef**s it and supplies an alternate set of definitions. Exactly which macro definitions are governed by each variable is listed below.

- **USE_DEFAULT_MACROS**

  Controls the definition of the macros which define how to marshall, unmarshall and convert values of each NDR scalar type as well as NDR string types. The following macros must be defined if **USE_DEFAULT_MACROS** is **#undef**'d:

  rpc_marshall_boolean

  rpc_marshall_byte

  rpc_marshall_char

  rpc_marshall_enum

  rpc_marshall_hyper_int

  rpc_marshall_long_float

rpc_marshall_long_int

rpc_marshall_short_float

rpc_marshall_short_int

rpc_marshall_small_int

rpc_marshall_uhyper_int

rpc_marshall_ulong_int

rpc_marshall_ushort_int

rpc_marshall_usmall_int

rpc_marshall_v1_enum

rpc_convert_boolean

rpc_convert_byte

rpc_convert_char

rpc_convert_enum

rpc_convert_hyper_int

rpc_convert_long_float

rpc_convert_long_int

rpc_convert_short_float

rpc_convert_short_int

rpc_convert_small_int

rpc_convert_uhyper_int

rpc_convert_ulong_int

rpc_convert_ushort_int

rpc_convert_usmall_int

rpc_convert_v1_enum

rpc_unmarshall_boolean

rpc_unmarshall_byte

rpc_unmarshall_char

rpc_unmarshall_enum

rpc_unmarshall_hyper_int

rpc_unmarshall_long_float

rpc_unmarshall_long_int

rpc_unmarshall_short_float

rpc_unmarshall_short_int

rpc_unmarshall_small_int

rpc_unmarshall_uhyper_int
rpc_unmarshall_ulong_int

rpc_unmarshall_ushort_int

rpc_unmarshall_usmall_int

rpc_unmarshall_v1_enum

- **USE_DEFAULT_MP_REP**

  Controls the definition of a type and the macros which define the marshalling pointer scheme used on a particular target system. The following macros need to be defined if **USE_DEFAULT_MP_REP** is **#undef**'d:

  rpc_advance_mop

  rpc_advance_mp

  rpc_advance_op

  rpc_align_mop

  rpc_align_mp

  rpc_align_op

  rpc_init_mp

  rpc_init_op

  rpc_synchronize_mp

  and the following types need to be **typedef**'d:

  rpc_mp_t

  rpc_op_t

- *TARGET_MACHINE*/**ndr_rep.h**

  This file contains code that lets you specify the data representations used by your system (for example, big-endian, little-endian, IEEE floating point, ASCII). You do so by defining the following constants as follows:

  - **NDR_LOCAL_CHAR_REP**

    Should be defined as either **ndr_c_char_ascii** or **ndr_c_char_ebcdic**.

  - **NDR_LOCAL_FLOAT_REP**

    Should be defined as **ndr_c_float_ieee**, **ndr_c_float_vax, ndr_c_float_cray**, or **ndr_c_float_ibm**.

  - **NDR_LOCAL_INT_REP**

    Should be defined as either **ndr_c_int_big_endian** or **ndr_c_int_little_endian**.

  These constants specify how a particular platform represents things like characters (ASCII/EBCDIC), integers (big-endian, little-endian), and floating-point numbers (IEEE, VAX, CRAY, and so on).

  In addition, the macro to specify the platform's natural alignment should, if needed, be defined here. The choices are:

  - **IDL_NATURAL_ALIGN_8**
  - **IDL_NATURAL_ALIGN_4**
  - **IDL_NATURAL_ALIGN_1**

2.2.1 Conditionally Built Characteristics of the IDL API The following symbols, all defined in the *dce-root-dir*/**dce/src/rpc/sys_idl/idlbase.h**

header file, conditionally control various aspects of the IDL API:

- **IDL_NO_PROTOTYPES**

  Define IDL_NO_PROTOTYPES to hide prototypes regardless of conditions.

- **IDL_PROTOTYPES**

  Define IDL_PROTOTYPES to control function prototyping in generated stubs.

- **USE_DEFAULT_NDR_REPS**

  Controls the definition of the macros which assign a particular target system type to each NDR scalar type. The following **typedef**s must be defined if **USE_DEFAULT_NDR_REPS** is undefined:

  - **ndr_boolean**
  - **ndr_byte**
  - **ndr_char**
  - **ndr_false**
  - **ndr_hyper_int**
  - **ndr_long_float**
  - **ndr_long_int**
  - **ndr_short_float**
  - **ndr_short_int**
  - **ndr_small_int**
  - **ndr_true**
  - **ndr_uhyper_int**
  - **ndr_ulong_int**
  - **ndr_ushort_int**
  - **ndr_usmall_int**

The following remarks are adapted from comments in **idlbase.h**.

Note that you should not redefine **volatile** except upon careful consideration of the consequences on your platform. If **volatile** is redefined for a compiler that actually supports it already, the result will be nasty program bugs.

Therefore, you should not redefine **volatile**. If your system in fact does not support it, use the **VOLATILE_NOT_SUPPORTED** macro instead (see ‘‘Porting System IDL Files’’, earlier in this chapter).

2.2.2 System IDL Preprocessor Variables The following C preprocessor variables are used in building the IDL compiler.

STUBS_USE_PTHREADS

This is normally **#define**’d in **stubbase.h**.

If you are using a threads package with an API different from Pthreads, you will need to redefine the following macros in **stubbase.h**:

RPC_SS_THREADS_CANCEL_STATE_T

RPC_SS_THREADS_CONDITION_CREATE

RPC_SS_THREADS_CONDITION_DELETE

RPC_SS_THREADS_CONDITION_SIGNAL

RPC_SS_THREADS_CONDITION_T

RPC_SS_THREADS_CONDITION_WAIT

RPC_SS_THREADS_DISABLE_ASYNC

RPC_SS_THREADS_ENABLE_GENERAL

RPC_SS_THREADS_INIT

RPC_SS_THREADS_KEY_CREATE

RPC_SS_THREADS_KEY_GET_CONTEXT

RPC_SS_THREADS_KEY_SET_CONTEXT

RPC_SS_THREADS_KEY_T

RPC_SS_THREADS_MUTEX_CREATE

RPC_SS_THREADS_MUTEX_DELETE

RPC_SS_THREADS_MUTEX_LOCK

RPC_SS_THREADS_MUTEX_T

RPC_SS_THREADS_MUTEX_UNLOCK

RPC_SS_THREADS_ONCE

RPC_SS_THREADS_ONCE_INIT

RPC_SS_THREADS_ONCE_T

RPC_SS_THREADS_RESTORE_ASYNC

RPC_SS_THREADS_RESTORE_GENERAL

RPC_SS_THREADS_X_CANCELLED

See ‘‘Porting System IDL Files’’, above.

2.3 Porting the RPC Runtime Library

The

dce-root-dir/**dce/src/rpc/runtime**

directory has subdirectories that contain RPC runtime library code for various hardware platforms. To port this code to a particular platform, you may need to modify the following files:

- **uuidsys.c**

  This file contains system-specific code for generating universal unique identifiers (UUIDs), together with all the necessary operations for doing so, such as getting the time, getting a process ID, and calling **dce_get_802_addr**( ), which is defined in:

  dce-root-dir/**dce/src/dce/utils/misc/platform/dce_802_addr.c**

  All UUIDs contain a 48-bit node-ID field which must uniquely identify a machine. The OSF/1 reference port uses the IEEE physical level address of the node network controller (either IEEE 802.3 (Ethernet) or IEEE 802.5 (token ring)) for this. Machines with a network controller only sometimes provide access to this number; you will have to use platform-specific techniques to retrieve this information.

If you do not have such an interface or cannot get the number, you will have to produce your own way to generate a unique number. IEEE will sell you a block of numbers from the Ethernet number space if that helps in your solution to this problem. A *Request Form for IEEE Assignment of a 48-bit LAN Globally Assigned Address Block* can be obtained by writing to the following address:

Mr. Vincent Condello
IEEE Standards Office
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331

Telephone: (908) 562-3812

- *machine*/**sysconf.h**

  This system-specific configuration file lets you override several default symbolic constants, include files that are not portable to your system, or both. The DCE source tree contains different versions of **sysconf.h** in platform-specific subdirectories of:

  dce-root-dir/**dce/src/rpc/runtime**

  The easiest way to build your own version of this file is by examining the existing versions of **sysconf.h**, selecting the one that comes closest to meeting your requirements, copying it to a new platform-specific subdirectory, then making the necessary changes. See the ''RPC Runtime Preprocessor Variables'' section later in this chapter.

- *machine*/**ipnaf_sys.c**

  Contains routines specific to the Internet Protocol (IP), the Internet Network Address Family extension service, and the Berkeley BSD) UNIX system. You will need to modify this file so that it runs on your platform. The DCE source tree contains different versions of **ipnaf_sys.c** in platform-specific subdirectories of:

  dce-root-dir/**dce/src/rpc/runtime**

  As with *machine*/**sysconf.h**, you will probably want to copy the closest version to a new, machine-specific directory, then modify the copy. Use the contents of

  dce-root-dir/**dce/src/rpc/runtime/ipnaf_bsd.c**

  as a beginning. If possible, you should use this file unchanged; the OSF/1 version does.

  Note that the **enumerate_interfaces( )** routine defined both in this file and in

  dce-root-dir/**dce/src/rpc/runtime/ipnaf_bsd.c**

  may not allocate space for a sufficient number of **ifreq** structures for your system. The routine as supplied allocates a 1024-byte buffer on the stack for the structures, which are returned by the **ioctl(..., SIOCGIFCONF, ...)** call. Each **ifreq** structure is at least 32 bytes long, so this means that space is allocated for at most 32 **ifreq**s. If this is an inadequate amount for your purposes, an alternative to using stack space would be to replace the stack allocation with a call to **malloc( )** with a **#define**d size. Note however that the **malloc( )** approach cannot be used in the kernel runtime version of the routine in **ipnaf_sys.c** (see the subsection ''Operating System-Specific Code'' in ''Porting the KRPC Runtime Library'', below).

2.3.1 RPC Runtime Preprocessor Variables The following C preprocessor variables are used in building **rpc/runtime**. Many of these are set in:

dce-root-dir/**dce/src/rpc/runtime/***TARGET_MACHINE*/**sysconf.h**

unix

This currently governs including certain Internet include files. See **ipnaf.h** and **twr*.c** files.

ATFORK_SUPPORTED

If a fork handler is available, this routine invokes **rpc__fork_handler** prior to and immediately after forking. See:

dce-root-dir/**dce/src/rpc/runtime/cominit.c**

AUTH_DEFS

The value of this variable is set in

dce-root-dir/**dce/src/rpc/runtime/Makefile**

and is dependent on the values of the following preprocessor variables:

- **AUTH_KRB**
- **AUTH_KRB_DG**
- **AUTH_KRB_CN**

If only **AUTH_KRB** is defined, then Kerberos support for both connection-oriented and datagram RPC is enabled. However, if *only one* of **AUTH_KRB_DG** and **AUTH_KRB_CN** is defined (together with **AUTH_KRB**), then support only for the specified RPC protocol is enabled. If **AUTH_KRB** is not defined, *no* Kerberos support is enabled.

For further information, see the comments in:

dce-root-dir/**dce/src/rpc/runtime/Makefile**

MAX_NETADDR_LENGTH

Maximum number of bytes in network address. Default is 14.

NON_CANCELLABLE_IO

For Pthreads implementations that do not allow cancels to be delivered in **stdio** (read, write, select, etc.,) this define enables a **timed** select in the listener thread that performs a **pthread_testcancel**( ) to receive cancels.

NO_SIOCGIFADDR

Define **NO_SIOCGIFADDR** if your network interface does not support the **ioctl SIOCGIFADDR** operation.

NAF_IP

Define **NAF_IP** if Internet Protocol is used. **NAF_IP** is set for the DCE reference platforms.

PROT_NCACN

Define **PROT_NCACN** to build connection-based support into RPC.

PROT_NCADG

Define **PROT_NCADG** to build datagram support into RPC.

RPC_MUTEX_DEBUG

Enables mutex lock and/or condition variable debugging.

RPC_MUTEX_STATS

Enables mutex lock and/or condition variable statistics.

RPC_DEFAULT_NLSPATH

Not used, although present in:

dce-root-dir/**dce/src/rpc/runtime/***TARGET_MACHINE*/**sysconf.h**

RPC_NLS_FORMAT

Not used, although present in:

dce-root-dir/**dce/src/rpc/runtime/dce_error.c**

_SOCKADDR_LEN

The layout of a 4.4 **struct sockaddr** includes a 1 byte ‘‘length’’ field which used to be one of the bytes of the ‘‘family’’ field. (The ‘‘family’’ field is now 1 byte instead of 2 bytes.) 4.4 provides binary compatibility with applications compiled with a 4.3 **sockaddr definition by inferring a default length when the supplied length is zero.**

Define _SOCKADDR_LEN if your socket’s **sockaddr struct** contains **sa_len**.

2.3.2 Correction of Mispacked RPC Headers on Certain PLatforms The **uuid_t** type is defined by IDL, and interface and object UUIDs are transmitted ‘‘over the wire’’ as part of the RPC message headers. However, not all C compilers (especially those for machines whose smallest addressible unit is not 8 bits) pack the RPC header structure ‘‘correctly’’ (that is, into a storage layout that can be overlayed on a vector of bytes that make up a packet that has just come off the wire). As a result, on some machines **rpc_dg_pkt_hdr_t** cannot be used ‘‘as is’’ on incoming packets, or used to set up outgoing packets. Machines that have this problem are called ‘‘mispacked header machines’’.

If the host machine is a mispacked header machine, the incoming RPC headers, which contain the UUID, will not be able to be overlaid correctly onto the host header struct (**rpc_dg_pkt_hdr_t**), which will be somewhat too ‘‘big’’ for it; instead, the header will have to be expanded first. It is the job of a porter whose target platform has this characteristic to add code to the skeleton of **unpack_hdr( )** in

dce-root-dir/**dce/src/rpc/runtime/dglsn.c**

to accomplish the unpacking, and to **compress_hdr( )**, which is called in

dce-root-dir/**dce/src/rpc/runtime/dgutl.c**

and

dce-root-dir/**dce/src/rpc/runtime/dgxq.c**

to pack the outgoing headers. (Note that no skeleton is supplied for the **compress_hdr( )** routine.) Calls to these routines are already present at the appropriate places in the RPC runtime; all that is needed is to activate them (after, of course, you have added the necessary code to the routines themselves) by defining **MISPACKED_HDR** in:

dce-root-dir/**dce/src/rpc/runtime/***TARGET_MACHINE*/**sysconf.h**

The **rpc_c_dg_rpho_...** constants (‘‘**rpho**’’ stands for ‘‘raw packet header offset’’) in the

dce-root-dir/**dce/src/rpc/runtime/dg.h**

file can be used to locate the logical header fields in a raw packet header.

2.4 Information on Porting ‘‘rpcd’’

The following sections contain information pertinent to porting the RPC daemon (**rpcd**, also referred to as the ‘‘endpoint mapper’’).

2.4.1 Enabling or Disabling Remote Endpoint Access The behavior of **rpcd** has been changed in DCE 1.0.3 so that requests from remote hosts to add or delete endpoints from the endpoint map will now be rejected (in previous versions of DCE, **rpcd** would fulfill such requests). The change has been made in order to prevent the possibility of unauthenticated users’ adding or deleting endpoints anywhere in a cell, simply by making calls through the RPC interface, or by issuing commands through **rpccp**.

The code that enables remote endpoint access is still present in the source, however, and it can be enabled or disabled by defining or undefining the preprocessor variable **REMOTE_ENDPOINT_ACCESS**. The code exists in two modules:

- *dce-root-dir*/**dce/src/rpc/rpcd/rpcdep.c**

  If the variable is *not* defined in this code, **rpcd** will reject remote requests to modify the endpoint map, but will allow the map to be read. If the variable *is* defined, **rpcd** will execute remote endpoint map modification requests (the pre-DCE 1.0.3 behavior).

  The default is that the variable is not defined, and remote modification requests are rejected.

- *dce-root-dir*/**dce/src/rpc/rpccp/rpccp.c**

  If the variable is *not* defined in this code, **rpccp** will reject remote requests to modify the endpoint map (via the **add mapping** and **remove mapping** subcommands), but will allow the map to be read by remote users. If the variable *is* defined, **rpccp** will execute remote endpoint map modification requests (the pre-DCE 1.0.3 behavior).

  The default is that the variable is not defined, and remote modification requests are rejected.

2.4.2 RPCD data file The RPCD service maintains a persistent database of endpoints in a file, located in

dcelocal/**var/rpc/rpcdep.dat**

on the OSF/1 reference platform (where *dcelocal* usually stands for

/opt/dcelocal

as set up by default by **dce_config** during cell configuration). The following comments were derived from the three source files

dce-root-dir/**dce/src/rpc/rpcd/dsm.idl**

dce-root-dir/**dce/src/rpc/rpcd/dsm.c**

dce-root-dir/**dce/src/rpc/rpcd/dsm_p.h**

and discuss the physical representation of this database.

2.4.3 DSM (Data Store Manager) Public Interface Definition The Data Store Manager is a heap storage allocation package wherein allocated records are strongly associated with storage in a backing file, such that they can be stably stored upon modification. The basic paradigm is that the client **ALLOCATE**s a block of some requested size, modifies it in memory, and **WRITE**s it; successful completion of the **WRITE** implies that the record has been stably stored in the file.

DSM uses OS page alignment to define an atomic operation (a write of or within a page is assumed to either succeed or fail, without any intermediate state). Records are laid out in the file such that the DSM header, as well as some reasonable chunk of the start of application data (e.g. the first 64 bytes total of each record) are contiguous in a page, and so can be written atomically. A write that spans a page boundary occurs in two phases (assuming the record being written was previously free and is so marked on disk): the data portion is written and synched first, then the DSM header (specifically the ‘‘inuse/free’’ mark) to commit the write.

Updates are not atomically supported. Changing the contents of a record requires conceptually adding a new version and deleting the old one. DSM provides an operation to ‘‘detach’’ a record (mark it free in the file, effectively deleting it if a crash occurs at this point), after which it can be written normally. This is adequate for applications like DSM which can recover from crashes by replaying the last operation on its propagation queue. Another approach would be to allocate a new record and make a copy, setting a ‘‘new’’ flag in its application header, then freeing the old copy, and finally clearing the ‘‘new’’ flag in the new copy. Upon crash recovery the application might see two versions of the same datum, one flagged ‘‘new’’, and can discard the other one (or it may see only the ‘‘new’’ version).

The DSM does not currently itself provide mutual exclusion, although it must be used in such a context (the caller is currently assumed to do the mutex).

```
typedef struct page_t {    /* generic page */
    unsigned char page_contents[PAGE_SIZE];
} page_t;

typedef struct block_t {       /* block preheader */
    struct block_t *link;      /* link to next block on (free) list */
                           /*   [meaningless in file]          */
    unsigned long   size;      /* size of user data */
    unsigned long   loc;       /* location (offset) of preheader in file */
    boolean         isfree;    /* true iff free */
    unsigned char   cookie;    /* magic number basic identification */
    unsigned char   unused[2]; /* preheader ends here */
    double          data;      /* user data begins here -- double to align */
} block_t;

typedef struct file_hdr_t {    /* first page of file contains global info */
    long            version;   /* file format version */
    long            pages;     /* number of initialized data pages */
    long            pad1[20];  /* reserve for DSM header expansion */
    unsigned char   info[INFOSZ];  /* space for client info */
    page_t          padding;   /* pad out past page boundary */
} file_hdr_t;
```

Note that strong assumptions are made about the size and alignments of this structure. The important thing is that the **data** field be naturally aligned for all potential user data (8-byte alignment), and the preheader should occupy the **PREHEADER** bytes just before the user data. It currently looks as follows (16 bytes):

```
+--------+--------+--------+--------+  <-
|         space for link ptr        |   |
+--------+--------+--------+--------+   |
|         size of user data         |   |
+--------+--------+--------+--------+    > preheader (16 bytes)
|     offset in file of preheader   |   |
+--------+--------+--------+--------+   |
|  FREE  | cookie |    (unused)     |   |
+--------+--------+--------+--------+  <-
|  user data...
+--------+
```

3. Building and Linking

The

dce-root-dir/**dce/src/rpc/runtime/Makefile**

may be used to configure **libnck.a** for various combinations of network protocols, authentication mechanisms, and name server usage.

3.1 Building the IDL Compiler

The DCE 1.1 IDL compiler sources must be built with the **-DMIA** option specified. If they are not, some of the files will be built in their 1.0 versions. The ODE Makefiles that accompany DCE 1.1 have been modified to include this flag.

4. Setup, Testing, and Verification

Make sure that the **rpcd** endpoint map service is running.

Change to the

/opt/dce1.1/bin

directory and enter

```
rpcd
```

to start **rpcd** (the RPC daemon). You can also include the **-d** flag to provide debugging output when you start **rpcd**.

4.1 The perf Tests

The **perf** test case tests a larger subset of the RPC runtime library than **v2test**. You must start the **perf server** as one process and then start the **perf client** as another process before running the **perf** test case. These processes can be run on the same or different hosts, as long as the server process is started first. The **server** and **client** can be found in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.1/test/rpc/runtime/perf**

directory. (Note that the contents of this directory are built from the contents of the

dce-root-dir/**dce/src/test/rpc/runtime/perf**

directory in the source tree.)

To test using the **perf** test case, make a number of remote procedure calls from the **perf** client to the **perf** server. The **perf** server waits for remote procedure calls from the **perf** client and then gives a response. The **perf** server then prints messages that give the results of the remote procedure call. To fully test using **perf** test, use different combinations of **perf** server and **perf** client testing options and observe the resulting messages.

To start the **server**, enter

```
server 1 ncadg_ip_udp
```

or:

```
server 1 ncacn_ip_tcp
```

at the command line. The following message will be printed:

```
Got Binding: ncadg_ip_udp:ip_addr [port]
```

where *ip_addr* is the IP address of the server and *port* is the number of the port the server is listening to.

To start the **client**, enter a command similar to the following:

```
client 1 ncadg_ip_udp:ip_addr[port] 10 5 n y 100
```

or:

```
client 1 ncacn_ip_tcp:ip_addr[port] 10 5 n y 100
```

at the command line, where *ip_addr* is the IP address of the server (printed out when you started the server) and *port* is the port number that the server is listening to (printed out when you started the server).

See the

dce-root-dir/**dce/src/test/rpc/runtime/perf/README**

file for further information, including information about several scripts that can be used to run the **perf** tests.

4.1.1 Help Messages You can get help messages on how to invoke both the **server** and **client** programs by entering the program name at the command line with no arguments. You can get additional help on a specific **client** test case by entering the program name followed by the test number. For example, entering **client 2** prints help on test number 2.

4.1.2 The perf server Program The **perf server** testing options are listed below:

```
server [-sD] [-S server_loops] [-d switch_level] [-p auth_proto, principal, [keytab_file]]
[-v {0|1}]
[-B bufsize] max_calls protseq_spec [protseq_spec ...]
```

where:

-s	Enables remote shutdown of the server. This parameter is optional, and is currently not implemented.
-D	This optional parameter specifies the default level of debug output.
-S *server_loops*	Specifies the number of times to run the server listen loop. If no value is specified for the *server_loops* parameter, the default value is 1.
-d *switch_level*	This optional parameter lets you specify the amount of debug output desired. Some useful *switch_level* settings are the following:

0-3.5	Maximum error/anomalous condition reporting and mutex checking. This amount of output is often too verbose for normal use. Also, there is extra overhead for mutex checking.
0-1.10	Same function as 0-3.5, but drops some transmit/receive informational messages.
2-3.4	Same function as 0-1.10.
0.10	Reports all error conditions plus a little more; no mutex checking.
0.1	Report error conditions only (same as specifying **-d**).

-p	Specifies an authenticated RPC call. You must enter the **-p** parameter with the *auth_proto* parameter and the *principal* parameter.
auth_proto	Specifies which authentication service to use when the server receives a remote procedure call. The following values are valid for *auth_proto*:

0	No authentication is used.
1	OSF DCE private key authentication is used.
2	OSF DCE public key authentication is used. This parameter is reserved for future use and is not yet supported.

Note that if private key authentication is desired, a keytab file must be set up (with the **rgy_edit ktadd** command) before the server program is run. Otherwise, the server will display the following message at startup:

```
***Error setting principal - Requested key is unavailable (dce/sec)
```

and terminate.

principal Specifies the principal name of the server to use when authenticating remote procedure calls. The content of the name and its syntax are defined by the authentication service in use.

-v 0 Enables verbose output.

-v 1 Disables verbose output. Verbose output is disabled by default if no **-v** flag is used with **perf server**.

bufsize Sets the connection-oriented protocol socket buffer size, specified in bytes.

max_calls Specifies the number of threads that are created to service requests.

protseq_spec Specifies one of the following:

protocol_sequence
Tells the server to listen for remote procedure calls using the specified protocol sequence (for example, network protocol) combined with the endpoint information in **perf.idl**. Valid values for this argument are described in the discussion of the **v2server** program. The server calls **rpc_server_use_protseq_if** to register the protocol sequence with the RPC runtime.

all Tells the server to listen for remote procedure calls using all supported protocol sequences. The RPC runtime creates a different binding handle for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC runtime. The server calls **rpc_server_use_all_protseqs** to accomplish this.

allif Tells the server to listen for remote procedure calls using all the specified protocol sequences and endpoint information in **perf.idl**. The server uses **rpc_server_use_all_protseqs_if** to accomplish this.

ep *protocol_sequence endpoint*
Tells the server to listen for remote procedure calls using the specified protocol sequence and endpoint information (for example, **ep ncadg_ip_udp 2000**). The server calls **rpc_server_use_protseq_ep** to accomplish this.

notif *protocol_sequence*
Tells the server to listen for remote procedure calls using the specified protocol sequence. The RPC runtime dynamically generates the endpoint. The server calls **rpc_server_use_protseq** to accomplish this.

4.1.3 The perf client Program the **perf client** testing options are listed below:

```
client [-Disf] [-d switch_level] [{-m | -M} nthreads] [-t timeout]\
    [-c timeout] [-w wait_point, wait_secs]\
    [-p auth_proto, authz_proto [, level, principal]\
    [-r frequency] [-R frequency] [-v {0|1}]\
    [-f opt] [-B bufsize] [-o] [-s]\
    test test_parms
```

where:

-D This optional parameter specifies the default level of debug output.

-i This optional parameter causes statistics to be dumped at the end of the test.

-s This optional parameter prints statistics at the end of the test.

-o Specifies that **perf** object UUID be used in bindings (default is that no object UUID is used).

-f Repeats the test after a **fork( )**.

-d *switch_level* Lets you specify the amount of debug output desired. Some useful *switch_level* settings are the following:

0-3.5	Maximum error/anomalous condition reporting and mutex checking. This amount of output is often too verbose for normal use. Also, there is extra overhead for mutex checking.
0-1.10	Same function as 0-3.5, but drops some transmit/receive informational messages.
2-3.4	Same function as 0-1.10.
0.10	Reports all error conditions plus a little more; no mutex checking.
0.1	Report error conditions only (same as specifying **-d**).

-m *nthreads* This optional parameter causes *nthreads* tasks to be run at the same time.

-M *nthreads* This optional parameter has the same function as the **-m** parameter, but uses a shared binding handle.

-t *timeout* Sets the communications timeout value to *timeout* seconds. The value specified for *timeout* must be between zero and ten.

-c *timeout* Sets the cancel timeout value to *timeout* seconds.

-w *wait_point, wait_secs*
Causes the client to wait at the *wait_point* for *wait_secs* seconds.

-p Specifies an authenticated RPC call. You must enter the *auth_proto* and *authz_proto* parameters when using **-p**; the *level* and *principal* parameters are optional.

-r *frequency* Resets bindings every *frequency* number of calls in a single pass.

-R *frequency* Recreates bindings every *frequency* number of calls in a single pass.

auth_proto Specifies which authentication service to use. The following values are valid for *auth_proto*:

0	No authentication is used.
1	OSF DCE private key authentication is used.
2	OSF DCE public key authentication is used. This parameter is reserved for future use and is not yet supported.

authz_proto Specifies the authorization service implemented by the server. The following values are valid for *authz_proto*:

0	The server performs no authorization.
1	Server performs authorization based on the client principal name.

2	Server performs authorization checking using the client DCE privilege attribute certificate (PAC) information sent to the server with each remote procedure call.

level Specifies the level of authentication to be performed on remote procedure calls. The following values are valid for *level*:

0	Use the default authentication level for the specified authentication service.
1	Perform no authentication.
2	Authenticate only when the client first establishes a relationship with the server (only on "connect.")
3	Authenticate only at the beginning of each remote procedure call.
4	Authenticate that all data received is from the expected client.
5	Authenticate that none of the data transferred between client and server has been modified.
6	Authentication includes all previous levels as well as encrypting each remote procedure call argument.

principal Specifies the expected principal name of the server. The content of the name and its syntax are defined by the authentication service in use.

-v 0 Enables verbose output.

-v 1 Disables verbose output. Verbose output is disabled by default if no **-v** flag is used with **perf client**.

-f *opt* Repeats test after fork. *opt* is a digit from 1 to 6, with the following meanings:

1	Repeat test in the original and child processes.
2	Repeat test in the original process only.
3	Repeat test in the child process only.
4	Repeat test in the child and grandchild processes.
5	Repeat test in the grandchild process only.
6	Run test in the child process only.

-B *bufsize* Sets the connection-oriented protocol TCP socket buffer size, where *bufsize* is the desired size, specified in bytes.

test Specifies which test to run. Each test requires different *test_parms*. The following values are valid for *test*:

0	Null call
1	Variable-length input argument
2	Variable-length output argument
3	Broadcast test
4	Maybe test
5	Broadcast/maybe test
6	Floating-point test

7	Call unregistered server interface
8	Forwarding test
9	Exception test
10	Slow call
11	Shutdown server
12	Callback (**Note:** This test is not supported.)
13	Generic interface test
14	Context test
15	Static cancel test
16	Statistics test
17	Interface identifiers test
18	One shot test

test_parms The following *test_parms* correspond to the test numbers:

Test Number	**Test_Parms**
0	*string_binding passes calls/pass verify? idempotent?*
1	*string_binding passes calls/pass verify? idempotent? nbytes*
2	*string_binding passes calls/pass verify? idempotent? nbytes*
3	*protocol_sequence*
4	*string_binding*
5	*protocol_sequence*
6	*string_binding passes calls/pass verify? idempotent?*
7	*string_binding*
8	*string_binding global?*
9	*string_binding*
10	*string_binding passes calls/pass verify? idempotent? seconds [mode]*
11	*string_binding*
12	*string_binding passes callbacks/pass idempotent?*
13	*string_binding*
14	Host passes *die?* seconds
15	Host passes *idempotent?* **[seconds[cancel_two_seconds]]**
16	*[host+ep]*
17	*[host+ep]*
18	*[host+ep] forward? idempotent?*

where:

string_binding
Contains the character representation of a binding in the form *protocol_sequence:network_address[port]*, where *protocol_sequence* is one of the valid protocol sequences discussed previously,

network_address is the network address of the server, and *port* is the port the server is listening to.

passes
: Specifies the number of times to run the test.

calls/pass
: Specifies the number of remote calls per pass.

verify?
: Specifies whether the test case must verify that there were no data transmission errors. Enter **y** to verify, **n** to not verify.

die?
: For the context test, this parameter specifies if the server's context is freed at the end of each pass. Enter **y** to free the context.

idempotent?
: Specifies whether or not to place an idempotent or nonidempotent call (enter **y** to place an idempotent call, **n** to place a nonidempotent call.)

nbytes
: Specifies the number of bytes transferred per call.

protocol_sequence
: Specifies one or more network protocols that can be used to communicate with a client. Valid values for this argument are specified in the discussion of the **v2server** program.

callbacks/pass
: Specifies the number of times the server calls back the client per pass.

seconds
: The *seconds* parameter specifies the number of seconds the server delays while executing a remote procedure call. For the context test, this parameter specifies the number of seconds the client will **sleep** after it checks if the test was successful.

mode
: For the *slow call* test, *mode* specifies the technique used by **perf** to slow down the call. The following values are valid for *mode*:

0	Sleep
1	Slow I/O
2	CPU loop

global
: This parameter is currently not checked. It can be set by entering **y** or **n**.

cancel_two_seconds
: Specifies the number of seconds that the client's RPC runtime will wait for a server to acknowledge a cancel. Note that the value of *cancel_two_seconds* must be greater than the value of the *seconds* argument (described above); otherwise Test 15 cannot be run successfully.

[host+ep]
: Specifies the host IP address and endpoint.

5. RPC Runtime Output and Debugging Output

The RPC component outputs server information of all kinds via the DCE serviceability component. The following sections describe how to control the various kinds of information (including debugging output) available from RPC via serviceability.

5.1 Normal RPC Server Message Routing

There are basically two ways to control normal RPC server message routing:

- At startup, through the contents of a routing file (which are applied to all components that use serviceability messaging).
- Dynamically, through the **dcecp log** object.

The following sections describe each of these methods.

5.1.1 Routing File If a file called

dce-local-path/**svc/routing**

exists when RPC is brought up (i.e., when **dced** is executed or when the cell is started through **dce_config**), the contents of the file (if in the proper format) will be used as to determine the routing of RPC serviceability messages.

The value of *dce-local-path* depends on the values of two **make** variables when DCE is built:

DCEROOT	its default value is: **/opt**
DCELOCAL	its default value is: **$DCEROOT/dcelocal**

Thus, the default location of the serviceability routing file is normally:

/opt/dcelocal/svc/routing

However, a different location for the file can be specified by setting the value of the environment variable **DCE_SVC_ROUTING_FILE** to the complete desired pathname.

The contents of the routing file consist of formatted strings specifying the routing desired for the various kinds of messages (based on message severity). Each string consists of three fields as follows:

*severity***:***output_form***:***destination* [*output_form***:***destination* . . .]

Where:

severity specifies the severity level of the message, and must be one of the following:

- **FATAL**
- **ERROR**
- **WARNING**
- **NOTICE**
- **NOTICE_VERBOSE**

(The meanings of these severity levels are explained in detail in Chapter 4 of the *OSF DCE Application Development Guide — Core Components* volume, in the section entitled ‘‘Specifying Message Severity’’.)

output_form specifies how the messages of a given severity level should be processed, and must be one of the following:

- **BINFILE**

  Write these messages as binary log entries

- **TEXTFILE**

  Write these messages as human-readable text

- **FILE**

  Equivalent to **TEXTFILE**

- **DISCARD**

  Do not record messages of this severity level

- **STDOUT**

  Write these messages as human-readable text to standard output

- **STDERR**

  Write these messages as human-readable text to standard error

Files written as **BINFILE**s can be read and manipulated with a set of logfile functions. See Chapter 4 of the *OSF DCE Application Development Guide — Core Components* volume, mentioned above, for further information.

The *output_form* specifier may be followed by a two-number specifier of the form:

.*gens***.***count*

Where:

gens is an integer that specifies the number of files (i.e., generations) that should be kept

count is an integer specifying how many entries (i.e., messages) should be written to each file

The multiple files are named by appending a dot to the simple specified name, followed by the current generation number. When the number of entries in a file reaches the maximum specified by *count*, the file is closed, the generation number is incremented, and the next file is opened. When the maximum generation number files have been created and filled, the generation number is reset to 1, and a new file with that number is created and written to (thus overwriting the already-existing file with the same name), and so on, as long as messages are being written. Thus the files wrap around to their beginning, and the total number of log files never exceeds *gens*, although messages continue to be written as long as the program continues writing them.

destination specifies where the message should be sent, and is a pathname. The field can be left blank if the *output_form* specified is **DISCARD**, **STDOUT**, or **STDERR**. The field can also contain a **%ld** string in the filename which, when the file is written, will be replaced by the process ID of the program that wrote the message(s). Filenames may *not* contain colons or periods.

Multiple routings for the same severity level can be specified by simply adding the additional desired routings as space-separated

*output_form***:***destination*

strings.

For example,

```
FATAL:TEXTFILE:/dev/console
WARNING:DISCARD:--
NOTICE:BINFILE.50.100:/tmp/log%ld STDERR:-
```

Specifies that:

- Fatal error messages should be sent to the console.
- Warnings should be discarded.
- Notices should be written both to standard error and as binary entries in files located in the **/tmp** directory. No more than 50 files should be written, and there should be no more than 100 messages written to each file. The files will have names of the form:

  /tmp/log*process_id***.***nn*

  where *process_id* is the process ID of the program originating the messages, and *nn* is the generation number of the file.

5.1.2 Routing by the dcecp log Object Routing of RPC server messages can be controlled in an already-started cell through the **dcecp log** object. See the **log.8dce** reference page in the *OSF DCE Command Reference* for further information.

5.2 Debugging Output

Debugging output from RPC can be enabled (provided that RPC has been built with **DCE_DEBUG** defined) by specifying the desired debug messaging level and route(s) in the

*dce-local-path***/svc/routing**

routing file (described above), or by specifying the same information in the **SVC_RPC_DBG** environment variable, before bringing up RPC (i.e., prior to starting the cell). Debugging output can also be enabled and controlled through the **dcecp log** object.

Note that, unlike normal message routing, debugging output is always specified on the basis of DCE component/sub-component (the meaning of ‘‘sub-component’’ will be explained below) and desired level.

The debug routing and level instructions for a component are specified by the contents of a specially-formatted string that is either included in the value of the environment variable or is part of the contents of the routing file.

The general format for the debug routing specifier string is:

"*component***:***sub_comp***.***level***,**. . .**:***output_form***:***destination* \
[*output_form***:***destination* . . .] "

where the fields have the same meanings as in the normal routing specifiers described above, with the addition of the following:

component	specifies the component name (i.e., **rpc**)
sub_comp.level	specifies a subcomponent name, followed (after a dot) by a debug level (expressed as a single digit from 1 to 9). Note that multiple subcomponent/level pairs can be specified in the string. A star (‘‘*’’) can be used to specify all sub-components. The sub-component list is parsed in order, with later entries supplementing earlier ones; so the global specifier can be used to set the basic level for all sub-components, and specific sub-component exceptions with different levels can follow (see the example below).

‘‘Sub-components’’ denote the various functional modules into which a component has been divided for serviceability messaging purposes. For RPC, the sub-components are as follows:

general	RPC general messages
mutex	RPC mutex messages
xmit	RPC xmit messages

recv	RPC receive messages
dg_state	RPC DG state messages
cancel	RPC cancel messages
orphan	RPC orphan messages
cn_state	RPC CN state messages
cn_pkt	RPC CN packet messages
pkt_quotas	RPC packet quota messages
auth	RPC authorization messages
source	RPC source messages
stats	RPC statistics messages
mem	RPC memory messages
mem_type	RPC memory type messages
dg_pktlog	RPC DG packetlog messages
thread_id	RPC thread ID messages
timestamp	RPC timestamp messages
cn_errors	RPC CN error messages
conv_thread	RPC conversation thread messages
pid	RPC pid messages
atfork	RPC atfork messages
cma_thread	RPC CMA thread messages
inherit	RPC inherit messages
dg_sockets	RPC datagram sockets messages
timer	RPC timer messages
threads	RPC threads messages

For example, the string

```
"rpc:*.1,cma_thread.3:TEXTFILE.50.200:/tmp/RPC_LOG
```

sets the debugging level for all RPC sub-components (*except* **cma_thread**) at 1; **cma_thread**'s level is set at 3. All messages are routed to **/tmp/RPC_LOG**. No more than 50 log files are to be written, and no more than 200 messages are to be written to each file.

The texts of all the RPC serviceability messages, and the sub-component list, can be found in the RPC sams file, at:

dce-root-dir/**dce/src/rpc/sys_idl/rpc.sams**

For further information about the serviceability mechanism and API, see Chapter 4 of the *OSF DCE Application Development Guide — Core Components* volume, ‘‘Using the DCE Serviceability Application Interface’’.

5.3 Restricting Protocol Sequences Used

There is a way to restrict the pool of protocol sequences eligible for use by RPC to a group of one or more that you specify. **RPC_SUPPORTED_PROTSEQS** is an environment variable tested at RPC startup by code in:

dce-root-dir/**dce/src/rpc/runtime/cominit.c**

It should be used *only* for debugging DCE.

The value of **RPC_SUPPORTED_PROTSEQS** is a colon-separated list of RPC protocol sequence strings. When **RPC_SUPPORTED_PROTSEQS** is defined, it restricts the set of protocol sequences that the RPC runtime will use to the list of sequences defined as its value. Normally, the RPC runtime uses any protocol sequences it can detect on the local host.

To use this debugging feature, just set **RPC_SUPPORTED_PROTSEQS** (*before* starting **dced**) to one or more DCE RPC protocol sequences (each sequence separated by a colon when using C shell). For example:

```
setenv RPC_SUPPORTED_PROTSEQS ncadg_ip_udp
```

will restrict RPC to only use UDP. If you wanted to use only the RPC connection-oriented protocol over TCP/IP and DECnet (assuming that your implementation supports the latter), you could set the variable as follows:

```
setenv RPC_SUPPORTED_PROTSEQS ncacn_ip_tcp:ncacn_dnet_nsp
```

The set of protocol sequences currently defined in DCE are:

- **ncadg_ip_udp**
- **ncacn_ip_tcp**

RPC_SUPPORTED_PROTSEQS is tested at RPC startup in each process. It lasts only for the life of that process. If **RPC_SUPPORTED_PROTSEQS** is not set, all protocol sequences that can be supported will be available for use by the application.

Note that if you build the RPC runtime library without defining **DEBUG** (i.e., without **-DDEBUG**), then **RPC_SUPPORTED_PROTSEQS** is ignored.

6. Some RPC Questions and Answers

This section contains several RPC questions and answers that have arisen during DCE porting and application development efforts so far. Some of this material is not directly applicable to porting but is included here as useful background information about the component and DCE.

Q1: Is it possible to add simultaneous TP4 and TCP support under the connection oriented protocol?

A: Yes. The RPC runtime is designed so that new protocols can be added in a modular fashion.

Assuming that you have a sockets interface to TP4, the work required would be to first implement a new network address family (NAF), and then implement the NSI tower support. More work would be required if you do not have a sockets interface to your TP4 implementation.

Q2: I'm not sure I understand how RPC functions work with the **[broadcast]** attribute. When an application is using the automatic binding method, why does the **RPC_DEFAULT_ENTRY** environment variable have to be set to the NSI entry that contains the server's exported bindings, even though the **broadcast** attribute is being used in making the client call? When I try to make such a call without setting **RPC_DEFAULT_ENTRY**, the client gives an IOT exception. But if **RPC_DEFAULT_ENTRY** is set to the correct namespace entry, the call succeeds. If the client still has to bind to the server before sending a broadcast, what point is there in using the broadcast attribute?

A: It's probably fair to say that it was not expected that applications would try to mix the use of the **auto_handle** and **broadcast** features. In your case, the IDL compiler is making a valiant attempt to do something useful with the combination, but it's not clear it shouldn't simply flag the mixed use as being an error.

The client doesn't *have* to bind to a server, but it *does* have to make a choice about what RPC protocol sequence it wants to use. The thing a client would generally do is something like:

```
{
    handle_t h;
    error_status_t status;

    rpc_binding_from_string_binding("ncadg_ip_udp:", &h, &status);
    bcast(h, ...);
}
```

—Where the choice of protocol sequence would, one hopes, be configured a little more cleanly than in the example.

The server needn't call **rpc_ns_binding_export**( ), and no one needs to set up **RPC_DEFAULT_ENTRY**; CDS doesn't get involved in this operation at all.

What's happened in your case is that the client stub is attempting to import a binding. If it succeeds (that is, if the server's done the export and the client process has **RPC_DEFAULT_ENTRY** set appropriately), the binding is passed to the RPC runtime, which promptly discards everything in it except for the protocol sequence. If it fails —as will happen, for example, when you haven't set **RPC_DEFAULT_ENTRY**— it should raise an exception, which is what's showing up as your IOT.