OSF® DCE Version 1.2.2

DCE Testing Guide

January 17, 1997
Revision 1.2.2

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

The information contained within this document is subject to change without notice.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein, or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

Contents

LIST OF FIGURES

LIST OF TABLES

Preface

The *DCE Testing Guide* describes how to test the OSF™ Distributed Computing Environment (DCE).

Audience

The *DCE Testing Guide* is for licensees who are porting DCE to a non-reference platform.

Applicability

This is Revision 1.0 of this guide. It applies to the OSF™ DCE Version 1.2.2 offering. See your software license for details.

Purpose

The purpose of this manual is to guide developers testing DCE. After reading this guide, you should be able to effectively test DCE.

Document Usage

This section describes the 13 chapters and 2 appendices that make up the guide.

- Chapters 1 - 10

  These chapters give information on testing the DCE components, with one chapter devoted to each component.

- Chapter 11: TET and DCE Testing

  This chapter describes how to install the Test Environment Toolkit (TET), which is used to execute many of the DCE functional and system tests, and how TET is used to execute tests and monitor their results.

- Chapter 12: DCE System Tests under TET

  This chapter describes the DCE system tests that are executed using TET.

- Chapter 13: DCE System Tests not under TET

  This chapter describes the DCE system tests that are not executed directly, not by TET.

- Appendix A: File and Path Names Cross-Reference

  This appendix lists the pathnames of many files mentioned in the DCE documentation.

- Appendix B: DCE Abbreviations List

  This appendix contains a list of DCE abbreviations met with both in the documentation and the source code, together with brief definitions.

Throughout this guide, the path variable *dce-root-dir* is used, and *dce-root-dir* is *your-root-dir*/**dce**, where *your-root-dir* is the directory in which you create the **dce** directory, and **dce** is the directory into which you unloaded the contents of the DCE distribution tape.

Related Documents

For additional information about the Distributed Computing Environment, refer to the following documents:

- *Introduction to OSF DCE*
- *OSF DCE Command Reference*
- *OSF DCE Application Development Reference*
- *OSF DCE Administration Guide*
- *OSF DCE DFS Administration Guide and Reference*
- *OSF DCE GDS Administration Guide and Reference*
- *OSF DCE Problem Determination Guide*
- *OSF DCE Application Development Guide*

- *Application Environment Specification (AES)/Distributed Computing*
- *OSF DCE Technical Supplement*
- *OSF DCE Release Notes*

Typographic and Keying Conventions

This document uses the following typographic conventions:

Bold	**Bold** words or characters represent system elements that you must use literally, such as commands, flags, and pathnames.
Italic	*Italic* words or characters represent variable values that you must supply.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times. Vertical ellipsis points indicate that you can repeat the preceding item one or more times.

This document uses the following keying conventions:

<Ctrl-*x***>** or ^*x*	The notation **<Ctrl-***x***>** or ^*x* followed by the name of a key indicates a control character sequence. For example, **<Ctrl-c>** means that you must hold down the control key while pressing **<c>**.
<Return>	The notation **<Return>** refers to the key on your terminal or workstation that is labeled with the word ‘‘Return’’ or ‘‘Enter,’’ or with a left arrow.
Entering commands	When instructed to *enter* a command, type the command name and then press **<Return>**. For example, the instruction ‘‘Enter the **ls** command’’ means that you must type the **ls** command and then press **<Return>** (enter = type command + press **<Return>**).

Problem Reporting

If you have any problems with the software or documentation, please contact your software vendor's customer service department.

Chapter 1. DCE Subsystems

This chapter contains information regarding porting DCE 1.2.2 subsystems and APIs. It consists of the following main sections:

- Miscellaneous

  Contains information about various platform-sensitive aspects of DCE subsystem code not peculiar to any single component.

- Internationalization

  Contains information about porting the DCE internationalization mechanisms.

- Serviceability

  Contains information about porting the DCE Serviceability routines.

- DCE configuration routines

  Contains information about porting the DCE configuration routines.

1.1 Internationalization

An ‘‘internationalized’’ RPC application uses a wide variety of languages other than U.S. English. DCE 1.2.2 contains RPC runtime support for character and code set interoperability for use by internationalized RPC applications. If you are porting DCE and plan for your DCE product to support internationalized RPC applications, you must create a character and code set registry from a ‘‘template’’ character and code set registry source file that OSF supplies on the DCE source tape. The file is installed at:

/usr/lib/nls/csr/code_set_registry.txt

The code set registry template source file contains unique identifiers that OSF has assigned to the character sets and code sets that have been registered with OSF. You must edit this source file and supply the names that your platform uses to refer to these character sets and code sets, then run the **csrc** utility to generate the binary-format code set registry, which is required for porting and testing the RPC runtime extensions for

character and code set interoperability. See the chapter entitled ‘‘Configuring DCE’’ in the *OSF DCE Administration Guide — Introduction*, and the **csrc(8dce)** reference page in the *OSF DCE Command Reference* for details on the template code set registry file and how to run **csrc**. See the chapter entitled ‘‘Writing Internationalized RPC Applications’’ in the *OSF DCE Application Development Guide — Core Components* volume for more information on character sets, code sets, and RPC runtime support for internationalized RPC applications.

1.1.1 Testing and Verification

See Chapter 12 for information on the DCE Internationalization system tests.

1.2 Serviceability

The Serviceability API is a library of routines used by the DCE components (with the exception of DCE Threads) to display or write server information of various kinds. It uses message catalogs (generated by the DCE **sams** utility), but it is more than simply a message catalog-manipulation library. Serviceability is also made available for application use; this is documented in the *OSF DCE Application Development Guide — Core Components* volume.

The DCE Serviceability source code is located at:

dce-root-dir/**dce/src/dce/utils/svc**

The DCE **sams** utility source code is located at:

dce-root-dir/**dce/src/tools/sams**

1.2.1 Testing and Verification

A version of the DCE example application **timop** which uses the serviceability API can be found at

dce-root-dir/**dce/src/examples/svc/timop_svc**

The **timop_svc** application was developed mainly during the writing of the OSF DCE Application Development Guide chapter on Serviceability. Although it was not designed to be used for test purposes, it does make use of most of the serviceability routines, and it is included in the DCE 1.2.2 release as (it is hoped) a useful functional hand test for the interface. Instructions for building and running the program, as well as sample run results, can be found in:

dce-root-dir/**dce/src/examples/svc/timop_svc/README**

An additional very simple Serviceability hand test can be found at:

dce-root-dir/**dce/src/examples/svc/hello_svc**

This program, when compiled and executed, does nothing more than print a ‘‘Hello world’’ message to standard output, using the serviceability API. Unlike **timop_svc**, **hello_svc** does not require you to have a DCE cell up and running in order to successfully execute it. (It does however require you to have the DCE Application Environment installed). For further information, see:

dce-root-dir/**dce/src/examples/svc/hello_svc/README**

Chapter 2. DCE Programs

This chapter contains information about testing the following DCE 1.2.2 programs and facilities:

- **dcecp** — the DCE command program
- **dced** — the DCE daemon
- DCE ACL Facility
- DCE Backing Store Library

2.1 dcecp

In DCE 1.1 a new control program called **dcecp** was added to the DCE administrative package. This program is meant to augment the use of the existing control programs in the present release, and ultimately to replace them entirely.

2.1.1 Testing and Verification

The **dcecp** functional tests are designed to run under the TET scaffold (Test Environment Toolkit; see ‘‘Overview of TET Use’’ in Chapter 11 for general information on TET).

The **dcecp** tests can be run in two different ways:

- Use **tcc** to run a block of tests
- Run individual test files as scripts

The second method is often helpful when you are porting and want to just run specific tests without going through the overhead of running **tcc**.

The general format of running the tests under the TET scaffold is:

tcc -e functional/admin/dcecp *test_suite_name*

where *test_suite_name* is one of the following, as defined in the TET scenario file:

- **all**
- **account**
- **acl**
- **attrlist**
- **aud**
- **audevents**
- **audfilter**
- **audtrail**
- **cdsalias**
- **cdsalias_hcell**
- **clearinghouse**
- **clock**
- **directory**
- **dts**
- **endpoint**
- **group**
- **group_era**
- **hostdata**
- **ktb**
- **link**
- **log**
- **misc**
- **object**
- **org_era**
- **organization**
- **principal**
- **principal_era**
- **registry**
- **registry_one**
- **rpcentry**
- **rpcgroup**

- **rpcprofile**
- **schema**
- **secval**
- **server**
- **utc**

The **all** *test_suite_name* is used to run the entire suite of **dcecp** tests.

Most of the above suites are named for the **dcecp** object they test. The names whose meanings are not obvious have the following explanations:

*obj*_**era**	Tests manipulating Extended Registry Attributes (ERAs) on *obj* objects.
*obj*_**hcell**	Tests the **cdsalias** command on *obj* objects in a hierarchical cells environment.
misc	Tests miscellaneous, non-object **dcecp** commands such as **login**.
ktb	Tests **keytab** objects.
registry_one	Destructive registry tests. These tests should be run individually outside of the test suite.
schema	Tests the **xattrschema** object.

Within each test suite are individual test files that are used to test subcomponents. The list of these subcomponents is too lengthy to be given here, but it can be found in the tests scenario file at:

dce-root-dir/**dce/src/test/functional/admin/dcecp/tet_scen**

Tests are divided into two groups. The first group consists of negative tests. These are found in files with the **_N.tcl** suffix in their name; they are designed to supply input that generates error conditions.

The second group consists of positive tests. These are found in files with the **_P.tcl** suffix in their name; they verify the functionality of **dcecp** commands.

For more information about this file and other files used for the **dcecp** functional test suite, see ‘‘Files Used By the Tests’’. below.

For the purpose of creating the **dcecp** functional test suite, a Tcl API to the TET scaffold was added to the previously existing C, Bourne shell (XPG3) and Korn shell APIs. The TET APIs are designed to allow tests to log test information and report results to the TET journal file. The source for all the TET APIs can be found in:

dce-root-dir/**dce/src/test/tet/src/tcl/api**
dce-root-dir/**dce/src/test/tet/src/posix_c/api**
dce-root-dir/**dce/src/test/tet/src/ksh/api**
dce-root-dir/**dce/src/test/tet/src/xpg3sh/api**

For more information about TET in general, see ‘‘Overview of TET Use’’ in Chapter 11. See ‘‘Running the Tests’’, below, for details on running the **dcecp** tests.

See the ‘‘Platform Definitions and Variables’’ section earlier in this chapter for information on the **_DCECP_TEST** preprocessor variable, which must be defined when

building **dcecp** for functional testing.

2.1.1.1 Building the Tests

The current source location of the **dcecp** functional tests is:

dce-root-dir/**dce/src/tests/functional/admin/dcecp**

In order to run any of the **dcecp** functional tests, you must first build and install all the files in the following directories:

dce-root-dir/**dce/src/test/functional/admin/dcecp**
dce-root-dir/**dce/src/test/functional/admin/dcecp/lib**
dce-root-dir/**dce/src/test/tet/src/posix_c**
dce-root-dir/**dce/src/test/tet/src/tcl/api**
dce-root-dir/**dce/src/test/tools**

The tests themselves are found in:

dce-root-dir/**dce/src/test/functional/admin/dcecp/ts/***

These may be installed by subcomponent or as a whole.

To build and install the entire **dcecp** test suite (without the required TET and tools directories) under ODE, do the following:

```
cd dce-root-dir/dce/src/test/functional/admin/dcecp
build
build install_all
```

(ODE is the OSF Development Environment; for more information on it, see Chapter 12 of the DCE 1.1 *OSF DCE Porting and Testing Guide*.)

2.1.1.2 Running the Tests

Note: The **dcecp** functional tests should be run under an ordinary user login, *not* as root or any other extraordinary identity. This is because some of the tests verify functionality running unauthenticated, and invoking the tests under an ordinary login is the only way to make sure that authentication does not occur when it is not supposed to.

There are two methods for running the **dcecp** functional tests. The first is the standard approach. After installing the tests, do the following:

cd *dce-root-dir*/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT:$PATH
tcc -e functional/admin/dcecp *test_suite_name*

where *test_suite_name* is one of the test suites listed in the TET scenario file. (See ‘‘Testing and Verification’’, above, for a list of valid test suite names.)

The second method for running tests can be helpful during the development and debugging process. The individual script files found in:

$TET_ROOT/functional/admin/dcecp/ts/*

can be run by hand. To do this, set the **TET_CONFIG** environment variable to the location of the **tetexec.cfg** file, as follows:

```
setenv TET_CONFIG $TET_ROOT/functional/admin/dcecp/tetexec.cfg
```

Prior to running the tests, the **tetexec.cfg** file must be modified to reflect your local configuration.

This file defines variables that are used throughout the tests. There are only a few variables that need to be changed. The following list shows the variables that you will need to modify, and what their values should be changed to (example values are given in parentheses):

DCP_CLIENT	The principal name of the cell administrator (**cell_admin**).
DCP_CLIENT_PW	The password for **DCP_CLIENT**.
DCP_CELLNAME_ONE	The name of the cell you are in. (**/.../name.foo.com**).
DCP_CELLNAME_TWO	The name of the cell used for intercell tests (**/.../name.foo.com**).
DCP_HOSTNAME_ONE	The simple name of the host you are on (**famagusta**).
DCP_HOSTNAME_TWO	The simple name of another host in your cell (**murgatroyd**).
DCP_ROOT_CH	The name of the clearinghouse that contains the master copy of the root directory (**machine_ch**).
DCP_INTERCELL_AVAIL	Do you want to run the intercell tests? (**0** or **1**).
DCP_SR_IP_ADDR	The IP address of the machine that you are running the tests on.
DCP_SR_STR_BINDING	A protocol sequence followed by the above IP address. Separated by a colon ‘‘**:**’’. (**ncacn_ip_tcp:127.0.0.1**).

After you have changed the above variables’ values as appropriate, **cd** to the directory that contains the **dcecp** functional test that you wish to run. For example:

```
cd $TET_ROOT/functional/admin/dcecp/ts/dts
```

You may now execute the test script by hand:

```
dts_modify_P.tcl
```

After the test completes, the results will be left in the file **tet_xres** in the current directory. Note that each test file invocation will overwrite this file, so you should either

view or save its contents, as desired, after each test run.

Note that the above sequence of commands assumes that the tests have been installed in their default location (by ODE) and that you wish to run them from that location. However, the

dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2**

test tree is self-contained (insofar as the tests and TET are concerned), and can be copied to any other preferred location on your system, and executed from there. If you do this, the first step given above becomes the following three steps:

cd *dce-root-dir*/**dce/install/***platform*/**dcetest**
cp -r dce1.2.2 *your_test_tree*
cd *your_test_tree*

If you execute the tests from their default installed location, test results will be found at:

dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet/functional/admin/dcecp/results/***pass_nr*/**journal**

where *pass_nr* is the number of the test iteration whose results are being written. The **results** subdirectory is created by TET in the subdirectory specified by **-e** to the **tcc** command, as shown above. For further information about TET output, see ‘‘Overview of TET Use’’ in Chapter 11.

For information on how to run specific tests within a suite, see the following section.

2.1.1.3 Special Requirements for Running the Tests

All of the **dcecp** tests must be run in a fully functioning DCE cell with the following specific characteristics:

- There must be a **dced** running on the current host.
- A security master and a CDS server must be running in the cell.
- The appropriate helper programs (such as a CDS advertiser (**cdsadv**) and a CDS clerk (**cdsclerk**)) must be running on the host on which the tests are run.

In addition, the following **dcecp** tests have the following special requirements:

- **clearinghouse** and **directory** tests

  These tests must be run on a machine that is running a CDS server.

- **dts** and **acl** tests

  These tests must be run on a machine that is running a DTS server.

- **audit** tests

  These tests must be run on a machine that is running an audit daemon, which must be started with the **-a** option.

- **registry** tests

  These tests must be run in a cell that has a security replica.

- **registry_one** tests

  Must be run on a machine on which a security replica is running. In addition, the tests must be run one at a time, and the security state of the cell has to be restored between each run.

2.1.1.4 Files Used By the Tests

The following files are used by TET when invoked to run **dcecp** functional tests:

- The TET configuration file

  dce-root-dir/**dce/src/test/admin/dcecp/tetexec.cfg**

  is where global variables should be defined for all **dcecp** tests.

- The TET scenario file

  dce-root-dir/**dce/src/test/admin/dcecp/tet_scen**

  is where TET gets the list of tests it must run for a specific test component. If you wish to run a specific ‘‘negative ACL modify’’ tests, you should edit the following section in the scenario file:

  ```
  "Starting negative ACL MODIFY tests"
  /ts/acl/acl_modify_N.tcl
  ```

  so that it reads:

  ```
  "Starting negative ACL MODIFY tests"
  /ts/acl/acl_modify_N.tcl{28-30}
  ```

  or:

  ```
  "Starting negative ACL MODIFY tests"
  /ts/acl/acl_modify_N.tcl{28,29,30}
  ```

  Either version will result in only tests 28, 29, and 30 in the negative ACL modify suite to be executed when the ACL test suite is run by invoking TET as follows:

  tcc -e admin/dcecp acl

2.1.1.5 Tcl Tests

The

dce-root-dir/**dce/src/test/admin/tcl_dce**

subdirectory contains a set of validation tests for the Tcl commands. See the **README** file located in this directory for instructions on how to build and run these tests.

1.2.2,Add note on LANG (start)

Note that before running the Tcl functional tests, you must set the **LANG** environment variable to ‘‘C’’.

1.2.2,Add note on LANG (end)

2.1.1.6 Hand Tests for dcecp registry set Functionality

dcecp contains support for several **sec_admin** commands, as follows:

TABLE 2-1

New dcecp Command	Equivalent sec_admin Command
registry set <replica_name>	change_master -to <replica_name>
registry set <replica_name> -slave	become -slave
registry set <replica_name> -master	become -master

This section contains procedures for hand testing this **dcecp** functionality.

To execute the test procedures successfully, the tester must first set up a master and at least one slave replica, as follows:

```
dcecp> registry cat
/.../cellname/subsys/dce/sec/spartacus
/.../cellname/subsys/dce/sec/caesar
```

where, in the example given here, **spartacus** is the name of a slave machine in *cellname* before the tests are performed, and **caesar** is the name of the master machine in the same cell.

Test 1: Bind to master and change master to slave.

The test is performed as follows:

```
dcecp> registry set subsys/dce/sec/spartacus
```

You should now be able to execute the **registry dump** command and get results similar to the following:

```
dcecp> registry dump
{name /.../cellname/subsys/dce/sec/caesar
{type slave}
{cell /.../cellname}
{uuid 08c199b6-b836-11cd-94b4-0800092734a4}
{status enabled}
{lastupdtime 1994-08-22-13:54:07.000-04:00I-----}
{lastupdseq 0.1100}
{addresses {ncacn_ip_tcp nnn.nnn.n.nnn}
           {ncadg_ip_udp nnn.nnn.n.nnn}}
{masteraddrs {ncacn_ip_tcp nnn.nnn.n.nnn}
             {ncadg_ip_udp nnn.nnn.n.nnn}}
{masterseqnum 0.1101}
{masteruuid 2e7ac32b-b846-11cd-a8cf-0000c0239a70}
{version secd.dce.1.2.2}

{name /.../cellname/subsys/dce/sec/spartacus}
{type master}
{cell /.../cellname}
{uuid 2e7ac32b-b846-11cd-a8cf-0000c0239a70}
{status enabled}
{lastupdtime 1994-08-22-14:10:25.000-04:00I-----}
{lastupdseq 0.1101}
{addresses {ncacn_ip_tcp nnn.nnn.n.nnn}
           {ncadg_ip_udp nnn.nnn.n.nnn}}
{masteraddrs {ncacn_ip_tcp nnn.nnn.n.nnn}
             {ncadg_ip_udp nnn.nnn.n.nnn}}
{masterseqnum 0.1101}
{masteruuid 2e7ac32b-b846-11cd-a8cf-0000c0239a70}
{version secd.dce.1.2.2}
{updseqqueue {0.1099 0.1101}}
```

Test 2: Change replica to a slave replica.

The test is performed as follows:

```
dcecp> registry set subsys/dce/sec/caesar -slave
```

You should now be able to execute the **registry dump** command and get results similar to the following:

```
dcecp> registry dump
{name /.../cellname/subsys/dce/sec/spartacus}
{type slave}
{cell /.../cellname}
{uuid 2e7ac32b-b846-11cd-a8cf-0000c0239a70}
{status enabled}
{lastupdtime 1994-08-22-12:26:39.000-04:00I-----}
{lastupdseq 0.1091}
{addresses {ncacn_ip_tcp nnn.nnn.n.nnn}
           {ncadg_ip_udp nnn.nnn.n.nnn}}
```

```
{masteraddrs {ncacn_ip_tcp nnn.nnn.n.nnn}
             {ncadg_ip_udp nnn.nnn.n.nnn}}
{masterseqnum 0.1091}
{masteruuid 08c199b6-b836-11cd-94b4-0800092734a4}
{version secd.dce.1.2.2}

{name /.../cellname/subsys/dce/sec/caesar}
{type slave}
{cell /.../cellname}
{uuid 08c199b6-b836-11cd-94b4-0800092734a4}
{status enabled}
{lastupdtime 1994-08-22-12:26:39.000-04:00I-----}
{lastupdseq 0.1091}
{addresses {ncacn_ip_tcp nnn.nnn.n.nnn}
           {ncadg_ip_udp nnn.nnn.n.nnn}}
{masteraddrs unknown}
{version secd.dce.1.2.2}
```

Test 3: Change replica to a master replica.

The test is performed as follows:

```
dcecp> registry set subsys/dce/sec/spartacus -master
```

You should now be able to execute the **registry dump** command and get results similar to the following:

```
dcecp> registry dump
{name /.../cellname/subsys/dce/sec/spartacus}
{type master}
{cell /.../cellname}
{uuid 2e7ac32b-b846-11cd-a8cf-0000c0239a70}
{status enabled}
{lastupdtime 1994-08-22-14:26:45.000-04:00I-----}
{lastupdseq 0.1104}
{addresses {ncacn_ip_tcp nnn.nnn.n.nnn}
           {ncadg_ip_udp nnn.nnn.n.nnn}}
{masteraddrs {ncacn_ip_tcp nnn.nnn.n.nnn}
             {ncadg_ip_udp nnn.nnn.n.nnn}}
{masterseqnum 0.1104}
{masteruuid 2e7ac32b-b846-11cd-a8cf-0000c0239a70}
{version secd.dce.1.2.2}
{updseqqueue {0.1103 0.1104}}

{name /.../cellname/subsys/dce/sec/caesar}
{type slave}
{cell /.../cellname}
{uuid 08c199b6-b836-11cd-94b4-0800092734a4}
{status enabled}
{lastupdtime 1994-08-22-14:26:45.000-04:00I-----}
{lastupdseq 0.1104}
```

```
{addresses {ncacn_ip_tcp nnn.nnn.n.nnn}
           {ncadg_ip_udp nnn.nnn.n.nnn}}
{masteraddrs {ncacn_ip_tcp nnn.nnn.n.nnn}
             {ncadg_ip_udp nnn.nnn.n.nnn}}
{masterseqnum 0.1104}
{masteruuid 2e7ac32b-b846-11cd-a8cf-0000c0239a70}
{version secd.dce.1.2.2}
```

2.2 dced

This and the following sections contain testing information about **dced**, the DCE Host Daemon, which replaces the (pre-DCE 1.1) RPC daemon (**rpcd**) and **sec_clientd**.

2.2.1 Testing and Verification

The installed location of the **dced** tests is:

your_install_path/**test/tet/functional/admin/dced**

which by default is:

dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet/functional/admin/dced**

2.2.1.1 Running the Tests

Before attempting to run the tests, you must edit the

dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet/functional/admin/dced/tetexec.cfg**

file and set the values of the following parameters:

T_CELL_ADMIN This should be the value of your Cell Adminstrator's principal name (the default value when setting up the cell with **dce_config** is **cell_admin**).

T_CELL_ADMIN This should be your Cell Administrator principal's password.

TET_SIG_IGN This should be the (system-dependent) value of the **SIGVTALRM** signal, defined in

/usr/include/sys/signal.h

for your platform.

To run the tests, do the following:

1. Make sure **dced** is running.
2. **dce_login** as a privileged user.
3. Change directory to the installed test subtree:

   cd *your_install_path*/**test/tet/**
   set TET_ROOT ‘pwd‘
   setenv PATH $TET_ROOT/bin:$PATH

4. To execute all of the tests, do:

   tcc -e functional/admin/dced

5. To execute a test suite, do:

   tcc -e functional/admin/dced *test_suite_name*

   where *test_suite_name* is one of the suites listed in the TET scenario file located at:

   dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet/functional/admin/dced/tet_scen**

   The existing test suites are:

   - **binding**
   - **common**
   - **hostdata**
   - **keytab**
   - **secval**
   - **srvrconf**
   - **srvrexec**

   You can also specify a *test_suite_name* of **all**, which will cause all of the suites to be run.

   If you execute the tests from their default installed location, test results will be found at:

   dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet/functional/admin/dced/results/***pass_nr*/**journal**

   where *pass_nr* is the number of the test iteration whose results are being written. The **results** subdirectory is created by TET in the subdirectory specified by **-e** to the **tcc** command, as shown above. For further information about TET output, see ‘‘Overview of TET Use’’ in Chapter 11.

2.2.2 dced Runtime Output and Debugging Output

The **dced** 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 **dced** via serviceability.

2.2.2.1 Normal dced Server Message Routing

There are several ways to control normal **dced** 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.
- Via environment variables (such as **SVC_ERROR**).
- Via command line options

The **svcroute(5dce)** reference page describes most of these methods; the **dced(8dce)** reference page should be referred to for the last method. Routing of an already-started **dced**'s messages can be controlled through the **dcecp log** object. See the **log.8dce** reference page in the *OSF DCE Command Reference* for further information.

2.2.2.2 Debugging Output

Debugging output from **dced** can be enabled (provided that **dced** 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_DHD_DBG** environment variable, before bringing up **dced**. 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* 6
[*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

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 **dced**, the sub-components are as follows:

aclmgr	The **dced** ACL managers
xattrschema	The **dced** attribute service
general	General **dced** facilities
hostdata	The **dced** hostdata service
rkeytab	The **dced** rkeytab service
secval	The **dced** secval service
srvrconf	The **dced** srvrconf service
srvrexec	The **dced** srvrexec service
locks	The **dced** lock manager
endpoint	The **dced** endpoint mapper service

For example, the string

```
dhd:*.1,general.3:TEXTFILE.50.200:/tmp/dced_LOG
```

sets the debugging level for all **dced** sub-components (*except* **general**) at 1; **general**'s level is set at 3. All messages are routed to **/tmp/dced_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 **dced** serviceability messages, and the sub-component list, can be found in the **dced** sams file, at:

dce-root-dir/**dce/src/admin/dced/idl/dhd.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''.

2.3 DCE ACL Facility and Backing Store Library

2.3.1 Testing and Verification

The source code for the functional tests for the DCE Backing Store library is located in the

dce-root-dir/**dce/src/test/dce/utils/acldb/ts/db**

subdirectory. The following tests are supplied:

- **dce_db_open**
- **dce_db_close**
- **dce_db_fetch**
- **dce_db_store**
- **dce_db_misc**
- **dce_db_delete**
- **dce_db_iter**

These programs test the DB APIs implied by their names. They are standalone (no server) tests which create, manipulate, and delete backing stores in the current directory.

The source code for the functional tests for the DCE ACL facility are located in the

dce-root-dir/**dce/src/test/dce/utils/acldb/ts/acl**

subdirectory. The following tests are supplied:

- **rdacl_svr_opers**

  Tests the **rdacl_*** routines.

- **dce_acl_perm_fcns**

  Tests the **dce_acl_*** permissions-related routines.

- **dce_acl_conv_fcns**

  Tests the **dce_acl_*** convenience routines.

- **acl_setup**

  This module implements the setup routines for the DCE ACL functional tests. FVT's.

Each of the test suites attempts to add a principal and account, called **test_princ1**, which they need. The setup script logs in as **cell_admin** and sets an ACL on

/**.:/hosts**/*host_name*

in preparation for the tests.

2.3.1.1 Running the Tests

To run the Backing Store or ACL library tests, do the following:

cd *dce-root-dir*/**dce/install/**_platform_/**dcetest/dce1.2.2/test/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/bin:$PATH
tcc -e functional/dce/utils/acldb *test_suite_name*

where *test_suite_name* is one of the suites listed in the TET scenario file located at:

dce-root-dir/**dce/install/**_platform_/**dcetest/dce1.2.2/test/tet/functional/dce/utils/acldb/tet_scen**

Note that the above sequence of commands assumes that the tests have been installed in their default location (by ODE) and that you wish to run them from that location. However, the

dce-root-dir/**dce/install/**_platform_/**dcetest/dce1.2.2**

test tree is self-contained (insofar as the tests and TET are concerned), and can be copied to any other preferred location on your system, and executed from there. If you do this, the first step given above becomes the following three steps:

cd *dce-root-dir*/**dce/install/**_platform_/**dcetest**
cp -r dce1.2.2 *your_test_tree*
cd *your_test_tree*

If you execute the tests from their default installed location, test results will be found at:

dce-root-dir/**dce/install/**_platform_/**dcetest/dce1.2.2/test/tet/functional/dce/utils/acldb/results/**_pass_nr_/**journal**

where *pass_nr* is the number of the test iteration whose results are being written. The **results** subdirectory is created by TET in the subdirectory specified by **-e** to the **tcc** command, as shown above. For further information about TET output, see ''Overview of TET Use'' in Chapter 11.

Chapter 3. DCE Threads

DCE Threads is a POSIX 1003.4a-compliant threading service which allows an application to create separate threads of execution within a process. These threads have low startup overhead and can share data among themselves.

The DCE Remote Procedure Call (RPC) service uses threads to let servers communicate with multiple clients concurrently. Many of the server functions in DCE use threads to allow simultaneous communication with multiple clients and for the concurrent processing of data while waiting for I/O operations to complete.

3.1 Testing and Verification

Nineteen types of DCE Threads tests are shipped with DCE. These tests are described below.

3.1.1 Installing Threads Functional Tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config** will install the tests you select at the path you specify, and will create a softlink (called **/dcetest/dcelocal**) to that location.

The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the sections below, which refer to the tests' source or build locations.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the location of the built test tree). If you are installing the DCE

Threads functional tests, you should give the pathname of the DCE **obj** tree, *not* the **install** tree, even though the prompt message contains as an example an install tree pathname. For example:

Figure 3-1. Supplying Threads Test Install-from Location

```
Location of DCE Test Install Binaries


 1. Filesystem
 2. Media

98. Return to previous menu
99. Exit

selection:  1

Enter the full path to the DCE binary install tree.
This will be the directory that contains the
.../<BUILD>/install/<machinetype>/dcetest/dce1.2.2
directory:  /myproject/dce/dce1.2.2/obj
```

Thus, **dcetest_config** will install the DCE Threads functional tests at:

/dcetest/dcelocal/test/threads/

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* (if you choose) functional tests with **dcetest_config**; for test configuration and execution you must follow the instructions in the sections below.

Refer to Chapter 11 of this guide for further information on using **dcetest_config**.

3.1.2 Testing Dependencies

Several of the test cases require the presence of Berkeley I/O, particularly the **ftime( )** system call. If you are porting to an operating system that is not compatible with the Berkeley Software Distribution (BSD) UNIX, you must link a compatibility library with the test cases for them to work properly. The test cases also make use of the ANSI C function **atexit( )**. If your system does not support this function, you will have to provide an equivalent.

Note that one of the Threads test cases (**cuvb_nbi_005**, which tests **cma_fcntl( )**) uses **lockd** to create a write lock for a file which it uses. If the file is NFS-mounted, the test will hang forever at this point if **lockd** and **statd** are not running on both the local and

remote machines. This means that you may have to make sure that the test is run on only one machine if your platform does not support **lockd** and **statd** (which is the case with the OSF/1 platform).

Note: Both reference platforms require that a compatibility library be used.

3.1.3 Threads Test Case Categories

The following sections describe categories of testing done for threads, and a brief description of the coverage of each category.

3.1.3.1 Test Case Naming Format

DCE Threads tests are named using the following format:

4 alphabetic characters,
a dash,
3 alphabetic characters,
a dash,
3 alphabetic characters.

For example,

abcd-efg-hij

is a test name where each of the following characters represents a certain type of test:

a	C for tests using the CMA Threads interface P for tests using the PThreads (POSIX) interface E for ‘‘extended’’ test, which may apply to Pthreads entry points, or to CMA threads entry points that are visible to them. Tests starting with E examine thread operation under error, exception, or excess conditions, such as writes to broken pipes, or situations where thread operations exceed process limits.
b	V for VMS specific test U for U*IX specific test R for reference implementation (portable) test
c	P for performance test Q for performance test using internal interfaces R for regression test U for unit test using internal interfaces V for validation test
d	B for batch mode test I for interactive test (for example, needs user input)

e, **f**, and **g**

The test topic. Tests having more than one topic have 3 additional characters (for example, **abcd-efg-efg-hij**). Topics have the following meanings:

ALT	Alerts
AQO	Atomic queue operation
ATT	Attributes objects
CAN	Pthread cancel
CVX	Condition variable operations, including barrier operations
ERR	Error reporting
EXC	Exceptions
HAN	Handles
INI	One-time initialization
MUT	Mutex operations
NBI	Nonblocking (UNIX) I/O
OBJ	Dynamic object management
PTC	Per threads context
SAM	Sample (or example) programs
SIG	U*IX signal handling
STK	Stacks
THD	Threads operations
TIM	Timer operations
WRP	Unix wrapper routines

h, **i**, and **j**

The sequence number of each test. Tests whose names differ only by this number typically exercise the same operations. However, they usually are not versions or revisions of each other, and may exercise the same operations quite differently.

For example, **crub_err_001** is an actual test name, specifying that it:

- is a CMA test
- is portable
- is a unit test that uses internal interfaces
- is a batch mode test
- is testing error reporting
- is number 1 in the sequence of tests of this kind.

3.1.3.2 Test Topic Abbreviations

The test topic abbreviations, represented by **efg** in the preceding test name example, specify test cases with the following functions:

Alerts (ALT)	These test cases attempt to alert threads with and without exception handlers and verify correct thread exit or handling. They alert compute-bound threads and threads in a **timed_wait** state and also ensure that deferred and synchronous alerts work.
Atomic Queue Operation (AQO)	Exercise the Atomic Queue Operations of the CMA library services. The operations are currently available only on VMS and are not part of DCE.
Attributes Objects (ATT)	Verify that attribute objects can be created and deleted for both default and specified values. They check deferred delete, cache sequencing, and cache reclamation and also verify locking during attribute deletion.
Pthread Cancel (CAN)	Test the functionality of the thread cancellation mechanism that allows a thread to terminate the execution of any other thread in the process in a controlled manner.
Condition Variables (CVX)	Measure wait/signal performance time when condition variables are used. They also verify timed wait functionality.
Error Reporting (ERR)	Ensure that calling **cma__error** or **cma__bugcheck** causes process termination and confirm the ability of the functions to raise warning and failure exceptions. Error return values are also ensured as per-thread.
Exception Handling (EXC)	Force various exceptions, including address and status exceptions, which are handled per-thread.
Handles (HAN)	Verify that the thread handle size is static.
One Time Initialization (INI)	Use one-time initialization and ensure that it executes only once.
Mutex Operations (MUT)	Lock and unlock a mutex, both with a single thread and with multiple threads, while measuring elapsed time. Threads attempt to lock and unlock mutexes to which they do not have access, as well as friendly mutexes. They also test nested locks and use global locks to gain exclusive access to libraries.
Nonblocking UNIX I/O (NBI)	Test the wrapper routines for the UNIX I/O system calls. These wrapper routines provide thread-synchronous I/O through the use of select and nonblocking I/O mode. This category verifies system calls such as **open( )**,

	close( ), and **select( )**. File descriptors need to be shared between threads.
Object Management (OBJ)	Test management of various dynamically allocated data objects, such as thread control blocks, mutexes, and condition variables.
Per Thread Context (PTC)	Use a PTC destructor that locks a TCB, which ensures proper behavior. A batch of threads is created with a context associated with them, and proper behavior of yields is verified.
Sample Program (SAM)	Demonstrate the use of threads. It creates 10 threads, terminates the odd-numbered threads with an alert, and allows even-numbered threads to terminate normally.
Signal Handling (SIG)	Test asynchronous, synchronous, terminating, and nonterminating signals. The tests send all possible signals and verify correct behavior.
Stack Handling (STK)	Test the stack management services. Stacks are created, deleted, reassigned, alternated, and shared. Multithreaded operations are used on stacks. One test case also checks limits by touching a stack guard page to simulate a stack overflow by a thread.
Thread Operations (THD)	Measure thread creation time, thread yield performance time, and time elapsed during a context switch. They also measure the time-slicing algorithm performance and ensure that **thread_exit** operations affect the current thread only. Use varying process priorities and policies when creating threads.
Timing (TIM)	Verify timed waits.
UNIX Wrapper Routines (WRP)	Test the implementation of CMA wrapper routines around certain UNIX system calls, particularly I/O calls and **fork( )**.

3.1.4 Test Case Execution

To execute the test cases, no parameters are required. A shell script, **runtest**, is provided for serial execution. This script can be found in the

dce-root-dir/**dce/obj/***machinetype*/**test/threads**

directory, where *machinetype* is your system type (for example, **rios** or **mips**). The test cases can be executed individually by entering the test case name on the command line.

Note: Any tests with ‘‘i’’ as the fourth character (such as **crvi_sig_003**, **cuvi_nbi_004**, and **puvi_nbi_004**) are not executed by **runtest** because they are interactive and must be invoked manually.

3.1.5 Test Case Results

Standard output for a successful execution includes a PASSED message. Some test cases, however, deliberately cause abnormal program termination, and may cause core dumps. The following test cases have nonstandard output:

- **crub_err_001**, **crub_err_002**, **crub_err_003**

  These tests correctly return a core dump.

- **crvb_exc_001**

  The first 10 loops of this test complete with the message

  ```
  Normal fall through ENDTRY.
  ```

  The eleventh loop correctly terminates with a core dump.

- **crvb_sam_001**, **prvb_sam_001**

  Even-numbered threads exit normally; odd-numbered threads exit prematurely due to an alert. The test then prints

  ```
  Program over.
  ```

  Note: In DCE 1.0.1, the **prvb_sam_001** test does not output the normal header and trailer lines (START and PASSED). However, the test does run correctly.

- **crvb_thd_007**

  This test generates reports that must be verified manually for scheduling accuracy.

- **crvi_exc_001**, **prvi_exc_001**

  These tests require that the **<Ctrl-Y>** debug sequence be entered during test case execution.

3.1.6 Test Plans

Refer to Chapter 1 of the *OSF DCE Release Notes* for the location of the DCE test plans on the DCE distribution tape.

3.2 Debugging DCE Threads

In the course of porting DCE Threads, you will probably need to debug applications that make use of them. These applications could be threads functional tests, DCE component programs, or applications of your own design. Because any application that uses DCE

Threads maintains execution state for multiple threads of execution, it will probably confuse your current debugger, unless the debugger has already been extended to understand the current DCE Threads implementation.

The amount of work necessary to extend your debugger to work correctly with DCE Threads applications will naturally depend on which one you use. Essentially, the debugger must relate the execution state of the currently-running thread to the tables internal to DCE Threads thatprovide information on all threads. Ideally, the debugger should also permit you to find out information on threads that are not currently running.

A simple example of such capabilities is described in the following section. It consists of additions that can be made to a standard, non-thread-aware version of **gdb**, in order to find out information about the currently-executing thread in a DCE Threads application.

3.2.1 Debugging with gdb

gdb is not aware of DCE Threads, how threads affect the stack, multiple contexts, or breakpointing in a particular thread. However, it is possible to find out which thread you are currently executing in with **gdb**. Calling the internal DCE Threads routine:

```
cma__get_self_tcb()
```

will return a pointer to the current thread's TCB (thread control block).

The ''.gdbinit File'' section below contains a listing for a **.gdbinit** file that contains three commands for identifying the currently executing thread.

The command **pthd** uses a fixed offset into the TCB to print the thread's ''sequence,'' or identifier. This integer identifier is the number output by DCE Threads to identify the thread to which a particular error or status message applies. The **pthd** command is probably the one you will use the most from this package.

Note: This fixed offset may be DCE Threads-revision dependent, but is not likely to cause problems in the near future.

The **pthdx** command prints this same thread sequence integer, but requires the module to have included **<cma.h>** and **<cma_tcb.h>** and to be compiled with symbol information (**-g**). This is a cleaner way to use the package, but most modules will not have **<cma_tcb.h>** included.

Finally, the command **ptcb** simply prints a pointer to the TCB. Like **pthd**, this command does not require your program to be built with any CMA symbols.

3.2.1.1 Breakpointing in a Particular Thread

You can use a **gdb condition** on a breakpoint to stop on a particular statement in a particular thread. To do this easily, you should include **<cma.h>** and **<cma_tcb.h>** in the module. For example, doing the following:

break 180
condition 1 (cma__get_self_tcb ( ) -> header.sequence == 15)

will stop execution on Line 180 of the current source file, whenever thread 15 is executing. (It is not possible to write a **.gdbinit** macro to do this breakpointing because **gdb** macros are not able to take arguments, such as line number or thread ID.)

3.2.1.2 The .gdbinit File

Put the following into a file called **.gdbinit** in your home directory:

```
define pthd
x/x (cma__get_self_tcb( ) + 8)
end

document pthd
Prints the CMA thread identifier in the TCB in a program
compiled without debug symbols.

NB: This command may be CMA rev dependent!!

end

define pthdx
print ((cma__t_int_tcb *) cma__get_self_tcb( )) -> header.sequence
end

document pthdx
Prints the CMA thread identifier in the TCB
end

define ptcb
print/a cma__get_self_tcb( )
end

document ptcb
Prints the address of this thread's TCB
end
```

3.2.1.3 Debugging Shared Object Core Files

One approach to the problem of debugging core files generated with shared objects is as follows. Begin by invoking **gdb**:

% gdb *program_to_be_debugged* **core**

(If the application dumped core while executing code in a shared library routine, **gdb** will at this point incorrectly report the name of the routine.) Continue as follows:

```
(gdb) break main
(gdb) run
(gdb) kill
Kill the inferior process? (y or n) y
(gdb) where
```

—and at this point a correct backtrace will be produced.

While this may not be the best solution to the problem of debugging with shared objects, running the application and breaking at **main** does allow **gdb** to build the shared object symbol tables needed for a backtrace from the core file.

3.2.2 Debugging with dbx

This section contains code for three DCE Threads-aware **dbx** commands for Ultrix, implemented as **dbx** scripts. These **dbx** scripts will allow you to display the call frames of each DCE thread in a process. You can also continue execution after doing this.

Note: To use these scripts to debug a DCE application, you must have built the application with a DCE Threads library with symbols (that is, with the **-g** flag), and you must use **dbx**.

Following is the code for **dbx_cma_stack_dump**:

```
#
#
set $dbxtcb = (struct CMA__T_INT_TCB *)((int)($dbxqueue) - \
              (int)(&(((struct CMA__T_INT_TCB*)0)->threads)))
#
set $dbxsp = ((struct CMA__T_INT_TCB *)$dbxtcb).static_ctx.sp

#>>>
#>>> The following numbers obtained from /usr/include/setjmp.h
#>>>
assign $s0 = *($address)($dbxsp + 19 * 4)
assign $s1 = *($address)($dbxsp + 20 * 4)
assign $s2 = *($address)($dbxsp + 21 * 4)
assign $s3 = *($address)($dbxsp + 22 * 4)
assign $s4 = *($address)($dbxsp + 23 * 4)
assign $s5 = *($address)($dbxsp + 24 * 4)
assign $s6 = *($address)($dbxsp + 25 * 4)
assign $s7 = *($address)($dbxsp + 26 * 4)
assign $s8 = *($address)($dbxsp + 33 * 4)
assign $ra = *($address)($dbxsp + 34 * 4)
```

```
assign $pc = *($address)($dbxsp + 34 * 4)

assign $sp = $dbxsp + 332
#>>> 332 should be (_JBLEN = 84) * 4

where
set $dbxqueue = ((struct CMA__T_QUEUE *)$dbxqueue)->flink
#
#
```

Following is the code for **dbx_cma_stack_dump_init**:

```
#
#
set $dbxhpc = $pc
set $dbxhsp = $sp
set $dbxhs0 = $s0
set $dbxhs1 = $s1
set $dbxhs2 = $s2
set $dbxhs3 = $s3
set $dbxhs4 = $s4
set $dbxhs5 = $s5
set $dbxhs6 = $s6
set $dbxhs7 = $s7
set $dbxhs8 = $s8
set $dbxhra = $ra
set $dbxptr = (&cma__g_known_threads.queue)
set $dbxqueue = ((struct CMA__T_QUEUE *)$dbxptr)->flink
set $dbxthdumpinit = 1;
#
#
```

Following is the code for **dbx_cma_stack_dump_restore**:

```
#
#
assign $pc = $dbxhpc
assign $sp = $dbxhsp
assign $s0 = $dbxhs0
assign $s1 = $dbxhs1
assign $s2 = $dbxhs2
assign $s3 = $dbxhs3
assign $s4 = $dbxhs4
assign $s5 = $dbxhs5
assign $s6 = $dbxhs6
assign $s7 = $dbxhs7
assign $s8 = $dbxhs8
assign $ra = $dbxhra
set $dbxthdumpinit = 0
#
#
```

3.2.2.1 Description of dbx Commands

Following is a description of what each of the three commands will do:

- **dbx_cma_stack_dump_init**

  dbx_cma_stack_dump_init will save some context and setup a pointer to the DCE Thread control block linked list. It does not display anything.

- **dbx_cma_stack_dump**

  dbx_cma_stack_dump will dump the stack of a thread using the **dbx** ‘‘where’’ command. It will then increment the pointer to the next thread control block. Running **dbx_cma_stack_dump** again will result in this thread’s stack being dumped and the pointer being set to point to the next thread control block. The thread control block linked list is circular: If executing **dbx_cma_stack_dump** causes numerous simultaneous memory violations, this means that the pointer has looped around to the front of the list. A subsequent invocation of **dbx_cma_stack_dump** will then display the first thread on the list again, and so on.

- **dbx_cma_stack_dump_restore**

  dbx_cma_stack_dump_restore will restore the context saved in **dbx_cma_stack_dump_init**, thus allowing you to use the **dbx** ‘‘continue’’ command.

3.2.2.2 Example

The following sample command lines are excerpted from a possible **dbx** session, and demonstrate how the three scripts should be invoked:

```
dbx> record output cma_dbx_stack_dump.log

dbx> source <location>/cma_dbx_stack_dump_init

dbx> source <location>/cma_dbx_stack_dump

      ......

dbx> source <location>/cma_dbx_stack_dump

      ......

dbx> source <location>/cma_dbx_stack_dump

      ......

dbx> source <location>/cma_dbx_stack_dump
```

.....

dbx> source <*location*>**/cma_dbx_stack_dump_restore**

dbx> continue

Chapter 4. DCE Remote Procedure Call

4.1 Overview

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.

4.2 Setup, Testing, and Verification

The following types of RPC test cases are shipped with DCE:

- IDL compiler tests (for testing compiled stubs)
- RPC application tests
- KRPC application tests
- RPC runtime library and IDL compiler tests

Before running the RPC runtime library and IDL compiler Name Service Interface (NSI) test cases, you must configure the namespace and start the namespace daemon and clerk. See the section on CDS setup in Chapter 5 of this guide for more information on configuring and starting CDS.

Before running the RPC runtime library and IDL compiler RPC authentication test cases, the DCE Security Service must be configured properly. See the section on DCE Security Service setup in Chapter 8 of this guide for more information on configuring and starting DCE Security Service.

Note: These setup steps are not required prior to running the IDL compiler tests. They may be tested once their code has been built.

4.2.1 Installing RPC Functional Tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config** will install the tests you select at the path you specify, and will create a softlink (called **/dcetest/dcelocal**) to that location. The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the sections below, which refer to the tests' source or build locations.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the final location of the built test tree). For the RPC functional tests, this path should be the location, on your machine, of:

dce-root-dir/**dce/install**

—which is the DCE **install** tree (for more information on the structure of the DCE tree, see Chapter 3 of the *OSF DCE Release Notes*).

Thus, **dcetest_config** will install the RPC functional tests at:

/dcetest/dcelocal/test/rpc/

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* (if you choose) functional tests with **dcetest_config**; for test configuration and execution you must follow the instructions in the sections below.

Refer to Chapter 11 of this guide for further information on using **dcetest_config**.

4.2.2 RPC Setup

The following steps are necessary in order to run the **perf** and **v2test** tests in normal configuration (that is, using the namespace to handle binding information). If you are testing only with full string bindings, the following steps are not necessary.

To configure RPC for OSF™ DCE Version 1.2.2 testing, do the following:

1. Make sure that

   /opt/dce1.2.2/etc/cds_attributes

   is available from the DCE CDS component.

2. Make sure that the **dced** endpoint map service is running.

3. You can optionally configure DCE CDS for **rtandidl** name service tests and DCE Security Service for authenticated RPC testing. For more information on configuring these components, see the sections on component setup in the CDS and Security Service chapters of this guide.

4.2.3 RPC Application Tests

The following test cases are shipped with DCE to test the user-mode version of RPC:

- **perf**
- **v2test**

The source code for these test cases can be found in the **perf** and **v2test_lib** subdirectories of the

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

directory of the DCE source tree.

Both of these test cases let you test authenticated remote procedure calls. However, running authenticated RPC requires special configuration of both the client and server machines. See Chapter 8 of this guide for information on how to perform this configuration.

4.2.3.1 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/dce1.2.2/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.

4.2.3.1.1 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.

4.2.3.1.2 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.

4.2.3.1.3 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:

Nr	**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.

4.2.3.2 The v2test Testcase

The **v2test** test suite tests the underlying packet-handling routines of the RPC runtime library. You must start the **v2server** program as one process and then start the **v2client** program as another process before running the **v2test** test suite. These processes can be run on the same host or on different hosts as long as the server process is started first. The **v2server** and **v2client** can be found in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/rpc/runtime/v2test_lib**

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

dce-root-dir/**dce/src/test/rpc/runtime/v2test_lib**

directory in the source tree.)

Essentially, the **v2test** bypasses the IDL stubs to test parts of the underlying RPC runtime. The following two scripts:

- **v2test_tcp.sh**
- **v2test_udp.sh**

contain useful test scenarios.

Note: It is possible to successfully pass illegal combinations of arguments to the **v2test**s; the tests should therefore be used carefully.

To test using the **v2test** suite, make a number of remote procedure calls from the **v2client** to the **v2server**. The **v2server** waits for remote procedure calls from the **v2client** and then gives a response. The **v2server** then prints messages that give the results of the

remote procedure call. To fully test using **v2test**, use different combinations of **v2server** and **v2client** testing options and observe the resulting messages.

To start the server, enter

v2server 1 ncadg_ip_udp

or:

v2server 1 ncacn_ip_tcp

at the command line. A message similar to the following 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 port number the server is listening to.

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

v2client io ncadg_ip_udp:*ip_addr[port]* 10 17 132 0

or:

v2client io ncacn_ip_tcp:*ip_addr[port]* 10 17 132 0

at the command line, where *ip_addr* is the IP address of the server (printed out when you started **v2server**) and *port* is the port number that the server is listening to (also printed out when you started **v2server**).

You can get help messages on how to invoke both the **v2server** and **v2client** programs by entering the program name at the command line with no arguments.

4.2.3.2.1 The v2server Program

The **v2server** testing options are as follows:

v2server [-Dbce] [-d *switch_level***] [-p** *auth_prot, auth_name***]** \
max_calls protocol_sequence

where:

-D	This optional parameter specifies the default level of debug output.
-b	Enables a break between the RPC runtime calls.
-c	This optional parameter causes the server to call back its clients.
-e	This optional parameter causes the server to register its endpoint with the local location broker daemon, unregister its endpoint, and print a message indicating whether these operations were

successful.

-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	Maximal error/anomalous condition reporting and mutex checking. This amount of output is often too verbose for normal use, and 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_prot* and the *auth_name* parameters.

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

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.

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

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

protocol_sequence — Specifies one or more network protocols that can be used to communicate with client applications. The following values are valid for *protocol_sequence*:

ncacn_ip_tcp	NCA connection over Internet Protocol: Transmission Control Protocol (TCP/IP).
ncadg_ip_udp	NCA datagram over Internet Protocol: User Datagram Protocol (UDP/IP).

4.2.3.2.2 The v2client Program

The **v2client** testing options are listed below:

v2client [-D] [-d *switch_level***] [-p** *auth_prot, authz_proto, level, auth_name***]**
test string_binding num_calls num_buffs buff_size call_opts

where:

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

-d *switch_level* This optional parameter lets you specify the amount of debug output desired. Some useful *switch_level* settings:

- 0-3.5 — Maximum error/anomalous condition reporting and mutex checking. This amount of output is often too verbose for normal use, and 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 — Reports error conditions only (same as specifying *-d*).

-p Specifies an authenticated RPC call. You must enter the **-p** parameter with the *auth_prot*, *authz_proto*, *level*, and *auth_name* parameters.

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

- 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 validity and trustworthiness of authorization data depends on the authentication service and authentication level selected. The following values are valid for *authz_prot*:

- 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.

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

test Specifies one of the following tests:

- n Null test. Makes remote procedure calls with no parameters.
- i Input test. Makes remote procedure calls with input parameters only.
- o Output test. Makes remote procedure calls with output parameters only.
- io Input/Output test. Makes remote procedure calls with both input and output parameters.

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.

num_calls Specifies the number of times **v2client** calls the server.

num_buffs Specifies the number of buffers that are sent with each call.

buff_size Specifies the number of bytes in each buffer.

call_opts Specifies one of the following call options:

- 0 Nonidempotent call

1	Broadcast call
2	Idempotent call
4	Maybe call
8	Nonidempotent call; actively keeps communications alive with the server
9	Broadcast call; actively keeps communications alive with the server
10	Idempotent call; actively keeps communications alive with the server
12	Maybe call; actively keeps communications alive with the server

4.2.4 IDL Compiler Tests

The test cases for IDL data types are found in the

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

directory. The compatibility testcases are provided for information purposes only; they do not compile properly. The

dce-root-dir/**dce/src/test/rpc/idl/README**

file contains additional information about the test cases.

Before running the IDL tests, be aware of the following:

- The stubs and the **server** and **client** programs for each test case are built when the source tree is built.
- The IDL compiler will not report an error if there is no **.acf** file corresponding to an **.idl** file, so always keep the **.acf** file in the directory where the **build** or **make** command is issued.

4.2.4.1 IDL Compiler Testcase Driver

To run the IDL compiler testcase driver, enter:

run_tests *repeat_count* **[** *testcase_name ...* **]**

where *repeat_count* specifies the number of times to repeat a test, and *testcase_name* specifies the testcase (or testcases) to run.

To test connection-oriented RPC, you must set the **PROTOCOL** environment variable to ‘‘ncacn_ip_tcp’’; **run_tests** defaults this to ‘‘ncadg_ip_udp’’.

4.2.4.2 Running Individual Testcases

To run a test, you must first start the **server** as one process, then start the **client** as another process. These processes can be run on the same or different hosts as long as the **server** process is started first.

The server and client processes exist under each built subdirectory (for example, in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/rpc/idl/array**

directory. In general these build locations correspond to subdirectories in the source tree; for example, the contents of the subdirectory mentioned just above are built from the contents of the

dce-root-dir/**dce/src/test/rpc/idl/array**

directory). To start the **server** for a test case, enter

server [- | -f *filename* **]** *protseq*

where **-** specifies that binding information be written to standard output, **-f** *filename* specifies that binding information be written out to the file *filename*, and *protseq* specifies the protocol sequence (usually **ncadg_ip_udp** or **ncadg_ip_tcp**) used. The command prints the line

protocol ip_addr port

where *protocol* is the protocol specified with the **server** command, *ip_addr* is the IP address of the server, and *port* is the number of the port the server is monitoring. Unless you specify a name for *filename*, information is written to a file called **binding.dat**.

To start the **client**, enter

client *protocol ip_addr port passes*

where *protocol*, *ip_addr*, and *port* are the values obtained from the output of the **server** command, and *passes* is the number of times the client calls each remote procedure call specified in the interface definition.

Entering **server** or **client** at the command line with no arguments prints a help message on how to invoke the programs.

The test case automatically generates data and verifies correct data transfer. See the

dce-root-dir/**dce/src/test/rpc/idl/README**

file for more information.

Testcases are provided that test:

- Simple data types like **char**, **byte**, and **float**, as well as structures that can be transmitted using the **pipe** data type
- Reference pointers with null or non-null values and directional attributes

- Reference pointers with directional attributes
- The field attribute for arrays
- Arrays of pointers and field attributes for arrays specified as pointers
- Attributes

4.2.4.3 IDL C++ Tests

The following are tests of the IDL compiler C++ functionality. The source for the tests is located at:

dce-root-dir/**dce/src/test/rpc/idlcxx**

- **account**

  Tests inheritance, binding to an object using another interface, binding to an object with an unsupported interface, and the reflexive, symmetric, and transitive relation properties of the **bind( )** API. A **Savings** interface is derived from an **Account** interface. A **nowAccount** implementation class is derived from the **Savings** and **Checking** interfaces. An **oldAccount** implementation class is derived from the **Savings** but not the **Checking** class which implies that an **oldAccount** does not support a **Checking** interface.

- **account2**

  Tests the same properties as **account** but combines the client and server stubs in the test to also verify a local object with multiple interfaces as an RPC argument.

- **accountc**

  Tests the same properties as **account** but uses the C interfaces for all the APIs.

- **bind**

  Tests the **bind( )** APIs for binding to named objects; specifically, the **bind((unsigned_char_t *))**, **bind(uuid &)** and **bind(rpc_binding_handle_t)** APIs.

- **card**

  Tests the passing of C++ objects as parameters using the **[cxx_delegate]** attribute and the polymorphism property of the base class. A **Player** implementation class is a generic sports card class. Derived from **Player** are a **BaseballPlayer** class and a **BasketballPlayer** class. The application interfaces with the **Player** class to invoke virtual operations in the derived classes.

- **handle**

  Tests the invocation of a remote static procedure using explicit and implicit handles. The operations are really remote constructors **([cxx_new])** which are static by definition.

- **local_object**

  Tests the **[local]** IDL attribute in conjunction with inheritance to verify side casts in a C++ class hierarchy. No RPC calls are made and the server is just a copy of the client

executable to be compatible with the test environment. C++ itself does not support side casts, but IDL helps get around this problem with the **rpc_object_reference** base class and the **rpc_object_reference::bind(rpc_object_reference *)** API.

- **lookup**

  Tests the **[cxx_lookup]** attribute and the passing of a **rpc_x_object_not_found** exception from the server to the client.

- **matrix**

  Tests many different basic features, such as: local and remote objects as parameters; structures; arrays and unions containing interfaces as parameters; **bind( )** APIs; registering named objects; and so on.

- **matrixc**

  Tests the basic C interfaces to member functions and IDL generated APIs.

- **native**

  Tests passing a native C++ object as a parameter using the **[represent_as]** attribute. A system supplied C++ **String** object is passed as an RPC argument. (This test is used as a model in the paper *Passing C++ Objects as DCE RPC Parameters* from the IDL WWW home page, and is included in the WWW examples link.)

- **refcnt**

  Tests the reference counting APIs and validates that the Object Table is maintained properly on the server by creating lots of remote dynamic objects, and then deleting them.

- **refmon**

  Tests the **rpc_object_reference::get_binding_handle( )** and **rpc_object_reference::secure( )** APIs and uses a reference monitor. The client must be logged into DCE in order to run this test. The **refmon.pwd** contains the principal's passwords for the **jail**, **idl** and **xidl** cells. To port this test to another cell will require adding the password to **refmon.pwd** and creating a DCE principal **refmon_test** with the password ''dce''. During DCE 1.2.2 testing the client was run as follows:

  ```
  dce_login refmon_test dce -exec client
  ```

- **retry**

  Tests the retry feature of a client proxy object. The server executable file produced by the make file is a script that runs two server processes in the background using 2 different protocols. The client connects to one of the server processes. That server process then exits. The client then tries to connect to the same process again; this fails. The client then selects another binding handle to communicate with the second server process. (The client and servers are synchronized with **pthread_delay_np( )** rather than **sleep( )** because of a bug in the **sleep( )** API on VMS, causing it to wake up prematurely.)

- **stack**

  Tests the passing of C++ objects as parameters using the **[cxx_delegate]** attribute and a user defined **Stack** class. This test implements a reverse Polish notation algorithm

where the binary arithmetic operations are performed on the server. (This test is used as a model in the paper *Passing C++ Objects as DCE RPC Parameters* from the IDL WWW home page, and is included in the WWW examples link.)

- **static**

  Tests the IDL **static** and ACF **[cxx_static]** and **[cxx_static(arg)]** attributes. There are three ways to specify static member operations in IDL.

- **stubexc**

  Tests the passing of a **rpc_x_no_client_stub** exception from a server to the client and the raising of a **rpc_x_no_server_stub** in the client application. These exceptions are raised at runtime if the client or server stub is not linked with the server or client application respectively and a RPC parameter requires it.

- **tiered**

  Tests the passing of an object reference from one client to another. A client is built as both a server and client of an interface. It creates a remote object on the server. A second client connects to the first client to get the object reference to the server's object.

Before running the tests, you should set the **RPC_DEFAULT_ENTRY** environment variable to a CDS pathname consisting of an object name (named after the test to be run) located in a CDS directory that can be set writable to all. For example, to create an **idltest** directory for this purpose, do the following:

```
cdscp create dir /.:/idltest
acl_edit /.:/idltest << EOF
    m unauthenticated:rwdtcia
    m anyother:rwdtcia
    co
EOF
```

After having done the above, you can run (for example) the **account** test as follows:

```
% cd dce-root-dir/dce/install/machine/dcetest/dce1.2.2/test/rpc/idlcxx/account
% setenv RPC_DEFAULT_ENTRY /.:/idltest/account_myhost
% server &
% client
```

1.2.2,IDL C++ Extension Tests (start)

4.2.4.4 IDL C++ Extension Tests

This test suite tests the capability added to IDL to generate and support C++. The tests are located under:

src_tree/**test/rpc/idlcxx/...**

The following areas are tested:

- Interface inheritance
- Object bind calls:
  bind(unsigned_char_t *)
  bind(uuid &)
  bind(rpc_binding_handle_t)
- C/C++ interface
- Passing of C++ objects:
  [cxx_delegate] attribute
  [represent_as] attribute
- Static member functions:
  IDL **file static** attribute
  ACF **[cxx_static]** attribute
  ACF **[cxx_static(arg)]** attribute
- **[cxx_lookup]** attribute:
  passing of **rpc_x_object_not_found** exception
- **[cxx_new]** attribute
- **rpc_object_reference** class testing:
  [local] attribute
  rpc_object_reference::bind(rpc_object_reference*)
  rpc_object_reference::get_binding_handle( )
  rpc_object_reference::secure( )
- Named objects:
  register_named_object(unsigned_char_t *)
- Dynamic objects
- Coexistence of local and remote objects
- Enhanced enumeration types

4.2.4.4.1 Prerequisites for Running the Tests

The following are prerequisites for running the tests:

- ODE 2.3.3 (or later version)
- C++ and C compilers
- For the C/C++ interface tests, C++ constructors must be invoked even if the **main( )** routine is written in C.

  For the native C++/C compilers on current versions of Digital Unix and AIX this happens automatically—you must only be sure to link using the C++ compiler itself instead of either the C compiler or the linker. (Use **xlC** on AIX or **cxx** on Digital Unix.) When using GNU's **g++** compiler for AIX, a call must be inserted into the C

main( ) to **__do_global_ctors( )**. For the **accountc** and **matrixc** tests this means inserting the special call into **client.c**. This necessity may or may not be consistent among the **g++** implementations. Other platforms may have different special requirements for causing the constructors to be invoked, or they may do it automatically when using the C++ compiler to link the C and C++ object modules.

4.2.4.4.2 Building TET and the Tests

To build TET, do the following:

1. Locate yourself in the

   src_tree/**test/tet**

   directory and type **build**.

2. Put the directory

   obj_tree/**test/tet/src/posix_c/tcc**

   into your path or copy the file

   obj_tree/**test/tet/src/posix_c/tcc/tcc**

   to some directory already in your path.

3. Adjust any necessary build flags for your particular platform.

   To do so, edit the

   src_tree/**test/rpc/idlcxx/idlcxx.mk**

   file, which is pulled into each Makefile for the tests below. Each machine may have a separate section of IDL, compile, and link flags as shown below:

   ```
   <...>

   .if ${TARGET_MACHINE} == "RIOS"
   # With TET builds
   CFLAGS              += -DTET -DIDL_CHAR_IS_CHAR
   IDLFLAGS      +=     -lang cxx -v -cc_opt "-DTET -DIDL_CHAR_IS_CHAR -DAIX32"
   RIOS_LIBS     +=     -lidlcxx -ldce -ltetapi -ltettcm -ltetapi
   .elif ${TARGET_MACHINE} == "ALPHA"

   <...>
   ```

   The flags shown above for the AIX/RIOS platform are expected to be required for all platforms with exception of **-DAIX32**. A circular link dependency requires **-ltetapi** to appear twice. The **-cc_opt** option for IDL must include all flags to be handed to the compiler from IDL—they are not additions to the C flags that IDL generates for the platform by default. You can find out what your default IDL-spawned C flags are by using the **-v** option with IDL on a test **.idl** file.

4. Build the tests themselves. Go to

src_tree/**test/rpc/idlcxx**

and type **build**.

4.2.4.4.3 Running the Tests and Checking Results

To execute these tests under TET, follow the directions below. It is not necessary to have any special privileges (root or cell_admin) to follow these steps, but the scripts **setup** and **refmon/run** assume that the password for **cell_admin** is ‘‘**-dce-**’’.

1. Go to the test directory:

   % cd *object_tree*/**test/rpc/idlcxx**

2. Run the ’setup’ script, which will create a CDS test directory and open up the permissions for anyone to write to the directory:

   % setup

3. Set TET’s root directory to your current directory:

   % setenv TET_ROOT ‘pwd‘

4. Begin execution of the tests with:

   % tcc -e -j tet_jrnl -s tet_scen -x tetexec.cfg all

This final step will look for failures and summarize which tests passed, failed, or were missing results. This summary will happen automatically through the **tcc** execution, but can be repeated by executing the **summary** script which is created in the same current directory as above.

1.2.2,IDL C++ Extension Tests (end)

4.2.5 RPC Runtime I18N Extension Functional Tests

This test suite tests the APIs for I18N extensions to the RPC runtime in OSF DCE 1.1. The following APIs are tested:

- NSI management:
  - **rpc_ns_mgmt_set_attribute**
  - **rpc_ns_mgmt_remove_attribute**
  - **rpc_ns_mgmt_read_codesets**
  - **rpc_ns_mgmt_free_codesets**

- Codeset Registry
  - **dce_cs_loc_to_rgy**
  - **dce_cs_rgy_to_loc**
  - **rpc_rgy_get_max_bytes**
  - **rpc_rgy_get_codesets**
- Evaluation
  - **rpc_ns_import_ctx_add_eval**
  - **rpc_cs_eval_without_universal**
  - **rpc_cs_get_tags** (default eval logic)
  - **rpc_ns_binding_lookup_next**
  - **rpc_ns_binding_lookup_done**
  - **rpc_cs_binding_set_tags**
  - **rpc_cs_char_set_compat_check**
  - custom evaluations (**CMIR**/**SMIR**)
- Stub Support
  - **rpc_cs_get_tags**
  - **cs_byte_net_size**
  - **wchar_t_net_size**
  - **cs_byte_to_netcs**
  - **wchar_t_to_netcs**
  - **cs_byte_local_size**
  - **wchar_t_local_size**
  - **cs_byte_from_netcs**
  - **wchar_t_from_netcs**

The test sources are located at

dce-root-dir/**dce/src/test/functional/rpc/runtime/i18n_api**

in the source tree; the built objects can be found at:

dce-root-dir/**dce/obj/***platform*/**test/functional/rpc/runtime/i18n_api**

4.2.5.1 Prerequisites for Running the Tests

The following things must be true in order to successfully run the I18N Extension RPC runtime tests:

- All platforms:
  - OSF character and code set registry must be installed as

    /usr/lib/nls/csr/code_set_registry.db

    This is a binary file, which is produced by **csrc** (the code set registry compiler). The input file should be found in:

    dce-root-dir/**dce/src/test/functional/rpc/runtime/i18n_api/ts/cs_rgy**/*platform*
  - The Japanese EUC and SJIS locales are required. This is because the test input data are Japanese. However, the contents of **i18n_input_data** can be changed to other data (for example, French), in which case the other appropriate locale will be required.
- HP-UX Platform:
  - HP-UX version 10 is required, since **nl_langinfo( )** is broken with HP-UX version 9.

4.2.5.2 Running the Test and Checking Results

To run the tests, do the following:

1. Compile the code set registry:

   ```
   % cd /usr/lib/nls/csr

   % csrc \
       -i dce-root-dir/dce/src/test/functional/rpc/runtime/i18n_api/ts/cs_rgy/platform/code_set_registry.txt \
       -o code_set_registry.db
   ```

   (Note that this step requires **root** permission, because **/usr/lib/nls** is a system directory.)

2. **dce_login** as **cell_admin**:

   dce_login cell_admin *password*

3. Go to the

   dce-root-dir/**dce/obj**/*platform*/**test/functional/rpc/runtime**

   directory.

4. Execute the following shell commands (the following is given in **csh** syntax):

   ```
   % setenv I18N_SERVER_ENTRY "/.:/i18n_test"
   % setenv TET_ROOT "‘pwd‘/i18n_api"
   % setenv TET_EXECUTE "‘pwd‘/i18n_api"
   % mkdir i18n_api/all
   ```

5. Add TET's path to your current execution path, for example:

   % setenv PATH /usr/dcetest/test/tet/bin:$PATH

6. Set the appropriate locale names for your system (locale names are system dependent). For example, on an HP-UX system:

   % setenv I18N_SERVER_LOCALE "japanese.euc"
   % setenv I18N_CLIENT_LOCALE "japanese"

   —or, on an OSF/1 system:

   % setenv I18N_SERVER_LOCALE "/usr/lib/nls/loc/ja_JP.AJEC"
   % setenv I18N_CLIENT_LOCALE "/usr/lib/nls/loc/ja_JP.SJIS"

7. Execute the test under TET with the following command:

   % tcc -e -s i18n_api/tet_scen -x i18n_api/tetexec.cfg -j *journal* **all**

   where *journal* is the pathname of the journal file where test results will be written. This command will execute all of the available test cases. Note that if you wish to execute the test more than once, you will have to either remove the journal file from the test's previous run or specify a different journal filename.

To verify the test results, check the journal output. The journal will be located in a numbered directory, where the number represents a test run. A numbered directory and journal is created for each invocation of the **tcc** command (for example, **0001e**, **0002e**, and so on).

For the evaluation/stub support test cases, go to the

dce-root-dir/**dce/obj/***platform*/**test/functional/rpc/runtime/i18n_api/ts/cs_eval/cs_byte**

and

dce-root-dir/**dce/obj/***platform*/**test/functional/rpc/runtime/i18n_api/ts/cs_eval/wchar**

directories, and run the **result_check.sh** script. The script will verify that the generated output is the same as the expected output.

4.2.6 RPC Runtime Library and IDL Compiler Tests

A suite of test cases is provided for verification of compiler and runtime interaction. Use the **testsh** shell script, which allows for summary statements and uniformly formatted output for each test case, to execute these test cases. Control program scripts are ''built'' in the directory:

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/rpc/rtandidl/control**

(The scripts all have file names ending with **.tsh**.) Note that the contents of this directory are built from the contents of the

dce-root-dir/**dce/src/test/rpc/rtandidl/control**

directory.

Each control program imports an environment from one or more configuration files (with names ending with the suffix **.tshrc**) and invokes the test case with the appropriate input parameters. Summary information can be printed prior to exit from the control program. This structure lets the user ignore complicated parameter requirements for individual test cases, thereby simplifying test case execution.

Before executing the Naming Service Interface (NSI) portion of this suite, be aware of the following:

- The namespace must be configured.
- The **NSTEST_DIR** directory must be created in the namespace for use by the NSI tests. See Chapter 5 of this guide, the chapters on configuring and starting up DCE in the *OSF DCE Administration Guide—Introduction*, and *Appendix A* of the *OSF DCE Administration Guide—Introduction* for details on namespace configuration.

Before executing the RPC Authentication testcases, the DCE Security Service must be properly configured. See Chapter 8 of this guide for information on configuring and enabling the DCE Security Service.

You must also do the following before running authenticated RPC tests:

- Login as the privileged user (root).
- Authenticate as cell_admin, or any user with privileges to modify the registry, using the **dce_login** command. The default password is ‘‘-dce-’’.

  ```
  dce_login cell_admin -dce-
  ```

- Set the following environment variables:

BACKTREE	The absolute path to the backing tree or sandbox.
CALLER_KEY	Password for the **cell_admin** account. The default is **-dce-**.
CLIENT_KEY	Password given to the client user account.
CLIENT_NAME	Account name for the client user.
PROTOCOL	Should be set to either ‘‘ncadg_ip_udp’’ or ‘‘ncacn_ip_tcp’’.
SERVERHOST	Should be set to the machine name of the machine that is to run the server daemon.
SERVER_KEY	Password given to the server account.
SERVER_NAME	Account name for the server user.

- Ensure the **run_server** shell script invokes the **sofserv** process with the appropriate value for the server account and *server_key*.

  Note: Typically *server_name* and *server_key* are set to ‘‘server,’’ and *client_name* and *client_key* are set to ‘‘client.’’

To run these tests, you must first start the **run_server** shell script, and then start the **run_client** shell script. Since **run_server** starts a server process, it must be executed

prior to **run_client**. The **run_client** script invokes the test case control files using the **testsh** program.

To start the server process, enter

run_server

at the command line. No parameters are required.

The **run_client** shell script executes the specified test cases and has the following syntax:

run_client *-testlist server_host testsh_dir testcase_dir include_dir testname*

where

-testlist	Provides a listing of all valid test case choices. Individual test cases are valid choices, as are categories of tests such as **all**, which requests execution of all test cases in this suite.
server_host	Specifies the name of the machine on which the **run_server** shell script was executed.
testsh_dir	Specifies the name of the directory containing the **testsh** executable.
testcase_dir	Specifies the name of the directory containing the test case executables.
include_dir	Specifies the name of the directory containing the DCE header files. It is used by the IDL compiler tests **nocode** and **cmd_line** so these tests can be run prior to final installation of the DCE RPC header files.
testname	Specifies the name of the test to run, or category of test cases to be run. The **run_client -testlist** command can be used to generate a list of valid test names.

See the

dce-root-dir/**dce/src/test/rpc/rtandidl/README**

file for further information, including information about the **do_rpc_test** script, which will run the **rtandid** test suite.

4.2.6.1 The testsh Program

The **testsh** program is a front end for execution of test programs. Source code for this program can be found in the

dce-root-dir/**dce/src/test/rpc/rtandidl/testsh**

directory. It provides a standard way for a test developer to create a test environment and it tallies subtotals and summaries of test results. It also allows error conditions to abort a test suite.

The default behavior for **run_client** is to run the test specified, and log results in *testname*.**log**.

The **testsh** testing options are as follows:

testsh [-d [*output_level***]] [-e] [-l** *filename* **|**
-L *filename***] [-s | -S] [-I** *path***]**

where:

-d Specifies an output level for all test programs. Using the **-d** option with no *output_level* integer returns a message only when a test fails.

output_level Specifies a specific output level for all test programs. The following list shows the valid integer values for *output_level* and the output levels they specify:

- 1 Prints message on failure.
- 2 Prints message on success.
- 3 Prints message on warning.
- 4 Prints message on trace.
- 5 Prints message on information.
- 63 Prints debug messages during test case execution.

-e Terminates the execution of the test case when an error is encountered.

-l Generates a log file and stores that log file in *filename*.

-L Sends the expanded **testsh** script commands from **testcase.tsh** to *filename*.

-s Prints output to the screen using the standard error.

-S Sends verbose output to the screen using the standard error.

-I Enables **testsh** to look in the *path* directory for test case executables.

4.2.6.2 RPC API Function Tests

This test suite includes a test for all RPC API functions. The tests are located in the

dce-root-dir/**dce/src/test/rpc/rtandidl/control**

directory and are grouped as shown in the following table:

Test Group	Control File	Function Tested
Binding tests	**all_binding.tsh**	**rpc.binding_*( )** & **string_( )*** calls
DCE error inquire text tests	**error_inq_text.tsh**	**dce_error_inq_text( )** call
NSI tests	**all_ns.tsh**	**rpc_ns_*( )** calls
RPC authentication tests	**all_auth.tsh**	**rpc_*_auth_*( )** calls
RPC management tests	**all_mgmt.tsh**	**rpc_mgmt_*( )** & **network_protseqs*** calls
Object tests	**object_set_type.tsh, object_inq_type.tsh**	**rpc_object_*( )** calls
UUID tests	**all_uuid.tsh**	**uuid_*( )** calls
IDL tests	**all_idl.tsh**	IDL compiler and application tests

4.2.6.3 Specification for control file and Command Descriptions

The **control file** is a template that directs the execution of test cases. The control file consists of commands that can be composed of keywords, function calls, literals, and values that are interpreted by the script as parameters to pass to test cases.

The valid commands are as follows:

echo *string* — Prints the specified string.

*string* — The # (number sign) character specifies a comment, which is ignored.

include *configfile* — Executes the *configfile* configuration file.

execute *runfile* — Spawns a subshell and executes *runfile*.

test *options testcase_parameters*
Executes a test case. The **-p***(iterations)* option can be used to execute multiple iterations of a test. The test case parameters must coincide with the parameters expected by the individual test case to be run.

run *program* — Executes the specified program.

summary — Generates and prints the number of successful and unsuccessful test cases. It is typically the last line of a control file.

subtotal — Prints the number of test cases that have passed or failed since the last **subtotal** command.

subtotal clear — Resets the subtotal counts to zero passes and zero failures.

remote *host program testsh_options*
Remotely executes a *program* on the machine *host*. The *program* is run under the **testsh** controller with the options specified by *testsh_options*.

set *VAR=value* — Sets an environment variable *VAR* to *value*.

pause — Prints the message

```
Press RETURN to continue or q to quit
```

on the screen and delays the execution of the program until the tester enters a valid response.

onerror *option* — Specifies default behavior of the control program when errors occur. The following values are valid for *option*:

stop	Causes **testsh** execution to halt if an error is encountered.
continue	Causes **testsh** execution to continue regardless of errors.
default	Consults the global parameter (set by the **-e** option to the **testsh** controller) to determine the appropriate behavior in the event of a failure.

4.2.7 Name Service Interface Test

dcesx is a test of the CDS NSI (Name Service Interface). Refer to Chapter 5 of this guide for information on running the test.

4.2.8 Test Plans

Refer to Chapter 1 of the *OSF DCE Release Notes* for the location of the DCE test plans on the DCE distribution tape.

4.3 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.

4.3.1 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.

4.3.1.1 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.

4.3.1.2 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.

4.3.2 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''.

Chapter 5. DCE Cell Directory Service

5.1 Overview

The DCE Cell Directory Service (CDS) provides the directory (naming) services for use within a cell in a DCE environment. CDS allows users to assign names to resources and then use those resources, without needing to know their physical locations in the network. CDS uses the client/server model, and provides both command line and programming interfaces for configuring services. CDS services can be accessed through two Application Programming Interfaces (APIs), provided as part of **libdce.a**. The first is the X/Open Directory Service (XDS) API, and the second is the Name Service Interface (NSI) of the RPC component, which accesses CDS in an RPC-specific way.

CDS allows clients to register named objects with the server and to bind a set of attributes, including an object's network addresses, to these objects. An object's attributes are stored in a distributed database, which is partitioned and partially replicated. CDS is composed of three programs:

- **cdsd**

  The CDS server. This program stores and maintains CDS names and handles requests to create, modify, or look up data in the CDS database.

- **cdsclerk**

  The CDS clerk. This is the interface between client applications and servers, and it must exist on every node. Several of these may be running on each node since one is spawned for each user.

- **cdsadv**

  The CDS advertiser, the program which makes distributed CDS servers aware of each other and known to clients. There must be one of these on every node.

In addition to these, there is also the **cdsbrowser** utility and the **cdscp** administration program (''CDS control program'').

5.2 Setup, Testing, and Verification

Eight types of CDS tests are shipped with DCE. Two ways to test CDS are provided: **cdstest** and the CDS test scripts. These tests are described in more detail in the following sections.

The **cdsd_diag**, **cadump**, and **catraverse** programs, and the **dcesx** test, are also useful in debugging CDS.

Before executing the test cases, you must configure CDS for testing using either the DCE Configuration script

dce-root-dir/**dce/src/config/dce_config**

or the instructions found in the next subsection. You can run tests on the configurations described in that section.

5.2.1 Installing CDS Functional Tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config** will install the tests you select at the path you specify, and will create a softlink (called **/dcetest/dcelocal**) to that location. The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the ‘‘CDS Test Scripts’’ and following sections below, which refer to the tests’ source or build locations.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the final location of the built test tree). For the CDS functional tests, this path should be the location, on your machine, of:

dce-root-dir/**dce/install**

—which is the DCE **install** tree (for more information on the structure of the DCE tree, see Chapter 3 of the *OSF DCE Release Notes*).

Thus, **dcetest_config** will install the CDS functional tests at:

/dcetest/dcelocal/test/directory/cds/

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* (if you choose) functional tests with **dcetest_config**; for test configuration and execution you must follow the instructions in the sections below.

Refer to Chapter 11 of this guide for further information on using **dcetest_config**.

5.2.2 CDS Setup

You can set up CDS for testing purposes in two ways: with or without the CDS advertiser, **cdsadv**. The **cdsadv** program automatically starts up a clerk-only system; you must start the clerk manually when running CDS without **cdsadv** for testing. You can specify the **-a** switch with **cdsd** to create a namespace, a clearinghouse, and the root directory. This process is called auto-initialization.

To debug the CDS commands **cdscp**, **cdsd**, **cdsclerk**, **cdsadv**, **cdsd_diag**, or **cdsbrowser**, you need to have built the code with the **DEBUG** macro defined. The debug output will go to a log file in:

*dcelocal***/var/adm/directory/cds/cdsd**

(for server daemons) and:

*dcelocal***/var/adm/directory/cds/cdsclerk**

(for clerk daemons) directories, if there exists a file with the command name and a file extension **.events** (with an asterisk as the only character in the file) in the respective directory. This file is checked only at startup.

Note: The CDS clerk will stop working if the contents of its events file:

*dcelocal***/var/adm/directory/cds/cdsclerk/cdsclerk.events**

exceeds 100 entries.

Both **cdsd** and **cdsadv** take **-d** and **-e** switches. The **-d** switch specifies debugging mode (that is, it does not fork) and if specified with the **-e** switch, routes event output to the standard output. The **-e*** switch requests debug event logging for all events to go to standard output. (The backslash is a shell escape character so that the asterisk is passed through the shell. The asterisk indicates that all events should be reported.)

5.2.2.1 CDS Setup without cdsadv

To configure CDS for testing without **cdsadv**, you need to be logged in as root and do the following:

1. Change directory to *dcelocal***/bin**.
2. The **dced** (DCE host daemon) process must be running before you start any CDS processes. See the **dced(8dce)** reference page for information on starting **dced**.
3. To start the **cdsd** daemon, enter:

   ./cdsd -a

where **-a** specifies auto-initialization. The auto-initialization information is stored in the

dcelocal/**etc/cds_config**

file, which is created by **cdsd**, and which can be used to configure clerks and servers manually.

You may also use the following optional switches:

-d Debug mode, events to **stdout**, does not fork, turns on tracing to your terminal.

-e Prints error messages. Use the \ character to escape the shell. Use the ***** character to indicate full error messages (-e*) or use a fully qualified filename.

-v Prints initialization progress messages; these verify that initialization sucessfully completed.

4. On the server machine (the machine on which you started the server), enter:

```
./cdsclerk -d -F
```

to start the **cdsclerk** process, since this configuration does not use **cdsadv** to start **cdsclerk**. The **-d** flag prohibits forking, and the **-F** flag deletes the old socket on startup.

You may also use the following optional switches with **cdsclerk**:

-e Prints error messages. Use the \ character to escape the shell. Use the ***** character to indicate full error messages (-e*) or use a fully qualified filename.

-m *number* Uses shared memory ID *number*. The shared memory ID can be found in:

dcelocal/**etc/cdscache.shmid**

5. If the machine on which you want to run the client is not the server machine, you need to run **cdsclerk**. Copy the

dcelocal/**etc/cds_config**

from the server machine to the client machine. Enter:

```
./cdsclerk -d -F
```

to start the **cdsclerk** process on the client machine, since this configuration does not use **cdsadv** to start **cdsclerk**. The **-d** flag prohibits forking, and the **-F** flag deletes the old socket on startup.

5.2.2.2 CDS Setup with cdsadv

To configure CDS for testing with **cdsadv**, you must be logged in as root and do the following:

1. The **dced** (DCE host daemon) process must be running before you start any CDS processes. See the **dced(8dce)** reference page for information on starting **dced**.
2. To start the CDS advertiser, enter:

   ./cdsadv

   You may also use the following optional switches:

   -c Specifies cache size in kilobytes.

   -e Prints error messages. Use the \ character to escape the shell. Use the ***** character to indicate full error messages (-e*), or use a fully qualified filename.

   -s Prohibits the sending or receiving of advertisements. This setting is useful for debugging and for setting up multiple cells on one LAN.

   -v Prints initialization progress messages; these verify that initialization completed successfully.

   The **cdsadv** program solicits responses from CDS Servers on the same LAN by broadcast RPC. The first response it receives becomes the default CDS Server used by that clerk.

   To promote some other server to default, edit

   dcelocal/**etc/cds_config**

   and change the desired defaults. You must then stop any clerks that are running, and restart **cdsadv**.
3. In the same directory, start the **cdsd** daemon by entering:

   ./cdsd -a

   where the **-a** flag specifies auto-initialization. The auto-initialization information is stored in the

   dcelocal/**etc/cds_config**

   file, which can be used to configure clerks and servers manually. You can also use the optional switches described for **cdsd** in the section on ‘‘CDS Setup Without cdsadv’’ in this chapter.

5.2.2.3 Using gdad

The **gdad** command starts the GDA daemon. The Global Directory Agent (GDA) enables intercell communication, serving as a connection to other cells through the global naming environment.

You may use the following optional switches:

-d For debugging use only. Ranges from d0 through d12, with d0 being the simplest level and d12 the most complex. The most useful level of debug output is d7 for diagnosing operational problems. Higher levels are useful when debugging coding errors.

-f Does not fork the child process.

-F Deletes old socket on startup.

-r Alternate pathname of **/etc/resolv.conf**.

-s Alternate pathname of **named.ca** file.

-u Does not update GDA information in CDS server.

-v Prints initialization progress messages; these verify that initialization completed successfully.

5.2.2.4 Resetting the CDS Environment

If it is necessary to reset the CDS environment to a ‘‘clean’’ state, there are several files that need to be removed and shared memory segments and semaphores to be deleted.

The shared memory segment(s) can be removed by performing the following steps:

1. Get the SHMID (shared memory ID) from the first line of the file:

   *dcelocal***/etc/cdscache.shmid**

2. Use **ipcs** to find the **shm_key** for the SHMID. The semaphore used by CDS uses the same key as the shared memory:

   ipcs | awk ’/*SHMID_from_step_1***/ {print $3}’**

3. Remove the semaphore:

   ipcrm -S *shm_key_from_step_2*

4. Remove the shared memory:

   ipcrm -m *SHMID_from_step_1*

The CDS files can be removed with the following script:

rm -rf *dcelocal***/var/adm/directory/cds/***

rm -rf *dcelocal***/var/directory/cds***

rm -rf *dcelocal*/**var/directory/cds/adm/cdsd/***

rm -rf *dcelocal*/**var/directory/cds/adm/gdad/***

rm -rf *dcelocal*/**etc/cds_config**

rm -rf *dcelocal*/**etc/cds_defaults**

rm -rf *dcelocal*/**etc/gda_id**

rm -rf *dcelocal*/**etc/cdsadv.pid**

rm -rf *dcelocal*/**etc/cdscache.shmid**

rm -rf *dcelocal*/**etc/cdsd.pid**

See also the **dce.rm** script.

It is sometimes useful to purge the CDS cache between runs. To remove the CDS on-disk cache (e.g., before starting up a new CDS server), execute the following commands:

kill -9 *cdsclerk PID*
/etc/dce.clean
cd /opt/dcelocal/var/adm/directory/cds
mv cds_cache.*number* **cds_cache.***number***.BAD**
/etc/rc.dce

If the CDS server and client cannot broadcast, you must also do the following:

cdscp define cached server *CDS_Server_Hostname* **tower** \
  ncadg_ip_udp:*CDS_Server_IP_Address*

For example:

cdscp define cached server west tower ncadg_ip_udp:130.105.201.10

5.2.2.5 CDS Configuration Files

The following files are used in CDS configuration:

dcelocal/**etc/cds.conf**	This file contains security information for CDS, such as the principal names of the **cdsd** and **gdad**, as well as the names of the **cds-server** and **cds-admin** groups.
dcelocal/**etc/cds_config**	This file contains configuration information about namespaces and clearinghouses, including the name

and UUID of each. In each case it also contains the internet address of the server that supports the clearinghouse.

dcelocal/**etc/cds_attributes** This text file contains a list of the DCE attributes and their OIDs. It maps OID, SYNTAX, and the label used by CDS to identify the displayed attribute. For example:

```
OID             LABEL            SYNTAX
1.3.22.1.3.42   CDS_LastSkulk    Timestamp
```

dcelocal/**etc/cds_globalnames** This file is a database of DCE-supported X.500 attribute types. Some of these are ‘‘naming attributes’’ (meaning that they occur in the names of objects, as specified by the schema), but most are not. The file maps the following for each Attribute Type:

- OID
- LABEL
- ASN.1-IDENTIFIER
- SYNTAX
- MATCHING RULE

cdscp.bpt Used by the **cdscp** parser.

cdscp.mbf Used by the **cdscp** parser.

5.2.3 CDS Test Scripts

The test scripts for CDS are in the

dce-root-dir/**dce/src/test/directory/cds**

directory. To run a test, enter:

cp_test.sh [*-switch ...*] *testname*

where

-switch This optional parameter specifies a certain testing option. The following values are valid for *switch*:

-cdscpdir *pathname* Specifies an alternative pathname for **cdscp**.

-cell *name1* Specifies *name1* as the cell name to perform local tests on.

-ch1 *name2*	Specifies *name2* as the primary clearinghouse (Clearinghouse 1).
-ch2 *name3*	Specifies *name3* as the secondary clearinghouse (Clearinghouse 2 - an existing clearinghouse).
-ch3 *name4*	Specifies *nam4* as Clearinghouse 3 - a clearinghouse to create.
-dir *dirname*	Specifies *dirname* as the top level test directory.
-disable	Do not strip **disable** commands from scripts.
-inet *address*	Specifies Internet address.
-keeplines	Do not delete the test script when done.
-nodeldir	Strips **deldir** commands from scripts.
-noch	Strips all clearinghouse information.
-noch1	Strips primary clearinghouse (Clearinghouse 1) information.
-noch2	Strips secondary clearinghouse (Clearinghouse 2) information.
-noch3	Strips create clearinghouse (Clearinghouse 3) information.
-nopipe	Specifies that commands not be piped into **cdscp**.
-noshow	Strips **show** commands from scripts.
-noskulk	Strip **skulk** commands from scripts.
-pid	Uses the process ID of **cp_test.sh** to generate unique log filenames. You can run multiple simultaneous tests using this option.
-remcell *cellname*	Specifies Remote cell name to reference for intercell testing.
-restart	Specifies that the DCE servers be restarted before starting the test.
-use_alias	Use **/.:** in tests to refer to the cellname.
-v	Specifies verbose mode.

testname Specifies the CDS test to run. The following tests are provided:

cp_abuse.tests	Stress tests.
cp_childpointer.tests	Tests childpointer operations.

cp_clearinghouse.tests	Tests clearinghouse operations.
cp_clerk.tests	Tests clerks.
cp_credir.tests	Tests directory operations and is a subset of **cp_directory.tests**.
cp_directory.tests	Tests directory operations.
cp_foreign.tests	Tests merges of foreign cell subtree dump files.
cp_intercell.tests	Tests references to foreign cell data (requires **-remcell** to be specified).
cp_misc.tests	Tests confidence, preferred clearinghouse.
cp_negative.tests	Tests multiple creates/deletes, and non-extant references.
cp_object.tests	Tests object operations.
cp_replica.tests	Tests replica operations.
cp_server.tests	Tests servers.
cp_softlink.tests	Tests softlink operations.
cp_subtree.tests	Tests subtree operations.

The

dce-root-dir/**dce/src/test/directory/cds/cp_killer.sh**

script runs all the tests listed above except:

- **cp_misc.tests**
- **cp_abuse.tests**
- **cp_intercell.tests**
- **cp_credir.tests**

To run **cp_killer.sh**, enter:

```
cp_killer.sh
```

Any of the **cp_test.sh** switches may be used when running **cp_killer.sh**. The **cp_killer.sh** script uses **cp_test.sh**.

5.2.4 Distributed ACL Tests

The driver script

dce-root-dir/**dce/src/test/directory/cds/dacl_testing.sh**

runs the distributed ACL tests:

- **dacl_creates.sh**
- **dacl_deletes.sh**
- **dacl_modifies.sh**
- **dacl_reads.sh**
- **dacl_replicas.sh**

It is invoked as follows:

dacl_testing.sh -ch1 *clearinghouse1* **-ch2** *clearinghouse2*

where *clearinghouse1* and *clearinghouse2* are the names of two clearinghouses, both of which must already have been created when the test is run, and neither of which should be the cell default clearinghouse.

Note that the clearinghouse arguments must *not* be specified with a leading ‘‘**/.:/**’’ or ‘‘**/.../**’’.

The following things must be true in order for the ACL tests to be run successfully:

- The driver script is running as the principal **notroot**.
- The CDS server is called

  */.../hosts/hostname/***cds-server**

  This is the default name as set up by **dce_config**.
- The **notroot** principal has write permission for the default clearinghouse.
- The **notroot** principal has insert and read permission for the root directory.

Because of these prerequisites for running the test, it is advisable to run **dacl_testing.sh** in a newly-configured DCE cell which has been specially set up for this purpose. The **dacl_setup.sh** script can be run to set up such a newly-configured cell so that it meets the above requirements.

dacl_setup.sh, which should be run as the **cell_admin** principal, is invoked as follows:

dacl_setup.sh -ch1 *clearinghouse1* **-ch2** *clearinghouse2* **-ch3** *default_clearinghouse*

where *default_clearinghouse* is the default clearinghouse for the cell; this usually has a name of the form *hostname***_ch**, where *hostname* is the name of the host machine.

Note that the clearinghouse arguments must *not* be specified with a leading ‘‘**/.:/**’’ or ‘‘**/.../**’’.

The output of the tests is written to the following logfiles:

- **dacl_creates.log**
- **dacl_deletes.log**
- **dacl_modifies.log**
- **dacl_reads.log**
- **dacl_replicas.log**

5.2.5 NSI Test

dcesx is a test of the CDS NSI (Name Service Interface). It is invoked as follows:

```
dcesx -K -M -R -V -i 10 -m 10 -p 99 -t 30
```

The flags have the following meanings:

Flag	Meaning
-K	Skulk whenever the namespace is changed.
-M	Use multiple threads.
-R	Re-randomize search context (only used if a directory search fails).
-V	Set maximum verbosity.
-i 10	Number of interfaces to enable (10 is the maximum).
-m 10	Number of call threads to configure (for RPC).
-p 99	Number of passes (**-p 0** means go forever).
-t 30	Number of seconds to delay after failure to import an interface.

Note that you should **dce_login** as **cell_admin** before running the **dcesx** test, so that the test will have the permissions necessary to perform the operations it will attempt on specific directories and objects.

5.2.6 Testing Intercell Lookup

The GDA clerk, unlike the CDS clerk which is an integral part of CDS, exists for test purposes only. Its source is located at:

dce-root-dir/**dce/src/directory/cds/gda/gda_clerk.c**

The **gda_clerk** test program performs the same GDA lookup that the CDS clerk performs; by running it you can eliminate all of the logic of the CDS clerk when testing the GDA. **gda_clerk** uses the same interfaces and the same progress records as the CDS clerk.

Its interactive inputs are:

- A string binding to the GDA. You can get this from the output of running the command:

  rpccp show mapping

- A **/.../***cellname* for the GDA to look up.

You should make sure that **gda_clerk** returns good results before you try remote cell access through CDS.

5.3 CDS Runtime Output and Debugging Output

The CDS 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 CDS via serviceability.

5.3.1 Normal CDS Server Message Routing

There are basically two ways to control normal CDS 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.

5.3.1.1 Routing File

If a file called

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

exists when CDS is brought up (i.e., when the CDS daemons are 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 CDS 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.

5.3.1.2 Routing by the dcecp log Object

Routing of CDS 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.

5.3.2 Debugging Output

Debugging output from CDS can be enabled (provided that CDS 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_CDS_DBG** environment variable, before bringing up CDS (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* 6
[*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

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 CDS, the sub-components are as follows:

adver	CDS Advertiser sub-component
child	CDS Clerk/Child/Client sub-component
gda	CDS GDA sub-component
server	CDS Server sub-component
cache	CDS Cache sub-component
library	CDS Library sub-component
general	CDS General sub-component
dthread	CDS dthreads sub-component
cdscp	CDS Control Program sub-component

For example, the string

"cds:*.1,server.3:TEXTFILE.50.200:/tmp/CDS_LOG

sets the debugging level for all CDS sub-components (*except* **server**) at 1; **server**'s level is set at 3. All messages are routed to **/tmp/CDS_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 CDS serviceability messages, and the sub-component list, can be found in the CDS sams file, at:

dce-root-dir/**dce/src/directory/cds/includes/cds.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''.

Chapter 6. DCE Global Directory Service

6.1 Overview

The DCE Global Directory Service (GDS) provides an X.500-compliant directory service. GDS includes the Directory User Agent (DUA), or client, and the Directory Service Agent (DSA), or server, as specified by the X.500 standard.

In conjunction with the directory service, GDS supplies the following services and interfaces:

Note: In the descriptions below, OSI means ‘‘Open System Interconnection,’’ an internationally recognized (ISO) term. However, in Chapter 10 of this guide, which covers porting and testing DFS, OSI means ‘‘Operating-System Independent.’’

- The standard XDS/XOM (X/Open Directory Service / X/Open OSI-Abstract-Data Manipulation) application programming interface to GDS
- The RTROS and CMX interface, and libraries for the OSI protocol stack upper layers
- An ASN.1 compiler (MAVCOD/MAVROS) and ASN.1 runtime library, used by GDS
- A screen-based menu-oriented administration interface
- A shell-based command interface to administer GDS
- A shell-based command interface to create and initialize a directory configuration
- An integrated ROS interface (RTROS) with AOM12 support for use by DME and DME applications

Note: Reference pages for the **mavros** and **mavcod** commands can be found in the *OSF DCE Technical Supplement*.

6.2 GDS Testing Overview

The following types of GDS tests are shipped with DCE:

- Admin

  Tests the menu-driven administration interface and the functionality it provides.

- API

  Tests the XDS/XOM/XOMS/MHS application programming interfaces.

- DUA Switch

  Tests the switching mechanism between CDS and GDS.

- **gdscp**

  Tests the command line interface.

- **gdssetup**

  Tests the command to create or initialize directory configuration.

- **gds_sec**

  Tests the use of DCE authentication.

- MAVROS

  Tests for the MAVROS compiler.

In addition, hand procedures for testing GDS intercell operation can be found in the section ‘‘Testing GDS Intercell Operation’’, later in this chapter.

Compiler and linker flags for building the GDS test cases reside in:

dce-root-dir/**dce/src/test/test.mk**

Machine-specific compiler and linker flags that affect the compilation or linking of the GDS test cases should be included in this file.

The following subsections describe how to install and set up GDS, and how to run tests for each of the categories in the previous list.

Note the following prerequisite conditions for testing various aspects of XDS:

- In order to test XDS access to GDS, you must have GDS running.
- You do not have to have GDS running in order to test XDS access to CDS.

6.2.1 Changes to the GDS Functional Tests Since DCE 1.0.3

The GDS and XDS functional tests were overhauled for DCE 1.1, and new functional tests were implemented for new functionality. The tests were converted to use the X/Open Test Environment Tool (TET) test harness. TET provides a common invocation mechanism for all GDS/XDS functional tests, a consistent means of determining testcase

outcome, and a common repository for testcase results.

Additionally, the administration tests are now completely automated.

The exception to the above statement is the MAVROS test. This still runs in the same manner as it did in the previous release.

Following are the tests that are available:

- GDS Tests
  - The admin tests located under the

    /dcetest/dcelocal/test/tet/functional/directory/gds/ts/admin

    directory:

    cacheadm cache administration testsuite

    dsa DSA administration testsuite

    shadow shadow administration testsuite

    subtree subtree administration testsuite

    scheme schema administration testsuite
  - The **gdscp** tests located under the

    /dcetest/dcelocal/test/tet/functional/directory/gds/ts/gdscp

    directory tests the GDS command program.
  - The GDS security tests located under the

    /dcetest/dcelocal/test/tet/functional/directory/gds/ts/gds_sec

    directory test the GDS security methods.
  - The **gdssetup** tests located under the

    /dcetest/dcelocal/test/tet/functional/directory/gds/ts/gdssetup

    directory test the GDS setup program.
- API Tests
  - The XDS API tests, located under the

    /dcetest/dcelocal/test/tet/functional/directory/xds/ts/xds

    directory:

    xds_st single-threaded mode

    xds_mt multi-threaded mode
  - XOM API tests, located under the

    /dcetest/dcelocal/test/tet/functional/directory/xds/ts/xom

    directory:

    xom_st single-threaded mode

    xom_mt multi-threaded mode

- XOMS API tests, located under the

  /dcetest/dcelocal/test/tet/functional/directory/xds/ts/xoms

  directory:

  xoms_st single-threaded mode

  xoms_mt multi-threaded mode

- MHS API tests, located under the

  /dcetest/dcelocal/test/tet/functional/directory/xds/ts/mhs

  directory:

  mhs_st single-threaded mode

- SWITCH API tests, located under the

  /dcetest/dcelocal/test/tet/functional/directory/xds/ts/switch

  directory:

  switch_st single threaded mode

  switch_mt multi-threaded mode

  switch_DNS typeless tests (uses DNS Cell Name)

The MAVROS tests are located at:

/dcetest/dcelocal/test/directory/gds/mavrostest

The TET binaries and scripts are located at:

/dcetest/dcelocal/test/tet/bin
/dcetest/dcelocal/test/tet/lib

6.2.2 Installing GDS Functional Tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config** will install the tests you select at the path you specify, and will create a softlink (called **/dcetest/dcelocal**) to that location. The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the sections below, which refer to the tests' source or build locations.

The GDS and XDS functional tests are available via option 4 (''Global Directory Service'') of the ''DCE Test Installation (Functional Tests'' menu. The TET binaries are available via option 3 (''TET'') of the DCE Test Installation menu.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the final location of the built test tree). For the GDS functional

tests, this path should be the location, on your machine, of:

dce-root-dir/**dce/install/***target_machine*/**dcetest/dce1.2.2**

—which is the DCE **install** tree (for more information on the structure of the DCE tree, see Chapter 3 of the *OSF DCE Release Notes*).

Thus, **dcetest_config** will install the GDS functional tests at:

/dcetest/dcelocal/test/tet/functional/directory/gds

and:

/dcetest/dcelocal/test/tet/functional/directory/xds

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

It is recommended that you not actually install the tests on your root filesystem; they are quite large. You will need at least 8 Megabytes of space in order to install the necessary software, and you should have another 8 Megabytes to allow for the creation of log files and test results journals.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* the functional tests with **dcetest_config**; you must use TET to run the tests (with the exception of the MAVROS tests). Information on running the individual tests can be found in the following sections.

See ‘‘Overview of TET Use’’ in Chapter 11 for general information on TET.

6.2.3 Running GDS Functional Tests with TET

The following subsections describe and explain various aspects of running the GDS functional tests that are run under TET.

6.2.3.1 Testing Tools: Test Drivers and Journal Filters

Several tools have been provided to make the testing process easier. These are not part of either TET or the functional tests, but are additions to ease the testing work load.

In

/dcetest/dcelocal/test/tet/functional/directory/gds/tools

are the following scripts:

local_TET.admin	GDS test driver
TET_filter.admin	Filter for **admin** test TET journal

TET_filter.gdscp	Filter for **gdscp** test TET journal
TET_filter.gdssetup	Filter for **gdssetup** test TET journal
TET_filter.gds_sec	Filter for **gds_sec** test TET journal

Similarly, in

/dcetest/dcelocal/test/tet/functional/directory/xds/tools

are the following:

local_TET.api	XDS test driver
TET_filter.api	Filter for XDS tests TET journal
xt_test	XDS test device

The test driver is a front-end to the TET test environment. It sets a number of environment variables used by the GDS tests and determines the location of results files produced by the tests. The filters scan the TET journal and produce a more concise and understandable summary of the test results.

6.2.3.2 Setting Up to Run the Tests

Before running either the GDS or API tests you must do the following things:

1. Set the **TET_ROOT** environment variable to

   /dcetest/dcelocal/test/tet

   For example (in a C shell):

   ```
   % setenv TET_ROOT /dcetest/dcelocal/test/tet
   ```

   Note that the above configuration steps are required only if the user starts with a newly-installed GDS. As soon as the tests have been started once, some Directory IDs will always be configured already.

2. Set the **OUTDIR** environment variable to specify a location to which the test-specific log files are to be written. If this variable is not set, the test driver will specify the default logfile destination to be:

   $TET_ROOT/functional/directory/gds/outdir.*hostname*

   or

   $TET_ROOT/functional/directory/xds/outdir.*hostname*

   —depending on which tests are being run.

You may now run whichever of the TET-executed tests you wish.

6.2.3.3

The tests listed below configure single-machine DCE cells as part of the test environment. The cellnames are hard-coded into the test scripts; thus you cannot run these tests on more than one machine on the same LAN at the same time. If two or more cells of the same name exist on the same LAN they will intercept and respond to each other's cell broadcasts. This will cause problems with CDS which will result in failures of calls to **rpc_binding_set_auth_info( )**, typically by the CDS clerk.

The workaround is to do only one iteration of these tests at a time on any subnet.

Test	*Cellname configured*
gds_sec	**c=ie/o=digital**
switch_mt	**c=ie/o=digital**
switch_st	**c=ie/o=digital**
switch_DNS	**snidec.sni.com**

6.2.3.4 Running the Admin Tests

The Administration test driver accepts options that specify which particular suite of tests to run. The driver is invoked as follows:

local_TET.admin *test_suite*

where *test_suite* is one of the scenarios listed in the **tet_scen** file. The principal scenarios are:

all	All admin tests
gdscp	GDSCP test suite
cadm	Cacheadm testsuite
dsa	DSA testsuite
scheme	Schema testsuite
shadow	Shadow testsuite
subtree	Subtree testsuite
gdssetup	GDS Setup test suite
gds_sec	GDS Security test suite

For example:

```
% ./local_TET.admin gdscp
% ./local_TET.admin subtree
```

6.2.3.5 Running the API Tests

For the API test driver, test suites are specified by switches followed by values. The driver also will print a ‘‘help’’ message when this is specified with the **-h** switch.

The driver is invoked as follows:

local_TET.api [-c] [-h] [-l] -s *test_suite*

where:

-c Specifies that XOMS Convenience functions be used (this is the default when **all** is specified as the *test_suite*; see below).

-h Specifies that a help message be displayed.

-l Specifies that API test logs not be removed after test run.

-s *test_suite* Specifies the *test_suite* to run; *test_suite* is one of the following:

xds_all_ST	All single-threaded XDS tests
xds_all_MT	All multi-threaded XDS tests
xds_all	All XDS tests (single- and multi-threaded)
xom_all_ST	All single-threaded XOM tests
xom_all_MT	All multi-threaded XOM tests
xom_all	All XOM tests (single- and multi-threaded)
xoms_all_ST	All single-threaded XOMS tests
xoms_all_MT	All multi-threaded XOMS tests
xoms_all	All XOMS tests (single- and multi-threaded)
switch_all_ST	All single-threaded SWITCH tests
switch_all_MT	All multi-threaded SWITCH tests
switch_all_DNS	All typeless SWITCH tests
switch_all	All SWITCH tests (single- and multi-threaded and typeless)
mhs_all	All MHS tests
all_no_switch	All of the above except switch tests
all	All of the above

For example:

```
% ./local_TET.api -h
% ./local_TET.api -s switch_all_ST
```

```
% ./local_TET.api -s xds_all
```

6.2.3.6 How to Interpret Test Results

Two kinds of output are generated by the GDS functional tests run under TET:

- The TET journals, found at

  $TET_ROOT/functional/directory/gds/results

  and:

  $TET_ROOT/functional/directory/xds/results

  Journals produced by TET provide a synopsis of what happened during a test's execution. Details about the kind of information contained in the journals may be found in the TET documentation in the source tree, at

  dce-root-dir/**dce/src/test/tet/doc**

  In general, the journals contain statements that indicate whether the testcase passed, failed, or did something else.

  The **TET_filter.*** scripts have been provided to help you organize the journal information into a more manageable format. The formats of the reports output by the filters vary, but each is self-explanatory. You run the filter by specifying the relative path to the journal file you wish to filter; for example:

  ```
  % tools/filter_TET.api results/0001e/journal
  ```

  This will produce a file called **journal.log** in your current working directory.

- The Test-Specific output files, found at

  $OUTDIR

  These files are not necessary for determining the pass/fail status of the test. They contain supplementary information not contained in the journal file; this information may be useful for debugging test problems, or simply as further verification that a test has passed.

  The number, content, and format of these files are all specific to the test being executed.

6.3 The XDS Test Tool xt_test

The following sections describe the procedures necessary to use the XDS test tool, **xt_test**, which can be used to run individual test cases. Note that the API test driver should be used to run suites of tests; this driver uses **xt_test** to invoke the individual tests. See ''Running the API Tests'', earlier in this chapter.

The **xt_test** program is an interpretive Directory test driver using the XDS/XOM API. It allows the construction of testcases using an interpreted notation which follows closely the form of the XDS interface, without the disadvantages of compilation. The XOM public objects used are hard-wired into the file **xt_parms.h**. As a result, the creation of new testcases using existing data is easy; however, alteration to the data or additions to it require recompilation and linking.

The **xt_test** tool is invoked as follows:

xt_test { **-i** *testcase* | **-a** *testcases* \
| **-t** *testcases*} [**-o** *logfile*] \
[**-n** *number*] [**-c**] [**-v**] [**-0**]

Where the flags and parameters have the following meanings:

-i *testcase* Specifies that a single testcase (named by the testcase file parameter) be run.

-a *testcases* Specifies that the parameter be interpreted as the name of a file containing a list of testcase file names, each of which is to be run in turn.

If no parameter is present, the filename **Testcases** is used.

-t *testcases* Specifies that the parameter be interpreted as the name of a file containing a list of testcase file names, which are to be run in parallel using threads, except for the first and last entries in the list, which are to be run in single-threaded mode before and after, respectively, the testcases specified between them.

Option **-t** is available only if the client and tester are built with **THREADSAFE** defined.

-o *logfile* Specifies the logfile name (if **D2_LOG_DIR** is defined, the default is **$D2_LOG_DIR/xt_test.log**; otherwise the default is **$HOME/xt_test.log).**

-n *number* Specifies the number of iterations (the default is 1).

-c Specifies conversion of objects to string and back (convenience library). A subset of XOM objects is converted to a string, which is logged, and then back to an object. **om_get( )** is performed on this object to test its syntactic validity. The subset of objects is that which is recognised by the standard version of the XOM Object Information file **xoischema**.

Note that the **-c** flag is required when running convenience library (**xoms**) testcases.

-v Specifies verbose output as an aid to debugging the tester itself; additional output is logged.

-0 Prints version information and exits.

When **xt_test** is invoked with no parameters, or with invalid parameters, it produces a brief message describing the usage options.

Before using the tester, GDS must be configured and activated. Since there are scripts to do this when runnning tests under TET, the simplest way to configure GDS is to run some tests under TET before using **xt_test**. The Admin tests scheme could be used, since

they run quickly.

6.3.0.1 XDS/XOM/XMH/Switch Tests

The following sections describe the XDS/XOM tests.

6.3.0.1.1 General

The **xom**, **xoms**, **xds**, **switch** and **mhs** testcases are found at:

dce-root-dir/**dce/src/test/functional/directory/xds/ts/xom/lib**
dce-root-dir/**dce/src/test/functional/directory/xds/ts/xoms/lib**
dce-root-dir/**dce/src/test/functional/directory/xds/ts/xds/lib**
dce-root-dir/**dce/src/test/functional/directory/xds/ts/switch/lib**
dce-root-dir/**dce/src/test/functional/directory/xds/ts/mhs/lib**

respectively.

The non-threaded tests in each suite are divided into four groups, whose expected result is always to pass:

basic	Basic functionality tests
valid	More advanced tests, expecting success
invalid	More advanced tests, expecting failure
stress	Tests of capacity limits (These tests are slow.)

Assessing the results of the threads tests is not always as straightforward as for the other tests, since the parallel-running tests can influence each other. There are four groups of threads tests:

threads_as	in which all actions are expected to succeed
threads_af	in which all actions are expected to fail
threads_os	in which one success is expected
threads_up	in which the outcome is unpredictable

The first two cases are easily interpreted: the outcome will be either success or failure of the test's action, which (if it is the outcome expected) will be equivalent to the test's passing. Thus all these tests should pass.

The third case requires an inspection of the outcomes of all the tests, and confirmation that only one action has succeeded. Since startup and shutdown should also succeed, the expected (successful) result consists in three threads passing and the rest failing.

In the last group, success or failure *per se* is not so important, since this depends on the non-deterministic interleaving of the tests: the purpose of these tests is to show whether

the directory system is robust enough not to crash or deadlock when confronted with a complex mix of simultaneous interacting requests.

In summary, the desired outcomes are:

threads_as	All tests will pass
threads_af	All tests will pass
threads_os	Three tests will pass, the rest fail
threads_up	There will be no hanging or crashing (passes/failures unimportant)

6.3.0.1.2 Running Individual Threads Tests

Each thread testcase consists of four parts. For example, for test **this1v**, there exist four files:

- **STARTthis1v**
- **BODYthis1v**
- **SHUTDOWNthis1v**
- **T10this1v**

To run a test, for example **T10list1v**, do the following:

```
xt_test -t T10list1v
```

and the other three parts will be called implicitly.

In general, the names of runnable threads tests start with an initial capital ‘‘T’’, followed by a number indicating how many threads will be created, and ending with the name of the test itself.

6.3.1 Examples

Following are some examples of **xt_test** usage.

- To run the testcase file *read1v*:

```
xt_test -i read1v
```

- To run the testcase files named in file *Testcases* sequentially:

```
xt_test -a
```

- To run the testcase files named in file *my_tests* sequentially:

```
xt_test -a my_tests
```

- To run the threaded testcase file *T10add_entry37i*:

```
xt_test -t T10add_entry37i
```

6.3.2 MAVROS Compiler Tests

The source files for the test drivers, input files, and reference output files for the MAVROS tests are located in the

dce-test-dir/**test/directory/gds/mavrostest**

directory.

Running the MAVROS compiler test consists of executing the **test_mvr.sh** script. The script executes the test program to verify the coding routines can be executed correctly. The **test_mvr.sh** shell script executes the test programs in the **install** tree. Both **test_mvr** and **oidt** are executed. If these programs execute correctly, the **test.errlog** (for **test_mvr**) and **oidt.errlog** (for **oidt**) error logs will be empty.

Note: Once **llib-ltest.ln** is up to date, **lint** is not actually executed. After reexecuting **test_mvr.sh**, **lint.log** may be empty even though there are **lint** errors in the code.

6.3.3 Testing GDS Intercell Operation

This section contains the steps followed to hand-test using GDS for intercell communications. The typical test scenario involves two single-machine cells configured with X.500 names; in the steps given below, these machines are named ‘‘prague’’ and ‘‘gemini’’. The cell names used are, respectively:

```
/.../c=us/o=osf/ou=dce/cn=prague
/.../c=us/o=osf/ou=dce/cn=gemini
```

The cell located on ‘‘prague’’ will be considered the foreign cell, and the cell located on ‘‘gemini’’ will be considered the local cell.

1. Configure the foreign cell, with GDA.
2. Configure the local cell, with GDA.
3. Start GDS on the local cell.
4. Administer the DUA Cache in the local cell with **gdssysadm**, as follows:
   a. Prime cache with client address (option 5):

```
T-selector: Client
NSAP:       TCP/IP!internet=127.0.0.1+port=21010
```

b. Prime cache with name of default DSA (option 1):

```
name:     c=us/o=osf/ou=dce/cn=gemini/cn=gemini-dsa
```

Select ‘‘DSA-Type’’ from the attribute list and provide the following values:

```
dsa-type:   default/local’
T-selector: Server
PSAP:       TCP/IP!internet=127.0.0.1+port=21011
```

5. Get UUIDs and towers of foreign cell.

Logon to the foreign machine and type:

cdscp show cell as gds

You will get output that looks like this:

```
                SHOW
                CELL   /.../c=us/o=osf/ou=dce/cn=prague
                  AT   1994-09-28-15:01:02
     Namespace Uuid = 6e22b59f-dad0-11cd-a4ac-0000c0a1de56
Clearinghouse Uuid = 6d17b15e-dad0-11cd-a4ac-0000c0a1de56
Clearinghouse Name = /.../c=us/o=osf/ou=dce/cn=prague/prague_ch
      Replica Type =   Master
             Tower = ncacn_ip_tcp:130.105.5.83[]
             Tower = ncadg_ip_udp:130.105.5.83[]
```

6. Create object for foreign cell in DSA of local cell:

a. Logon to the local DSA on the local cell.

.b. Create all superior objects; for example:

```
c=us/o=osf/ou=dce
```

c. Create object for the foreign cell:

```
c=us/o=osf/ou=dce/cn=prague
application-process
```

Select ‘‘CDS-Cell’’ and ‘‘CDS-Replica’’ from the attribute list and provide the following values:

CDS-Cell	Cut and paste namespace UUID
	Root directory UUID is same as namespace UUID
	Name of cell is root directory name
CDS-Replica	Replica type is MASTER
	Cut and paste clearinghouse UUID

Cut and paste clearinghouse name

Cut and paste towers

7. Have the cells exchange keys.

   On the local cell, **dce_login**, enter **rgy_edit**, and type the following:

```
rgy_edit=> cell /.../c=us/o=osf/ou=dce/cn=haven
Enter group name of the local account for the foreign cell: none
Enter group name of the foreign account for the local cell: none
Enter org name of the local account for the foreign cell: none
Enter org name of the foreign account for the local cell: none
Enter your password:    enter local cell cell_admin password
Enter account id to log into foreign cell with: cell_admin
Enter password for foreign account: enter foreign cell cell_admin password
Enter expiration date [yy/mm/dd or 'none']: (none)
```

8. Verify GDS intercell operation.

   Test unauthenticated access. Type:

   cdscp show dir /.../c=us/o=osf/ou=dce/cn=prague'

   You should perform this command not as root, but as an unauthenticated system user. Type **klist** to verify that you in fact have no credentials.

9. Test authenticated access.

   dce_login and issue the same **cdscp** command as in the previous step.

6.4 GDS Runtime Output and Debugging Output

The GDS component outputs server information of all kinds via the DCE serviceability component. The *OSF DCE Administration Guide*, Chapter 5, Section 5.5 describes how to control the various kinds of information (including trace output) available from GDS via serviceability.

6.4.1 Test Plans

Refer to Chapter 1 of the *OSF DCE Release Notes* for the location of the DCE test plans on the DCE distribution tape.

Chapter 7. DCE Distributed Time Service

7.1 Overview

The DCE Distributed Time Service (DTS) synchronizes the clocks on computer systems connected by a network. DTS uses the client/server model and provides a command-driven management interface for configuration and management functions. An Application Programming Interface (API) is provided for application developers to write programs that use DTS services. Finally, DTS provides a Time-Provider Interface to obtain Coordinated Universal Time (UTC) from time-provider devices, as well as several example time provider implementations.

The UTC-based time structure in DTS uses 128-bit binary numbers to represent the time value internally. These binary timestamps consist of the time in terms of 100-nanosecond units since 00:00:00:00, October 15, 1582, the count of 100-nanosecond units of inaccuracy applied to the preceding time, the time differential factor expressed as the signed number of minutes east or west of Greenwich mean time, and the DTS version number. The inaccuracy represents the upper bound on all nonfaulty sources of inaccuracy (for example, clock drift, resolution of discrete computer clock, software communication path lengths, and so on). The clerks and servers compute a correct time from time values obtained from several servers or from a time provider. The synchronization is accomplished by intersecting intervals. This algorithm provides fault detection and withstands the failures of a small number of servers.

7.2 Setup, Testing, and Verification

The following types of DTS tests are shipped with DCE:

- API tests

- Synchronization tests
- Control program tests
- Time conversion tests
- Kernel (or user-mode) tests

These tests are described in more detail in following sections. Results from tests described in the test plan are also included.

The DTS test directory contains three subdirectories: **control**, **common** and **service**. The first, **control**, contains a script which tests **dtscp** command line syntax. The second, **common**, contains the tests. The third, **service**, contains **dtss-graph.c**, the graph tool for displaying the test environment.

Before executing the test cases, you must configure DTS for testing, using the instructions found in the following section of this chapter. You can run tests on the configurations described in that section.

7.2.1 Installing DTS Functional Tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config** will install the tests you select at the path you specify, and will create a softlink (called **/dcetest/dcelocal**) to that location. The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the sections below, which refer to the tests' source or build locations.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the final location of the built test tree). For the DTS functional tests, this path should be the location, on your machine, of:

dce-root-dir/**dce/install**

—which is the DCE **install** tree (for more information on the structure of the DCE tree, see Chapter 3 of the *OSF DCE Release Notes*).

Thus, **dcetest_config** will install the DTS functional tests at:

/dcetest/dcelocal/test/time/

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* (if you choose) functional tests with **dcetest_config**; for test configuration and execution you must follow the instructions in the sections below.

Refer to Chapter 11 of this guide for further information on using **dcetest_config**.

7.2.2 Building the Tests

The tests are delivered as source: you must build the executables on your system. To do so, you must pick up the appropriate headers to define structures such as **timespec**, **timeval**, and **utc_t**. Depending on the platform to which you are porting, you may have to change the include files. Kernel structures can differ from non-kernel structures of the same name, so you will have to keep straight which structures correspond with which symbols. For example, you may have to modify **test_kernel.c** in the **common** subdirectory to define **_TIMESPEC_T_** and include <**utctime.h**>. In addition, you may have to include <**sys/time.h**> instead of <**time.h**>. Once you start to build the tests, these constraints will become obvious; if the wrong files are included, you are likely to get compiler warnings.

7.2.3 DTS Setup

Before running many DTS tests, you must first configure a DCE cell. Refer to the following chapters of the *OSF DCE Administration Guide—Introduction* for information on configuring a DCE cell:

- Overview of The DCE Configuration Script
- Phase One: Initial Cell Configuration
- Phase Two: Configuring a DCE Client and Other DCE Services

7.2.4 API Tests

The **rantest_api.c** file in the

dce-root-dir/**dce/src/test/time/common**

directory generates random test cases for the API. The program stops and displays information if a failure is found.

Note that these tests do not require to be executed in a DCE cell; only a built and installed **libdce** (DCE library) is needed.

The test is invokes as follows:

rantest_api [*count***]**

where *count* is an integer specifying how many iterations the test should execute.

The following compiler arguments, which are defined in **common/Makefile**:

- **-Dunix**
- **-DSYSTEM_FIVE**

generate test invocations of the standard C library functions **gmttime( )** and **localtime( )**, respectively.

Note: Certain operating systems have a bug in the **localtime(3)** code which manipulates the Daylight Savings Time switch on the last Sunday of October 1974. The presence of this bug will cause a failure in the **rantest_api** test for that date. See the comments under **#ifdef NOV1974_BUG** in **rantest_api.c** for further information.

7.2.5 Synchronization Testing

In order to perform useful synchronization testing, you should have at least three **dtsd** servers running (in a running DCE cell).

The **dtscp** control program command **set synch trace true** tells the time service daemon **dtsd** (see the **Makefiles** under **control** and **service**) to write the input and output values for each synchronization to:

*dcelocal***/var/adm/time/dts-inacc.log**

A separate program, **dtss-graph**, located in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/time/service**

directory (where *machine* is your system type), processes the trace into a PostScript file of a graph of the synchronization. See the **Makefile** under:

dce-root-dir/**dce/src/test/time/service**

The **dtss-graph** command allows the user to see a large number of synchronizations quickly and in detail. The last page of the output includes statistics describing the interaction between the tracing node and all the servers it queried during the test.

To use these tools to perform synchronization testing, do the following:

1. Set up a test environment that includes one or more (preferably three) servers.
2. Enable the graph trace (using the **dtscp** control program command **set synch trace true**) on a sample of the nodes, including at least one client and one server. This causes DTS to write a trace file for the **dtss-graph** command in:

   *dcelocal***/var/adm/time/dts-inacc.log**

   If there is a time-provider in the test, the test should include a trace from the daemon connected to the time-provider. (Note that DTS starts a new trace file each time the service restarts.)
3. Process the traces with **dtss-graph** when the test run is complete. Enter **dtss-graph -help** for options.

4. Print the graphs on a PostScript printer and examine the output.

7.2.6 dtscp Testing

The **test_dtscp.ksh** script is a functional test which runs **dtscp** commands and compares the resulting output to the contents of an ‘‘expected results’’ file.

The test consists of the following parts, all located in the

dce-root-dir/**dce/src/test/time/control**

directory:

test_dtscp.ksh	Test driver script.
dtscp.ksh	The test script.
test_dtscp_clerk.templ	Expected results template for **dtscp** clerk.
test_dtscp_server.templ	Expected results template for **dtscp** server.

The test is invoked as follows:

test_dtscp *type* [*remote_hostname*] [*machine_type*]

— where *type* is either **server** or **clerk**, depending on what type of DTS machine the test is being executed on, server or clerk; and *machine_type* is **AIX**, **OSF1**, or **HPUX**.

When invoked, the test edits the template files with local information such as the machine’s hostname, clock adjustment rate, and the next TDF change. This information is placed in a file named **test_dtscp_clerk.exp** or **test_dtscp_server.exp**, depending on whether the clerk or server form of the test is running. (The contents of this file is used to determine the expected results when the remote test is executed.)

The test will report whether the expected matches the actual output, and will record any differences between the two in a file named **test_dtscp.diff**. **test_dtscp.ksh** will also test commands which have variable output (such as the current time) and report any failures. The actual results of running the test will be placed in a file named **test_dtscp.act**.

Note that the server test should be run with a server that has just been started, with no time provider, in a cell with no other DTS servers running. The clerk test should be run with a clerk that has just been started, in a cell with no *global* servers, and at least one local server running.

Sample output from a clerk test:

```
START DCE time functional test: dtscp.ksh; DATE: Thu Oct 21 11:53:41 EDT 1993
The value of NODE_TYPE is clerk
Begin test of dtscp command structure (Thu Oct 21 11:53:43 EDT 1993)
You are running this test on a dts clerk (right?)
Actual output is in file test_dtscp.act
Expected output is in file test_dtscp_clerk.exp
Compare actual output to expected output
Actual output matches expected output
```

```
Execute variable commands
PASSED, Test 1 "TEST_DTSCP" : test ran successfully
END DCE time functional test: dtscp.ksh; DATE: Thu Oct 21 11:59:16 EDT 1993
```

Sample output from a server test:

```
START DCE time functional test: dtscp.ksh; DATE: Thu Oct 21 12:10:39 EDT 1993
The value of NODE_TYPE is server
Begin test of dtscp command structure (Thu Oct 21 12:10:42 EDT 1993)
You are running this test on a dts local server (right?)
Actual output is in file test_dtscp.act
Expected output is in file test_dtscp_server.exp
Actual output matches expected output
Execute variable commands
PASSED, Test 1 "TEST_DTSCP" : test ran successfully
END DCE time functional test: dtscp.ksh; DATE: Thu Oct 21 12:15:22 EDT 1993
```

7.2.7 Additional DTS Testing

The following subsections describe how to run and interpret the output of some additional tests.

7.2.7.1 Timezone Conversion Test

The DTS timezone conversion test (**test_zonecvt**) is invoked as follows:

test/time/common/test_zonecvt < time/common/zonecvt.dat

Note that you must install all of the files built for **/etc/zoneinfo** in order to run this test (however, a running DCE cell is *not* required). The files should be located at:

dceshared/**etc/zoneinfo**

where *dceshared* is a link that **dce_config** will try to create from **/etc/zone/info** (note however that your operating system may already have something at this location and if it does **dce_config** will not overwrite it).

How To Set up DTS to use Local Zone Information

By default, DTS uses the GMT time zone, so time information you get from either

dtscp show current time

or the API function **utc_gettime( )** will be in GMT.

The default time zone information used will be that in **localtime**; **dce_config** will usually link this name to the correct zone, so that (for example) **/etc/zoneinfo/localtime** on an HP-UX machine will have been linked to **/etc/zoneinfo/US/Eastern**. If this has not been done, simply set the **TZ** environment variable to the desired zone. For example:

TZ=US/Eastern

or:

TZ=EST5EDT

for a POSIX system.

If necessary, you can obtain the full distribution of **localtime** from:

ftp.uu.net:usenet/comp.sources.unix/volume18/localtime3/part*XX***.Z**

where *XX* runs from 01 to 07.

When using API functions, remember to call **tzset( )** before anything else.

To change the localtime to a new zone, you can use **zic** with the **-l** option.

7.2.7.2 Kernel Library Tests

The following tests:

- **test_kernel**
- **test_kernel-kernel**
- **test_kernel-user**

are built from source in the

dce-root-dir/**dce/src/test/time/common**

directory. The tests have similar output. The first, **test_kernel**, links in **libdce**. The **test_kernel-kernel** test links in **libutc-kernel.a** and runs in kernel mode; the **test_kernel-user** test links in **libutc-user.a** and runs in user mode.

Note that your platform must support both the kernel-mode and the user-mode DTS libraries in order for all three versions of this test to be built. See ‘‘Building and Linking’’ earlier in this chapter.

7.2.7.3 DTS Hand Tests

The text file

dce-root-dir/**dce/src/test/time/hand-tests**

consists of instructions for performing, by hand, further DTS functional testing. These tests are intended to be run by hand in the configurations specified.

7.2.8 Test Run Examples

Some annotated examples of test runs follow.

 # test_kernel

```
> resolution = 3970000 nanosecs
> drift = 1/20000
> frequency = 1000000000 nanosec / sec
```

This test checks various pieces of information that the kernel knows and DTS needs. The 3.97 milliseconds shown is the correct clock tick for the DECstation 3100. (The clock ticks at 256 hz = 3.90625, but since the kernel actually uses micro-seconds, once a second there's an extra 64 micro-seconds added to keep the clock correct; thus the longest tick is: 3.906 + 0.064 = 3.970). The drift is equal to 1 part in 20,000. The frequency of 1,000,000,000 nanosecs/sec indicates that no clock training is occuring (yet). Note that the first two numbers will be different on different platforms.

```
> 1992-06-09 22:04:40.045538
> 1992-06-09 22:04:40.045537000 +/- 0 00:00:00.052798900 (GMT)
> 1992-06-09 17:04:40.045537000 +/- 0 00:00:00.052798900 (GMT-5:00 = -18000)
> 1992-06-09 22:04:40.049443000 +/- 0 00:00:00.052798900 (GMT)
> 1992-06-09 17:04:40.049443000 +/- 0 00:00:00.052798900 (GMT-5:00 = -18000)
```

This section shows the output of three system calls: one to **gettimeofday( )**, and two to **utc_gettime( )**. They should give the same answer to within a few milliseconds. (If two calls to **utc_gettime( )** should monotonically increase, they do.)

```
> 1992-06-09 22:04:40.068973000 +/- 0 00:00:00.052799900 (GMT)
> 1992-06-09 17:04:40.068973000 +/- 0 00:00:00.052799900 (GMT-5:00 = -18000)
> 1992-06-09 22:04:40.088504
> Setting time forward 1 second
> Leap second set to: 1992-06-09 22:04:45.088504000 +/- 0 00:00:00.000000000 (GMT)
> 1992-06-09 22:04:41.088504
> 1992-06-09 22:04:41.084597000 +/- 0 00:00:00.004021000 (GMT)
> 1992-06-09 18:04:41.084597000 +/- 0 00:00:00.004021000 (GMT-4:00 = -14400)
```

This section show a **set** of the time forward 1 second. Note that the time did in fact go forward about one second (from 40 to 41).

```
> 1992-06-09 22:04:44.108035
> 1992-06-09 22:04:44.104127000 +/- 0 00:00:00.004171000 (GMT)
> 1992-06-09 18:04:44.104127000 +/- 0 00:00:00.004171000 (GMT-4:00 = -14400)
> 1992-06-09 22:04:47.115847
> 1992-06-09 22:04:47.111939000 +/- 0 00:00:01.004322000 (GMT)
> 1992-06-09 18:04:47.111939000 +/- 0 00:00:01.004322000 (GMT-4:00 = -14400)
> 1992-06-09 22:04:47.127564
```

This section verifies that the inaccuracy increases, due to drift, and also the second should be increased by one second because of the (possible) leap second.

```
> Adjusting time backwards 0.1 second
> Leap second set to: 1992-06-09 22:04:52.127564000 +/- 0 00:00:00.000000000 (GMT)
> 1992-06-09 22:04:47.127565
> 1992-06-09 22:04:47.123657000 +/- 0 00:00:00.103971000 (GMT)
> 1992-06-09 18:04:47.123657000 +/- 0 00:00:00.103971000 (GMT-4:00 = -14400)
> 1992-06-09 22:04:50.109214
> 1992-06-09 22:04:50.105306000 +/- 0 00:00:00.074675000 (GMT)
> 1992-06-09 18:04:50.105306000 +/- 0 00:00:00.074675000 (GMT-4:00 = -14400)
> 1992-06-09 22:04:53.090863
> 1992-06-09 22:04:53.086955000 +/- 0 00:00:01.045340000 (GMT)
> 1992-06-09 18:04:53.086955000 +/- 0 00:00:01.045340000 (GMT-4:00 = -14400)
```

This section executes an **adjust** and verifies that the inaccuracy decreases. The inaccuracy decreases during an **adjust** under the assumption that the clock is being made more correct.

```
> Ending adjustment prematurely
> 1992-06-09 22:04:53.098596
> 1992-06-09 22:04:53.094689000 +/- 0 00:00:01.045262000 (GMT)
> 1992-06-09 18:04:53.094689000 +/- 0 00:00:01.045262000 (GMT-4:00 = -14400)
> 1992-06-09 22:04:53.110314
> 1992-06-09 22:04:53.106407000 +/- 0 00:00:01.045263000 (GMT)
> 1992-06-09 18:04:53.106407000 +/- 0 00:00:01.045263000 (GMT-4:00 = -14400)
> 1992-06-09 22:04:53.118126
```

This section stops the adjustment and verifies that inaccuracy starts increasing again.

```
> Adjusting time backwards 0.1 second
> Leap second set to: 1992-06-09 22:04:58.118126000 +/- 0 00:00:00.000000000 (GMT)
> TDF change set to : 1992-06-09 22:05:03.000000000 +/- 0 00:00:00.000000000 (GMT)
> 1992-06-09 22:04:53.118127
> 1992-06-09 22:04:53.114219000 +/- 0 00:00:00.103971000 (GMT)
> 1992-06-09 17:04:53.114219000 +/- 0 00:00:00.103971000 (GMT-5:00 = -18000)
> Adjustment should have completed.
> 1992-06-09 22:06:52.417993
> 1992-06-09 22:06:52.414085000 +/- 0 00:00:00.268508400 (GMT)
> 1992-06-09 18:06:52.414085000 +/- 0 00:00:00.268508400 (GMT-4:00 = -14400)
```

```
> 1992-06-09 22:06:52.445062
> 1992-06-09 22:06:52.441154000 +/- 0 00:00:00.268275400 (GMT)
> 1992-06-09 18:06:52.441154000 +/- 0 00:00:00.268275400 (GMT-4:00 = -14400)
```

This section allows the adjustment to complete by itself and verifies that the inaccuracy starts increasing again. It also verifies that the timezone changes back from -5:00 to -4:00.

```
> Ending adjustment again
> 1992-06-09 22:06:52.452795
> 1992-06-09 22:06:52.448888000 +/- 0 00:00:00.268198400 (GMT)
> 1992-06-09 18:06:52.448888000 +/- 0 00:00:00.268198400 (GMT-4:00 = -14400)
> 1992-06-09 22:06:52.460607
> 1992-06-09 22:06:52.456700000 +/- 0 00:00:00.268198400 (GMT)
> 1992-06-09 18:06:52.456700000 +/- 0 00:00:00.268198400 (GMT-4:00 = -14400)
```

This section confirms that ending the **adjust** (after it has run out) does not cause any problems, and that the inaccuracy increases (or stays the same).

7.3 DTS Runtime Output and Debugging Output

The DTS 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 DTS via serviceability.

7.3.1 Normal DTS Server Message Routing

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

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

The following sections describe each of these methods.

7.3.1.1 Routing File

If a file called

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

exists when DTS is brought up (i.e., when **dtsd** 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 DTS 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.

7.3.1.2 Routing by the dcecp log Object

Routing of DTS server messages can be controlled in an already-started cell through the **dcecp log** object. The name used to manipulate the routes is the server entry name, which is usually:

hosts/*machine_name***/dts-entity**

See the **log.8dce** reference page in the *OSF DCE Command Reference* for further information.

7.3.2 Debugging Output

Debugging output from DTS can be enabled (provided that DTS 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_DTS_DBG** environment variable, before bringing up DTS. 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* 6
[*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

*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 DTS, the sub-components are as follows:

general	General server administration
events	Events received and acted upon
arith	Math operations
ctlmsgs	Control messages received
msgs	Messages received
states	Server state transitions
threads	Thread interactions
config	Server/cell configuration
sync	Server/synchronization interactions

For example, the string

"dts:*.1,events.3:TEXTFILE.50.200:/tmp/DTS_LOG

sets the debugging level for all DTS sub-components (*except* **events**) at 1; **events**’s level is set at 3. All messages are routed to **/tmp/DTS_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 DTS serviceability messages, and the sub-component list, can be found in the DTS sams file, at:

dce-root-dir/**dce/src/time/common/dts.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’’.

7.3.3 Test Plans

Refer to Chapter 1 of the *OSF DCE Release Notes* for the location of the DCE test plans on the DCE distribution tape.

Chapter 8. DCE Security Service

8.1 Overview

The DCE Security Service manages the rights and identities of users within a given cell. It does so primarily by representing and certifying that identity to applications running on separate systems in the environment

Some local system functions are also provided in an effort to preserve the location transparency of the distributed environment. By supplanting the conventional machine login and account management utilities with replacements that consult the network user registry, users are free to use any system in the environment, provided that the local administrator does not restrict access. In this way, systems become sharable resources related to objects in the file or name system.

The DCE Security Service consists of the following cooperating subcomponents:

- Registry Service

  The Registry Service manages user, group, and account information and stores administrative policies regarding the characteristics of accounts that can access the distributed system. The Registry Service is composed of a set of client services to add, manipulate, and delete entries in the server's registry database. The Kerberos database, containing the secret keys of all registered principals, is contained in the registry database. You can replicate the registry database within a cell, and any changes to the master registry are propagated to the replicas. With this single logical registry, a user can log in and authenticate from any system in the cell.

- Authentication Service

  The Authentication Service is an encryption-based authentication protocol that uses a modification of the Needham-Schroeder authentication algorithm.

  The Authentication Service allows principals defined as accounts in the registry to exchange credentials and establish mutually authenticated communications. The Authentication Service is the network service that supplies the simple tickets and

session keys necessary for such communications. DCE's Authentication Service is analogous to Kerberos' Key Distribution Center (KDC).

- Access Control List (ACL) facility

  All objects in DCE can have an ACL. The ACL facility consists of a single ACL editor tool (**acl_edit**) and a set of APIs for ACL manipulation. Each DCE component implements its own ACL managers to process and interpret the ACL when access to the object is requested.

- Privilege Service

  The Privilege Service is a certification authority that provides a trusted mechanism to derive authorization information about principals. Authorization information includes a principal's identity expressed as a universal unique identifier (UUID) and the principal's group memberships. The Privilege Service packages this information into a privilege attribute certificate (PAC), which is then sealed in a Kerberos V5 ticket's authorization data area. After the target principal receives and verifies the ticket, the unsealed authorization data is trusted and used to make access decisions.

  The Privilege Service and the ACL facility provide authorization services to the cell.

The servers — the registry server, the authentication server, and the privilege server — are encompassed within one daemon, called **secd**.

8.2 Setup, Testing, and Verification

The following types of DCE Security Service tests are shipped with DCE:

- Basic functionality tests
  - The **update** and **query** tests
- Command tests
- API tests

These tests are described in more detail in the following sections.

Before executing the test cases, you must configure the DCE Security Service for testing using either the DCE configuration script

dce-root-dir/**dce/install/***machine*/**opt/dce1.2.2/etc/dce_config**

or the instructions found in the next section of this chapter. You can run the tests on the configurations described in that section.

8.2.1 Installing DCE Security Functional Tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config**

will install the tests you select at the path you specify, and will create a softlink (called **/dcetest/dcelocal**) to that location. The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the sections below, which refer to the tests' source or build locations.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the final location of the built test tree). For the DCE Security functional tests, this path should be the location, on your machine, of:

dce-root-dir/**dce/install**

—which is the DCE **install** tree (for more information on the structure of the DCE tree, see Chapter 3 of the *OSF DCE Release Notes*).

Thus, **dcetest_config** will install the Security functional tests at:

/dcetest/dcelocal/test/security/

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* (if you choose) functional tests with **dcetest_config**; for test configuration and execution you must follow the instructions in the sections below.

Refer to Chapter 11 of this guide for further information on using **dcetest_config**.

8.2.2 Basic Security Setup

Before running the test, configure your machine as a DCE client machine, or configure your machine as a DCE Security Server machine and run the test there. To configure the DCE Security Service for basic testing, do the following:

1. Using **mkdir**, create the **/krb5** directory on your machine.
2. Use the **dce_config** script to install the necessary files on your machine. You may install the Security Server code, the security client code, or both. Note that the **dce_config** script places the executables in

   dcelocal/**bin**

   and creates symbolic links to **/usr/bin**. Therefore, you should not need to add any paths to your **$PATH** environment variable to execute the tests.
3. Create *dcelocal*/**dce_cf.db**.

   This file is used by the Security Service to find the machine name and the name of the cell. This file should be in the following format:

```
cellname  /.../cellname
hostname  hosts/machine
```

where *cellname* is the name of your cell and *machine* is the IP host name of your machine.

4. Create the master registry database.

   The **sec_create_db** tool is used to create the initial database. This database is populated with the default principals and accounts needed to bootstrap the system. The accounts are created with the default password "**-dce-**." An alternative may be specified with the *-password* option to **sec_create_db**. This tool creates the database in the directory:

   *dcelocal***/var/security/rgy_data**

   Run **sec_create_db** as the privileged user (root) so that the database is protected appropriately.

   The **sec_create_db** command must be issued with the *-myname* switch to identify the CDS name of the server entry for this server. This name can be anything, but by convention is:

   /.../*cellname***/subsys/dce/security/master**

   - To create the database, enter

     sec_create_db -myname subsys/dce/security/master

     at the command line.

   At that point, **sec_create_db** will issue the prompt:

   ```
   Enter keyseed for initial database master key:
   ```

   In response, enter any character string, to a maximum length of 1024 characters.

   This string seeds a random key generator, which generates a random master key used to encrypt keys in the database. The master key is stored in

   *dcelocal***/var/security/.mkey**

   and can be read and written only by the privileged user (root).

   A default keytab file, **/krb5/v5srvtab**, is created to store the server keys created at this time

   The **sec_create_db** tool also creates the file

   *dcelocal***/etc/security/pe_site**

   which contains the name of the machine running the **secd**. This file contains one or more lines with the name of the target cell and the RPC string binding for a server providing security services for that cell. It has the following format:

   ```
   /.../cellname UUID@ncadg_ip_udp:XXX.XX.XXX.XXX[]
   ```

where *UUID* is the cell's security service object UUID and *XXX.XX.XXX.XXX* is the host machine's IP address.

This file provides access to security services in the absence of CDS. Therefore, if you are setting up a client machine, be sure to copy this file from the Security Server machine.

Note: The *BIND_PE_SITE* environment variable controls client use of NSI. If the environment variable is set to any value other than 0, the security code will not bother to make NSI calls. Set and export this environment variable if your configuration does not include a running CDS.

When running **sec_create_db** more than once on a host (for example, when installing a new release), you must delete the old registry database files and the default keytab file by entering:

rm -r *dcelocal***/var/security/rgy_data**
rm /krb5/v5srvtab

at the command line.

If you fail to delete the **rgy_data** directory, you will see the following error message

```
Registry: Fatal Error - at line 426 of file rgy_create.c -
- 0x171220ab - cannot create database (dce / sec)
```

If you fail to delete **v5srvtab**, you will see the following error:

```
Registry: Error - Error setting local host's key -- try
deleting old entry -
- 0x17122048 - Specified key already exists in key
store (dce / sec)
```

5. Run the servers.

The program **secd** is the process that provides the Authentication Server, Privilege Server, and Registry Server. This program must be run as the privileged user (root) and must be run on a machine that contains the database created by **sec_create_db**. In addition, the Authentication Server requires **syslogd** to be running on the local server machine.

Running the server with the **-debug** switch causes it to run in the foreground. The **-verbose** switch prints diagnostic and auditing information. This mode is recommended for early integration testing. It is also recommended that you enable **syslog** and examine the log while running the authentication server.

To do so, use the following **syslog.conf** information:

***.mark.info /usr/spool/adm/syslog**
***.err /usr/spool/adm/syslog**

and **tail** the

/usr/spool/adm/syslog

output file.

6. Make sure the **dced secval** service is running.
7. Set up a Security client.

   Use the **dce_config** script to install the Security client executables.

   Assume that a registry has been created and a Security Server started on host ‘‘laurel’’ which has IP address 15.22.144.215. Assume that the cell name is ‘‘/.../com/hp/apollo.’’

   The contents of the

   dcelocal/**dce_cf.db**

   file should appear as follows:

   ```
   cellname /.../com/hp/apollo
   hostname hosts/laurel
   ```

   The contents of the

   dcelocal/**etc/security/pe_site**

   file should appear as follows:

   ```
   /.../com/hp/apollo UUID@ncadg_ip_udp:15.22.144.215[]
   ```

   To make host ‘‘hardy’’ a client, follow these steps:

   1. On ‘‘hardy’’, create:

      dcelocal/**dce_cf.db**

      Its contents should appear as follows:

      ```
      cellname /.../com/hp/apollo
      hostname hosts/hardy
      ```

   2. On ‘‘hardy’’, create:

      dcelocal/**etc/security/pe_site**

      Its contents should appear as follows:

      ```
      /.../com/hp/apollo UUID@ncadg_ip_udp:15.22.144.215[]
      ```

      You may copy this file directly from the Security Server machine.

   3. Return to the host ‘‘laurel’’. On ‘‘laurel’’, do the following:

      - Run **dce_login** to login as a user with privileges to edit the registry database. See the following section, ‘‘The dce_login Utility,’’ for more information about **dce_login**.

- Run **rgy_edit**. Add the principal ‘‘hosts/hardy/self’’ and an account for that principal. Remember the key (password) you specified for ‘‘hardy’s’’ account. See the ‘‘DCE Security Service’’ part of the *OSF DCE Administration Guide—Core Components* for instructions on how to use **rgy_edit**.

4. Return to the host ‘‘hardy,’’ and perform the following steps.
   - Run **rgy_edit** unauthenticated (without using **dce_login**). Use the **ktadd** command to add the key for ‘‘hosts/hardy/self.’’
   - Make sure the **dced secval** service is running on ‘‘hardy’’.

   Now you can run the security tests on either the server machine ‘‘laurel’’ (which is also a client) or on the client machine ‘‘hardy.’’

8.2.2.1 The dce_login Utility

The **dce_login** sample application allows users to obtain DCE credentials without modifying their local OS state. This application constructs a credential cache that supports authorization service **dce** and then execs the user’s shell. The shell is inherited from the parent process if the **SHELL** environment variable is set. Command usage is:

dce_login [*user_name* **[** *password* **]]**

If the user’s password or the user name is not specified on the command line, **dce_login** will prompt you for the data.

You can use **dce_login** to login as a registry user with privileges to edit the registry database. You will have to have these privileges for most of the tests described in this chapter.

8.2.3 Basic Functionality Tests

These tests can be used to ensure that the basic functionality of the Security Service is working properly.

8.2.3.1 The update Test

You must execute the **dce_login** command as a user with privileges to modify the registry before running this test. If you configured your machine using the **dce_config** script, then whatever user the script’s **celladmin** variable was set to has registry-modifying privileges.

The

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/commands/rgy/update**

test checks basic update functionality by adding some specified number of principals, groups, and organizations to the registry database. Only PGO (principal, group, organization) and account objects are checked; policy and property updates are not checked.

To run the **update** test, **cd** to its directory and enter (on one line):

update -a | -r [-p *principal* **-pw** *password* **] **
**num_accts site [person_prefix [group_prefix [org_prefix]]] **
[-d | -d*rpc_debug_flags*]

where:

-a Specifies that entries are to be added to the registry.

Note that either the **-a** or the **-r** flag *must* be specified.

-r Specifies that entries are to be removed from the registry.

Note that either the **-r** or the **-a** flag *must* be specified.

-p <*principal*> Specifies the principal name to be logged in. *principal* should be a principal with registry-modifying privileges.

-pw <*password*> Specifies the password of the principal.

Note that either *both* **-p** and **-pw** must be specified or *neither* should be specified. In the latter case, the test will prompt for the name and password of the principal.

-d Specifies the minimal level of debug output. This parameter is optional.

-d*rpc_debug_flags* Allows you to specify the amount of debug output desired. Some useful *rpc_debug_flags* settings are the following:

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

num_accts Specifies the number of new accounts to add to the registry database.

cellname Specifies the cell whose registry is to be updated. This cellname should include the global prefix ‘‘**/.../**’’.

person_prefix Specifies a prefix for all update entries added to the **person** domain. The default prefix is **up.da.te._.te.st/per**.

group_prefix Specifies a prefix for all update entries added to the **group** domain. The default prefix is **upd_test/grp**.

org_prefix Specifies a prefix for all update entries added to the **org** domain. The default prefix is **upd_test/org**.

For example, enter

update 100 *cellname*

where **100** is the number of new accounts and *cellname* is the name of the cell. **update** will then prompt you for your principal name and password. Note that if you are not authorized to edit the registry (if you have not executed **dce_login** to login as a user with those privileges), then the test will fail. If the update is successful, the output looks like the following:

```
Opening registry at site /.../cellname
TIMING: Account add [1.380000]user+sys [20.100334] real time (20 in, 20 out)
TIMING: (Per call aggregate) [0.069000]user+sys [1.005017] real time
TIMING: (Per call period 20) [0.069000]user+sys [1.005017] real time
TIMING: Account add [2.690000]user+sys [39.817963] real time (40 in, 40 out)
TIMING: (Per call aggregate) [0.067250]user+sys [0.995449] real time
TIMING: (Per call period 20) [0.065500]user+sys [0.985881] real time
TIMING: Account add [4.020000]user+sys [60.174643] real time (60 in, 60 out)
TIMING: (Per call aggregate) [0.067000]user+sys [1.002911] real time
TIMING: (Per call period 20) [0.066500]user+sys [1.017834] real time
TIMING: Account add [5.200000]user+sys [80.262026] real time (80 in, 80 out)
TIMING: (Per call aggregate) [0.065000]user+sys [1.003275] real time
TIMING: (Per call period 20) [0.059000]user+sys [1.004369] real time
TIMING: Account add [6.360000]user+sys [100.262032] real time (100 in, 100 out)
TIMING: (Per call aggregate) [0.063600]user+sys [1.002620] real time
TIMING: (Per call period 20) [0.058000]user+sys [1.000000] real time
No errors during update test
```

Note that **update** also provides information about the time needed to perform blocks of 20 updates. This information varies among systems.

You can use the **rgy_edit** tool to view the registry to verify that the correct number of principals, groups, organizations, and accounts are added. See the ‘‘DCE Security Service’’ part of the *OSF DCE Administration Guide—Core Components* for instructions on using **rgy_edit**.

8.2.3.2 The query Test

You must **dce_login** before running this test. You do not need to have registry-modifying privileges, but you must be authenticated to query the registry.

The

dce-root-dir/**install/***machine*/**dcetest/dce1.2.2/test/security/commands/rgy/query**

test checks basic query functionality. It searches through the registry database, performing every query operation. The data returned for a particular object is checked for consistency when it can be returned using different query paths. Only PGO and account objects are checked; policy and property queries are not made.

To run the **query** test, **cd** to its directory and enter

query [-d | -d*rpc_debug_flags***] [***cellname***]**

where:

-d Specifies the minimal level of debug output. This parameter is optional.

-d*rpc_debug_flags* Allows you to specify the amount of debug output desired. Some useful *rpc_debug_flags* settings are the following:

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

cellname Specifies the cell whose registry is to be queried. The default (if *cellname* is not specified) is that the registry of the cell from which **query** is being run will be queried.

For example, entering

query *cellname*

performs the **query** test for *cellname* with no RPC debug output. If *cellname*'s registry has been updated successfully by 100 accounts, **query** displays the following:

```
Contacting registry at site /.../cellname
Processing People...
  10   20   30   40   50   60   70   80   90  100
 110
```

```
Processing Groups...
  10   20   30
Processing Orgs...
No errors during query test
```

8.2.4 ERA, Delegation, and Extended Login Tests

The ERA, Delegation, and Extended Login functional tests were new in DCE 1.1. They are run under TET.

The test sources are located at:

dce-root-dir/**dce/src/test/security/tet-tests**

The following subsections explain how to build, install, and run the tests. For more information on TET, see ‘‘Overview of TET Use’’ in Chapter 11.

8.2.4.1 Building and Installing

To build and install the tests, do the following:

1. Build TET (if you have not already done so):

   cd *dce-root-dir*/**dce/src/test/tet**
   build

2. As root, execute the following command, which will create an install area in / (the root directory), and install TET there:

   build TOSTAGE=/ install_all

   Note that in order to get the **TOSTAGE** value specified in the command line to take effect, you must comment out the following line in the

   dce-root-dor/**dce/src/test/test.mk**

   file:

   ```
   TOSTAGE =  ${SOURCEBASE}/../install/${target_machine}/dcetest/dce1.2.2
   ```

   Note also that TET (and the tests) can be installed elsewhere by supplying a different value for **TOSTAGE** in the command line in the example above (and, for the tests, in the following examples).

3. Build the ERA, Delegation, and Extended Login tests:

   cd ../security/tet-tests
   build

4. As root, execute the following command to install the tests:

   build TOSTAGE=/ install_all

5. As root, do the following:

   ln -s ../../tet/test /test/tet/test
   mkdir /test/tet/tet_tmp_dir
   chmod 777 /test/tet/tet_tmp_dir
   mkdir /test/tet/test/security/results

8.2.4.2 Running the Tests

To run the tests, do the following:

1. Set the following environment variables:

   TET_ROOT=/*your_path_to_installed_tests*/**test/tet**

   If security replication is being tested, set the following environment variables:

   SEC_TEST_REPLICATION=True
   SEC_REPLICA_SITE_NAME=*replica_name_of_the_slave_security_server*

   For example:

   SEC_REPLICA_SITE_NAME=/.../r_d.com/subsys/dce/sec/rs_server_250_2

   or:

   SEC_REPLICA_SITE_NAME=ncacn_ip_tcp:15.22.144.248

   If security replication is not being tested, set the following environment variable:

   SEC_TEST_REPLICATION=False

2. Add the following to your execution path:

   ${TET_ROOT}/bin

3. **dce_login** as **cell_admin**.
4. To execute all of the security TET test cases, execute the following command:

   tcc -e test/security

   Specific test cases can be executed individually. For example:

```
tcc -e test/security sec_rgy_attr-tc
```

The following test cases are available:

- **sec_rgy_attr**

  Tests to verify that the functions within **sec_rgy_attr.c** are working correctly.

- **sec_rgy_attr_sch**

  Tests to verify that the functions within **sec_rgy_attr_sch.c** are working correctly.

- **pwd_expiration**

  Tests to verify that the local **sec_pwd_mgmt_strength_chk_prvcy** function is functioning correctly.

  This test case makes the following assumptions:

  - The host machine is a DCE client.
  - The tester is **dce_login**'d as **cell_admin** and that the password is **-dce-**.

- **pwd_strength**

  Tests to verify that the local **sec_pwd_mgmt_strength_chk_prvcy** function is functioning correctly.

  This test case makes the following assumptions :

  - Machine is a DCE client
  - The Password Strength Server is running and exporting the **sec_pwd_mgmt_strength_chk_prvcy** operation.
  - The Password Strength Server running is the sample server (**pwd_strengthd**) provided with DCE 1.2.2.
  - The tester is **dce_login**'d in as **cell_admin**.
  - The **PWD_STRENGTHD_STRING_BINDING** TET configuration variable has been set correctly.

- **login**

  Tests to verify that the local **sec_login** functions associated with the new EPAC/Delegation work are functioning correctly.

8.2.4.3 Verifying the Results

Following is an example of output from a successful run of all the tests on an HP-UX platform. Note that one **FAILED** message for the **sec_pwd_mgmt_strength_chk_prvcy** test should be expected.

```
# tcc -e test/security
journal file name is: /path_to_installed_tests/test/tet/test/security/results/0007e/journal
```

```
PASSED sec_rgy_attr_update() integer test
PASSED sec_rgy_attr_lookup_by_id() integer test.
PASSED verification of integer test.
PASSED sec_rgy_attr_update() test_void
PASSED sec_rgy_attr_lookup_by_id() test_void.
PASSED verification of test_void.
PASSED sec_rgy_attr_update() test_any
PASSED sec_rgy_attr_lookup_by_id() test_any.
PASSED verification of test_any.
PASSED sec_rgy_attr_update() test_printstring
PASSED sec_rgy_attr_lookup_by_id() test_printstring.
PASSED verification of test_printstring.
PASSED sec_rgy_attr_update() test_printstring_array
PASSED sec_rgy_attr_lookup_by_id() test_printstring_array.
PASSED verification of test_printstring_array.
PASSED sec_rgy_attr_update() test_bytes
PASSED sec_rgy_attr_lookup_by_id() test_bytes.
PASSED verification of test_bytes.
PASSED sec_rgy_attr_update() test_confidential_bytes
PASSED sec_rgy_attr_lookup_by_id() test_confidential_bytes.
PASSED verification of test_confidential_bytes.
PASSED sec_rgy_attr_update() test_i18n_data
PASSED sec_rgy_attr_lookup_by_id() test_i18n_data.
PASSED verification of test_i18n_data.
PASSED sec_rgy_attr_update() test_uuid
PASSED sec_rgy_attr_lookup_by_id() test_uuid.
PASSED verification of test_uuid.
PASSED sec_rgy_attr_update() test_attr_set
PASSED sec_rgy_attr_lookup_by_id() test_attr_set.
PASSED verification of test_attr_set.
PASSED sec_rgy_attr_update() test_binding
PASSED sec_rgy_attr_lookup_by_id() test_binding.
PASSED verification of test_binding.
PASSED sec_rgy_attr_update() test them all
PASSED sec_rgy_attr_lookup_by_id() test them all.
PASSED verification of test them all.
PASSED sec_rgy_attr_lookup_by_id() for 1 attr id.
PASSED sec_rgy_attr_lookup_by_id() for 0 attr ids.
PASSED sec_rgy_site_open_update()
PASSED sec_rgy_attr_sch_create_entry()
PASSED sec_rgy_attr_sch_lookup_by_id()
PASSED sec_rgy_attr_sch_lookup_by_name()
PASSED sec_rgy_attr_sch_cursor_init()
PASSED sec_rgy_attr_sch_scan()
PASSED sec_rgy_attr_sch_cursor_release()
PASSED sec_rgy_attr_sch_update_entry()
PASSED sec_rgy_attr_sch_delete_entry()
PASSED SEC_LOGIN_DELEG: sec_login_become_initiator()
PASSED SEC_LOGIN_DELEG: sec_login_cred_get_initiator()
PASSED SEC_LOGIN_DELEG: sec_cred_get_pa_data()
PASSED SEC_LOGIN_DELEG: sec_cred_get_delegation_type()
```

```
PASSED SEC_LOGIN_DELEG: sec_login_cred_init_cursor()
PASSED SEC_LOGIN_DELEG: sec_login_cred_get_delegate()
PASSED SEC_LOGIN_DELEG: (attrs) sec_login_set_extended_attrs()
PASSED SEC_LOGIN_DELEG: (attrs) sec_login_cred_get_initiator()
PASSED SEC_LOGIN_DELEG: (attrs) sec_cred_initialize_attr_cursor()
PASSED SEC_LOGIN_DELEG: (attrs) sec_cred_get_extended_attrs()
Startup for sec_pwd_mgmt_strength_chk_prvcy() tests
FAILED: PWD_STRENGTHD_STRING_BINDING not defined in tetexec.cfg
Startup for password expiration tests
pwd_expiration, test purpose 1, login attempt using expired password
PASSED password expiration: login with expired password
Cleanup for password expiration tests
```

1.2.2,PKSS Tests (start)

8.2.5 PKSS Functional Tests

The Public Key Storage Server (PKSS) is supplied with five functional tests that specifically exercise different portions of the API. All tests begin with a call to **sec_pvtkey_pkss_open** to obtain a handle to use the API, and all tests end with a complementary call to **sec_pvtkey_pkss_close** to detach from the API. Most of the tests demonstrate that once inserted, a PKSS client can retrieve a record (that is, an asymmetric key pair) from the PKSS database. Most of the tests also demonstrate that the same record may be deleted. Note that all of the tests exercise the PKSS database as well as PKSS client/server communication.

Module **test_pkss_1.cxx** is primarily intended to demonstrate that, given an asymmetric key pair, the PKSS can store it, retrieve it, and delete it. It makes the following PKSS API calls:

sec_pvtkey_pkss_open
sec_pvtkey_pkss_store
sec_pvtkey_pkss_get
sec_pvtkey_pkss_delete
sec_pvtkey_pkss_close

Module **test_pkss_2.cxx** is primarily intended to demonstrate that when requested by a PKSS client, the PKSS server can generate a new asymmetric key pair on the client's behalf, retrieve it, and delete it. It makes the following PKSS API calls:

sec_pvtkey_pkss_open
sec_pvtkey_pkss_generate
sec_pvtkey_pkss_get
sec_pvtkey_pkss_delete
sec_pvtkey_pkss_close

Note that **test_pkss_2.cxx** differs from **test_pkss_1.cxx** only in who generates the new asymmetric key pair.

Module **test_pkss_3.cxx** is primarily intended to demonstrate that, after asking the PKSS server to generate a new asymmetric key pair on the client's behalf, using the

management API one can:

1. Change the asymmetric key pair by supplying a new one; and
2. Change the asymmetric key pair by requesting that the PKSS server generate one.

It also demonstrates that it can retrieve the latest version of the asymmetric key pair and delete it. It makes the following PKSS API calls:

sec_pvtkey_pkss_open
sec_pvtkey_pkss_generate
sec_pvtkey_pkss_update (mgmt client version)
sec_pvtkey_pkss_update_generate (mgmt client version)
sec_pvtkey_pkss_get
sec_pvtkey_pkss_delete
sec_pvtkey_pkss_close

Module **test_pkss_4.cxx** is primarily intended to demonstrate that, after asking the PKSS server to generate a new asymmetric key pair on the client's behalf, using the login client API that client can:

1. Change that client's asymmetric key pair by supplying a new one; and
2. Change that client's asymmetric key pair by requesting that the PKSS server generate one.

It also demonstrates that it can retrieve the latest version of the asymmetric key pair and delete it. It makes the following PKSS API calls:

sec_pvtkey_pkss_open
sec_pvtkey_pkss_generate
sec_pvtkey_pkss_update (login client version)
sec_pvtkey_pkss_update_generate (login client version)
sec_pvtkey_pkss_get
sec_pvtkey_pkss_delete
sec_pvtkey_pkss_close

Note that **test_pkss_4.cxx** differs from **test_pkss_3.cxx** only in who initiates the asymmetric key pair change requests, either a PKSS login client or a PKSS management client.

Module **test_pkss_5a.cxx** and **test_pkss_5.cxx** work in tandem to demonstrate that a PKSS management client can insert a record and a PKSS login client can retrieve the record and modify it by supplying a new asymmetric key pair.

Module **test_pkss_5a.cxx** calls:

sec_pvtkey_pkss_open
sec_pvtkey_pkss_store
sec_pvtkey_pkss_close

Module **test_pkss_5.cxx** calls

sec_pvtkey_pkss_open
sec_pvtkey_pkss_get
sec_pvtkey_pkss_update (login client version)

The test sources are located in

dce-root-dir/**src/test/security/api/pkss**

In the build tree, the built objects can be found at:

dce-root-dir/**obj/***platform*/**test/security/api/pkss**

8.2.5.1 Running the Tests

To run the tests, do the following:

1. In your sandbox, you should build TET, if you have not done so already:

   ```
   % cd sandbox/src/test/tet
   % build
   ```

   Next, go into

   sandbox/**src/test/functional/security**

   to build the tests:

   ```
   % build
   ```

   This will build images in the object tree *but does not* install the scripts or create a usable test directory.

2. Run the ‘‘build install_all’’ pass:

   ```
   % cd sandbox/src/test/tet
   % build install_all TOSTAGE=full path to sandbox/install
   % cd sandbox/src/test/functional/security
   % build install_all TOSTAGE=full path to sandbox/install
   ```

   All this will install the obj’s and scripts under

   sandbox/**install/test/tet**

   The tests will be run out of this directory. (Note that these directories are now owned by root.)

3. The **tet_*** files were installed under:

   sandbox/**install/test/tet/functional/security**

4. Set up an environment for running tests as follows:

   a. Become root.

   b. Make sure that **pkssd** (as well as the other DCE daemons) is running.

   c. Do the following:

      ```
      % cd sandbox/install/test/tet
      % setenv PATH sandbox/install/alpha/test/tet/bin:$PATH
      % setenv PATH sandbox/install/test/tet/lib/ksh:$PATH
      ```

d. Set up some TET environment variables:

% setenv TET_ROOT *sandbox*/**install/test/tet**
% setenv TET_EXECUTE *sandbox*/**install/test/tet/functional/security**

5. To run the tests, choose from different scenarios in

sandbox/**install/test/tet/functional/security/tet_scen**

To run the first functional test for pkss, do:
tcc -e functional/security test-pkss1.sh

To run the second functional test for pkss, do:

tcc -e functional/security test-pkss2.sh

To run the third functional test for pkss, do:

tcc -e functional/security test-pkss3.sh

To run the fourth functional test for pkss, do:

tcc -e functional/security test-pkss4.sh

To run the fifth functional test for pkss, do:

tcc -e functional/security test-pkss5a.sh

and:

tcc -e functional/security test-pkss5.sh

1.2.2,PKSS Tests (end)

1.2.2,Certification API Tests (start)

8.2.6 Certification API Tests

This section describes how to run the Certification API tests on the reference platform (IBM AIX). Where necessary, problems are mentioned that you may encounter in running tests on platforms other than the reference platform.

Note: Before running the Certification API tests, it is important to apply changes described in OT CRs 13665 and 13667 to your DCE1.2.2 code.

The tests can be run standalone (without **tcc**, the TET controller program), or they can be run using **tcc**. The stand-alone tests are built under

dce-root-dir/**dce/obj**/*platform*/**test/security/api/capi**

where *dce-root-dir* is the top directory of your source distribution, and *platform* is ‘‘rios’’, if you are running on the reference platform.

The test versions that can be run using **tcc** are installed in

dcetest/dcelocal/**test/tet**

where *dcetest/dcelocal* is the following path:

dce-root-dir/**dce/install/***platform*/**opt/dcetest/dce1.2.2**

The Certification API tests are divided into three basic phases:

A. Testing Certification API with the registry retrieval policy (RRP).

B. Testing Certification API with the hierarchical retreival policy (HRP) with CDS.

C. Testing Certification API with the HRP with a GDS server.

For each part, it is necessary to have a specific DCE or GDS server configuration before running the tests. Below are described the exact configuration required and the steps necessary to run these tests (either standalone or through **tcc**).

8.2.6.1 Testing the Certification API with the Registry Retrieval Policy

You need to configure your machine as a DCE client or DCE server. The name of the DCE cell is not important to these tests. However, it is required that your **cell_admin** principal name be ‘‘**cell_admin**’’ and that the password for **cell_admin** be ‘‘**-dce-**’’.

8.2.6.1.1 Running the Tests under TET

There are two tests under this part:

- **test-registry**
- **test-registry-second**

To run these tests, do the following:

su (Become superuser)
cd *dcetest/dceloca*/**test/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH ${PATH}:${TET_ROOT}/../test/test/bin
tcc -e functional/security test-registry

The last command creates a journal file. At the end of the journal file a summary of all parts of the test that succeeded or failed will be listed.

You must also do the following:

tcc -e functional/security test-registry-second

This will create another journal file. At the end of the journal file, a summary of all parts of this test that succeeded or failed will be listed.

8.2.6.1.2 Running the Tests Standalone

Alternatively, you can run without the overhead of **tcc** by doing the following

1. **su** (Become super user)
2. cd /*root-dir*/**obj**/*platform*/**test/security/api/capi**
3. **dce_login -c cell_admin -dce-**

   Note: It is important to get certified credentials by specifying the **-c** switch.
4. **rgy_edit < create_foo**
5. **dcecp < create_era**
6. **./test_registry direct > first.log**
7. **kdestroy**
8. **dce_login cell_admin -dce-**

   Note: Credentials are this time not certified (no **-c** switch).
9. **./test_registry untrusted > second.log**
10. **rgy_edit < delete_foo**

Check **first.log** and **second.log** to verify test results.

8.2.6.2 Testing Hierarchical Policy Retrieval (HRP) with CDS

- The DCE cell name must be **/.../dceaix2_cell**.

  The machine where you run the tests can be a DCE client or the DCE server machine.
- The **cell_admin** principal name must be ‘‘**cell_admin**’’
- The **cell_admin** password must be ‘‘**-dce-**’’

8.2.6.2.1 Running the Tests under TET

To run the tests under TET, do the following:

1. **su** (Become superuser on UNIX machines)
2. cd *dcetest/dcelocal*/**test/tet**

3. **setenv TET_ROOT ‘pwd‘**
4. **setenv PATH ${PATH}:${TET_ROOT}/../test/test/bin**
5. **tcc -e functional/security test-hierarchy**

   This will create a journal file. At the end of the journal file, a summary of all parts of the test that succeeded or failed will be listed.

8.2.6.2.2 Running the Tests Standalone

Alternatively, you can run without the overhead of **tcc** by doing the following:

1. **su** (Become superuser)
2. **cd** *root-dir*/**obj**/*platform*/**test/security/api/capi**
3. **dce_login cell_admin -dce-**
4. **./trycase_a pc1 pc2 pc3 pc4 pc5 pc6 pc7 > hierarchy_cds.log**

Afterwards, check **hierarchy_cds.log** for a summary of all testcase components that succeeded or failed.

8.2.6.2.3 Possible Problems

If the journal or log file indicates failure, check whether you have applied changes to modules in

src/directory/gds/dua/switch

as specified in OT CRs 13665 and 13667. If necessary, rebuild and install the tests.

If the tests fail even after you have applied these fixes, the reason may be that the XDS-CDS API on your platform does not allow ‘‘Attribute/Value’’ pairs to be added to CDS directory entries. If this is the case, it is necessary to change the files **pc1**, **pc2**, **pc3**, **pc4**, **pc5**, **pc6**, and **pc7**. Each of these files is present in two directories:

dcetest/dcelocal/**test/tet/functional/security/ts/capi/testcases**
root-dir/**obj**/*platform*/**test/security/api/capi/testcase**

If you are using **tcc**, you need to modify these files under the first directory (as described below); otherwise, modify them under the second directory.

Change the string ‘‘**X500**’’ to ‘‘**CDS**’’ in all lines in files **pc1**, **pc2**, **pc3**, **pc4**, **pc5**, **pc6**, and **pc7** that begin with one of the following strings:

- ‘‘ca’’
- ‘‘cross’’
- ‘‘user’’
- ‘‘urevoke’’

- ‘‘carevoke’’

For example, in the file **pc1**, the following lines:

```
ca:1_0_0_0.cert:X500:/.../dceaix2_cell/capi/a/b
user:0_4_0_0.cert:X500:/.../dceaix2_cell/capi/a/c
```

must be changed to:

```
ca:1_0_0_0.cert:CDS:/.../dceaix2_cell/capi/a/b
user:0_4_0_0.cert:CDS:/.../dceaix2_cell/capi/a/c
```

Make the same changes in any similar lines in **pc2**, **pc3**, **pc4**, **pc5**, **pc6** and **pc7**.

After modifying these files, rebuild the tests and run them as described above.

8.2.6.3 Testing Hierarchical Policy Retrieval (HRP) with GDS Server

It is necessary to configure your machine either as a GDS server or as a GDS client in order to run these tests. The configuration parameters are as follows:

- The GDS namespace must be capable of storing entries below the DN:

  /C=us/O=dec/OU=dceaix2

  Thus, you must create this entry when you configure GDS. Refer to the *OSF DCE GDS Administration Guide and Reference* for information on configuring GDS.

- The schema for the GDS server must be updated.

  To perform this update, you must become root, invoke the **gdsditadm** program, log on to the default DSA, and select ‘‘2’’ (Schema Administration). This should bring you to Mask 9, which is the common starting point for each of the following changes:

  1. Allow for creation of at least three-level **OU** entries under:

     /C=us/O=dec/OU=dceaix2

     The default schema shipped with GDS allows only a single **/OU=RDN** within an Organizational Unit object. The tests use **DN**s containing up to 3 levels of **OU**.

     To allow this, you must add rules to the schema SRT as follows:

     Select 4 (Add SRT entry) to add the following rules:

```
Rule Number                     20
Superior Rule Number            4
Acronymns of Naming Attributes  OU
Structural Object Class         OU

Rule Number                     21
Superior Rule Number            20
Acronymns of Naming Attributes  OU
```

```
Structural Object Class          OU

Rule Number                      22
Superior Rule Number             21
Acronymns of Naming Attributes   OU
Structural Object Class          OU

Rule Number                      23
Superior Rule Number             22
Acronymns of Naming Attributes   OU
Structural Object Class          OU

Rule Number                      24
Superior Rule Number             23
Acronymns of Naming Attributes   OU
Structural Object Class          OU
```

2. Allow for adding Auxiliary Object Classes **SAU** and **CA** to the Organizational Unit object class. This requires you to modify the Object Class Table (OCT).

   Go back to Mask 9, select 10 (Modify OCT entry), and select **OU** as the Object Class Acronym.

   In the Menu presented after you have done this, change the Auxiliary Object Classes Field as follows:

   ```
   Auxiliary Object Classes:        SAU CA_________________
   ```

3. Modify the **SAU** and **CA** object classes so that adding Certificates, CRLs, and Cross-certificates is optional.

   Go back to Mask 9, select 10 (Modify OCT entry), and select **SAU** as the Object Class Acronym.

   In the Menu presented after you have done this, change the following two fields:

   ```
   Mandatory Attributes:   _____________________________
   Optional Attributes:    UC___________________________
   ```

   Now remove **UC** from Mandatory Attributes and add it to Optional Attributes. Perform this step for Object Class **CA** as follows:

   Go back to Mask 9, select 10 (Modify OCT entry), and select **CA** as the Object Class Acronym.

   In the Menu presented after you have done this, change the following two fields:

   ```
   Mandatory Attributes:        _____________________________
   Optional Attributes:         CCP CAC CRL ARL_____________
   ```

   Remove **CAC**, **CRL**, and **ARL** from Mandatory Attributes and add them to Optional Attributes.

4. Modify the Attribute Table (AT) to allow use of ASN.1 encoding for the five certificate attribute (**CAC**, **UC**, **CRL**, **ARL**, and **CCP**) types as follows:

   Go back to Mask 9, select 14 (Modify AT entry), and select **CAC** as the Object Class Acronym.

   In the Menu presented after you have done this, change the Attribute Syntax field to the following:

   ```
   Attribute Syntax:          ASN1 Syntax
   ```

   Repeat these steps for Object Classes **UC**, **CRL**, **ARL**, and **CCP**.

After you have made these changes, commit them by choosing selection 1 (Store Schema) in the Mask 9 menu screen.

8.2.6.3.1 Running the Tests under TET

To run the tests under TET, perform the following steps:

1. **su** (Become super user on UNIX machines)
2. **cd** *dcetest/dcelocal*/**test/tet**
3. **setenv TET_ROOT 'pwd'**
4. **setenv PATH ${PATH}:${TET_ROOT}/../test/test/bin**
5. **tcc -e functional/security test-hierarchy-second**

This will create a journal file. At the end of the journal file, a summary of all parts of this test that succeeded or failed will be listed.

8.2.6.3.2 Running the Tests Standalone

Alternatively, you can run without **tcc** by doing the following:

1. **su** (Become super user)
2. **cd** *root-dir*/**obj**/*platform*/**test/security/api/capi**
3. **./trycase_a xc1 xc2 xc3 xc7 > hierarchy_xds.log**

Check **hierarchy_xds.log** for a summary of all testcase components that succeeded or failed.

1.2.2,Certification API Tests (end)

1.2.2,Kerberos Tests (start)

8.2.7 Kerberos 5 Functional Tests

The following security functional tests were developed as part of the Kerberos 5 integration work.

All the tests are coded to run under TET, and were developed to run in a single-machine cell environment. The first two tests use TET's C-binding API and the second two use TET's TCL-binding APIs.

8.2.7.1 Sample Client Test

This test was developed to run in a single-machine cell environment. It uses TET's C-binding API.

Before building the test, you must update the

dce1.2.2-root-dir/**dce/src/test/functional/security/tetexec.cfg**

file with the following lines:

KRB5_SAMPLE_PORT=*sample_server port#*
KRB5_SAMPLE_SERVER_HOST=*sample_server host*

After you have made the above changes and built the test, it can be invoked as follows:

cd dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=.1 functional/security sclient**

where:

/dcetest/dcelocal	Represents the path *dce-root-dir*/**dce/install**/*platform*/**opt/dcetest/dce1.2.2** (where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional_security** directory) for the test results journal file.
-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
sclient	Specifies the name of the test (TET scenario) to be run.

8.2.7.2 User-to-user Test

This test was developed to run in a single-machine cell environment. It uses TET's C-binding API

Before building the test, you must update the

dce1.2.2-root-dir/**dce/src/test/functional/security/tetexec.cfg**

file with the following information:

KRB5_UU_PORT=*user-to-user port#*
KRB5_UU_SERVER_HOST=*user-to-user_server host*

After you have made the above changes and built the test, it can be invoked as follows:

cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=.1 functional/security uu-client**

where:

/dcetest/dcelocal	Represents the path *dce-root-dir*/**dce/install**/*platform*/**opt/dcetest/dce1.2.2** (where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.
-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
uu-client	Specifies the name of the test (TET scenario) to be run.

8.2.7.3 Rsh Test

This test was developed to run in a single-machine cell environment. It uses TET's TCL-binding APIs.

Before building the test, you must update the

dce1.2.2-root-dir/**dce/src/test/functional/security/lib/test_setup.tcl**

file with the following information:

```
# cell admin's
  set env(KRB5_CELL_ADMIN) cell_admin name
  set env(KRB5_CELL_ADMIN_PW) cell_admin pw

  # test user name has to be less than 8 char for satisfying AIX limitations
  set env(KRB5_TESTER) test user name
  set env(KRB5_TESTER_PW) test user pw
```

After you have made these changes and built the test, it can be invoked as follows:

cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=.1 functional/security rsh**

where:

/dcetest/dcelocal	Represents the path *dce-root-dir*/**dce/install/***platform*/**opt/dcetest/dce1.2.2** (where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.
-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
rsh	Specifies the name of the test (TET scenario) to be run.

8.2.7.4 Rlogin Test

This test was developed to run in a single-machine cell environment. It uses TET’s TCL-binding APIs.

Before building the test, you must update the

dce1.2.2-root-dir/**dce/src/test/functional/security/lib/test_setup.tcl**

file with the following information:

```
# cell admin's
  set env(KRB5_CELL_ADMIN) cell_admin name
  set env(KRB5_CELL_ADMIN_PW) cell_admin pw

  # test user name has to be less than 8 char for satisfying AIX limitations
  set env(KRB5_TESTER) test user name
  set env(KRB5_TESTER_PW) test user pw
```

After you have made these changes and built the test, it can be invoked as follows:

cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=.1 functional/security rlogin**

where:

/dcetest/dcelocal	Represents the path *dce-root-dir*/**dce/install**/*platform*/**opt/dcetest/dce1.2.2** (where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.
-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
rlogin	Specifies the name of the test (TET scenario) to be run.

8.2.7.5 ASN.1 Test

This test was developed to run in a single-machine cell environment.

The test is invoked as follows:

cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=.1 functional/security asn.1**

where:

/dcetest/dcelocal	Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.

-vRUN_TIME=0.1 Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.

functional/security Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

asn.1 Specifies the name of the test (TET scenario) to be run.

8.2.7.6 kinit Test

This test was developed to run in a single-machine cell environment.

The test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security kinit
```

where:

/dcetest/dcelocal Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.

-vRUN_TIME=0.1 Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.

functional/security Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

kinit Specifies the name of the test (TET scenario) to be run.

8.2.7.7 ccache Test

This test was developed to run in a single-machine cell environment.

The test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security ccache
```

where:

/dcetest/dcelocal Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.

-vRUN_TIME=0.1 Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.

functional/security Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

ccache Specifies the name of the test (TET scenario) to be run.

8.2.7.8 keytab Test

This test was developed to run in a single-machine cell environment.

The test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security keytab
```

where:

/dcetest/dcelocal Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC

System/6000 running AIX).

-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.
-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
keytab	Specifies the name of the test (TET scenario) to be run.

1.2.2,Kerberos Tests (end)

1.2.2,Public Key Tests (start)

8.2.8 Public Key Login API Tests

The source for all of these tests is located under:

dce1.2.2-root-dir/**dce/src/test/functional/security/ts/client/login/pk_login**

8.2.8.1 Exportability Check Test

This test checks that there are no **sec_pvtkey** or **sec_bsafe** symbols in a **libdce** built for export.

The test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security export_check
```

where:

/dcetest/dcelocal	Represents the path *dce-root-dir*/**dce/install**/*platform*/**opt/dcetest/dce1.2.2** (where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.

-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
export_check	Specifies the name of the test (TET scenario) to be run.

8.2.8.2 Kerberos Public Key Cache Test

This test tests the Kerberos public key cache.

The test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security rsec_pk
```

where:

/dcetest/dcelocal	Represents the path *dce-root-dir*/**dce/install**/*platform*/**opt/dcetest/dce1.2.2** (where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.
-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
rsec_pk	Specifies the name of the test (TET scenario) to be run.

8.2.8.3 sec_psm_ API Test

The test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security sec_psm
```

where:

/dcetest/dcelocal Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.

-vRUN_TIME=0.1 Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.

functional/security Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.

sec_psm Specifies the name of the test (TET scenario) to be run.

8.2.8.4 Public Key API Test

The public key API test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security sec_pubkey
```

where:

/dcetest/dcelocal Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.

-vRUN_TIME=0.1 Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.

functional/security Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.

sec_pubkey Specifies the name of the test (TET scenario) to be run.

8.2.8.5 Private Key API Test

The private key API test is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security sec_pvtkey
```

where:

/dcetest/dcelocal Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.

-vRUN_TIME=0.1 Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.

functional/security Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

sec_pvtkey Specifies the name of the test (TET scenario) to be run.

8.2.8.6 Registry Public Key API Test

The test of Registry public key functionality is invoked as follows:

```
cd /dcetest/dcelocal/test/tet
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_path -vRUN_TIME=.1 functional/security sec_rgy
```

where:

/dcetest/dcelocal Represents the path

dce-root-dir/**dce/install**/*platform*/**opt/dcetest/dce1.2.2**

(where *platform* is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **functional/security** directory) for the test results journal file.
-vRUN_TIME=0.1	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
functional/security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
sec_rgy	Specifies the name of the test (TET scenario) to be run.

1.2.2,Public Key Tests (end)

8.2.9 GSSAPI Tests

The GSSAPI test program, the source code for which is located at:

dce-root-dir/**src/test/security/api/gssapi/test-gssapi.c**

is not compiled as part of an ODE DCE build. It must be compiled manually, against an installed DCE environment.

To build and run the GSSAPI tests, do the following:

1. Compile **test-gssapi.c** as a normal DCE application.
2. Create two DCE principal accounts (for example, **test1** and **test2**).
3. Use **rgy_edit**’s **ktadd** command to create a keytable (called **keytab** in the example below) containing **test2**’s key.
4. Use **dce_login** to login as the **test1** principal.
5. Run the test program as follows:

% test-gssapi {**-i** |*test1_principal test2_principal keytable*} [**-l**]

where:

-i (‘‘interactive’’) requests a menu of individual separately-runnable tests.

-l (‘‘long-form’’) specifies that additional logging information be sent to standard output.

Once invoked, the above command will:

- run the specified test(s)
- determine whether the GSSAPI is exportable or not (i.e., whether it has been compiled to support privacy protection)
- print out various progress messages during execution
- print out either a final success or failure message

8.2.10 Commands Tests

The **acl_edit.sh** and **rgy_edit.sh** shell scripts test DCE Security Service commands.

8.2.10.1 The acl_edit Tests

Because the tests are not put into an install tree like the source executables, these tests can be cumbersome to execute. This section includes explicit instructions for executing the **acl_edit** tests directly from the

dce-root-dir/**dce/install**

tree. You may find it easier to copy or link all of the control files, located in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/control**

directory, as well as all shell scripts and test case executables, located in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

directory, and the test case driver **testsh**, located in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/testsh**

directory. You must execute each test case from the directory in which its control file resides. The general syntax for tests run by the **testsh** driver is:

path-to-testsh -d*output-level* -I*path-to-test-execs control-file*

If you have copied or linked all of the relevant files into a single directory, the command for running a test case reduces to:

testsh -d*output-level* -I. *control-file*

The

dce-root-dir/**dce/install/***machine***dcetest/dce1.2.2/test/security/api/moretests/acl_edit.sh**

shell script runs tests for the **acl_edit** command. The **acl_edit** tests are structured in the same way as the RPC and IDL unit tests except for the fact that there is no shell script driver to invoke **testsh**.

To run the **acl_edit.sh** tests, do the following:

1. Change to the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory and enter:

   chmod +x *.sh

to make sure that necessary shell scripts are executable.

2. **dce_login** as a user with privileges to modify the registry.

   If you configured your machine using the **dce_config** script, then whatever user the script's **celladmin** variable was set to has registry-modifying privileges.

3. Change to the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory and enter:

   acl_edit_setup.sh

   This script creates an account for ''flintstone.none.none'' in the registry. This account has the password ''yabadabado.'' The script then modifies the **user_obj** entry on the ACL on this account so that user ''flintstone'' may modify the ACL.

4. **dce_login** as ''flintstone'':

   dce_login flintstone yabadabado

   Change to the

   dce-root-dir/**dce/install/***machine***dcetest/dce1.2.2/test/security/api/control**

   directory and enter:

   ../testsh/testsh -d [*output_level***] -I../moretests acl_edit.tsh** > *output_file*

   where:

   -d Specifies an output level for all test programs. Using the **-d** option with no *output_level* integer returns a message only when a test fails.

   output_level Specifies an output level for all test programs. The following list shows the valid integer values for *output_level* and the output levels they specify:

1	Prints message on failure.
2	Prints message on success.
3	Prints message on warning.
32	Prints message on trace.
33	Prints message on trace with failure.
34	Prints message on trace with success.
63	Prints debug messages during test case execution.

The log information generated in *output_file* varies with the *output_level* specified, but test run and execution results are obvious in the log.

8.2.10.2 The Local Registry Test

You must log in as as a user with privileges to modify the registry before running the test. If you configured your machine using the dce_config script, then whatever user the script's *celladmin* variable was set to has registry-modifying privileges. The following examples assume that *celladmin* is set to **cell_admin** and the password for **cell_admin** is **-dce-**.

Note: This test uses the program **binlogin**, which in turn uses the call *sec_login_valid_and_cert_ident( )*, which is a privileged operation. Hence the need for Step 4 outlined below.

To run the local registy test do the following:

1. Login as the privileged user (**root**) on the system.
2. **dce_login cell_admin -dce-**
3. This test uses the **sec_admin** command to stop **secd** so the location of the **sec_admin** command must exist in your **PATH** environment variable.
4. Change to the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory, and execute the following commands:

   ```
   chmod +x *.sh
   chmod u+s binlogin
   ```

   (Note that you may not have to do the **chmod u+s binlogin** if you are already logged in as root.)
5. Change directory to

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/control**

   and type the following:

   ```
   ../testsh/testsh -d[output_level] -I../moretests local_rgy.tsh
   ```

8.2.10.3 The Locksmith Test

There is no automated script for testing locksmith functionality. Instead, the tests have to be done manually as described below. Furthermore, the tests use the **acl_edit** test for which the setup must be done as described in Section 8.2.11.1, ‘‘The acl_edit Tests,’’ Steps 1, 2, and 3.

Note: This test uses the **sec_admin** command to stop **secd**, so the location of the **sec_admin** command must exist in your **PATH** environment variable. When **secd** is started in the locksmith mode, it runs in the foreground.

Test 1: Testing the Basic Locksmith Mode

1. Kill **secd** using the script **kill_secd.sh**.
2. Restart **secd** in locksmith mode as follows:

   secd -locksm locksmith-principal

3. **dce_login** as **flintstone**

   dce_login flintstone yabadabado

4. Run the **acl_edit** test.

Test 2: Testing the -rem option

This test requires a cell to be configured with at least one client machine and one server machine. To test, do the following.

On the server:

1. Kill **secd** using the script **kill_secd.sh**.
2. Restart **secd using the -rem** option as follows:

   secd -locksm locksmith-principal -rem

On the client:

1. Verify that principals other than locksmith-principal can still **dce_login**.
2. **dce_login** as **flintstone**

   dce_login flintstone yabadabado

3. Run the **acl_edit** test.

Test 3: Testing without -rem option

This test requires a cell to be configured with at least one client machine and one server machine. To test, do the following.

On the server:

1. Kill **secd** using the script **kill_secd.sh**.
2. Restart **secd** without the **-rem** option as follows:

   secd -locksm locksmith-principal

On the client:

1. Verify that the locksmith-principal cannot **dce_login**.
2. Verify that other principals (e.g., **cell_admin**) can still **dce_login**.

Test 4: Testing the -lockpw option

1. Kill **secd** using the script **kill_secd.sh**
2. Restart **secd** with the **-lockpw** option as follows:

   secd -locksm principal -lockpw

3. Verify that the principal can only **dce_login** with the password set by the **-lockpw** option.

8.2.10.4 The rgy_edit Tests

You must **dce_login** as a user with privileges to modify the registry before running this test. If you configured your machine using the **dce_config** script, then whatever user the script's **celladmin** variable was set to has registry-modifying privileges. There is no **.tsh** control file for the **rgy_edit** tests.

The

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests/all_rgy_edit.sh**

shell script runs tests for the **rgy_edit** command.

To run the **rgy_edit** tests, do the following:

1. Change directory to the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory and enter:

   chmod +x *.sh

   to make sure that all of the **rgy_edit** test scripts are executable.

2. In the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory, enter:

   all_rgy_edit.sh -d[*output_level] > output_file*

   where:

   -d — Specifies an output level for all test programs. Using the **-d** option with no *output_level* integer returns a message only when a test fails.

   output_level — Specifies a specific output level for all test programs. The following list shows the valid integer values for *output_level* and the output levels they specify:

1	Prints message on failure.
2	Prints message on success.
3	Prints message on warning.
32	Prints message on trace.
33	Prints message on trace with failure.
34	Prints message on trace with success.
63	Prints debug messages during test case execution.

The log information generated in *output_file* varies with the *output_level* specified, but test run and execution results are obvious in the log.

8.2.11 API Tests

The API tests in the

dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

directory are structured similarly to the **acl_edit** tests; also similarly, there is no shell script driver to invoke **testsh** for these tests. See the section describing the **acl_edit** tests for information on how the test cases using the **testsh** driver are structured and hints on how to make executing them easier.

Note: This section gives explicit instructions for executing the API tests directly from the

dce-root-dir/**dce/install**

tree.

The **sec_acl** API test assumes that the principal with registry modifying privileges is **cell_admin** and that the password is **-dce-**. If either of these is different, then the script **sec_acl.tsh** must be modified. Currently only subtest case 10 in **sec_acl.tsh** needs to be modified.

Note: Some tests use the program **binlogin**, which in turn uses the call *sec_login_valid_and_cert_ident( )*, which is a privileged operation. Hence the need for Step 3 outlined below.

To run the API tests, do the following:

1. Make sure you are starting with a clean registry. It is not necessary to re-create the registry after each individual API suite is run.
2. Run the **dce_login** tool to login as the registry principal ‘‘cell_admin’’ or the registry privileged user so that the test process (which inherits your credentials) has the necessary privileges. To run the **passwd_import** test, you need to define a variable **CELLADMIN** to either ‘‘cell_admin’’ or the registry privileged user.

3. Before running the passwd_import test ensure that:
   - The location of the passwd_import command exists in your **PATH** environment variable.
   - The registry is clean.
   - The variable **CELLADMIN** is defined to be either **cell_admin** or the registry privileged user.
4. Change to the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory, and execute the following commands:

   ```
   chmod +x *.sh
   chmod u+s binlogin
   ```

   (Note that you may not have to do the **chmod u+s binlogin** if you are already logged in as root.)
5. Change to the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory, and enter:

   sh rgy_setup.sh

   This script sets up necessary accounts in the registry.
6. Also in the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/moretests**

   directory, enter:

   sh key_mgmt_setup.sh

   This script creates keyfiles necessary for the key management API tests.
7. To actually run the tests, change to the

   dce-root-dir/**dce/install/***machine*/**dcetest/dce1.2.2/test/security/api/control**

   directory and enter:

   ../testsh/testsh -d [*output_level***] -I../moretests **
   control_file > *output_file*

   where:

   -d — Specifies an output level for all test programs. Using the **-d** option with no *output_level* integer returns a message only when a test fails.

   output_level — Specifies an output level for all test programs. The following list shows the valid integer values for

output_level and the output levels they specify:

1	Prints message on failure.
2	Prints message on success.
3	Prints message on warning.
32	Prints message on trace.
33	Prints message on trace with failure.
34	Prints message on trace with success.
63	Prints debug messages during test case execution.

control_file Specifies what control file to use. All files in the

dce-root-dir/**install/***machine*/**dcetest/dce1.2.2/test/security/api/control**

directory which have a **.tsh** extension are valid control files. Refer to the table at the end of this chapter to find which control file will test a given API.

The log information generated in *output_file* varies with the *output_level* specified, but test run and execution results are obvious in the log.

8.2.11.1 Registry Group Override Tests

The Registry Group Override tests are found in:

dce-root-dir/**dce/src/test/functional/security/grp_override**

There are two tests:

- **grp_override.c**

  This test exercises a non-documented functional API that supports group overrides. The new, documented, routine **sec_rgy_pgo_get_by_eff_unix_num( )** is also tested here.

- **passwd_export_grp_override.c**

  This test ensures that **passswd_export** correctly conveys overriden registry information to **/etc/group**.

Note that the DCE 1.2.2 versions of these tests do *not* run under TET, although some of the files and some aspects of the tests' directory structure may make it appear as if they do.

To build the tests under ODE (see Chapter 11 of the DCE 1.2.2 version of the *OSF DCE Porting and Testing Guide* for more information on ODE), change directory to

dce-root-dir/**dce/src/test/functional/grp_override/ts**

and run **build**. To run the tests, change directory to the

dce-root-dir/**dce/obj/***platform*/**test/functional/security/grp_override/ts**

directory, **dce_login** as **cell_admin**, and execute:

```
# ./grp_override
# ./passwd_export_grp_override
```

No failure messages should appear in output.

8.2.11.2 Additional API Test Information

The following table shows the available suites of API tests along with the control file that will execute all of the tests for each suite:

Control File	Function Tested
all_login.tsh	**sec_login**
all_pgo.tsh	**sec_rgy** (PGO management)
all_acct.tsh	**sec_rgy** (account management)
all_auth_pol.tsh	**sec_rgy** (auth policy management)
all_policy.tsh	**sec_rgy** (policy management)
all_props.tsh	**sec_rgy** (properties management)
all_key_mgmt.tsh	**sec_key** (key management)
all_misc_test.tsh	**sec_rgy** (miscellaneous interfaces)
site_bind.tsh	**sec_rgy** (site bind)
site_mgmt.tsh	**sec_rgy** (site management)
sec_acl.tsh	**sec_acl**
id_map.tsh	**sec_id**
local_rgy.tsh	**sec_login** (local registry)
passwd_import.tsh	**passwd_import**
passwd_override.tsh	**password_override**
most_sec.tsh	Most of the **.tsh** files besides **acl_edit.tsh, rgy_edit.sh, local_rgy.tsh, passwd_import.sh, passwd-override.tsh** and **unix.tsh**.
unix.tsh	**unix** (UNIX interfaces)

Additional API test information is available in **secp.gpsml** in the

dce-root-dir/**doc/testplans/security**

directory.

8.2.12 Use of the ‘‘compile_et’’ Program

The following command is used in testing.

- **compile_et**

This command is used to create message catalogs from error table files. It is part of Kerberos and is used by Security and DFS. Its source directory is:

dce-root-dir/**dce/src/security/krb5/comerr**

Error table files (usually ending with a **.et**) are input to **compile_et**, and a **.h** and a **.msf** are output. The **.h** file is included in source code to have macros defined for each error code and the **.msf** is used as input to **gencat** to create message catalog files.

The following is excerpted from the file **src/security/h/sad_err.et**:

```
This symbolic message source file (SMSF) defines the errors produced by
the security admin tools.  The first token on each line is the symbolic
name of an error.  The rest of the line is the text that describes that
error. An SMSF is like an XPG message source file (MSF) except
that symbolic message identifiers are used instead of numbers.
```

Example lines from the **.et** file are as follows

```
ec ek_null_handle,    "Unable to allocate handle (Registry Edit Kernel)"
ec ek_bad_format,     "Data string format not valid for the specified
                       field (Registry Edit Kernel)"
ec ek_required_field, "Kernel operation invoked on incomplete data
                       set (Registry Edit Kernel)"
```

The **.h** file produced contains lines as follows:

```
#define ek_null_handle    (386412545L)
#define ek_bad_format     (386412546L)
#define ek_required_field (386412547L)
```

The **.msf** file is used as input to **gencat** to generate message catalogs. Its contents have the following appearance:

```
1  Unable to allocate handle (Registry Edit Kernel)
2  Data string format not valid for the specified field
   (Registry Edit Kernel)
3  Kernel operation invoked on incomplete data set (Registry
   Edit Kernel)
```

8.2.13 Test Plans

Refer to Chapter 1 of the *OSF DCE Release Notes* for the location of the DCE test plans on the DCE distribution tape.

Chapter 9. DCE Audit Service

This chapter contains porting and testing information for the DCE Audit Service.

9.1 Audit Service Overview

Audit plays a critical role in distributed systems, where there is widespread sharing of data and resources, as well as the use of remote systems management facilities. Adequate audit facilities are necessary for detecting and recording critical events in a distributed application.

Audit is a key component of DCE and is provided by the DCE Audit Service. It has the following features:

- An audit daemon is provided which performs the logging of audit records based on specified criteria.
- Application Programming Interfaces (APIs) are provided which can be used as part of application server programs to actuate the recording of audit events. These APIs can also be used to create tools that can analyze the audit records.
- An administrative command interface to the audit daemon is provided which directs the daemon in selecting the events that are going to be recorded based on certain criteria.
- An event classification mechanism is used to logically group a set of audit events, allowing for ease of administration.
- The display of audit records can be directed to logs or to the console.

9.2 Testing and Verification

The test cases provided for the DCE Audit Service test the audit API and the command line interface.

There are three types of audit test cases:

- API Tests

  These test the audit logging and analysis APIs.

- Command Tests

  These test the use of **dcecp** to control the audit daemon.

- Event Class File Tests

  These test the configurability of event classes.

Each of these types and their corresponding test cases are described in the following sections. Eight audit test cases are shipped with DCE.

9.2.1 Description of the Audit API Test Cases

In the API test cases, the audit and logging APIs are first tested together independently of the filters. The filter mechanism is then tested by invoking the audit logging API functions. Finally, the audit logging API functions are invoked, specifying the audit daemon as the target of audit records.

The audit analysis API functions are used to prove the correctness of test results.

Following are the API test cases and their descriptions:

api_log	Invokes the logging APIs without using filters.
api_filter	Invokes the logging APIs and use filters.
api_log_to_daemon	Invokes the logging APIs without using filters, and logs to the audit daemon (that is, the central audit trail file).

In the Command test cases, the audit daemon is started and stopped using different combinations of command line parameters. All other features are tested by having the audit daemon audit its own control interface operations by linking the audit library with the audit daemon, and starting the daemon using the **-a** option.

The DCE Control Program (**dcecp**) is used to check how the audit daemon handles filters and the audit trail file.

Following are the Command test cases and their descriptions:

cp_filter	Issues **dcecp** commands that display and manipulate filters.
cp_auditd	Issues **dcecp** commands that display and modify the attributes of the audit daemon, as well as well as to

	enable or disable audit logging, or stop the daemon.
auditd_startup	Starts the audit daemon using the different options of the **auditd** command.
auditd_acl	Checks that the default ACL of the audit daemon object contains the specified ACL entries.

9.2.2 Description of the Event Class Test Case

In this test case, an event is added to an event class file. The test case then verifies that the event generates an audit record when the event class is selected by a filter.

The event is then excluded from the event class. The test case verifies that the event does not generate an audit record when the same filter is used.

The name of the test case is **ec_filter**.

9.2.3 Installing the Audit functional tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config** will install the tests you select at the path you specify, and will create a softlink (called **/dcetest/dcelocal**) to that location. The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the sections below, which refer to the tests' source or build locations.

The DCE Audit functional tests are available via option 8 (''Audit'') of the ''DCE Test Installation (Functional Tests'' menu. The TET binaries are available via option 3 (''TET'') of the DCE Test Installation menu.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the final location of the built test tree). For the Audit functional tests, this path should be the location, on your machine, of:

dce-root-dir/**dce/install**

—which is the DCE **install** tree (for more information on the structure of the DCE tree, see Chapter 3 of the *OSF DCE Release Notes*).

Thus, **dcetest_config** will install the Audit functional tests at:

/dcetest/dcelocal/test/tet/functional/security/audit

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* (if you choose) functional tests with **dcetest_config**; you must use TET to run the tests. Information on running the individual tests can be found in the following sections.

Refer to Chapter 11 of this guide for further information on using **dcetest_config**. See ''Overview of TET Use'' in Chapter 11 for general information on TET.

9.2.4 Audit Test Configuration Requirements

All Audit test suites are run from the TET environment. Before running the Audit test suites, ensure that:

- You are logged in as root.
- The DCE cell is up and running; that is, that the DCE daemons (**secd**, **cdsd**, and the DCE client daemons) have been started.
- The Audit daemon (**auditd**) is *not* running.
- You are *not* authenticated in the cell. The tests are designed to be run using the machine principal.
- In the CDS namespace, the Audit ACL object

  .:/hosts/*hostname***/audit-server**

  does not contain server binding entries (i.e., the **RPC_ObjectUUIDs** attribute for both entries should be null). If it does contain such entries, you should remove the object from the namespace before running the tests.

Note that since the test suites are run under TET, many of the configuration requirements are taken care of in the test code.

9.2.5 Running the Audit Test Cases

To run the audit test cases, enter the following command:

```
tcc -e functional/security/audit
```

The test results can be viewed from the journals that TET creates in the

/dcetest/dcelocal/test/tet/functional/security/audit/results

directory. The journal is located in a numbered directory, where the number represents a test run. A numbered directory and journal is created for each invocation of the **tcc** command (for example, **0001e**, **0002e**, and so on).

Following is an example of an Audit TET journal which shows the test cases that succeeded and those that failed:

```
0|1.10 12:59:18 19940525|User: weisz (0) TCC Start, Command Line:
dce-install-path/install/rios/dcetest/dce1.1/test/tet/bin/tcc -e functional/security/audit
20|dce-install-path/install/rios/dcetest/dce1.1/test/tet/functional/security/audit/tetexec.cfg 1|Config Start
30||TET_VERSION=1.10
30||TET_OUTPUT_CAPTURE=False
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_NSIG=31
30||TET_SIG_IGN=34
40||Config End
70||"Starting AUDIT Test Suite"
10|0 /ts/api_filter/api_filter 12:59:18|TC Start, scenario ref 11-1
15|0 1.9 1|TCM Start
520|0 0 18265 1 1|START DCE audit functional test:
dce-install-path/install/rios/dcetest/dce1.1/test/tet/tet_tmp_dir/24146a/api_filter/api_filter; \
    DATE: Wed May 25 12:59:22 EDT 1994
400|0 1 1 12:59:48|IC Start
200|0 1 12:59:48|TP Start
520|0 1 18265 1 2|START: api_filter1 started
520|0 1 18265 1 3|PASS: api_filter01 passed
520|0 1 18265 1 4|PASS: api_filter02 passed
520|0 1 18265 1 5|ERROR: api_filter03 failed
520|0 1 18265 1 6|PASS: api_filter04 passed
520|0 1 18265 1 7|PASS: api_filter05 passed
520|0 1 18265 1 8|PASS: api_filter06 passed
520|0 1 18265 1 9|PASS: api_filter07 passed
520|0 1 18265 1 10|PASS: api_filter08 passed
520|0 1 18265 1 11|PASS: api_filter09 passed
520|0 1 18265 1 12|PASS: api_filter10 passed
520|0 1 18265 1 13|PASS: api_filter11 passed
220|0 1 0 13:04:58|PASS
410|0 1 1 13:04:58|IC End
520|0 0 18265 1 1|END DCE audit functional test:
dce-install-path/install/rios/dcetest/dce1.1/test/tet/tet_tmp_dir/24146a/api_filter/api_filter; \
    DATE: Wed May 25 13:05:16 EDT 1994
80|0 0 13:05:19|TC End
70||"Completed AUDIT Test Suite"
900|13:05:19|TCC End
```

9.2.6 Test Plans

Refer to the *OSF DCE Release Notes* for the location of the DCE test plans on the DCE distribution tape.

9.3 Audit Runtime Output and Debugging Output

The Audit 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 Audit via serviceability.

9.3.1 Normal Audit Server Message Routing

There are basically two ways to control normal Audit 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.

9.3.1.1 Routing File

If a file called

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

exists when Audit is brought up, the contents of the file (if in the proper format) will be used as to determine the routing of Audit 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/var/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.

9.3.1.2 Routing by the dcecp log Object

Routing of Audit 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.

9.3.2 Debugging Output

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

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

routing file (described above), or by specifying the same information in the **SVC_AUD_DBG** environment variable, before bringing up Audit. 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* 6
[*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

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 Audit, the sub-components are as follows:

general	General server administration
esl	Event selection list (filters) management
evt	Audit record management
trl	Audit trail management
msgs	Debugging messages

For example, the string

"aud:*.1,trl.3:TEXTFILE.50.200:/tmp/AUD_LOG

sets the debugging level for all Audit sub-components (*except* **trl**) at 1; **trl**'s level is set at 3. All messages are routed to **/tmp/AUD_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 Audit serviceability messages, and the sub-component list, can be found in the Audit sams file, at:

dce-root-dir/**dce/src/security/audit/libaudit/aud.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''.

Chapter 10. DCE Distributed File Service

The DCE Distributed File Service (DFS) provides data sharing services for use within the DCE environment by extending the local file system model to remote systems. It provides the ability to store and access data at remote locations and utilizes the client/server model common to other distributed file systems.

10.1 Overview

DFS consists of the following components:

- DCE Local File System (LFS), which can store the file system data on the disk.

  Note: This component, alone among the DFS components, is optional. You can retain your existing file system instead of DCE LFS and use DFS to export that file system. However, there are advantages to bringing up LFS in conjunction with DFS.

- The File Exporter, which exports data using Remote Procedure Call (RPC).
- The Token Manager, installed on DFS servers, which synchronizes access to exported file systems on DFS servers.
- The Cache Manager, installed on DFS clients, which retrieves and stores data from the File Exporter.
- The Token Cache Manager, installed on DFS clients, maintains liaisons with the Token Manager, and controls server access to exported local filesystems.
- Fileset services, which handle administrative file system functions. These include the following servers:

  1. the Fileset Location Server, which supplies network locations for filesets.
  2. the Fileset Server, which provides access to entire filesets for administrative functions, such as moving and backing them up.
  3. the Replication Server, which provides fileset replication on different machines (for greater availability).

- The Basic Overseer (*bos*) service, which monitors other server processes and facilitates system administration tasks.
- *Scout*, which gathers file server statistics.
- *Backup*, which provides a mechanism for backing up data stored on the file server.

Command interfaces are provided for these server processes and tools.

DFS lets users access a remote file by its location-independent DCE pathname. It then finds the file, just as if it existed locally. Users do not have to know the physical location of files. The *Cache Manager*, which runs on client machines, translates file system calls into references to the client machine's file system cache. If necessary, it then executes RPCs to the file server machine containing the data.

The local file system (LFS) on the DFS server stores the master copy of filesystem data. The *File Exporter* can export any Virtual File System (VFS) resident on the server machine. DFS uses a token-based cache synchronization mechanism to maintain cache consistency and provide single-site semantics.

DCE LFS is a log-based file system that supports filesets, access control lists, and extended fileset features. These include copy-on-write clones, quotas, and multiple filesets per partition.

The DCE LFS code is designed to run in the server's kernel. It is based on a standard UNIX disk partition, using the facilities of the kernel device driver. DCE LFS operations are accessed through the system call layer, which calls the VFS switch.

10.2 Setup, Testing, and Verification

Since DFS interacts with various other DCE components, functional testing for it is not necessarily simple, particularly with a port of DCE to a new platform. The detailed operation of other DCE components may not be known, and there will not be a baseline of component behavior under different conditions of usage and loading. Therefore, testing interactions between DFS and the other components may indicate a need for modifications in those other components as well as DFS, and necessitate a cyclical or incremental approach to functional testing, as well as system test.

When you start testing DFS, a reference platform is particularly useful, since the code on it has been tested to known standards of functionality and robustness. In addition, the reference platform lets you address interoperability issues with a partner that works correctly.

10.2.1 Installing DFS Functional Tests with dcetest_config

You can install the functional tests described in the following sections by running the menu-driven **dcetest_config** script described in Chapter 11 of this guide. **dcetest_config** will install the tests you select at the path you specify, and will create a softlink (called

/dcetest/dcelocal) to that location. The functional tests for a given component will thus be installed under a:

/dcetest/dcelocal/test/*component_name*/

directory, where the **test/***component_name* elements of this path are equivalent to the **test/***component_name* elements in the pathnames given in the sections below, which refer to the tests' source or build locations.

Note that **dcetest_config** will prompt you for the location *from which* the tests should be installed (in other words, the final location of the built test tree). For the DFS functional tests, this path should be the location, on your machine, of:

dce-root-dir/**dce/install**

—which is the DCE **install** tree (for more information on the structure of the DCE tree, see Chapter 3 of the *OSF DCE Release Notes*).

Thus, **dcetest_config** will install the DFS functional tests at:

/dcetest/dcelocal/test/file/

where **/dcetest/dcelocal** is the link to whatever path you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

Note that you can only *install* (if you choose) functional tests with **dcetest_config**; for test configuration and execution you must follow the instructions in the sections below.

Refer to Chapter 11 of this guide for further information on using **dcetest_config**.

10.2.2 Debugging Notes

DFS involves the interaction of many different programs, which operate on different machines (servers, clients) in both kernel and user space. It uses the services of various other DCE components, such as RPC, Threads, DTS and Security. It also uses the services of non-DCE components, such as the native file services of at least one and possibly more host platforms.

Therefore, porting DFS to a new platform presents a broad set of challenges. The subcomponents must be built and integrated in a distributed and possibly heterogeneous environment, interactively with other development efforts. Porting and development work in different areas of DCE can proceed asynchronously, and the DFS port effort must bridge changes in the software environment.

10.2.2.1 Running Tests on the HP/UX Platform

Note following before running the DFS functional tests on the HP/UX platform:

- You should not use **/bin/sh**, but rather the **/bin/posix/sh** shell, when running the DFS functional tests. Otherwise errors will occur as a result of the way **/bin/sh** handles arguments when function calls are made.
- The **diff** command supplied with HP/UX 9.0.1 will not perform

  diff -r

  correctly under certain circumstances, returning a non-zero exit code even when there are no differences in the directory trees specified. Functional tests such as **low** and **fs** which use the **diff** command will incorrectly report failures.

10.2.2.2 Distributed Development Environments

Often, multiple versions of a particular source file are in use simultaneously, which complicates the debugging process when responsibilities are divided among developers. Distributed development environments, such as OSF *Open Development Environment* (ODE), packaged with the DCE sources, can support this type of work. The source control software included in such environments provides a handle for managing distributed development with tracing tools to find the filenames, file revisions and line of code affecting a particular variable or data object.

If you define the **AFSL_USE_RCS_ID** preprocessor directive on the command line when compiling a file, **osi_assert** failures return the source code file, its version number, the assertion's line number, and (if possible) the results of the assertion. Otherwise, the compiler's version of the filename is returned.

The DFS code implements the file and version information with Revision Control System (RCS). RCS is available from the Free Software Foundation. It is packaged with the *OSF Development Environment* (ODE), which is provided on the DCE source tape.

However, the package is general enough that you can apply it to your own source code control system, if you use a different development environment than ODE. To modify the code which lets the AFSL_USE_RCS_ID construct return information in a form appropriate to your source code control conventions, check and update the following files under the

dce-root-dir/**dce/src/file**

directory:

config/stds.h **osi/afsl_trace.c** **osi/osi.h**

Note: The code in the

dce-root-dir/**dce/src/file/osi**

directory contains various debugging aids for porting the **osi** layer. Some of this code may be applicable to other portions of DFS.

10.2.2.3 Kernel Debugging Considerations

You need a kernel debugger as well as user space debugging facilities to bring up DFS on your platform. At a minimum, such a debugger must be able to set breakpoints and execute stack traces. Increasing the debugger's capabilities and its integration into your computational environment can improve your debugging efficiency. Specific desiderata for a debugging environment include the following:

- Remote debugging, where the machine running the code differs from the machine doing the testing.
- Source code debugging.
- Structure format conversion facilities (dumpers): DFS kernel code includes multiple layers of nested structures. Written out in raw hexadecimal format, they can be tedious to interpret. Format conversion facilities which cast the information into a readable format, and trace out succeeding nested substructures, can speed the debugging process significantly.

If your kernel debugging tools have any shortcomings, you may find that an investment in improving them, particularly to provide the facilities listed above, will be repaid in shortened debugging time as you bring up DFS.

Note: When you plan the porting process, you should evaluate the costs and benefits of investing in improved development tools before you begin working with DFS.

10.2.2.4 Debugging Facilities in the DFS Source Code

The DFS source code provides several built-in debugging tools, particularly in the

dce-root-dir/**dce/src/file/osi**

directory.

For example, **osi_assert**, which checks for internal consistency, and debugging-related compiler switches can be found in

dce-root-dir/**dce/src/file/osi/osi.h**

If an **osi_assert** fails, the program uttering it restarts, typically dumping core. You may wish to build a soft restart facility into your kernel code, so such **osi_assert** failures do not cause a kernel panic. Doing so can speed up code development and testing. However, in production systems, **osi_assert** failures are normally only associated with critical problems and possible data corruption. You must decide how to handle such failures in

your final product.

Note that some debugging features must be ported separately for the different libraries in which they run, once for kernel and once for user-space code.

10.2.2.5 Debug Levels

You can select the level of debugging feedback with a numerical value for **AFSL_DEBUG_LEVEL**, defined in **osi.h**. Currently, three values of **AFSL_DEBUG_LEVEL** are implemented:

0 Only critical code reports errors.

3 Consistency checks are reported if they are not computationally expensive. For example, Boolean expressions of simple variables are checked, function calls or complex macros are not.

5 All consistency checks are performed, regardless of expense.

You can tune the debugging level, including definition of intermediate levels for **AFSL_DEBUG_LEVEL**, to suit your needs, depending where you are in the porting process.

Because DFS code involves interaction among many modules on different machines, expect to maintain a fairly high level of debugging reporting through most of the development process. Typically, **AFSL_DEBUG_LEVEL** will remain at 5, even for modules already built and separately functionally tested.

Once you have finished the debugging, and do not intend to trace operations again, do not define **AFS_DEBUG** or **AFSL_DEBUG_LEVEL** in:

dce-root-dir/**dce/src/file/osi/osi.h**

Then only critical **osi_assert**s, where failures are associated with possible data corruption, are turned on.

10.2.3 Test Types

There are several functional test suites available for DFS. Some are packaged with DCE, and some which are not, but are probably already present on your system. There are three sets of tests of overall DFS functionality, namely:

- Basic tests, such as the NFS connectathon suite, which are not packaged with DCE.
- The low-level functionality tests, in the

  dce-root-dir/**dce/src/test/file/low**

  directory.
- More extensive tests, in the

dce-root-dir/**dce/src/test/file/fs**

directory.

At least with the latter two sets of tests, you can modify the stress level by changing various parameters, such as the sizes and numbers of objects created, listed, modified or removed.

Besides testing basic DFS functionality, the

dce-root-dir/**dce/src/test/file**

directory has subdirectories for tests of specific functions associated with DFS.

10.2.3.1 Basic Testing with External Test Packages

If your platform also supports NFS, you can use tests packaged with it, particularly the *connectathon* test suite, to check basic DFS functions, such as creating, deleting, listing, reading and writing files and directories. Alternatively, you may be able to modify other low-level external filesystem test suites to test DFS during the porting process.

10.2.3.2 The Standard DFS Test Suites

Once your implementation passes such basic tests, you can begin stress tests, from the

dce-root-dir/**dce/src/test/file/low**

and

dce-root-dir/**dce/src/test/file/fs**

directories.

These tests let you specify sizes and number of objects to be manipulated, and the mix of operations on those objects, so you can increment the stress on your code along various parameters. In addition, the context in which the tests are run, for example heterogeneous machines or split servers, let you generate a matrix of performance stresses.

Beside the basic tests listed above, the following types of function-specific DFS tests are shipped with DCE:

- DFS kernel modification tests
- DCE Local File System tests
- DFS server process tests
- DFS command interface tests
- DFS administrative tool tests

These tests are contained in subdirectories of the

dce-root-dir/**dce/src/test/file**

directory and are described in the DFS Test Plan.

Before executing the test cases, you must configure DFS for testing, using the instructions in the following section of this chapter (‘‘DFS Test Setup’’). You can run tests on the configurations described in that section.

Because some DFS code runs in kernel space, many of the interfaces cannot be called directly in order to test them. Therefore, testcases have been written using user interfaces that in turn access and exercise the kernel space code. In addition, tests are included to exercise those subroutines not tested through traditional UNIX interfaces.

User-level code is tested using shell scripts that exercise the interfaces.

10.2.4 DFS Test Setup

Before running any DFS tests, you must first configure a DCE cell. Refer to the *OSF DCE Administration Guide—Introduction* for information on configuring a DCE cell, specifically Chapter 6 ‘‘Overview of The DCE Installation and Configuration Script,’’ Chapter 7 ‘‘Phase One: Initial Cell Configuration,’’ and Chapter 8 ‘‘Phase Two: Configuring a DCE Client and Other DCE Services.’’

10.2.5 DCE Distributed File Service Tests

The following sections describe functional tests for the DCE Distributed File System. These tests are packaged on the distribution tape, in the

dce-root-dir/**dce/src/test/file/**

directory. In addition, many of the DFS source subdirectories include test programs for individual functions and and subcomoponents.

Note: Before building and running the test programs packaged with the DFS sources, check them for platform and operating system dependencies. They may need to be modified to operate correctly in your target environment, and to exercise ported code.

In addition to the DFS system call tests described in the section immediately following, other following sections describe a number of development level tests which are built in the individual subcomponent directories. These can be used to test various phases of your port. Included are tests for the token manager, aggregate operations, free pool management, system calls, and others.

For information on DFS system testing, refer to the ‘‘DFS System Tests’’ section of Chapter 12 (‘‘DCE System Testing’’) of this guide.

10.2.5.1 System Call Tests

The

dce-root-dir/**dce/src/test/file/low**

and

dce-root-dir/**dce/src/test/file/fs**

subdirectories contain testcases for testing the file system-related system calls affected by DFS. Once your ported DFS code passes all tests in these two subdirectories, it can provisionally be considered ready for integration with other DCE functions.

10.2.5.1.1 The low Tests

The tests in

dce-root-dir/**dce/src/test/file/low**

are C programs with shell script drivers that use DFS to exercise low-level system calls. Brief descriptions of the **low** tests are listed below.

Note: Tests 2 and 4 are not listed. They exist, but are computationally expensive and are not considered necessary for testing DFS functionality.

These tests are specific to UNIX platforms. If you are porting to a different operating system, you will have to rewrite them, using your target environment's system calls.

Test 1 — Performs **stat( )** calls to check for existence of two test files, one of 16 bytes, one of half a megabyte. Does repeated **open( )**s and **close( )**s on each file, then repeated cycles of **open( )**-**write( )**-**read( )**-**close( )** on each. It then performs cycles of **lseek( )** and **open( )**-**read( )**-close( ) on as many as three files. Does not check data.

Test 3 — Performs sequential and random **write( )**s to a file, then a **close( )** followed by **fsync( )**. Then it **open( )**s and **read( )**s the file, and compares the data with what it wrote.

Test 5 — Writes out a file, marches through the file with successive read( ) and lseek( ) calls. Compares the first byte of each buffer for data integrity.

Test 6 — File and directory manipulation: Performs **mkdir( )** and **chdir( )** system calls. Uses **opendir( )** and **readdir( )** to confirm that what it created actually exists.

Test 7 — Creates symbolic links, performs **lstat( )**s on them.

Test 8 — Creates different files with all permission modes, opens, renames, and unlinks them. Checks whether the modes stay correct on **open( )**.

Test 9 — Creates a file, manipulates its mode and time with **chmod( )**, **fchmod( )** with the file open, and **utimes( )**. It then checks the file's status with **stat( )**, and unlinks the file.

Test 10 — File descriptor status manipulation: creates a file, **open( )**s it, performs **fcntl_sets** and **fcntl_gets** on it, does some **read( )**s and **write( )**s. It then calls **fcntl_sets** again. Then if truncates the file with **ftruncate( )**. Finally, it checks the file's status flags with **stat( )**, and unlinks the file.

Test 11 — Deadlock testing: a parent process forks a child, then both processes lock and unlock a file.

Test 12 — Creates a file, reads and writes vectors of data to it with **readv( )** and **writev( )**.

RTest 1 — Tests the **chroot** command.

RTest 2 — Tests the **chown** command.

Information on running these tests can be found in:

dce-root-dir/**dce/src/test/file/low/READ_ME**

10.2.5.1.2 The fs Tests

The tests in

dce-root-dir/**dce/src/test/file/fs**

are shell scripts that execute a number of common UNIX commands relating to files. These tests exercise the Cache Manager and Protocol Exporter functionality, as well as verify that UNIX filesystem semantics are maintained. These tests check that the DFS implementation adjudicates filesystem contention among multiple processes, as happens in a multi-user environment. They are summarized below. (Note that test 5 has been removed from the suite.)

As with the low-level tests described above, these tests are specific to UNIX systems, and will have to be rewritten for other target environments.

err1 — Tests file error conditions by issuing incorrect commands. For instance, this test attempts to **cp** to a directory, **cd** to a file, and perform invalid **chmod** and **chgrp** commands.

Test 1 — Run up to 9 simultaneous copies of a program, which modify different parts of the same file at the same time.

Test 2 — Creates a new subdirectory, then spawns multiple processes which performs various standard file operations in that subdirectory.

Test 3 — Performs hundreds of file creations and removals in the current directory, then checks that all the correct files (and no others) are present at the end of the process.

Test 4 — Concatenates files: multiple processes **cat** sets sixteen 1K files into 16K files, then repeat the process with the larger files, forming 256 kilobyte files.

Test 6 — Tests process contention: one process attempts to delete a file while another has the file open.

Test 7	Tests directory management integrity: creates a directory structure containing a variable number of directories, each of which contains a variable number of 16-kilobyte files. The tree is repeatedly created and then removed.
RTest 1	Checks the **chgrp**, **chmod** and **chown** commands.

Information on running these tests can be found in:

dce-root-dir/**dce/src/test/file/fs/READ_ME**

1.2.2,added DFS Delegation tests (start)

10.2.6 Delegation Tests

Delegation tests are located in the

dce-root-dir/**dce/src/test/file/delegation.system**

directory. Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/delegation.system/README**

These tests do not run under TET.

1.2.2,added DFS Delegation tests (end)

1.2.2,added Multihome Server tests (start)

10.2.7 Multihome Server Tests

The Multihome Server Tests are located in the

dce-root-dir/**dce/src/test/file/cmmhs**

directory. Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/cmmhs/README**

These tests do not run under TET.

1.2.2,added Multihome Server tests (end)

1.2.2,added File Exporter Authorization tests (start)

10.2.8 File Exporter Authorization Tests

The File Exporter Authorization Tests are located in the

dce-root-dir/**dce/src/test/file/cmfxauth**

directory. Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/cmfxauth/README**

These tests do not run under TET.

1.2.2,added File Exporter Authorization tests (end)

10.2.8.1 DFS Cache Consistency Tests

The DFS cache consistency tests are located in:

dce-root-dir/**dce/src/test/file/cache_mgr**

Descriptions of the tests and instructions on how to run them can be found in:

dce-root-dir/**dce/src/test/file/cache_mgr/README**

10.2.8.2 UNIX Filesystem Tests

UNIX filesystem tests are located in:

dce-root-dir/**dce/src/test/file/fs**

Descriptions of the tests and instructions on how to run them can be found in:

dce-root-dir/**dce/src/test/file/fs/README**

10.2.8.3 DFS ACL Tests

The DFS ACL tests are located in:

dce-root-dir/**dce/src/test/file/acl**

Descriptions of the tests and instructions on how to run the tests can be found in:

dce-root-dir/**dce/src/test/file/acl/README**

10.2.8.4 DFS Token Manager Tests

These tests verify DFS token manager functionality and are contained in the

dce-root-dir/**dce/src/file/tkm**

directory. Note that there is no **README**.

10.2.8.5 DFS Zero Link Count Tests

These tests verify the correctness of handling zero link count files in DFS and are contained in the

dce-root-dir/**dce/src/test/file/zlc**

directory. The directory contains a **README**.

10.2.8.6 DFS Token State Recovery Tests

The DFS token state recovery hand tests are located in:

dce-root-dir/**dce/src/test/file/tsr**

Descriptions of the tests and instructions on how to run them can be found in:

dce-root-dir/**dce/src/test/file/tsr/TSR_README**

10.2.8.7 DFS File Exporter Stress Tests

The DFS file exporter stress tests are located in:

dce-root-dir/**dce/src/test/file/fx**

A descriptions of the test script and instructions on how to run the tests can be found in:

dce-root-dir/**dce/src/test/file/fx/README**

10.2.8.8 ubik Failure Recovery Tests

The **ubik** failure recovery hand tests are located in:

dce-root-dir/**dce/src/test/file/ubik**

Descriptions of the tests and instructions on how to run them can be found in:

dce-root-dir/**dce/src/test/file/ubik/READ_ME**

10.2.9 DCE Local File System Tests

The following sections describe tests for the DCE Local File System.

10.2.9.1 System Call Tests for LFS

The **low** and **fs** tests described in the ‘‘System Call Tests’’ section earlier in this chapter can also be run on the DCE Local File System to test file system-related calls affected by DCE LFS.

10.2.9.2 LFS Fileset Operations Tests

The fileset (‘‘ftutil’’) test tools for testing DCE LFS fileset operations are located in:

dce-root-dir/**dce/src/test/file/fset**

Instructions on running the tests can be found in:

dce-root-dir/**dce/src/test/file/fset/README**

10.2.9.3 LFS Authorization Salvage Test

The LFS authorization salvage hand test is located in:

dce-root-dir/**dce/src/test/file**

A description of the test and instructions on how to run it can be found in the comment at the top of the

dce-root-dir/**dce/src/test/file/salvage/AuthCheckTest**

file. Test tools for the LFS salvager are located in

dce-root-dir/**dce/src/test/file/ravage**

and:

dce-root-dir/**dce/src/test/file/scavenge**

10.2.9.4 LFS ACL and LFS Recovery Tests

The LFS ACL and LFS recovery and associated POSIX compliance tests are located in:

dce-root-dir/**dce/src/test/file/recovery**

A description of the **checkaggr** tool, which is used by these tests, and which is located in this directory, can be found in the comments at the top of

dce-root-dir/**dce/src/test/file/recovery/checkaggr**

and in:

dce-root-dir/**dce/src/test/file/recovery/README.checkaggr**

10.2.9.5 Other DCE LFS Tests

The tests in the following directories test additional functions specific to the DCE LFS:

- *dce-root-dir*/**dce/src/file/episode/anode/test_anode.c**

  Described in:

  dce-root-dir/**dce/src/file/episode/vnops/README**

  and:

  dce-root-dir/**dce/src/file/episode/anode/README**
- *dce-root-dir*/**dce/src/file/episode/async/astest.c**
- *dce-root-dir*/**dce/src/file/episode/dir/test_dir.c**
- *dce-root-dir*/**dce/src/file/episode/vnops/test_vnodeops.c**

  Described in:

  dce-root-dir/**dce/src/file/episode/vnops/README**

Many of these tests are porting tests that run in user space. It is recommended that these tests only be used before placing your ported code into kernel space to help verify that the basic function is working correctly. In most cases, the tests accept scripts that tell them which subroutines or operations to perform in sequence. Functions covered include the following:

- Initializing aggregates
- Creating aggregates
- Verifying aggregates
- Creating filesets
- Closing filesets
- Mounting and unmounting tests
- Checking mode bit settings and access times
- Testing **vnode** operations

- Testing locks (**file** and **record**)

10.2.10 DFS Server Process Tests

DFS server processes are exercised both by the cache manager and protocol exporter operations described previously, and through DFS command tests. These tests are described in the ‘‘DFS Command Interface Tests’’ section of this chapter.

10.2.10.1 Ubik Database-Replication Tests

A test server and client process, **utst_server** and **utst_client**, are provided for testing replicated database functionality. These tests are described in the DFS Test Plan and are in the

dce-root-dir/**dce/src/test/file/ubik**

directory. You must create entries in the CDS namespace in order to run these tests.

10.2.11 DFS Command Interface Tests

Tests for the **bos** command are located in the

dce-root-dir/**dce/src/test/file/bos**

directory. Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/bos/READ_ME**

Tests for the **cm** command are located in the

dce-root-dir/**dce/src/test/file/cm**

directory. Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/cm/README**

The DFS Server Preference tests are located in the

dce-root-dir/**dce/src/file/cm/test**

directory. These tests verify correct operation of server preferences in DFS. The directory contains a **README** that explains how to run the tests.

Tests for the **fts** commands are located in the

dce-root-dir/**dce/src/test/file/fts**

directory. Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/fts/README**

The DFS Test Plan describes these tests and explains how to execute them. The **runtests** script for the **cm** and **fts** tests contains a number of variables which should be configured for the environment being tested. For the **fts** tests, two DCE LFS aggregates should be available to test against, and two more DCE LFS aggregates should be exported.

The **fts** tests for fileset replication are more effective if two fileserver machines are available for use. However, basic replication can be tested with a single file server.

The DFS replication tests verify DFS fileset replication functionality. The tests are contained in the

dce-root-dir/**dce/src/test/file/rep**

directory. The directory contains a **README** which describes the tests in detail and explains how to run them.

10.2.12 DFS Administrative Tests

Tests for the DFS administrative tools are available in the

dce-root-dir/**dce/src/test/file**

directory. Details about the separate tests appear in the following sections.

10.2.12.1 Update Tests

The **upserver** and **upclient** distribution tools should be tested with the

dce-root-dir/**dce/src/test/file/update**

tests. Comments at the beginning of

dce-root-dir/**dce/src/test/file/update/uptest**

explain how to run these tests.

10.2.12.2 Scout Tests

The Scout interactive monitoring tool is tested manually. Descriptions of the manual tests are located in:

dce-root-dir/**dce/src/test/file/scout/READ_ME**

10.2.12.3 Backup System Tests

The DFS backup system is tested using the scripts in:

dce-root-dir/**dce/src/test/file/backup**

A comment at the top of the

dce-root-dir/**dce/src/test/file/backup/runtests**

script explains the necessary configuration and how to run the tests.

10.2.13 DFS Gateway Tests

Tests for the DFS Gateway are located in the

dce-root-dir/**dce/test/file/gateway**

directory. Details about the separate tests appear in the following sections.

10.2.13.1 Gateway Daemon Tests

The **dfsgwd** should be tested using the tests in:

dce-root-dir/**dce/src/test/file/gateway/dfsgwd**

Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/gateway/dfsgwd/README**

10.2.13.2 Gateway Administration Tests

The **dfsgw** command line interface should be tested using the tests in:

dce-root-dir/**dce/src/test/file/gateway/dfsgw**

Information on setting up and running these tests can be found in:

dce-root-dir/**dce/src/test/file/gateway/dfsgw/README**

10.2.13.3 Gateway Client Tests

dfs_login and **dfs_logout** should be tested using the tests in:

dce-root-dir/**dce/src/test/file/gateway/dfs_login**

Information on setting up and running these tests can be found in:

dce-root-dir/dce/src/test/file/gateway/dfs_login/README

10.2.14 Test Plans

Refer to Chapter 1 of the *OSF DCE Release Notes* for the location of the DCE DFS test plans, describing the DFS test cases and how to execute them, on the DCE distribution tape.

Chapter 11. TET and DCE Testing

Many of the DCE system tests have been modified to use the Test Environment Toolkit (TET) version 1.10.

Source code for TET is provided in the source tree under

dce-root-dir/**dce/src/test/tet**

TET is built and placed in the release area as part of the default source tree build. The X/Open release notes, specifications and user guides for TET can be found in the

dce-root-dir/**dce/src/test/tet/doc**

directory.

TET provides support for building, running and for cleaning up the test suites. However, to provide better integration with OSF's software process, TET is used only to execute the tests, and ODE is used to build and install the test suites.

11.1 Installing TET

Once DCE has been built and installed, the system test directory should lie by default at:

dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/systest**

(for most of the tests that do not run under TET), and:

dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet/system**

(for the tests that do run under TET).

The **systest** directory contains the following:

- **admin**

  Directory containing DCE Administrative automated tests and checklists.

- **directory/gds**

Directory containing DCE Global Directory Service system tests.

- **profile.dcest**

  File containing definitions for environmental variables used by the system tests.

- **dcetest_config**

  The DCE Test Installation and Configuration script.

- **file** Directory containing DFS system tests.

- **tools** Directory containing DCE system test tools used by system tests that are not run under TET.

The **tet/system** directory contains the following:

- **I18N**

  DCE Internationalization system tests.

- **audit**

  DCE Audit Service system tests.

- **dced**

  DCE Host Daemon (**dced**) system tests.

- **directory/cds**

  DCE Cell Directory Service system tests.

- **profile.dcest.tet**

  File containing definitions for environmental variables used by the system tests.

- **rpc**

  DCE RPC system tests. Note that this directory contains the **rpc.cds.3** system test, which *is not* run under TET.

- **security**

  DCE Security Service system tests.

- **svc**

  DCE Serviceability system tests.

- **threads**

  DCE Threads system tests.

- **time**

  DCE Distributed Time Service system test.

- **tools**

  DCE system test tools.

For the remainder of this chapter, the name:

systest-root

will be used to signify the correct path on your system to the **systest** directory in the DCE install tree.

11.1.1 Using dcetest_config

dcetest_config is a menu-driven utility which can be used to do the following things:

- Install any of the DCE system tests.
- Install DCE functional tests, found in:

  dce-root-dir/**dce/install**/*platform*/**dcetest/dce1.2.2/test/***component_name*

  (for non-TET functional tests), and:

  dce-root-dir/**dce/install**/*platform*/**dcetest/dce1.2.2/test/tet/functional**/*component_name*

  (for functional tests run under TET).
- Install TET

dcetest_config operates much like **dce_config**, the script used to install and configure DCE itself. As long as you are using **dcetest_config** only to install tests, there is no requirement to have run **dce_config**; the only requirement is that DCE must have been built. However, if you wish to execute tests for any component other than GDS, you must of course have a DCE cell up and running—which means that you must have run **dce_config**. For GDS testing, the only requirement is that GDS be installed on the test systems.

You start **dcetest_config** by typing:

systest-root/**dcetest_config** *[***DEBUG***]*

(Specifying **DEBUG** will keep **dcetest_config** from clearing the screen when it changes menus.)

You may find it necessary to make the following environment variable setting:

MORE= -W notite -e

to prevent **dcetest_config** from prompting you to press a key to continue after each menu is displayed.

The following sections describe the various uses of **dcetest_config**.

11.1.2 Installing TET with dcetest_config

To install TET, become root and start the **dcetest_config** script. From the **dcetest_config** Main Menu, select ‘‘1’’ to install.

Figure 11-1. Installing TET: Step 1

```
DCE Test Main Menu

 1. Install
 2. Configure

99. Exit

selection: 1
```

After you have selected the ‘‘Install’’ menu option, the ‘‘Location of DCE Test Install Binaries’’ menu will be displayed. You can install TET either from a filesystem or from media.

Figure 11-2. Installing TET: Step 2

```
Location of DCE Test Install Binaries


  1. Filesystem
  2. Media

 98. Return to previous menu
 99. Exit

 selection: 1

Enter the full path to the DCE binary install tree.
This will be the directory that contains the
.../<BUILD>/install/<machinetype>/dcetest/dce1.2.2
directory:  /myproject/dce/mybuild/nb_ux/install/hp800/dcetest/dce1.2.2


Enter the path to the DCE test tree location.
This will be the directory that will contain all the tests.
Please locate this directory somewhere other than the root
partition, if possible.  A softlink /dcetest/dcelocal will be made
to this location.
directory: /usr/dcetest
```

After you have specified the location information and typed **<RETURN>**, the ‘‘DCE Test Installation Menu’’ will be displayed. Select ‘‘3’’ to install TET.

Figure 11-3. Completion of Installation

```
DCE Test Installation Menu
```

```
 1. Functional Tests
 2. System Tests
 3. TET

98. Return to previous menu
99. Exit

selection: 3
```

As TET is being installed, you should see the following messages:

```
installing test/tet/bin ...
installing test/tet/lib ...
```

After TET has been installed, you will be returned to the **dcetest_config** Main Menu.

Figure 11-4. Return to Main Menu

```
DCE Test Main Menu

 1. Install
 2. Configure

99. Exit

selection: 99
```

You have now installed TET.

11.1.3 Installing the DCE Functional Tests with dcetest_config

To install any or all of the DCE functional tests, you should select ‘‘1’’ in the **dcetest_config** Main Menu:

Figure 11-5. Selecting Test Installation

```
 DCE Test Main Menu

 1. Install
 2. Configure

99. Exit

selection:  1
```

You will then be prompted for the location of the test binaries. You can install the tests either from media (e.g., tape) or from a filesystem. In the following example, ‘‘1’’ (filesystem) has been selected; this causes the user to be prompted for the location of the filesystem and, following that, for the path at which the tests are to be installed:

Figure 11-6. Supplying Test Location

```
Location of DCE Test Install Binaries


 1. Filesystem
 2. Media

98. Return to previous menu
99. Exit

selection:  1

Enter the full path to the DCE binary install tree.
This will be the directory that contains the
.../<BUILD>/install/<machinetype>/dcetest/dce1.2.2
directory:  /myproject/dce/mybuild/nb_ux/install/hp800/dcetest/dce1.2.2

Enter the path to the DCE Test tree location.
This will be the directory that will contain all the tests.
Please locate this directory somewhere other than the root
partition, if possible. A softlink /dcetest/dcelocal will be made
to this location.
directory: /usr/dcetest
```

Following these steps, you will be returned to the Test Installation menu, where you can now select ‘‘1’’ to actually install the tests:

Figure 11-7. Functional Test Installation Menu

```
 DCE Test Installation Menu

 1. Functional Tests
 2. System Tests
 3. TET

98. Return to previous menu
99. Exit

selection:  1
```

Note that if you have previously installed tests at the destination path that you have given, **dcetest_config** will warn you of this and give you the chance to go no further:

Figure 11-8. Previously Installed Tests

```
Location of DCE Test Install Binaries


 1. Filesystem
 2. Media

98. Return to previous menu
99. Exit

selection:  1


Enter the full path to the DCE binary install tree.
This will be the directory that contains the
.../<BUILD>/install/<machinetype>/dcetest/dce1.2.2
directory:  /myproject/dce/mybuild/nb_ux/install/hp800/dcetest/dce1.2.2

Tests have previously been installed in /usr/dcetest
Do you want to continue storing the tests in the same location (y) y
```

The final menu for functional test installation allows you to select one or all of the functional suites for installation:

Figure 11-9. Installing Functional Tests

```
DCE Test Installation (Functional Tests) Menu

 1. Cell Directory Service
 2. Distributed File Service
 3. Global Directory Service
 4. Remote Procedure Call
 5. Security
 6. Threads
 7. Distributed Time Service
 8. Audit
 9. DCE Control Program
10. DCE Host Configuration Server

97. All of the above
98. Return to previous menu
99. Exit

selection:  7
        installing test/time ...
        installing test/systest/profile.dcest ...
        installing test/systest/dcetest_config ...
```

As was shown in the screen example above, **dcetest_config** will install the tests at the path you give it, and will create a softlink called **/dcetest/dcelocal** to that location. For example, it would install the DTS functional tests at:

/dcetest/dcelocal/test/time/

where **/dcetest/dcelocal** is a link to the path:

/usr/dcetest

which you supplied as the install destination.

The advantage in using **dcetest_config** to install the functional tests is that it will install *all* that is needed and *only* what is needed out of the DCE build, thus avoiding the mistakes that can occur with manual installation.

For instructions on how to run the installed functional tests, refer to the section on functional testing in the appropriate component chapter of this guide.

11.1.4 Installing the DCE System Tests with dcetest_config

Installing the DCE system tests is similar to functional test installation. From the Main Menu, select ‘‘1’’:

Figure 11-10. Installing System Tests: Step 1

```
 DCE Test Main Menu

 1. Install
 2. Configure

99. Exit

selection:  1
```

You will then be prompted for the location of the to-be-installed tests, as well as the location you wish them to be installed at:

Figure 11-11. Installing System Tests: Step 2

```
Location of DCE Test Install Binaries


 1. Filesystem
 2. Media

98. Return to previous menu
99. Exit

selection:  1


Enter the full path to the DCE binary install tree.
This will be the directory that contains the
.../<BUILD>/install/<machinetype>/dcetest/dce1.2.2
directory:  /myproject/mybuild/nb_ux/install/hp800/dcetest/dce1.2.2

Enter the path to the DCE Test tree location.
This will be the directory that will contain all the tests.
Please locate this directory somewhere other than the root
partition, if possible. A softlink /dcetest/dcelocal will be made
to this location.
directory: /usr/dcetest
```

In the Test Installation Menu you can now select ‘‘2’’ to install the tests:

Figure 11-12. Installing System Tests: Step 3

```
 DCE Test Installation Menu

  1. Functional Tests
  2. System Tests
  3. TET

 98. Return to previous menu
 99. Exit

selection:  2
```

You will then be shown the System Test Installation Menu, from which you can select one or all of the system tests for installation:

Figure 11-13. Installing System Tests: Step 4

```
 DCE Test Installation (System Tests) Menu

  1. Admin Tests
  2. Cell Directory Service
  3. Distributed File Service
  4. Global Directory Service
  5. Remote Procedure Call
  6. Security
  7. Threads
  8. Distributed Time Service
  9. Audit
 10. I18N
 11. Serviceability
 12. DCED

 97. All of the above
 98. Return to previous menu
 99. Exit

selection:  97
```

As **dcetest_config** installs the tests, it will display a series of messages updating you on its progress:

Figure 11-14. Installing System Tests: Installation Messages

```
installing test/systest/admin ...
installing test/tet/system/directory/cds ...
installing test/systest/file ...
installing test/systest/directory/gds ...
installing test/tet/system/rpc ...
installing test/tet/system/security ...
installing test/tet/system/threads ...
installing test/tet/system/time ...
installing test/tet/system/audit ...
installing test/tet/system/I18N ...
installing test/tet/system/svc ...
installing test/tet/system/dced ...
installing test/tet/system/profile.dcest.tet ...
installing test/systest/dcetest_config ...
installing test/tet/system/tools ...
installing test/systest/profile.dcest ...
```

To install some subset of tests, simply select the appropriate choice in the System Test Installation Menu instead of ‘‘97’’, until you have installed all the tests you want.

11.1.5 Configuring for System Test with dcetest_config

The configuration step for system testing is mainly a matter of specifying where logs and temporary files are to be created by the tests. Select ‘‘2’’ from the Main Menu:

Figure 11-15. Configuring for System Test

```
 DCE Test Main Menu

 1. Install
 2. Configure

99. Exit

selection:  2


You need to specify the directory where the logs would be stored.
Please locate this directory somewhere other than the root
partition, if possible. A softlink would be established to
point to this directory from /dcetest/dcelocal/status
Enter directory to store the logs: /dcetest/dcelocal/results

Directory /dcetest/dcelocal/results does not exist...
Do you want it it to be created (y) y

You need to specify the directory where the temporary files
would be stored.
Enter directory to store the temporary files: /usr/tmp

Please ensure that /usr/tmp is periodically cleaned...
```

You will be prompted for the paths at which you want logfiles and temporary files to be created. Note that **dcetest_config** will create a soft link (called **/dcetest/dcelocal/status**) to the directory you specify.

At the end of this step, you will be returned to the Main Menu:

Figure 11-16. End of Configuration

```
DCE Test Main Menu

1. Install
2. Configure

99. Exit

selection:  99
```

You have now completed the configuration step, and can exit from **dcetest_config**.

11.2 Using TET

The DCE system tests that are run under TET fall into two categories:

- Tests that are run directly by invoking TET itself

  There is only one DCE system test in this category, namely **rpc.sec.2**. The user invokes TET (**tcc**), which runs the **dcerpsec** script.

- Tests that are run by invoking a **run.***component_name* script. The following table shows these tests:

TABLE 11-1. DCE System Test Suites and TET Scenarios

Component	Test Suite Name	Scenario Name
cds	**systest/directory/cds**	**cdsserv** **dcecdsacl6** **hclcfg001** **hclrel001**
I18N	**systest/I18N**	**I8NSAN001** **I8NSAN002**
audit	**systest/audit**	**audstr001** **audrel001**
svc	**systest/svc**	**svccfg001** **svccfg002** **svccfg003** **svccfg004**

Component	Test Suite Name	Scenario Name
		svccfg005 **svccfg006**
dced	**systest/dced**	**dcdrel001** **dcdrel002** **dcdrel003**
rpc	**systest/rpc**	**dcerpbnk** **dcerpcrun** **dcerpbnk_auth**
security	**systest/security**	**dceseacl** **dceseact** **dcesepol** **dcesestr** **dcesergy** **dlgcfg001** **eraobj001** **erarel001**
threads	**systest/threads**	**dceth002** **dcethmut** **dcethrpc** **dcethrpc_auth**
dts	**systest/time**	**dcetmsyn**

The following section gives a basic overview of TET operation. For more detailed information consult the following documents:

- *Test Environment Toolkit: Architectural, Functional, and Interface Specification*

  located at:

  dce-root-dir/**dce/src/test/tet/doc/tet_spec.ps**

- *Test Environment Toolkit: Programmer's Guide*

  located at:

  dce-root-dir/**dce/src/test/tet/doc/tet_prog_guide.ps**

Unformatted **nroff** source (using the **mm** macro package) for each of the above documents is also available in the directories in the

dce-root-dir/**dce/src/test/tet/doc**

directory.

The following sections describe the use of TET to invoke the DCE system tests.

In the examples given, it is assumed that the tests are being run in a DCE cell that contains at least three machines configured as follows:

machine1: CDS Server, Security Server, Local Time Server — NTP provider

machine2: CDS Client, Security Client, Local Time Server

machine3: CDS Client, Security Client, Local Time Server

11.2.1 Overview of TET Use

Following is the structure of tests within the TET environment:

test suite contains a related group of test cases. Test cases are grouped together in scenarios.

testcase A testcase is an independent executable (a shell script or compiled C program) which contains one or more test *purposes*. Test purposes are combined together into invocable components within a testcase.

test purpose A test purpose is the component of the tests that report PASS/FAIL results. Each test purpose is a shell of a C function.

scenario A scenario is a collection of test cases that are executed together. Scenarios are defined in the **tet_scen** file at the top of each test suite. Every test suite has an ''all'' scenario that runs all test cases within a test suite.

invocable component An invocable component (IC) consists of one or more test purposes. There can be one or more IC per testcase. An IC is the smallest group of test purposes that can be executed independently. ICs are defined in data structures that are located inside of each testcase.

Before any test cases can be run you must define the **TET_ROOT** environment variable as follows:

```
TET_ROOT=/dcetest/dcelocal/test/tet
```

TET_ROOT defines the location of all the test suites and support utilities. When combined, **TET_ROOT** and the test suite name will define the location of the top of the test suite.

To run a test suite that uses TET you use the **tcc** command in the following form:

tcc -e [*optional_switches***]** *test_suite* **[***scenario***]**

For the DCE system tests, the **-e** flag is required. It tells **tcc** to **e**xecute the specified test suite. **tcc** has other modes that are not used by the DCE system tests.

There are many other switches that you may find useful, including:

-p tells **tcc** to print the name of each testcase as it executes it. This is a good way to track the progress of the running tests.

-j *filename* tells **tcc** to write the journaled test results to the designated *filename*.

-v *variable=value* Sets a TET variable to be used by the testcase. Default values for TET variables are specified in the **tetexec.cfg** file located in the top of the test suite. Values specified on the command line override the values in the **tetexec.cfg** file. The variables that are used by the specific test suites are documented in the sections specific to each test suite.

For information on other **tcc** command line options, consult the TET specification.

test_suite is the name of the test suite you wish to run. It also specifies the relative path from **TET_ROOT** to the location of the test suite to be run.

scenario tells **tcc** which pieces of the test suite to run. If you do not specify a scenario, the ''all'' scenario will be run. The scenarios for each test suite are defined in the **tet_scen** file at the top of the test suite tree.

When you run **tcc** the first thing that it will report is the location of the *journal file*. The journal file contains the results of the test scenario run. Each line in a journal file starts with a number code indicating the type of information appearing on that line. For example, lines that contain result codes start with ''220''. To get a quick view of the results of a test run you can do the following:

grep "^220" *journal_file*

—which will cause all the PASS/FAIL results from the journal file to be displayed. For more details on possible errors and causes of failures you will have to read the details of the results file.

Other important journal line codes are:

50 Identifies lines that contain test case execution error messages from **tcc**.

200 Identifies lines marking the beginning of each test purpose.

220 Identifies lines marking the end of each test purpose and containing the result from the test purpose.

520 Identifies lines that contain text printed by the test purpose.

11.2.2 Running DCE System Tests under TET

TET assumes when running DCE system tests that the following environment variables have the following values:

TET_ROOT **/dcetest/dcelocal/test/tet**

This is the base directory for all tests which run under TET.

DCELOGDIR **/dcetest/dcelocal/status**

This is the base directory for DCE Functional and System test output.

STTMPDIR **/tmp**

This is the base directory for temporary files.

11.2.2.1 DCE System Tests that can be Invoked with ‘‘Run’’ Scripts

The installed names of the scripts and tests that can be run from a **run** script are as follows:

- **$TET_ROOT/system/directory/cds/bin/run.cds**

  dcecdsacl6 CDS ACL Manager Test (formerly **cds.acl.6**)

  cdsserv CDS Directory Service Stress Test (formerly **cds.server.4**)

- **$TET_ROOT/system/directory/cds/bin/run.hcell**

  hclcfg001 Establishes intercell authentication with a list of cells using **rgy_edit**.

  hclrel001 Performs intercell testing to specified list of cells.

- **$TET_ROOT/system/dced/ts/lib/run.dced**

  dcdrel001 **dced** endpoint reliability test.

  dcdrel002 **dced** server configuration and server execution service reliability test.

  dcdrel003 **dced** hostdata, keytab, and ACL service reliability test.

- **$TET_ROOT/system/rpc/bin/run.rpc**

  dcerpsec RPC-Security System Test

  dcerpper RPC system test version of RPC **perf** functional tests

  dcerpbnk RPC Object Registry, Threads, CDS, and Security Test

  dcerpbnk_auth Authenticated RPC version of **dcerpbnk**

  dcerpcrun RPC Stress Test (formerly **rpc.runtime.1**)

- **$TET_ROOT/system/security/bin/run.sec**

  dceseacl Registry Access Control List (ACL) and Stress Test

  dceseact Tests Additions and Deletes in the Security Registry

  dcesepol Security policy option test

  dcesergy Security Registry Login and Administration Stress Test (formerly **sec.rgy.7**)

  dcesestr Multiple-client Security Registry Stress Test

  dlgcfg001 Delegation Configuration Test

  dlgcfg002 Delegation Configuration Test

  dlgstr001 Delegation Stress Test

eraobj001	Extended Registry Attributes ACL Test
erarel001	Extended Registry Attributes Stress Test

- **$TET_ROOT/system/audit/bin/run.aud**

audstr001	Audit Service Stress Test
audrel001	Audit Service Reliability Test

- **$TET_ROOT/system/threads/bin/run.thr**

dceth002	Threads Creation Test
dcethmut	Threads Creation and Mutex Exclusion Test
dcethrpc	RPC Server and Client Threads Test
dcethrpc_auth	RPC Server and Client Threads Test — authenticated version

- **$TET_ROOT/system/time/bin/run.time**

dcetmsyn	Test DTS Local Synchronization with DTS Servers

- **$TET_ROOT/system/svc/bin/run.svc**

svccfg001	Serviceability Configuration Test 1
svccfg002	Serviceability Configuration Test 2
svccfg003	Serviceability Configuration Test 3
svccfg004	Serviceability Configuration Test 4
svccfg005	Serviceability Configuration Test 5
svccfg006	Serviceability Configuration Test 6

11.2.3 Using the ‘‘Run’’ Scripts: An Example

Note: You *must* be using the Korn shell (**ksh**) in order to run the DCE system tests under TET, as described in this and the following sections.

To run DCE system tests which use TET and the **run.***component_name* scripts, do the following after installing the DCE systems tests and TET:

```
$ cd /dcetest/dcelocal/test/tet/system
$ . profile.dcest.tet          # Set up System Test Environment
$ run.thr -l 2 dceth002        # Run dceth002 just as an example
```

The example shown above will run two iterations (**-l 2**) of **dceth002**, creating some output in and under the standard directory, e.g.:

$DCELOGDIR/system/dceth002.*hostname***.931022124807**

—where *hostname* is the name of the machine the test was invoked on, and the series of concluding digits is a starting timestamp in the form of *yymmddhhmmss*. The **run** script

you invoke will tell you the name of the directory to which it writes its output. The contents of this directory will look something like the following:

```
$ ls -lsFR $DCELOGDIR/system/dceth002.hostname.931022124807
total 6
   2 drwxrwxrwx   2 root     system         512 Oct 22 12:49 fail/
   2 drwxrwxrwx   2 root     system         512 Oct 22 12:48 pass/
   2 -rw-rw-rw-   1 root     system         326 Oct 22 12:49 pass_fail_log

/dcetest/dcelocal/status/system/dceth002.hostname.931022124807/fail:
total 0

/dcetest/dcelocal/status/system/dceth002.hostname.931022124807/pass:
total 4
   4 -rw-rw-rw-   1 root     system        1276 Oct 22 12:48 journal.00001
```

When the **run.thr** script was invoked, a directory was created for TET journal files for the iterations of the test that passed, and another was created for any failed iterations. The **pass_fail_log** contains a header, one status line for each iteration, and a trailer. The contents of the **pass_fail_log** file from the example above would look something like the following:

```
COMMAND: run.thr -l 2 dceth002
PLATFORM: hostname(osf1i386)
TEST NAME: dceth002
STARTED AT: 10/22/93-12:48:07
NEWEST /OPT/DCELOCAL/BIN: Oct 21 23:55
10/22/93-12:48:42       PASS    pathname of journal file
10/22/93-12:49:45       PASS    <journal.00002 deleted>
COMPLETED: 10/22/93-12:49:45
```

To view the results of the test, you would enter the following:

```
$ run_summary.ksh $DCELOGDIR/system/dceth002.hostname.931022124807
```

—which will produce output something like the following (assuming that no errors occurred during the test; if there were errors, they will be listed in the results as well):

```
hostname(osf1i386)dceth002:       pass = 2, fail = 0
            "run.thr -l 2 dceth002" completed at 10/22/93-12:49:45
            /opt/dcelocal/bin complete: Oct 21 23:55
            Failures under /dcetest/dcelocal/status/system/dceth002.hostname.93102212480
                    None
```

For more information on **run_summary.ksh** see ‘‘Checking Test Results’’ later in this chapter.

Finally, to clean up when you had finished evaluating the results, you would enter:

```
$ rm -r $DCELOGDIR/system/dceth002.hostname.931022124807
```

11.2.4 Prerequisites for Running System Tests Using the ‘‘Run’’ Scripts

Each **run.***component_name* script contains some test-specific option processing code of its own and a call to the

$TET_ROOT/system/tools/run_loops.ksh

script, which is used in common by all the **run** scripts. **run_loops.ksh** controls test iteration, checks test output for pass/fail, reports totals, and writes the test output to a standard location.

Before running any of the DCE system tests, note the following.

The DCE System Tests should be run in a standalone (i.e., a non-production) cell. The tests place a heavy load both on DCE and on the host machines, and they do so for many hours or days. Such behavior is generally considered incompatible with a production environment. Furthermore, the only automatic way to finish cleaning up after running the DCE System Tests is to shut down the cell. All DCE credentials are deleted, and the unusable memory which accumulates in the DCE servers when these tests run is reclaimed.

The cell in which the tests are to be run must be created with the default cell administrator name (**cell_admin**) and password (**-dce-**). This is necessary because these names are hardcoded in the tests themselves. Such a configuration is obviously inappropriate for a cell intended for general use.

For the DCE system tests runnable under **run** scripts, the following things must be true before the tests can be successfully run:

- The **/.rhosts** or equivalent file on each machine in the test cell must include all machines in the cell, since the DCE System Tests use **rsh** or its equivalent to start processes on other machines in the cell.
- All DCE System Test and TET software must have been installed by **dcetest_config** on all machines in the DCE System Test cell. For instructions on how to do this, see ‘‘Installing TET and the DCE Functional and System Tests’’, earlier in this chapter.
- The DCE System Tests *require* the following environment variables to have the following values:

  TET_ROOT /dcetest/dcelocal/test/tet

  STTMPDIR /tmp

  Note that this must be true on *every* system in the test cell. It is acceptable to achieve arrange this via symbolic links. In any case, failure to do this will result in unpredictable test behavior.
- A number of quotas and limits must be set and/or monitored in order to safely and successfully run the DCE System Tests. All such account-specific changes should be done to the ‘‘root’’ account, which is the account from which DCE and all DCE System Tests must be run.
- Substantial disk space is required to run the tests. The tests will fail and possibly bring down both DCE and the system if the disks fill up. Disk usage varies greatly, depending on test choice and run duration. Twenty megabytes of free space is

recommended as an *absolute minimum* for the disk holding the top-level directory for DCE System Test output,

$DCELOGDIR/system

- Too little swap space is another potential problem. Some of the DCE servers grow slowly as the DCE System Tests run. Again, the rate and degree of such behavior depends on the platform used, which tests are run, and test run duration. Yous should consult your platform's system manuals and tune your machines for heavy memory usage, including allocating large swap files.
- Note that CPU time limits are a problem for DCE servers. Set ''root'' time limits to unlimited.
- Make sure you are using the Korn shell (**ksh**) and that you have sourced the

  /dcetest/dcelocal/test/tet/system/profile.dcest.tet

  file in your current shell before running any of the DCE System Tests. This will setup the shell variables the tests need. The main variables defined are:

  TET_ROOT **/dcetest/dcelocal/test/tet**

  This is the base directory for all tests which run under TET.

  DCELOGDIR **/dcetest/dcelocal/status**

  This is the base directory for DCE Functional and System test output.

  STTMPDIR **/tmp**

  This is the directory for creation of temporary files.

  Note: Note that sourcing **profile.dcest.tet** clears the **ENV** environment variable, thus affecting the behavior of all lower level Korn shell invocations. This will be a problem for any site that attempts to access **MANDATORY** Korn shell definitions via the **ENV** variable.

  There is no requirement that the machines in the system test cell have the ''root'' account *default* shell be the Korn shell (/bin/ksh). If the default shell is something other than **ksh**, you need only invoke the Korn shell before sourcing **profile.dcest.tet**.

11.2.5 Standard DCE System Test Output Location

Each time you run a system test with a **run** script, a new directory will be created in the ''standard location''. The standard output location for the DCE System Tests is:

$DCELOGDIR/system/*testname*.*hostname*.*yymmddhhmmss*

If **DCELOGDIR** is non-existent or empty, the default directory

/dcetest/dcelocal/status

will be used. No matter what directory name is specified by **DCELOGDIR**, the **run_loops.ksh** script will append **/system** to it.

Note that there is no **run** script option by which **DCELOGDIR** can be specified or overridden. You must either set the variable yourself to the desired pathname, or the

/dcetest/dcelocal/status

default directory must already exist when the **run** script is invoked.

The final directory name

*testname***.***hostname***.***yymmddhhmmss*

is designed to allow you to pick out a particular run by what you ran, where you ran it, and when you started it. The fine granularity of this name enables you to direct the output of multiple runs to a single collection point without worrying about name collisions.

Because all the normal output from one run of these tests is created under a single directory, deleting the output of that run when you are finished with it is easily done with a

rm -r *dir_name*

command.

The normal output of a test run is found in a structure of files underneath the standard location. The directories and files found there are as follows:

- **pass**

  Directory containing results of passed iterations.

- **pass/***journal.NNNNN*

  Pass journal files.

- **fail**

  Directory containing results of failed iterations.

- **fail/***journal.NNNNN*

  Failure journal files.

- **pass_fail_log**

  Log file of all test iterations, both passed and failed.

The journal files are TET format journals. *NNNNN* is a digit group that represents the iteration number of the test whose results are recorded. You should refer to the TET documentation for the details of the format of these files. In general, the format is that each line has a TET-defined header before and between a vertical bar pair, followed by the test- or TET-generated text. Test-generated messages start on the line labelled with a ‘‘520’’ code. The following example shows part of the contents of a typical journal file; the last two lines were generated by the test itself.

```
10|0 /ts/cds.server.4/cdsserv.ksh 14:19:03|TC Start, scenario ref 35-1
15|0 1.9 1|TCM Start
520|0 0 25874 1 1|Starting test CDSSERV
400|0 1 1 14:19:06|IC Start
200|0 1 14:19:06|TP Start
520|0 1 25874 1 2|The test will execute for: 900 sec.
```

```
520|0 1 25874 1 3|Executing in cell: /.../dce3_cell.qadce.osf.org
```

The TET journal files are always created and written in the **fail** directory and only moved into the **pass** directory if and when the test iteration has passed. At least one passed test iteration will have its journal file saved, assuming that any iterations passed at all. All journals from failed iterations are left in the **fail** directory.

The **pass_fail_log** file is created by the **run.***component_name* script and has the following format:

```
COMMAND: command invoking the run
PLATFORM: name of machine the test was run on (platform type)
TEST NAME: test name
STARTED AT: time stamp recording when the run was started
NEWEST /OPT/DCELOCAL/BIN: time stamp of when DCE was built
Pass/fail lines, one per iteration.  Each line contains:
  iteration completion timestamp
  <tab>
  PASS or FAIL keyword
  <tab>
  full journal file specification or delete message
COMPLETED: timestamp indicating when all iterations of run completed
```

The ‘‘COMPLETED’’ line at the end of the file shows that the requested testing was run to normal completion, whether successful or not; i.e. that the run did not hang.

Each of the tests sends test-specific output to standard output. However, since all these tests ultimately run under **run_loops.ksh**, the text sent to standard output is always surrounded by a series of standard lines of information, as in the following example:

```
Starting DCETH002 iteration 1 at 0 seconds executed, 11/12/93-14:37:22
journal file name is:
/dcetest/dcelocal/status/system/dceth002.hostname.931112143719/fail/journal.00001
  Output from DCETH002
  Output from DCETH002
  Output from DCETH002
PASSED,     Test "DCETH002_C":      Test ran successfully.
Completed iteration 1 successfully at 18 seconds.
Journal file moved to
/dcetest/dcelocal/status/system/dceth002.hostname.931112143719/pass/journal.00001

Command "run.thr -l 1 dceth002" completed at 11/12/93-14:37:41
All iterations on DCETH002 completed without error!
See synopsis of iteration status in
/dcetest/dcelocal/status/system/dceth002.hostname.931112143719/pass_fail_log
or use $TET_ROOT/system/tools/run_summary.ksh for more information.
```

There is a header and a trailer for each iteration of the test. Totals are output when all iterations have completed. Other information includes the pass/fail status of each iteration and of the test run as a whole.

11.2.6 Command Line Options Common to Some or All of the ‘‘Run’’ Scripts

The basic **run.***component_name* script command line to invoke a DCE system test has the following general form:

run.*component_name {***-l** *loops* | **-t** *hours} [other_opts] test_name [parameters]*

Either the **-l** or **-t** flag is required, as is the test name. In DCE 1.1 the names of the **run** scripts are as follows:

- **run.cds**
- **run.rpc**
- **run.sec**
- **run.thr**
- **run.time**
- **run.hcell**
- **run.dced**
- **run.aud**
- **run.svc**

The following command line options are common to some or all of the tests:

-h List test-specific options.

Causes the test-specific options for a test to be listed. For **run.thr**, you must specify the test name as an argument to this option. Note that there is no test-specific help for **run.time**. However, all the **run** scripts give basic help messages if invoked with no arguments.

-l *number* Number of external loops (iterations) to run.

-t *hours* Time allowed for external loops (iterations) to run.

-L *number* Number of internal loops to run.

-T *hours* Time allowed for internal loops to run.

The above four flags specify in various ways the number of times or hours that the test is to be run. An iteration count is most useful for quickly checking the test, e.g. invoking a test with something like ‘‘**-l 1**’’ or ‘‘**-l 2**’’ will allow you to quickly verify whether the test is present, whether it runs, and so on.

For longer test runs it is more useful to specify a time rather than an iteration count; for example ‘‘**-t 60**’’ for a Friday night-to Monday morning, 60 hour run. The *hours* parameter may contain a decimal point; e.g., ‘‘**-t 1.5**’’ is a valid specifier for a 90 minute run. Note that the **run** scripts make *no* time estimates. If at the conclusion of an iteration only one second is left in a specified time interval, the **run** script will start another

iteration of the test. Note also that the **run** scripts make extensive use of the Korn shell **SECONDS** variable. You should not alter the tests in any way that affects this variable's value.

The **-l** and **-t** flags both control *external* test iterations, that is, loops in which the entire test is repeated, including:

- TET invocation
- creation of a new journal file
- test initialization
- invocation of the test itself
- cleanup

The **-L** and **-T** flags accept the same parameters as **-l** and **-t**. For tests for which they are available, they control *internal* test looping, in other words: the number of times the test itself is executed within a single invocation of TET (including journal file creation and cleanup). The **-L** and **-T** options are available for the following **run** script/test combinations:

run.cds	**cdsserv**
run.rpc	**dcerpcrun**
run.sec	**dcesergy**

The **-l** and **-t** options are mutually exclusive, but either one or the other is required for most tests. The **-L** and **-T** are likewise mutually exclusive, but for the tests listed above it is acceptable to specify one internal loop control (**-L** or **-T**) along with the required external iteration control (**-l** or **-t**).

-c Keep all journal files from successful iterations.

Normally, when a successful external iteration of the test is completed, the journal file is deleted, and the only record of the iteration is a single line in **pass_fail_log** noting that the iteration passed, when it completed, and that **journal.***NNNNN* was deleted.

There are three possible reasons why a journal file will not automatically be deleted:

- The test iteration failed, in which case the journal file is saved in the **fail** subdirectory.
- The journal file was for the first successful iteration (usually **journal.00001**).
- The **-c** option was specified, in which case all journal files are kept.

-e *number* Maximum number of consecutive errors allowed before quitting (default is 50)

-E *number* Total number of errors allowed before quitting (default is 500)

There is seldom any useful information to be gained from the contents of a large number of failure journal files. Moreover, some failure modes can result in a large number of test executions occurring in a very short time, possibly filling up the disk. In order to prevent this, upper limits on test failures are imposed by the **run** scripts by default. You can use

the **-e** and **-E** flags to modify these limits. For example, a common expedient is to specify ‘‘**-e 1**’’ which will cause the run to terminate as soon as one error is detected.

-m *name(s)* List of machine(s) for the test to use.

-M List of machine(s) to use should be read from <testname>.data.

Several tests require one or more additional machines for execution, and the **-m** option is used to specify that information. Using the **-m** option to specify the machine the test is invoked on is legal but reduces the usefulness of the test. Both multiple ‘‘**-m** *name*’’ groups or a single ‘‘**-m** *name1 . . . nameN*’’ are accepted.

The tests can also obtain the list of additional machines from the *testname***.data** file associated with each test. The **-M** flag is used to tell the **run** script that the absence of the **-m** option for a test that requires it is not an error. The use of the **-M** option is discouraged, because it requires changing files whenever different machines are needed to run a test. The **-m** and **-M** flags are mutually exclusive.

The tests that require a list of additional machines are the following:

run.sec	**dceseact**, **dcesepol**, **dcesestr**
run.rpc	**dcerpbnk**, **dcerpcrun**, **dcerpper**
run.thr	**dcethrpc**, **dcethrpc_auth**

11.2.7 External and Internal Looping

In general, test ‘‘looping’’ can be classified into external loops (iterations) and internal loops. A somewhat confusing collection of common and test-specific options exists for controlling looping of the system tests.

There is no ‘‘best way’’ to run the system tests with respect to the division between internal and external loops. Tests that support internal loops and/or execution threads have default count parameters, and the user is thus not required to specify them. However, explicit biasing may be done. See ‘‘Command Line Options Common to Some or All of the ‘‘Run’’ Scripts’’ for more information.

Biasing towards more internal loops makes the tests more efficient testers of DCE because they spend less time in initializing and cleaning up. Furthermore, since many tests set up accounts and such, running for an equal length of time with higher internal bias creates fewer accounts and causes the servers to grow less.

Biasing towards a greater proportional number of external iterations affords TET more opportunities to indicate test success or failure, which is usually desirable in long runs. However, if the bias on internal looping is too large, there is a risk of the test’s credentials expiring. And, too, failures sometimes occur that affect several system tests at once. Having too large a time granularity as a result of high internal bias makes it difficult to correlate such failures. Thus keeping the internal loop time down is desirable even though this adds to test overhead.

It is still desirable to avoid the extreme case where the test is biased completely internally, for example as the following run would be:

run.cds -l 1 -T 48 cdsserv

The above command specifies that one external iteration of **cdsserv** be run with a 48 hour internal duration. No matter what goes wrong during this run, there will only be a single failure from TET as a record of it., and there will be a rather large journal file to evaluate. A more balanced approach would be to run the test as follows:

run.cds -t 48 -T .25 cdsserv

—that is, with 48 hours' worth of 15 minute runs.

There is also the question of how to increase the load on DCE during a system test run. Increasing the internal loop bias increases the actual DCE work done per test executed, but that approach suffers from diminishing returns. Running more tests simultaneously on different machines in the test cell is the right way to make the servers busier.

Note: The DCE 1.1 system tests were known to have mutual interference problems within a cell, causing test (not DCE) failures. These failures were due to name collisions both in the filespace and in the DCE namespace.

The possibility of interference should be considered when planning simultaneous DCE system test runs. Interference of some tests with themselves has been noted where two or more copies of the test were run simultaneously on the same machine or even in the same cell. However, interference has not been noted with multiple, different tests run simultaneously on different machines, one test per machine.

Using DCE DFS or NFS to create common areas for the DCE system tests to use, especially directories for temporary files, makes the interference problem significantly worse. However, the standard output location provided by the **run** scripts is a *known safe* exception.

11.2.7.1 Checking Test Results

A reporting script has been provided that produces a summary of all the DCE system test run output collected in a single directory tree. To run it, enter:

run_summary.ksh *directory*

run_summary.ksh does a

find *directory* **-name pass_fail_log**

to find all the DCE system test run records under *directory*, and then summarizes and displays the results, including any journal file error messages from iteration(s) that failed, if any.

Following is an example of the output, showing in this case the error messages from one iteration (out of 2569 total) of **dcethrpc**. The test was run on an HP/UX platform named ‘‘dce3’’, and the machines ‘‘dce5’’ and ‘‘west’’ were also used:

```
dce2(hpux)      dcethrpc:       pass = 2568, fail = 1
        "run.thr -t48 -m west -m dce5 dcethrpc" completed at 11/10/93-14:18:44
        /opt/dcelocal/bin complete: Nov 4 21:15
        Failures under /dcetest/dcelocal/status/system/dcethrpc.dce2.931108141817
                There were 6 ERRORs and/or FAILures total in
                the 1 failed iterations.  Here's the breakdown:
                   1 - DCE_ERROR
                   1 - ERROR: Copying thd_server to west:/tmp failed
                   1 - ERROR: dcethrpc threads test failed
                   1 - ERROR:Failed to start thd_server on dce5
                   1 - Exiting the test due to failure in check_servers
                   1 - Exiting the test due to failure in kill_servers
```

The error messages (identified by the case-insensitive keywords ‘‘error’’ or ‘‘fail’’) from journal files of failed tests are collected and sorted, duplicate messages are counted and eliminated, and each unique error message is reported. This simple summary can tell you a lot about whether the same or different errors were occurring during a run, and you can learn something of the nature of the errors as well.

You can do a

run_summary.ksh *directory*

while the test is running; in this case you will see a ‘‘`did NOT complete!`’’ message in the command output.

To collect the output from different test runs under a single directory, define **DCELOGDIR** for each test process before running the test.

See also ‘‘Performing a Quick Check of DCE on a Machine’’ later in this chapter for information on monitoring DCE status during DCE system test runs.

11.3 System Test Tools

This section describes the tool set developed to support the DCE System Test. These tools are generalized enough for system vendors to use them when developing their own test suites.

11.3.1 Performing a Quick Check of DCE on a Machine

dce.ps is a script that provides **ps** (process status) data only for the configured DCE processes that are supposed to be running on the machine it is executed on. It will

identify any missing configured processes and any unconfigured processes. If everything seems in order, **dce.ps** will go on to attempt to derive the cell name from a CDS clearinghouse name via **cdscp**. If that works, **dce.ps** will report the cell name, and you can be reasonably sure that the cell is running.

dce.ps returns a 0 (success) status code only if it successfully completes all its checks; otherwise it returns a non-zero code. It provides ‘‘**ps**’’-style output for the DCE processes and helpful messages for the user as well. An attempt has been made to standardize the **dce.ps**’s process status output across platforms. Following is an example of its output:

```
$ $TET_ROOT/system/tools/dce.ps

The following DCE components are running on "dce2".
   PID     STIME   TIME COMMAND
 17075  10:18:05   0:15 rpcd
 17194  10:18:54   0:28 secd -bootstrap
 18689  10:31:13   0:03 dts_ntp_provider -h paperboy -p 600 -i 30
 17654  10:21:50   0:32 cdsd -a
 18529  10:30:19   0:19 dtsd
 18556  10:30:29   0:00 dtstimed
 17625  10:21:40   0:03 cdsadv
 18481  10:30:06   0:06 sec_clientd
DCE on "dce2" seems to be running as configured.
Configured: dts_ntp_provider dtstimed dtsd cdsd cdsadv sec_clientd secd rpcd
CDSCP says "dce2" is responding in the cell "/.../my_cell".

$ echo $?
0
```

Note that if you have sourced **profile.dcest.tet** in your current or ancestor shell then the proper path exists in **PATH**, and you need only type **dce.ps**. Note also that **dce.ps** will give incorrect results while **dced** system tests are being run.

11.3.2 TET Tools

The following sections describe several utilities that have proven useful in integrating tests with TET.

11.3.2.1 tet_setup

tet_setup is a utility used by various DCE tests run under TET. When invoked, it executes (as root, and **dce_login**’d as the machine machine principal (\m for example, **hosts/foobar/self**) a program specified to it. The program is typically a TET-run test; executing it via **tet_setup** allows it to assume the principal identities necessary to test

desired ACLs.

It is invoked as follows:

tet_setup *program* [*args* . . .]

where:

program is the name of the program to be executed

args are the arguments, if any, to be passed to the program to be executed

For an example of **tet_setup use, see the contents of**

dce-root-dir/**dce/src/test/admin/dcecp/ts/secval/secval_cleanup.tcl**

or:

dce-root-dir/**dce/src/test/admin/dcecp/ts/secval/secval_setup.tcl**

tet_setup is installed in:

dce-root-dir/**dce/install/***platform*/**dcetest/dce1.2.2/test/tet/tools**

Its source is located in:

dce-root-dir/**dce/src/test/tools**

11.3.2.2 TET Utility Routines

Source for various miscellaneous TET utility routines is located in:

dce-root-dir/**dce/src/test/lib/libdcetst**

The utilities are built into a library **libdcetst.a** which is placed in:

dce-root-dir/**dce/export/***platform*/**usr/lib**

when DCE is built. Following are brief descriptions of the routines.

- **extern int tst_tet_printf(const char *_format_, ...)**

  Sends the contents of a **printf( )** to the TET journal file. Allows a tester to use the different format directives accepted by **printf( )** when sending a message to the journal file (**tet_infoline( )** by itself does not allow this). If successful, a 0 is returned; otherwise, a non-zero value is returned.

- **extern void tst_dce_login(char *_user_, char *_password_, error_status_t *_status_)**

  Attempts a **dce_login** as the specified principal. If successful **error_status_ok** is returned; otherwise, a non-zero value is returned.

- **extern int tst_chk_command(char *_command_, pid_t *_pid_)**

  Checks whether the specified command is in the process table: if it is, a 0 is returned; otherwise, a non-zero value is returned.

- **extern int tst_chk_process(pid_t *_pid_)**

Checks whether the specified *pid* is in the process table: if it is, a 0 is returned; otherwise, a non-zero value is returned.

11.3.3 Multi-Vendor Test Case Development Tools

The test cases have been designed to be easily ported to other flavors of the UNIX operating system. This is aided by a suite of tools which are considered multi-vendor because they are aware of the flavor of UNIX which they are running under and adjust their nature of operation accordingly.

A good example of the types of porting problems you may encounter is the use of the **ps** command. If a test case needed to determine the process identification (PID) of some daemon process, it would search the output of the **ps** command for the name of the daemon in question and extract its PID. The **ps** command has a different syntax for the BSD and System V flavors of UNIX. For example, BSD UNIX syntax is **ps -ax** while System V syntax is **ps -ef**. The test case needs to be aware of the type of system it is executing under in order to be able to choose the proper syntax. The problem grows very quickly. A test case needs a special case for each difference of each flavor of UNIX. Not only can this cause the test cases to be hard to port and maintain, but the readability and modularity of the code can suffer as well. For example, the test case would need a large **case** statement to handle the various syntaxes of the same command offered on the different flavors of UNIX. Maintaining every instance of the command's usage in every test case is time-consuming and costly.

The object of the multi-vendor tools is to abstract the differences of the flavors of UNIX into a set of commands. The commands determine the type of operating system automatically, if they have been ported to a particular flavor. Once the type of operation system is known, it is easy to use the correct syntax of the command.

The tools currently support the following operating systems:

- AIX 3.2.4
- OSF/1 1.2 (on DECpc 450ST)
- HP/UX 9.0.1
- SINIX 5.41 (SVR4 on MX300i)

To port the tools to another operating system, you need to begin with the **expmachinfo** command. In the system test environment, this command is executed by:

systest-root/**profile.dcest**

or

/dcetest/dcelocal/test/tet/system/profile.dcest.tet

It creates the environment variables necessary for the other commands to determine under what operating system they are executing.

The following commands are referred to as the core set, and since they use the information created by the **expmachinfo** command, they may also need to be ported:

chkproc Returns 0 if a process exists and 1 if it does not.

getproc Returns the process id (PID) of a given process.

The rest of the commands increase the usability of the core set. They are as follows:

killproc Kills processes that match the given strings.

rshsp Enhances the usability of **rsh** by sourcing a file of environment variables before running the command on the remote machine and by returning the return code of the remote command.

All the commands are located in the

/dcetest/dcelocal/test/tet/system/tools

directory.

11.3.4 Test Case Logging Facilitators for System Tests Not under TET

The tools in this section were developed to support faster analysis of scenario executions. They provide standard mechanisms for logging results, and several tools for examining the status.

11.3.4.1 Logging Results

The tools that are used to log results print the message you provide, with a header attached to the front to indicate what has occurred (success, failure, etc.). The message is printed to **stdout** and to the file whose name is stored in the environment variable **JOURNAL** (see the ‘‘Test Logging During Iterations’’ and ‘‘Test Logging After Iterations’’ sections earlier in this chapter). This allows testers to watch the progress of tests scroll by on the screen while also recording the results in a permanent file.

These tests are divided into two groups: tests used by testcases and tests used by testcase drivers. The following commands should be used within testcases:

xx_log Records something that worked successfully (or adds comments to the journal file).

xx_warning Records something that may not have been an actual error but should be examined.

xx_error Records something that did not work successfully (but the testcase will continue to execute).

xx_severe Records something that failed and was so important that the testcase should not continue to execute.

These commands are available from the command level (through shell scripts), and at the API level through a library called **libxx_.a**. The scripts are installed and the library built via:

dce-root-dir/**dce/src/test/systest/tools/Makefile**

The following commands should be used only by testcase drivers:

xx_begin Marks the beginning of an iteration of a testcase.

xx_pass Indicates that a testcase iteration has completed successfully.

xx_fail Indicates that a testcase iteration has completed with errors.

xx_example Checks journal files for errors.

These commands are available only at the command level, not at the API level.

11.3.5 Execution Tools

You can use the following tools to set up and execute System Test scenarios:

test/tet/system/profile.dcest.tet Defines all the necessary environment variables used by the DCE system tests run under TET.

test/systest/profile.dcest Defines all the necessary environment variables used by all DCE system tests not run under TET.

11.3.6 Miscellaneous Tools

The following tools are also available:

gdskill Deactivates a directory system installation of GDS, then deletes the configuration data.

gdsSetup Sets up GDS on a system, based on the contents of a configuration file. See the contents of the **gds_xds_str_001.data** file for more information.

worldSetup Sets up GDS on a system, based on the contents of a configuration file. See ‘‘Running the dcegdshd Driver’’ in Chapter 13 for more information.

su.dce Provides DCE authentication and accepts passwords on the command line. This program should be owned by **root** and have the **setuid** bit set.

rcheck Checks a return code value against an expected value.

Chapter 12. DCE System Tests under TET

The following subsections list the test-specific options and descriptions of the DCE system tests that have been converted to run under TET and the **run** control scripts.

All of these tests are run using the command format and common options described above, and produce TET journal file and **stdout** output also as described above.

It should be noted that some tests are intended to generate errors. Some of the resulting error messages appear in the standard output and may appear to be test errors, although they are not. The test journal files are always the final authority as to whether a test passed or failed.

The following subsections also contain information about the tests' associated ''data'' scripts. These data scripts contain variables and default values for: internal loop counts; thread counts; protocols; organization, group, and principal names; passwords; directory and file names; test data; file sizes; wait times; and other, more esoteric runtime parameters.

Some data script variables have test-specific command line options associated with them; it is recommended that you use the command line options to change the value of such variables at run time. If you wish to change variables that are not accessible from the command line, you should consult the test and data scripts for information.

Note: All DCE system test verification was done with the default values for all data file variables which are not alterable by command line option. It is left entirely to the user to resolve problems arising from alteration of variables not accessible from the command line.

12.1 Threads

The following sections describe the DCE Threads system tests run under TET.

12.1.1 dcethcac

Tests how many threads will co-habitate in an operating system by caching a user-specified number of threads and yields (calls). The test may be used for stress testing by specifying a large number of threads (via the **NUMBER_OF_THREADS** environment variable).

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
set TET_ROOT=‘pwd‘
set PATH=$TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=.1 threads dcethcac**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **threads** directory) for the test results journal file.
-vRUN_TIME=0.25	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
threads	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
dcethcac	Specifies the name of the test (TET scenario) to be run.

12.1.2 dceth002

dceth002 is designed to exercise the threads-creation capability. It calls **dceth002_c**, creating a number of threads in each of a number of processes. The threads then loop and perform some simple computations.

Test Script: **$TET_ROOT/threads/ts/dceth002/dceth002**

To run **dceth002**, do the following:

1. Set (if desired) the following environment variables:

   NUMBER_OF_THREADS
   Specifies the number of threads to create in each process. Default is 40.

   PROCESSES Specifies the number of processes to run. Default is 4.

2. Invoke the test as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-i** *intermediate_results_path* **threads dceth002**

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **threads** directory) for the test results journal file.

-i *intermediate_results_path*
Specifies a file pathname (relative to the **threads** directory) for the intermediate test results file.

threads Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.

dceth002 Specifies the name of the test (TET scenario) to be run.

This test can be used for stress testing by specifying a large number of threads and a large number of processes.

12.1.3 dcethmut

dcethmut is designed to exercise the threads-creation capability and the use of mutual exclusion primitives. It runs a number of copies of **dcethmut_c** in separate processes, each creating a number of threads which lock and unlock the same mutex repeatedly.

Test Script: **$TET_ROOT/threads/ts/dcethmut/dcethmut**

To run **dcethmut**, do the following:

1. Set (if desired) the following environment variables:

   NUMBER_OF_THREADS
   Specifies the number of threads to create in each process. Default is 40.

   PROCESSES Specifies the number of processes to run. Default is 4.

2. Invoke the test as follows:

   cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
   setenv TET_ROOT 'pwd'
   setenv PATH $TET_ROOT/../test/tet/bin:$PATH
   tcc -e -j *journal_path* **-i** *intermediate_results_path* **threads dcethmut**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **threads** directory) for the test results journal file.
-i *intermediate_results_path*	Specifies a file pathname (relative to the **threads** directory) for the intermediate test results file.
threads	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
dcethmut	Specifies the name of the test (TET scenario) to be run.

12.1.4 dcethrpc

dcethrpc tests RPC servers’ and clients’ ability to spawn multiple threads. It primarily tests the DCE Threads and RPC components but can also use the Security component.

Test Script: **$TET_ROOT/threads/ts/dcethrpc/dcethrpc**

Sets the following values:

- **THREAD_EXEC**

  Specifies a pathname for executing the threads. Its value should be ‘‘**$TET_ROOT/threads/ts/dcethrpc**’’.

- **PROTOCOL**

  Specifies the protocol sequence to use: ‘‘**ncadg_ip_udp**’’ (connectionless, the default) or ‘‘**ncacn_ip_tcp**’’ (connection-oriented). This option is useful for testing all the protocols DCE supports.

- **NUMBER_OF_THREADS**

  Specifies number of threads to create (default: 100).

- **CHUNK_SIZE**

  Specifies the size of the portion of array that is read by the server. The default is 100.

- **MAX_CALLS**

  Specifies the maximum number of calls the server can handle concurrently. The default is 5.

- **RPC_MACHINES**

Specifies machines to use for servers. For example, ‘‘**osf1 osf2 osf3 osf4 osf5**’’.

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-i** *intermediate_results_path* **threads dcethrpc**

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **threads** directory) for the test results journal file.

-i *intermediate_results_path*
Specifies a file pathname (relative to the **threads** directory) for the intermediate test results file.

threads Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

dcethrpc Specifies the name of the test (TET scenario) to be run.

This test can be used for stress testing by specifying a large number of threads to create (note that the number of threads that can be created is dependent on the memory capacity of the machine), setting a large chunk size, or by specifying many machines with the **-m** option.

dcethrpc_auth is simply a variant of the normal, unauthenticated running of **dcethrpc**, so it supports all the **dcethrpc** and **run** options, as well as the following additional options:

-A *number* Specifies the authentication level (Default: 0. 0 maps to default level.)

-V *number* Specifies the authentication service (Default: 1. 1 maps to private key authentication.)

-Z *number* Specifies the authorization service (Default: 2. 2 maps to DCE PAC authorization.)

-N Specifies unauthenticated RPC; overrides **-p**, **-A**, **-V** and **-Z** flags.

-p *principal_name* Specifies the account to authenticate with. This account must already exist in the security registry, and its password must be identical to its name. Moreover, the name must be registered locally on each machine you want to target with the **-m** option. Use the following **rgy_edit** command on each target machine to register the name locally and to verify that the name was registered locally with the **ktlist** command:

ktadd -p *principal_name* **-pw** *principal_name*

You must set the above options for the client and server by setting the **CLIENT_PARMS** and **SERVER_PARMS** environment variables to the desired option values. Once this has been done, **dcethrpc** will run as **dcethrpc_auth** when executed.

Note: The account added for **dcethrpc_auth** must have a password identical to its name. However, it is a severe security breach to leave this account extant after running the test. Make sure that you delete the account when you have completed running this test.

12.2 RPC

The following sections describe the DCE RPC system tests that are run under TET.

12.2.1 dcerpary

dcerpary is designed to test the ability of the RPC runtime to transmit arrays of arrays and arrays of pointers structures.

Refer to the comments in

dce1.2.2-root-dir/**dce/src/test/systest/rpc/ary_client.c**

and

dce1.2.2-root-dir/**dce/src/test/systest/rpc/ary_server.c**

for full details on how this testing is accomplished.

Note that only *one* **ary_server** process can run on a single machine, because the process listens on a well known port.

The test is invoked as follows:

```
cd dce-root-dir/dce/install/platform/opt/dcetest/dce1.2.2/tet
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -vRUNNING_TIME=.01 rpc dcerpary
```

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **rpc** directory) for the test results journal file.

-vRUNNING_TIME=0.01
Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.

rpc Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

dcerpary Specifies the name of the test (TET scenario) to be run.

12.2.2 dcerpidl

This test runs a selected number of DCE IDL tests. The idea is to run the tests between machines that have different endian representations.

However, note that the test programs are compiled only through ODE (that is, when DCE is built), not on the fly.

Also note that the **dcerpidl** tests run only on similar HP-UX machines.

The tests are invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH

tcc -e -j *journal_path* **-vRUNNING_TIME=0.25 rpc dcerpidl1**

tcc -e -j *journal_path* **-vRUNNING_TIME=0.30 rpc dcerpidl2**

tcc -e -j *journal_path* **-vRUNNING_TIME=0.30 rpc dcerpidl3**

tcc -e -j *journal_path* **-vRUNNING_TIME=0.30 rpc dcerpidl4**

tcc -e -j *journal_path* **-vRUNNING_TIME=0.30 rpc dcerpidl5**

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **rpc** directory) for the test results journal file.

-vRUNNING_TIME=0.*nn*
Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.

rpc Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

dcerpidl*n* Specifies the name of the test (TET scenario) to be run.

12.2.3 dcerprec

This test is designed to test the ability of the RPC library to handle heavy stress loads. The test is in two sections: a server side and a client side.

The client reads a file locally and remotely, and then compares the buffers to verify that the identical information was read both locally and remotely.

The server offsets into the file the required amount of bytes, reads the specified amount of bytes from that point, and passes this buffer back to the client.

Note: The stress levels of this test are low.

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/***platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_file* **-vRUNNING_TIME=.50 rpc** *dcerprec*

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **rpc** directory) for the test results journal file.

-vRUNNING_TIME=0.50
Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.

rpc Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.

dcerprec Specifies the name of the test (TET scenario) to be run.

12.2.4 dcerpbnk

The **dcerpbnk** DCE System Test is a small scale simulation of a banking operation. It tests most aspects of DCE and resembles an actual application.

dcerpbnk tests the RPC component (in particular the Object Registry table and Interface Registry table), as well as the Threads, CDS and Security components.

Test Script: **$TET_ROOT/rpc/ts/dcerpbnk/dcerpbnk**

dcerpbnk is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUNNING_TIME=0.25 rpc dcerpbnk**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **rpc** directory) for the test results journal file.
-vRUNNING_TIME=0.25	Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.
rpc	Specifies the ''test suite'' name.
dcerpbnk	Specifies the name of the test (TET scenario) to be run.

12.2.4.1 Running dcerpbnk_auth

dcerpbnk_auth is simply an authenticated variant of the unauthenticated **dcerpbnk**.

The following additional setup is required before running the test:

1. **dce_login** as the Cell Administrator (**cell_admin**).
2. Invoke **rgy_edit** and add the test principal as follows:

```
$ rgy_edit
=> domain principal
=> add bankd
=> domain account
=> add bankd -g none -o none -pw password -mp -dce
=> ktadd -p bankd -pw password
=> quit
```

The test itself is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUNNING_TIME=0.25 rpc dcerpbnk_auth**

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **rpc** directory) for the test results journal file.

-vRUNNING_TIME=0.25
Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.

rpc Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

dcerpbnk_auth Specifies the name of the test (TET scenario) to be run.

12.2.5 RPC Runtime Stress Test

This test first determines the platform’s maximum number of concurrent multiple client calls allowed to a server, and then repeatedly transmits an array of structures of ten members back and forth between its clients and server.

Test Script: **$TET_ROOT/rpc/ts/rpc.runtime.1/dcerpcrun**

Data Script: **$TET_ROOT/rpc/ts/rpc.runtime.1/dcerpcrun.data**

dcerpcrun is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vNMIN=15 rpc dcerpcrun**

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **rpc** directory) for the test results journal file.

-vNMIN=15 Sets the **NMIN** environment variable, which specifies the number of minutes the test should run.

rpc Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

dcerpcrun Specifies the name of the test (TET scenario) to be run.

12.2.5.1 Description of dcerpcrun

The **dcerpcrun** system test is a low level test of the DCE RPC runtime. It is designed to perform load-testing on RPC at the same time as other DCE system tests are exercising DCE upper layer functions (for example, in Security or CDS). **dcerpcrun** is derived from RPC functional tests, and thus does not itself exercise this upper layer functionality. The test contains the following enhancements over the functional test version:

- The test executes multi-threaded client calls to one server interface.
- A structure of 10 numbers is built into an array of 1000 elements and piped to and sent back from the server. Error checking is performed on both sides of the bi-directional pipe.

12.2.6 RPC-Security System Test

The **rpc.sec.2** system test is designed to stress the RPC and Security components of DCE.

The Security component is stressed via frequent identity updates and validations, and the RPC component is stressed via continuous RPC requests by multiple clients using full authentication and a complex data type (a conformant structure). The client side test code defaults to making calls as fast as possible so as to put as much load as possible on the server.

This test can also be used for performance testing of authenticated RPC, though this is not its default behavior. Note that an attempt has been made in the coding of this test to observe good programming practice from the DCE point of view.

In outline, the test operates as follows:

1. The test first determines the implementation's maximum number of concurrent calls for multiple clients to a server, using the highest level of authentication offered by the runtime library (**rpc_c_protect_level_pkt_privacy**), and transmitting structures with conformant array members. The concurrent call maximum will be sensed by the receipt of the RPC status **rpc_s_server_too_busy** (if the client is using a connection-oriented protocol) or **rpc_s_comm_failure** (if the client is using a connectionless protocol).
2. Following the determination of the call maximum, the test is run in a loop after a reset of the registry server ticket lifespan to five minutes for the test client and server principals in order to validate ticket renewal operations near the maximum call rate.

12.2.6.1 Logic Flow of the RPC-Security System Test

dcerpsec, the script invoked by TET, calls **rpc.sec.2_setup.sh** and **rpc.sec.2_runtest.sh**.

In outline, the operation of **rpc.sec.2_setup.sh** is as follows:

1. Checks to make sure that the user has a valid identity as **cell_admin**.
2. Checks to make sure that all of the variables used by the setup script are assigned values in the configuration file.
3. Creates the client and server principals.
4. Creates the client and server accounts and keytab files. If the path to the keytab file does not exist, the script attempts to create it. Note that you will be prompted for the **cell_admin** password twice during this part of the setup.
5. Creates the CDS directory into which the server interface entry will be exported.
6. Attempts to create a client keytab file on any systems named in the configuration file as client machines in the test.
7. Attempts to create a server keytab file on any systems named in the configuration file as server machines in the test.

TABLE 12-1. Objects Created by the rpc.sec.2 System Test

DCE Object Needed	Variable in config file	Default value as shipped
Server principal and account	**RPCSEC2_SRV_PRINC_NAME**	rpc.sec.2_srv
Client principal and account	**RPCSEC2_CLI_PRINC_NAME**	rpc.sec.2_cli
Group for the server test	**RPCSEC2_SRV_GROUP_NAME**	subsys/systest/cds_test
Server key file	**RPCSEC2_SRV_KEYTAB_FN**	rpc.sec.2_srv.keytab
Server key file directory	**RPCSEC2_CLI_KEYTAB_DIRPATH**	/tmp
Client key file	**RPCSEC2_CLI_KEYTAB_FN**	rpc.sec.2_cli.keytab
Client key file directory	**RPCSEC2_CLI_KEYTAB_DIRPATH**	/tmp
CDS directory for server interface object	**RPCSEC2_SRV_CDS_NAME**	/.:/test/systest/srv_ifs
Configuration file with test defaults and parameters	**RPCSEC2_CONF**	rpc.sec.2.conf

Logic Flow of ‘‘rpc.sec.2_runtest.sh’’

1. Reads the default configuration file **rpc.sec.2.conf**, or specified by the **RPCSEC2_CONF** environment variable (if it was not specified with the **-f** option).
2. Parses the command line options.
3. Determines the number of UDP and TCP clients to be run.
4. Reports to the user on the parameters that will be used for the run, describing the number of UDP and TCP clients, total clients, machines involved, the status of various flags, the duration of the run, where log files will be kept, and so on. In this way the parameters are recorded for later reference.
5. Builds a list of the clients to run.
6. Verifies the presence on each client machine of: a client binary, the configuration file, and a keytab file; and then starts the client(s) specified for that machine.
7. Waits a specified duration of time for the clients to start.
8. Verifies that all clients are started and ready to make remote calls. If all clients are not ready, and the **-I**[gnore] option has not been specified, then a message detailing the failures is output, all clients are killed, and the script exits.

9. Creates the synchronization file (i.e., the file for whose creation each client has been waiting as its cue to begin operations) on all the client machines to signal the clients to begin making RPCs to the server.
10. (*Boundary mode only*) Waits a specified amount of time for the clients to make their single RPCs.
11. (*Boundary mode only*) Parses the logs from the clients' runs and outputs a report that describes in detail: the number of clients run; how many of each type (UDP or TCP) failed or passed, etc; and a declaration of whether the test as a whole passed or failed.

12.2.6.2 Logic Flow of the RPC-Security System Test

As is possible with any program, this test evolved over its development. A description of the post-implementation state of the test follows:

Server side

The server side of the **rpc.sec.2** system test (**rpc.sec.2_srv**) performs the following operations:

1. Reads the configuration file and parses the command line options.
2. Assumes its own identity.
3. Looks for an entry in the CDS namespace for the interface it is to export.
4. Obtains binding handles from the endpoint mapper.
5. Exports its bindings and a UUID to the CDS namespace entry for the interface (unless a UUID is already present in the entry, in which case the existing UUID is used).
6. Starts a timer thread to automatically refresh the server's identity at the ticket lifetime's halfway point.
7. Starts a thread to catch and handle signals.
8. (If compiled with **-DRPCSEC2_SRV_REPORTING**) Starts a report thread to periodically generate reports of calls accepted, calls parsed, and calls failed.
9. Services requests for the **rpcsec2_cnf_str** RPC. In doing so it performs authorization based on client name, authorization service, authentication service, and protection level specified by binding handles in incoming RPCs. The incoming calls must specify the correct client and server name, DES authentication, DCE default authorization, and protection level.

Client side

The client side of the **rpc.sec.2** system test (**rpc.sec.2_cli**) performs the following operations:

1. Reads the configuration file and parses the command line options.
2. Assumes its own identity.
3. Spawns a thread to maintain its identity.
4. Spawns a thread to catch and handle signals.
5. If the run was specified to be for a duration of time, spawns a thread to generate reports at specified intervals.
6. Builds the structure to be passed to the server.
7. Looks in the namespace for a binding to the **rpc.sec.2** server. If a protocol is specified, only a binding with the specified protocol will be imported.
8. Tests the imported binding to make sure the server is active.
9. Displays a message indicating that it is ready to make calls.
10. (*Boundary mode only*) If a synchronization file is specified, loops until the sycnchronization file has been created (by the test driver script).
11. (*Boundary mode only*) Makes one call to the server, reports the result, and exits.
12. Loops, making **rpcsec2_cnf_str( )** calls to the server, checking results, and tracking successes and failures.

12.2.6.3 Test Options

All parameters for this test are specified in the test configuration file. Refer to the ‘‘Configuration File’’ section below for information about the variables and their format.

12.2.6.4 Compile-Time Switches for Optional Functionality

There are several areas of optional functionality available in the **rpc.sec.2** system test that can be used to expand the scope of the test or to provide additional runtime information. These areas of optional functionality are compiled into the program via the definition of tags which can be specified in either of two ways:

- On the **build** command line; for example:

  ```
  build -DRPCSEC2_ALL_OPTS
  ```

- In the environment variable **CENV**; for example (in a C shell):

  ```
  % setenv CENV RPCSEC2_ALL_DEBUGS
  % build
  ```

The default **build** flag value is **RPCSEC2_ALL_OPTS**.

The table below lists the supported compiler flags, the functionality that they control, and the operation of the test depending on whether the flag is or is not specified.

TABLE 12-2. Compile-Time Switches for rpc.sec.2

Flag	Functionality	Test Operation
RPCSEC2_KEEP_SYMBOLS	Keeps debugging symbols in compiled objects	If defined, code is compiled with debugging symbols; else debugging symbols usually stripped from objects.
RPCSEC2_SRV_REPORTING	Turns on server status reporting	If defined, server reports on call requests received, calls passed and failed, id refreshes, and time of last id refresh at an interval specified by RPCSEC2_CLI_DEF_REP_INTVL in configuration file. If not defined, server reports only upon receipt of SIGQUIT.
RPCSEC2_ADD_DUMPERS	Compiles extra routines into the server to aid debugging	If defined, several routines are compiled into the server for dumping the contents of various DCE data structures in people-readable form. These routines are not called from the code, but can be called from the debugger.
RPCSEC2_ALL_DEBUGS	turns on all debugging options	Has the same effect as specifying both RPCSEC2_KEEP_SIGNALS and RPCSEC2_ADD_DUMPERS.
RPCSEC2_ALL_OPTS	turns on all optional code	Has the same effect as specifying RPCSEC2_SRV_REPORTING.
RPCSEC2_DRVR_HNDLS_SIGCHLD	turns on code to stagger client exits	If defined, client will wait to exit after processing is complete, in an attempt to give the driver time to process client logs.
RPCSEC2_ALL_EXTS	turns on all extension code	Has the same effect as specifying RPCSEC2_DRVR_HNDLS_SIGCHLD

Specifying server reporting can provide useful information about the server side of the test.

12.2.6.5 Configuration File

Setting up to run the **rpc.sec.2** system test consists of one step, namely customizing the configuration file:

/dcetest/dcelocal/test/tet/rpc/ts/rpc.sec.2/rpc.sec.2.conf

The present section describes this step.

The scripts and programs that make up the **rpc.sec.2** system test get most of the information they need from a single configuration file whose default name is **rpc.sec.2.conf**. The information normally contained in this file can be split up roughly into two categories: default runtime parameters, and environment information.

Examples of default runtime information in the file are: the time duration a test should run; the names of machines on which clients will be run; etc.

Examples of environment information stored in this file are: the name of the CDS namespace entry to which the server exports its bindings; the name of the client and server principals; etc.

Before running the test, it is important to inspect the configuration file to see if any changes should be made for the site at which the test is to be performed. This is particularly important in regard to the environment configuration information. For example, you may wish to use a different client or server principal, a different CDS entry name, etc. All of these things, if they are to be changed, must be changed in the configuration file before running the test.

Note that all machines that the test is to be run on must have identical **rpc.sec.2.conf** files.

12.2.6.6 Format of the Configuration File

The contents of the test configuration file consist of text lines conforming to normal Bourne shell syntax.

Note, however, the following restriction. The configuration file, as implied above, is read by shell scripts, and by the **rpc.sec.2_cli** and **rpc.sec.2_srv** binary programs. In order to simplify the routine used by these programs to read the file, lines that set values for the **rpc.sec.2_cli** and **rpc.sec.2_srv** programs *must* be in one of the two following formats:

```
<string>=<string1>  # NOTE: in this case, string1 cannot
                    #       contain any spaces.
```

or:

```
<string>="<string1>"  # NOTE: in this case string1 may
                      #       contain spaces.
```

Any lines that are not in this format will either be ignored by the routine (**rpcsec2_rd_conf( )**, in the file **rpc.sec.2_rdconf.c**) that the client and server use to read the configuration file, or will generate an error. Comments are begun by a ‘‘#’’ character anywhere on a line, as shown above, and continue to the end of the line.

12.2.6.7 Contents of the Configuration File

The assignments in the configuration file as it is shipped represent the minimum set required to run the tests scripts and programs. You may add to the configuration file, but you should not remove any of the original assignments.

The information in the configuration file determines the way that your Security and CDS namespaces are set up. This being the case, you may want to modify the configuration information to tailor the namespace to your preferences. If you do not want to use the default values in the configuration file for the client or server principal name, CDS directory, CDS name, or for any of the other configuration file variables, you will have to modify the configuration file in accordance with your preferences before running the setup script.

TABLE 12-3. Configuration File Contents

Variable in Config File	Description	Default Value as Shipped
RPCSEC2_PROT_LEVEL	Default protection level	priv
RPCSEC2_CLI_PRINC_NAME	Client principal name	rpc.sec.2_cli
RPCSEC2_CLI_INIT_PW	Client initial password	"rpc&secC"
RPCSEC2_CLI_KEYTAB_DIRPATH	Directory for client keytab file	/tmp
RPCSEC2_CLI_KEYTAB_FN	Client keytab file name	rpc.sec.2_cli.keytab
RPCSEC2_CLI_MACHINES	Remote client machines	"rptest"
RPCSEC2_CLI_DEF_RUN_INTVL	Client interval to run	0 (hours)
RPCSEC2_CLI_DEF_REP_INTVL	Client report interval	1 (minutes)
RPCSEC2_CLI_SYNC_DELAY	Client delay for synchronization	60 (seconds)
RPCSEC2_CLI_START_DELAY	Clients startup delay	180 (seconds)
RPCSEC2_CLI_ARRAY_ELEMS	Number of array elements	15
RPCSEC2_SRV_PRINC_NAME	Server principal name	rpc.sec.2_srv
RPCSEC2_SRV_GROUP_NAME	Server group	subsys/dce/cds-test-group
RPCSEC2_SRV_INIT_PW	Server initial password	rpc&secS"
RPCSEC2_SRV_KEYTAB_DIRPATH	Directory for server keytab file	/opt/dcelocal/var/security/keytabs
RPCSEC2_SRV_KEYTAB_FN	Server keytab file name	rpc.sec.2_srv.keytab
RPCSEC2_SRV_CDS_NAME	Server interface name	/.:/test/systest/srv_ifs/rpcsec2_if
RPCSEC2_SRV_CDS_DIR_ACL	Directory ACL for server interface	rwditca
RPCSEC2_SRV_CDS_IO_ACL	Object ACL for server interface	rwdtc--
RPCSEC2_SRV_MAX_CALLS	Max concurrent call for server	5

Variable in Config File	Description	Default Value as Shipped
RPCSEC2_SRV_MAX_EXEC	Max concurrent execs for server	1
RPCSEC2_SRV_MACHINES	Server machine	"rptest"
RPCSEC2_BIN_DIRPATH	Directory with rpc.sec.2 binaries	$TET_ROOT/rpc/ts/rpc.sec.2
RPCSEC2_TEMP_DIRPATH	Directory for temporary files	/dcetest/dcelocal/tmp
RPCSEC2_LOG_DIRPATH	Directory for log files	/dcetest/dcelocal/status
RPCSEC2_UDP_PCT	Percentage of udp clients	50
RPCSEC2_CLI_TO_RUN	Number of clients	9
PRIN_PASSWD	Principal password	"-dce-"

12.2.6.8 Running rpc.sec.2

To run **rpc.sec.2** in the example cell described earlier in this chapter, you would do the following:

1. Edit the configuration file:

   /dcetest/dcelocal/tet/rpc/ts/rpc.runtime.1/dcerpcrun.data

   and make the appropriate changes to the configuration variables.

2. Define and export the **TET_ROOT** environment variable:

   export TET_ROOT=/dcetest/dcelocal/test/tet

3. Source the TET version of the system test profile:

   . $TET_ROOT/../test/systest/profile.dcest.tet

4. **dce_login** as **cell_admin**:

   dce_login cell_admin *cell_admin_password*

5. Invoke the test via TET:

   cd /dcetest/dcelocal/tet
   $TET_ROOT/bin/tcc -e -j *journal_filename* **rpc dcerpsec**

12.2.6.9 Generating Test Reports

If you are not running the test in boundary mode, then after all the clients have exited, you can generate a report of the results of the test by executing the following commands

in a Bourne or Korn shell:

```
$ cd <logdir>

$ for i in 'ls cli_logpid.*'
> do
> grep -v READY $i | awk -f bindir/rpc.sec.2_gen_summ.awk >> runpid.summ
> done

$ awk -f <bindir>/rpc.sec.2_gen_rep.awk run<pid>.summ > run<pid>.results
```

12.2.6.10 Implementation Notes

The size of the array passed to the server by the client determines how long the **rpcsec2_cnf_str( )** call will take. The server divides the array size by three, then waits in the **rpcsec2_cnf_str( )** call for the resulting number of seconds before processing the array and returning. The number of array elements should be six or more if a goal of the test is to force the server runtime to buffer and unbuffer call requests.

The observed maximum number of concurrent calls for an **rpc.sec.2** server running with a single execution thread (specified in the configuration file by setting **RPCSEC2_SRV_MAX_EXEC** to 1) is nine. If testing is desired with more than nine threads, the number of execution threads in the server must be increased.

Note that if the test is run with the observed maximum of test clients and a server with one execution thread (the default), then the connection-oriented protocol clients will report large numbers of **server_too_busy** errors. This is caused by the clients' finding the server call request buffer full because a slot that would normally have been available to accept a client request has been taken by a housekeeping call regularly made by the RPC daemon to determine whether the server is still active. The client then goes into a tight loop, continuing to call and continuing to receive the error until a slot does open up. To avoid this scenario, either run the server with more execution threads, or add a delay to the client call loop when **rpc_s_server_too_busy** is detected (if your **sleep( )** is not wrapped and hence not threadsafe, use **pthread_cond_timed_wait( )** or **pthread_delay_np( )** instead). Datagram clients will receive a few **comm_failure** errors for this same reason, but these will be far fewer than the **server_too_busy** errors received by connection-oriented clients, due to the different retry semantics of the datagram runtime in case of call failure.

It has been observed that if the test client, for some reason, loses its credentials, it will begin to consume swap space at the rate of about 1 megabyte per hour. However, the case of a client losing its credentials is quite rare (in the instance in which this phenomenon was observed, the clients had lost their credentials because the ticket lifetime was changed *after* the test had been started).

Note that if the clients are running in debug mode at the very end of the test, the report generation scripts will not work correctly on the raw output.

If you wish to run the **rpc.sec.2** test with a large number of clients, you will will to start the clients in groups. If you attempt to start too many clients concurrently, all making

calls to the same server, some number of the clients will receive the error status **rpc_s_connect_rejected**, and the **rpc.sec.2_runtest.sh** script will abort the test run. This is caused by too many client call requests arriving at the server machine's socket at the same time, filling up the listen backlog buffer associated with the socket faster than the RPC runtime can dequeue the requests and buffer them in the call request buffer; calls arriving when the listen backlog buffer is full are rejected. The number of clients that can be started at one time will vary from platform to platform; the larger the listen backlog size and the faster the machine, the greater the number of clients that can be started at once. For DCE 1.0.2, the maximum number of clients that could be successfully started at the same time on the AIX platform was between 10 and 20.

12.2.6.11 Ticket Expiration

It is possible in some circumstances for a test client's network credentials (i.e., ticket) to expire, in spite of the fact that a thread is spawned to maintain the ticket. If a client's ticket does expire, the test as shipped will almost certainly fail soon afterwards.

The client ticket's expiration is generally caused by starvation of the ticket-maintainance thread, and is more likely to occur in clients that access the **rpc.sec.2** server using the connection-oriented protocol —especially if the ticket lifetime is short (i.e., in the neighborhood of five minutes or less).

The chain of events that leads to the starvation generally begins when any unrecoverable error occurs in the test server runtime. From then on, all remote calls will return errors to the callers. (The test clients do not perform error handling for remote calls; instead, they are designed to simply log errors and continue test activity.) Further, with the connection-oriented protocol, any error in the server runtime causes an immediate return from the remote call to the client. Thus when all this happens, the client thread making the remote call goes into a tight loop, re-attempting immediately over and over again to successfully complete the remote call. If the client's ticket expiration time is short, the time taken up by the call thread's looping can deprive the ticket maintainance thread of sufficient CPU cycles to refresh the client's ticket before it expires. Then, once the ticket has expired, the remote call thread begins generating messages that describe the last time the ticket was refreshed, along with other (normally pertinent) information. This has the result of making the call thread take up even more time, and as a result the ticket maintainance thread is never allowed to refresh the ticket.

This failure scenario generally does not occur for test clients using the connectionless protocol; its semantics prevent the sequence of events that leads to the tight looping described above.

12.2.6.12 Runtime Errors that Should be Handled

As noted above, the **rpc.sec.2** clients do not currently perform any error handling of the communication status value returned from a remote call. The lack of such error handling is responsible for the spurious test failure scenario described above, and this scenario can

probably be avoided if you add code to handle the three following errors:

- **rpc_s_server_too_busy**

  (Returned only by TCP clients.) The server does not have a thread available to service the client request, nor does it have space in any call request buffer to queue the request. When a test client receives this error, it will go into a tight loop as described in the previous section, making RPCs and continuing to receive this same status, until sufficient resources are freed at the server to permit the call to be serviced or queued. While testing did not prove this looping to have a significant impact on the overall success rate of the TCP clients, it is wasteful of CPU cycles. One way to avoid the tight looping would be to have the TCP clients wait for a few seconds if they receive this status before doing anything. Another approach would be to allocate more server threads to begin with, and thus avoid the situation altogether.

- **rpc_s_connection_closed**

  A protocol error has occurred in the connection to the server. This means (with a connection-oriented protocol) that the binding to the server has become permanently useless, and the thread in the server runtime that listens for connection-oriented protocol requests is probably unavailable, so that no connection-oriented protocol calls will succeed. The only remedy for this condition is for the server to re-export its binding handles.

- **rpc_s_auth_tkt_expired**

  The client's network credentials (i.e., ticket) have expired. The client thread receiving this error can recover from the situation by notifying the ticket maintainance thread that it should now refresh the ticket.

12.2.7 dcerpper

The **dcerpper** DCE System Test is based on the RPC **perf** functional tests. It utilizes the **perf** functional test server and client programs to perform the following tests:

- Null call
- Null call, idempotent
- Variable length input arg
- Variable length input arg, idempotent
- Variable length output arg
- Variable length output arg, idempotent
- Broadcast
- Maybe
- Broadcast/maybe
- Floating point

- Unregistered interface
- Forwarding
- Exception
- Slow call
- Slow call, idempotent

The **perf_server** is run on the machine on which **dcerpper** is being executed, and **perf_client** is started on the specified client machines. The client machines are started simultaneously in order to put stress on the server machine.

Test Scripts: **$TET_ROOT/tet/rpc/ts/dcerpper/dcerpper**

Test Programs: **$TET_ROOT/tet/rpc/ts/dcerpper/perf_server**

$TET_ROOT/tet/rpc/ts/dcerpper/perf_client

dcerpper is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUNNING_TIME=0.25 rpc dcerpper**

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **rpc** directory) for the test results journal file.

-vRUNNING_TIME=0.25
Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.

rpc Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.

dcerpper Specifies the name of the test (TET scenario) to be run.

12.3 DCE Host Daemon (dced)

The **dced** system tests exercise the functionality provided by the DCE Host Daemon (**dced**):

- Endpoint resolution
- Remote Key Table Management

- Remote Host Data Management
- Remote Server Configuration and Execution
- ACL operations on all the above functions

The test suite consists of three reliability tests which use a **run** control script as a test driver in the same way as the other DCE system tests executed under TET. The tests must be installed on each machine on which they will be run, using **dcetest_config**. Setup requirements are the same as for the other **run** script-based DCE system tests.

All sub-components and other executables for the tests are installed at:

$TET_ROOT/../test/tet/system/dced/ts/rel/

Test Scripts: **$TET_ROOT/system/dced/ts/rel/dcdrel001/dcdrel001**
$TET_ROOT/system/dced/ts/rel/dcdrel002/dcdrel002
$TET_ROOT/system/dced/ts/rel/dcdrel003/dcdrel003

The tests are invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/test/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/bin:$PATH

tcc -e -j *journal_path* **system/dced dcdrel001**

tcc -e -j *journal_path* **-vNMIN=15 system/dced dcdrel002**

tcc -e -j *journal_path* **-vNMIN=15 system/dced dcdrel003**

where:

platform — Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e — Specifies to run the test.

-j *journal_path* — Specifies a file pathname (relative to the **system/dced** directory) for the test results journal file.

-vNMIN=15 — Sets the **NMIN** environment variable, which specifies the number of minutes the test should run.

system/dced — Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory (located under **system**) of the test to be run.

dcedrel00*n* — Specifies the name of the test (TET scenario: **dcdrel001**, **dcdrel002**, or **dcdrel003**) to be run.

The tests can also be invoked through the **run.dced** script as follows (see Chapter 11 for details on using the ‘‘**run**’’ scripts):

run.dced *{***-l** *loops* | **-t** *hours } [other_options] testname*

where:

-l *loops* Specifies the number of loops or passes to run.

-t *hours* Specifies the time in hours to run. A decimal point is accepted; e.g. ‘‘.5’’ is interpreted as 30 minutes.

testname Specifies the name of the test; see below.

The following tests can be run by specifying their name as *testname* in the command line:

dcdrel001 Exercises some of the endpoint operations provided by **dced**. Two test servers and a test client are started on each machine included in the test. The test servers register themselves via CDS and are then contacted by the test clients on all machines involved in the test. For each series of client-server interactions, the client requests the server to register its interface and endpoints with a newly-generated list of object UUIDs, and then contacts the server using each of the newly-registered object UUIDs and requests that the endpoint be unregistered.

As many machines as desired can be included in the test run, via the command line options **-m** or **-M**, and the number of endpoints registered can be modified by recompiling the tests with a different value for the constant **UUID_VEC_COUNT**, which is defined in

dce-root-dir/**src/test/systest/dced/ts/rel/dcdrel001/dcdrel001_client.c**

The test starts two servers, both of which register endpoints using the **rpc_ep_register( )** routine. This test could be readily enhanced by modifying one of the two servers’ manager routines to use the **dced** API functions for registering and unregistering endpoints instead.

dcdrel002 Exercises some of the server configuration and execution operations provided by **dced**. Four variations of a test server are configured, using the **dcecp server create** command. Then several sequences of starting, stopping, and restarting the servers are executed.

At present the test creates the test servers on each of the machines identified on the **run.dced** command line, and then executes **dcecp** operations on those servers from the machine that the servers are executing on. The test could enhanced by having it execute the **dcecp** commands on each machine involved in the test to control servers on the other machines.

dcdrel003 Exercises some of the keytab, hostdata, and ACL **dced** service operations. A **dcecp** script is executed on each of the macines specified on the command line, to test some of the hostdata operations. Following this, separate **dcecp** script is executed on each of the machines to test some of the keytab operations. Finally, another **dcecp** script is executed on each of the machines to test some of the ACL operations.

Note: When **dcdrel003_acl.tcl** is running, no other **dced**-related testing should be taking place. This test subcomponent changes ACLs to disallow **dced** operations, and will thus cause any other active **dced** tests to fail.

all Causes all of the tests to be run in turn, with the specified command line options.

This test also uses the following standard **run** options:

- **-e** *number*
- **-E** *number*
- **-m** *name(s)*
- **-M**

For explanations of these options, see ‘‘Command Line Options Common to Some or All of the ‘‘Run’’ Scripts’’, in Chapter 11.

12.4 Security

All of the security systest directory scripts are run as ‘‘root’’ UID, with the systest environment file **profile.dcest.tet** sourced. All tests are run from the test ‘‘driver’’ level and use the **rgy_edit -update** control program interface for all registry operations. The drivers all use the

/dcetest/dcelocal/tet/security/ts/sec.functions

file for determining the security-related operations (**rgy_edit** functions) to be tested, as listed below under each test driver name.

Note that all of these details are automatically taken care of when you run the tests through **dcetest_config**, as is recommended; if you run the tests manually, you must source **profile.dcest.tet** yourself.

12.4.1 secrep

This test consists of 9 test cases to test the **change -master** and **become -master** functionality. The test cases are as follows:

Test case name	*Description*
tc_bm	Tests **become master**.
tc_bm_restart	Become master and restart new master.
tc_cm_basic	Basic change master test.
tc_cm_q_update	Change master with updates in progress.
tc_cm_login_query	Change master while logins and queries are in progress.
tc_cm_c_update	Change master while master is being updated.
tc_cm_restart	Change master and restart new master.

tc_cm_slvinit Change master with **initrep** in progress.

tc_cm_slvdel Change master with **delrep** in progress.

The global cleanup function, **tc_secrep_cleanup**, will delete *all* replicas in the cell. The function is written as if it were a test case, and is the last TET scenario to be executed.

Note that at the end of the test (after **tc_secrep_cleanup** has been run), there is only master; however, this may not be the original master. This is because the test performs a change/become master.

To see both the result of each test as well as the start and end time of each test do the following:

```
grep TEST  <journal file name> | awk -F"|" '{print $NF}'
```

The test is invoked as follows:

```
cd dce-root-dir/dce/install/platform/opt/dcetest/dce1.2.2/tet
setenv TET_ROOT `pwd`
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j journal_file security all_secrep
```

where:

platform Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **security** directory) for the test results journal file.

security Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

all_secrep Specifies the name of the test (TET scenario) to be run.

12.4.2 dceseacl

dceseacl tests security registry ACLs and authorization operation, and can serve as a registry load or stress test. It does this by creating a number of accounts, principals, groups, and organizations; modifying permissions; and verifying appropriate ACL management operation.

Test Script: **$TET_ROOT/security/ts/dceseacl/dceseacl**

Data Script: **$TET_ROOT/security/ts/dceseacl/dceseacl.data**

dceseacl is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vNMIN=15 security dceseacl**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **security** directory) for the test results journal file.
-vNMIN=15	Sets the **NMIN** environment variable, which specifies the number of minutes the test should run.
security	Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.
dceseacl	Specifies the name of the test (TET scenario) to be run.

12.4.3 eraobj001

eraobj001 is a variant of the **dcesseacl** test. It is implemented as a wrapper around the latter test. When invoked, it sets the value of the **USE_ERA** environment variable to ''yes'' and then invokes **dceseacl**. **dceseacl** is then run with the extended attribute functionality (i.e., ACLs on the registry schema object, and extended registry attribute instances attached to principals, groups, and organizations).

Test Script: **$TET_ROOT/security/ts/eraobj001/eraobj001**

Data Script: **$TET_ROOT/security/ts/era.data**

eraobj001 is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=0.25 security eraobj001**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.

-j *journal_path*	Specifies a file pathname (relative to the **security** directory) for the test results journal file.
-vRUN_TIME=0.25	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
eraobj001	Specifies the name of the test (TET scenario) to be run.

12.4.4 dceseact

dceseact adds, deletes, and changes information about principals, groups, organizations, and accounts to test the security registry.

Note: This test must be run by a user who has write access to the registry database.

Test Script: **$TET_ROOT/security/ts/dceseact/dceseact**

Data Script: **$TET_ROOT/security/ts/dceseact/dceseact.data**

dceseact is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=.25 security dceseact**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **security** directory) for the test results journal file.
-vRUN_TIME=0.25	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
security	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
dceseact	Specifies the name of the test (TET scenario) to be run.

12.4.5 dcesepol

dcesepol tests security policy options through the use of the registry editor (**rgy_edit**) and repeated login attempts.

Note: In order to test account expiration, this test must be running at midnight (i.e., during the interval 11:59 P.M. and 12:01 A.M.). The test must be run by a user who has write access to the registry database.

dcesepol creates three organizations to test password expiration date, password life span, and account life span, respectively. Principals and accounts are created for the organizations in order to perform policy verification via authenticated login. The local registry password override login function is tested by disabling the first account's first machine login.

Test Script: **$TET_ROOT/security/ts/dcesepol/dcesepol**

Data Script: **$TET_ROOT/security/ts/dcesepol/dcesepol.data**

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-i** *intermediate_results_path* **security dcesepol**

where:

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **security** directory) for the test results journal file.

-i *intermediate_results_path*
Specifies a file pathname (relative to the **security** directory) for the intermediate test results file.

security Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.

dcesepol Specifies the name of the test (TET scenario) to be run.

12.4.6 dcesestr

dcesestr exerts stress on the registry server (**secd**) by attempting to access information from the server through multiple clients. It adds principals, groups, and organizations to the registry and then invokes multiple clients (**resestr**) which in turn perform valid and invalid logins.

Note: This test must be run by a user who has write access to the registry

database.

Test Script: **$TET_ROOT/security/ts/dcesestr/dcesestr**

Data Script: **$TET_ROOT/security/ts/dcesestr/dcesestr.data**

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-i** *intermediate_results_path* **security dcesestr**

where:

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **security** directory) for the test results journal file.

-i *intermediate_results_path*
Specifies a file pathname (relative to the **security** directory) for the intermediate test results file.

security Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.

dcesestr Specifies the name of the test (TET scenario) to be run.

The **SEC_MACHINES** environment variable, whose value is set in the data script mentioned above, can be used for stress testing by giving it a long list of machines to act as clients. Each of these clients will attempt logins at the same time.

For additional stress testing, you can specify a large number of users for **SEC_NUM_USERS** in the test data script. This will cause a large number of accounts to be added to the registry, each of which will be used by the clients. This can be used to force overflows of any caches that may be used in **secd** or **sec_clientd**.

12.4.7 erarel001

erarel001 is a variant of the **dcesestr** test. It is implemented as a wrapper around the latter test. When invoked, it sets the value of the **USE_ERA** environment variable to ''yes'' and then then invokes **dcesestr**. **dcesestr** is then run with extended registry attributes functionality, manipulating extended attributes on principals, groups, and organizations during logins.

Test Script: **$TET_ROOT/security/ts/erarel001/erarel001**

Data Script: **$TET_ROOT/security/ts/era.data**

erarel001 is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUN_TIME=0.25 security erarel001**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **security** directory) for the test results journal file.
-vRUN_TIME=0.25	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
security	Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.
erarel001	Specifies the name of the test (TET scenario) to be run.

12.4.8 dlgcfg001

dlgcfg001 is a basic delegation configuration test.

Test Script: **$TET_ROOT/security/ts/dlgcfg001/dlgcfg001**

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-vRUNNING_TIME=0.25 security dlgcfg001**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **security** directory) for the test results journal file.
-vRUNNING_TIME=0.25	Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.

security Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

dlgcfg001 Specifies the name of the test (TET scenario) to be run.

12.4.9 Security Registry System Test dcesergy

The purpose of this test is to stress the security registry servers by performing a number of DCE logins and logouts while administrators are at the same time performing registry updates and queries. Five Security registry administrators on each host in the local cell create new organizations, groups and accounts, setting registry policy on the new accounts and creating password override local registry login policy, while verifying these policies and performing logins on each machine. Ten principals on each host machine concurrently perform logins while the registry administration is in progress. The test also provides override file support for local registry account information.

The test was derived from the RPC API functional tests, and it conforms to the basic RPC client-server model. Execution of the test operations is provided by the server; the client learns the result of an operation via RPC status or exception. Success is also indicated by a text message which is displayed for most otherwise silent operations.

Test Script: **$TET_ROOT/security/ts/sec.rgy.7/dcesergy**

Data Script: **$TET_ROOT/security/ts/sec.rgy.7/dcesergy.data**

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-i** *intermediate_results_path* **security dcesergy**

where:

-e Specifies to run the test.

-j *journal_path* Specifies a file pathname (relative to the **security** directory) for the test results journal file.

-i *intermediate_results_path*
Specifies a file pathname (relative to the **security** directory) for the intermediate test results file.

security Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.

dcesergy Specifies the name of the test (TET scenario) to be run.

You can increase the stress on the security server by running multiple copies of **dcesergy** on different machines in the cell, but you must do so manually at each machine, because the DCE 1.2.2 version of **dcesergy** does *not* use the **-m** *machine*

option.

12.4.9.1 Logic Flow of the Security Registry System Test

The server and client execution sequence can be displayed by building the test with the **ST_DEBUG** symbol defined. The sequence is:

Server: Initialize Pthread synchronization data

Export the server binding to the RPC runtime, endpoint mapper and the CDS data base

Start the credential refresh thread

Start the RPC listen thread

Client: Login as client principal

Import the server binding handle from CDS

Call the test operation (which was specified on the command line; for example, **-u** *principal* will add the principal to the registry database)

Wait for RPC status/exceptions or server return message

The server creates and uses a local key file **/usr/tmp/v5srvtab** to be used by the RPC runtime in decrypting incoming tickets from authenticated clients (for authenticated RPC, the **-a** option).

12.4.9.2 Test Setup Script

dcesergy adds test servers to the registry before the test server processes are started on the respective test machines. This is done via a test setup script.

After setup is completed, the script will execute internal loops for the specified number of loops or hours. It will execute the **login-logout** test for a specified number of call threads, followed by the add principal account operation for the specified number principals beginning with ‘‘basename0’’, finally ending the loop with the deletion of the previously created principals.

Note that this script does not perform other operations (password override functionality or get password entry); it is intended to be used only as an extended duration test driver for continuous operation testing.

The script also invokes the client program, which logs in, binds to the CDS-named server, and executes the login/logout operation on the server 10 times concurrently (i.e., with 10 client call threads).

12.4.9.3 Running the Security Registry System Test Components by Hand

The **dcesergy** system test can be manually invoked as follows:

1. Add the test servers to the registry by running the setup script:

   secrgy_setup -n *server_name* **-p** *password*

2. Run the server:

   secrgy_server -n *server_name* **-p** *server_pwd* \
   *[***-c** *cell_name] [***-a***] [***-d***] [***-i** *prot_seq]*

3. Run the client program:

   secrgy_client -n *client_name* **-p** *client_pwd* **-s** *server_name* \
   *[***-w***] [***-o** **-r** *integer] [***-x***] [***-c** *cell_name]* \
   *[***-u** *principal] [***-k** *principal] [***-a***] [***-f** *filename]* \
   *[***-t** *integer] [***-l** *integer] [***-d***] [***-i** *prot_seq]* \
   *[***-P***] [***-j** *aggregate_nr]*

Where:

-n *client_name* The *client_name* (client principal name) specifies the principal identity under which the client process is to execute.

-p *client_pwd* Specifies the client principal password.

-s *server_name* (*Client program only*) Specifies the server principal name (in the NSI namespace) which the client will import and bind to.

-w Specifies the get password entry operation for the client principal.

-o Specifies that the **sec_login_validate_and_certify( )** call be used by the client during login operations. This is a privileged operation, so the server must be running under the root UID in order to be able to execute this option.

-r *integer* Specifies the number of concurrent client calls for login operations.

-x Client flag to specify server clean-up and exit operations.

-c *cell_name* Specifies the cell within which client/server NSI import/export and registry operations should occur.

-u *principal* Specifies that the principal name and account be added to the registry. The password for all added principals is the same as the server's.

-k *principal* Specifies that the principal name and account be deleted from the registry.

-a Specifies authenticated RPC.

-f *filename* Specifies the file to transfer from the client host machine to the server host machine as the **password_override** file. Note that this requires that the server be running under root UID in order to have write permission to the *dcelocal*/**etc** directory.

-t *integer* Specifies how long (in minutes) each single client call should continue execution, repeatedly looping through the login and logout operations.

-l *integer* Specifies how long (in seconds) to wait in each client call between login and logout operations. Use default or specify zero for maximum test loading.

-d Specifies debug mode.

-i *prot_seq* Specifies the RPC protocol sequence to be used; either ‘‘**ncacn_ip_tcp**’’ or ‘‘**ncadg_ip_udp**’’.

-P Specifies that the client should perform a ping of the server (by calling **rpc_mgmt_is_server_listening()**).

-j *aggregate_nr* Specifies the number of aggregate accounts to add or delete from the registry.

Both the client and the server program will detect conflicting parameters and output an appropriate error message to the invoker.

12.4.9.4 Usage Examples

Following is a sequence of example commands showing how to run the **sec.rgy.7** test by hand.

First, the setup script is run:

```
secrgy_setup -n foo -p bar
```

—This adds the server name and password to the registry.

Next, the server is started:

```
secrgy_server -n foo -p bar
```

—This invokes the server program, which adds the server name to the namespace and starts a thread to refresh the server’s credentials at the midpoint of their default registry lifetime.

```
secrgy_client -n foo -p bar -s foo -r 10
```

Invokes the client program, which logs in as the principal **foo** using the password **bar**, binds to the CDS-named server **foo**, and executes the login-logout operation on the server 10 times concurrently (i.e., with 10 client call threads).

12.5 CDS

The following sections describe the DCE CDS system tests run under TET.

12.5.1 dcecdsrep

The CDS replication system test consists of the following main components:

Test Script: **$TET_ROOT/cds/ts/dcecdsrep/dcecdsrep**

Data Script: **$TET_ROOT/cds/ts/dcecdsrep/dcecdsrep.data**

Function Script: **$TET_ROOT/cds/ts/dcecdsrep/dcecdsrep.functions**

When invoked, the test does the following:

1. Creates a replica clearinghouse and skulks the root.
2. Creates a directory in the master clearinghouse and skulks the root.
3. Replicates the directory in the replica clearinghouse and skulks the root.
4. Disables the master clearinghouse to ensure that addition of an object to the replica clearinghouse is not possible.
5. Enables the master clearinghouse and adds the object to the directory, adds its attribute, and skulks the directory.
6. Tries to get the attribute of the object; this should succeed.
7. Adds a new attribute (Note: currently does NOT use the same attribute) to the object, and does NOT skulk the directory.
8. Disables the replica clearinghouse and tries to skulk; this should fail to propagate the attribute.
9. Enables the replica clearinghouse.
10. Tries to get the attribute; this should fail.
11. Skulks the directory.
12. Tries to get the attributes again; this should succeed.
13. Cleans up.

All test functions (except for **cleanUp** and related functions) will trap **SIGINT**, **SIGKILL**, **SIGTERM** and **SIGQUIT**. **cleanUp** ignores all of these except for **SIGQUIT**, and the functions called by **cleanUp** ignores all of them.

The test is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH

tcc -e -j *journal_path* **-vRUNNING_TIME=0.25 cds dcecdsrep**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **cds** directory) for the test results journal file.
-vRUNNING_TIME=0.25	Sets the **RUNNING_TIME** environment variable, which specifies the number of hours the test should run.
cds	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
dcecdsrep	Specifies the name of the test (TET scenario) to be run.

12.5.2 CDS Server System Test

cdsserv performs access of local and remote cell (DNS naming) objects, using ten clients per cell host. The test sets **cdscp** confidence to ‘‘high’’ and gathers statistics on time and number of server read accesses.

Test Script: **$TET_ROOT/cds/ts/cds.server.4/cdsserv.ksh**

Data Script: **$TET_ROOT/cds/ts/cds.server.4/cdsserv.cfg**

The following environment variables are set in **cdsserv.cfg**:

- **CELLS**

  The name of the cells in the form ‘‘**/.../cell1 /.../cell2**’’, and so on. Default is ‘‘**.:**’’ (i.e., the current cell).

- **PRINCS**

  The names of the principals used to login to each cell specified in **CELLS**. The default is **cell_admin**.

  Specifying additional principals starts additional, simultaneous processes to access CDS, so this is a good way to increase the load on CDS.

- **PWS**

  A list of passwords for the list of principal names. The list of passwords must match the order of the corresponding principal name list.

- **CONFIDENCE**

  The CDS clerk confidence level (low, medium, or high).

cdsserv is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-i** *intermediate_results_path* **cds cdsserv.ksh**

where:

-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **cds** directory) for the test results journal file.
-i *intermediate_results_path*	Specifies a file pathname (relative to the **cds** directory) for the intermediate test results file.
cds	Specifies the ''test suite'' name, equivalent to the component subdirectory (located under **system**) of the test to be run.
cdsserv.ksh	Specifies the name of the TET test scenario to be run.

12.5.2.1 Logic Flow of the cdsserv System Test

The test consists of three nested control loops:

- The outermost loop is controlled by the number of cells in the cell list.
- The next inner loop is controlled by the number of principal logins.
- The innermost loop is controlled by the number of passes (loops) or the number of hours of execution specified on the command line. This loop is entirely contained in a separate process.

The test executes for all cells in the cell list and for each principal login. This establishes the authenticated login context for all subsequent **cdscp** operations. The CDS clerk, however, is invoked by the CDS advertiser on a UNIX ID basis, not by DCE authenticated login context. Therefore, in order to increase the number of CDS clerks which would apply localized stress to the **cdsd** server, the test should be executed using differing multiple UIDs.

The innermost loop performs two **cdscp set confidence**, and **show directory** operations, followed by a single **show clearinghouse** operation. The output of the **cdscp show clearinghouse** command is parsed to gather individual CDS server statistics on ''read'' access count and response timing.

The **cdscp** operations are monitored and success failure results compiled and sent to the test process standard output and TET journal file.

12.5.3 CDS ACL Manager System Test

This test exercises the CDS ACL manager via client access requests to local and foreign cells. If clearinghouse replicas are available, they are tested. Five administrators on each host in the specified cell(s) sequentially verify valid and invalid ACL entry type permissions and management on replicas, soft links, objects, and directories.

Test Script: **$TET_ROOT/cds/ts/dcecdsacl6/dcecdsacl6**

Data Script: **$TET_ROOT/cds/ts/dcecdsacl6/dcecdsacl6.data**

dcecdsacl6 is invoked as follows:

```
cd dce-root-dir/dce/install/platform/opt/dcetest/dce1.2.2/tet
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -vRUN_TIME=0.25 cds dcecdsacl6
```

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **cds** directory) for the test results journal file.
-vRUN_TIME=0.25	Sets the **RUN_TIME** environment variable, which specifies the number of hours the test should run.
cds	Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory of the test to be run.
dcecdsacl6	Specifies the name of the test (TET scenario) to be run.

12.5.4 dcecdsacl6 Initialization

During initialization the necessary user and administrative groups are first added to the registry, then the administrative and user principals are added to those groups, and finally the associated principal accounts are added. CDS ACLs for the cell and clearinghouses are modified to include the **st_admin** group initial permissions as follows:

cell: **rwcidta** permissions

master clearinghouse: **rwdtc** permissions

The ACL of each clearinghouse server is modified to include the **st_admin** group initial permissions (**rwcidta**).

12.5.5 Logic Flow of dcecdsacl6 Test

When invoked, **dcecdsacl6** executes a series of three nested control loops:

- The outermost loop is controlled by the number of cells.
- The next inner loop is controlled by the number of administrative principals.
- The innermost loop is controlled by the number of clearinghouses.

The principal loop sequence is executed for each administrative principal passed into the test driver on the command line and for every user principal included in the clearinghouse operations files.

In each complete pass of the test, the following is done:

- some miscellaneous initialization;
- **cdscp show directory** and **set confidence** operations;
- the specified ACL management-related operations.

The ACL management operations are determined by reading the clearinghouse specific operation files created in advance and parsing output based on principal name. The operation sequence and expected result (pass or fail) is parsed in the order it appears in the file.

The state of the tested cell at the end of the clearinghouse operation sequence for each complete test pass using default test setup parameters will be the same as the cell's state at the beginning of the pass, so that the test can continue execution indefinitely.

12.5.6 Hierarchical Cell Tests

These are tests of the DCE 1.2.1 hierarchical cell functionality. **hclcfg001** tests intercell authentication with a list of cells using **rgy_edit**. **hclrel001** performs intercell testing to a specified list of cells.

Note: Before attempting to run these tests, you must insert entries for them in the appropriate TET scenario file. You can use either of two scenario files to run the tests, i.e.

$TET_ROOT/system/directory/cds/tet_scen

(the CDS system test scenario file), or

$TET_ROOT/system/tet_scen

(the master system test scenario file). The entries you must insert are as follows:

```
dcecdsacl6
        "Starting dcecdsacl6 Test Suite"
```

```
        /ts/dcecdsacl6/dcecdsacl6
        "Completed dcecdsacl6 Test Suite"

-----insert the following entries:-----
hclcfg001
        "Starting hclcfg001 Test Suite"
        /ts/hclcfg001/hclcfg001
        "Completed hclcfg001 Test Suite"

hclrel001
        "Starting hclrel001 Test Suite"
        /ts/hclrel001/hclrel001
        "Completed hclrel001 Test Suite"
-----...end of inserted material-----

cdsserv.ksh
        "Starting cdsserv.ksh Test Suite"
        /ts/cds.server.4/cdsserv.ksh
        "Completed cdsserv.ksh Test Suite"
```

Test Scripts: **$TET_ROOT/system/directory/cds/ts/hclrel001/hclrel001**
$TET_ROOT/system/directory/cds/ts/hclcfg001/hclcfg001

Data Scripts: **$TET_ROOT/system/directory/cds/ts/hclrel001/hclrel001.data**
$TET_ROOT/system/directory/cds/ts/hclcfg001/hclcfg001.data

The tests are invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.1/test/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/bin:$PATH

tcc -e -j *journal_path* **-vNMIN=15** *test_suite* **hclcfg001**

tcc -e -j *journal_path* **-vRUN_TIME=.50** *test_suite* **hclrel001**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **system/** directory) for the test results journal file.
-vNMIN=15	Sets the **NMIN** environment variable, which specifies the number of minutes the **hclcfg001** test should run.
-vRUN_TIME=0.50	Sets the **RUN_TIME** environment variable, which specifies the number of hours the **hclrel001** test should

run.

test_suite — Specifies the ‘‘test suite’’ name, equivalent to the component subdirectory (located under **system**) of the test to be run.

The value of this parameter will depend on which TET scenario file you added the test entries to (see the Note at the beginning of this section). If you added the entries to the

$TET_ROOT/system/tet_scen

file, the ‘‘test suite’’ name will be simply **system**. If you added the entries to the

$TET_ROOT/system/directory/cds/tet_scen

file, the ‘‘test suite’’ name will be **system/directory/cds**.

hclcfg001 — Specifies the name of the test (TET scenario) to be run. **hclcfg001** establishes intercell authentication with a list of cells (specified in the environment variable **CELLS**, set in **hclcfg001.data**) using **rgy_edit**.

hclrel001 — Specifies the name of the test (TET scenario) to be run. **hclrel001** performs intercell testing to a list of cells (specified in the **CELLS** environment variable).

12.6 DCE Audit Service System Tests

The Audit system tests are located at

$TET_ROOT/../test/tet/system/audit

where **$TET_ROOT** is

dce1.2.2-root-dir/**dce/install/***platform*/**opt/dcetest/dce1.2.2/tet**

The tests are invoked as follows (see Chapter 11 for details on using the ‘‘**run**’’ scripts):

run.aud *{* **-l** *loops* | **-t** *hours } test_name*

where:

-l *loops* — *loops* specifies number of loops or passes to run.

-t *hours* — Specifies the time in hours. A decimal point is accepted; e.g. ‘‘.5’’ is interpreted as 30 minutes.

-c	Specifies that the log files from successful iterations be kept.
test_name	The name of the test to be run, which must be one of the following:
	audstr001 Audit stress test.
	audrel001 Audit reliability test.

This test also uses the following standard **run** options:

- **-e** *number*
- **-E** *number*
- **-c**
- **-h**

For explanations of these options, see ‘‘Command Line Options Common to Some or All of the ‘‘Run’’ Scripts’’, in Chapter 11.

See also the

dce-root-dir/**dce/src/test/systest/audit/README**

file.

12.7 DTS

The following sections describe the DCE DTS system tests run under TET.

12.7.1 dcetmsyn

dcetmsyn tests that synchronization occurs when the **dtscp synchronize** command is executed.

The state is tested to see if a synchronization is occurring, and if so, the test will enter a loop to wait for the state to be ‘‘on’’, which will occur when the synchronization is completed. The **dtscp show last synchronization** command is then executed and the output saved. The current time is saved. A **synchronize** command is then executed, and a loop is entered in order to wait for the synchronization to complete. The **dtscp show last synchronization** command is again executed and compared to the previous saved output to verify that a synchronization did occur after the **dtscp synchronize** command was entered.

Before running the test you should do a

dtscp set synch hold down 24:00:00

—this will set the default time to synchronize on the machine to every 24 hours. If you do not do this, failures may occur when the test attempts to do a synchronization at the same time that the machine is trying to do one of its own. This test can be run on DTS local and global servers and clerks.

Test Script: **$TET_ROOT/time/ts/dcetmsyn/dcetmsyn**

Note that there is no data script for this test.

dcetmsyn is invoked as follows:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**
setenv TET_ROOT 'pwd'
setenv PATH $TET_ROOT/../test/tet/bin:$PATH
tcc -e -j *journal_path* **-i intermediate_results_path time dcetmsyn**

where:

platform	Is the name of the platform on which you are testing DCE (for example, *platform* is **rios** for the IBM RISC System/6000 running AIX).
-e	Specifies to run the test.
-j *journal_path*	Specifies a file pathname (relative to the **time** directory) for the test results journal file.
-i *intermediate_results_path*	Specifies a file pathname (relative to the **time** directory) for the intermediate test results file.
time	Specifies the ''test suite'' name, equivalent to the component subdirectory of the test to be run.
dcetmsyn	Specifies the name of the test (TET scenario) to be run.

12.8 Internationalization System Tests

The files **I8NSAN001** and **I8NSAN002**, found in the

$TET_ROOT/../test/tet/system/I18N/ts

directory (where

dce-root-dir/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/tet**

is the value of **$TET_ROOT**), are the Internationalization system tests; they test Internationalization support in the CDS and Security components. The tests are written as **dcecp** scripts, and are run under TET, but they are not run under a **run** script.

Following is a list of the files and directories that make up the test; they are located in the

dce-root-dir/**dce/src/test/systest/I18N**

directory and installed in the

*dce-root-dir***dce/install/**/*platform*/dcetest/dce1.2.2/test/tet/system/I18N

directory:

lib	Directory containing files that define common procedures called by the test main procedure.
tet_code	File containing error definitions known by TET.
tet_scen	TET scenario file.
tetexec.cfg	File containing test configuration variables and assignments.
ts	Directory where the main test scripts reside.

12.8.1 Prerequisite Setup

Before attempting to run the internationalization tests, you must do the following:

1. Select a locale for testing and ready the message catalogs corresponding to that locale.
2. Make sure that the host machine's operating system has I18N support for the desired locale.
3. Install and configure the DCE cell.
4. Install and configure the system tests using **dcetest_config**. For information on **dcetest_config**, see Chapter 11.
5. Edit the value of the variables defined in the

   tetexec.cfg

   file. For example:

   ```
   LOCALE_NAME=c-french
   MESSAGE_CAT=/u1/piglatin/%N
   LOCALE_DATAFILE=french.short
   ```

   The above settings mean that the test will use the French locale, and will look for message catalogs in the

   /u1/piglatin

   directory.
6. Create the datafile defined by the environment variable **LOCALE_DATAFILE**. This is the input file for the tests. It should contain a list of at least 20 words, arranged one word per line.

12.8.2 Running the Tests

To run the tests, do the following:

cd *dce-root-dir*/**dce/install/**/*platform*/**opt/dcetest/dce1.2.2/test/tet**
setenv TET_ROOT ‘pwd‘
setenv PATH $TET_ROOT/bin:$PATH
tcc -e -j *journal_file* **system/I18N** *test_suite_name*

where *test_suite_name* is one of the following:

- **I8NSAN001**
- **I8NSAN002**

and *journal_file* is the name of the file to which you want the test results written.

After the test has executed, the results will be found in the journal file (which you specify). If a journal file is not specified, TET will create the file under the

results

directory.

12.9 DCE Serviceability System Tests

The DCE 1.2.2 Serviceability system tests are found in the

$TET_ROOT/../test/tet/system/svc/ts

directory (where

dce-root-dir/**dce/install/***platform*/**opt/dcetest/dce1.2.2/tet**

is the value of **$TET_ROOT**). These are tests of different ways of configuring serviceability at DCE startup. The tests are installed by **dcetest_config**. They are invoked as follows (see Chapter 11 for details on how to use the ‘‘**run**’’ scripts:

run.svc *{***-l** *loops* | **-t** *hours } [other_options] testname*

where:

-l *loops*	Specifies the number of loops or passes to run.
-t *hours*	Specifies the time in hours to run. A decimal point is accepted; e.g. ‘‘.5’’ is interpreted as 30 minutes.
testname	Specifies the name of the test, and is one of the following:

- **svccfg001**
- **svccfg002**
- **svccfg003**
- **svccfg004**
- **svccfg005**
- **svccfg006**

The contents of

$TET_ROOT/../test/tet/system/svc/README

contains additional information about running the tests.

Chapter 13. DCE System Tests not under TET

The following sections describe the set of DCE system tests that are not run under TET.

13.1 Security Administrative Tests

The following sections describe checklists for system testing DCE Security Service administrative functions.

A checklist is a series of instructions and manually-entered commands, together with a description of the expected results of executing the commands. Checklists are used to document test functions for which no automated test exists.

The DCE Administration tests are installed with **dcetest_config**. See ‘‘Installing the DCE System Tests’’, in Chapter 11, for instructions on running **dcetest_config**.

13.1.1 Backup and Restore Registry Checklist

The purpose of **backup_restore_rgy_checklist** is to verify that the backup and restore of the master registry function properly.

13.1.1.1 Prerequisites for Performing ‘‘backup_restore_rgy_checklist’’

The checklist must be performed as root, and the machine the checklist is being executed on must have root access via **.rhosts** to all machines in the cell.

Note: You should not execute this checklist in any DCE cell which you cannot afford to corrupt as a result of performing the steps.

13.1.1.2 ‘‘backup_restore_rgy_checklist’’ Logic Flow

When performed, the steps in **backup_restore_rgy_checklist** accomplish the following series of operations:

1. Logins are attempted.
2. Registry is set to maintenance mode.
3. Master registry is backed up locally.
4. A test entry is attempted in the registry, which should fail.
5. Registry is set to service mode.
6. Logins are attempted.
7. Test entries are made to the registry.
8. Logins of test entries are attempted.
9. Registry is set to maintenance mode.
10. Test master registry is backed up locally.
11. Master registry is restored from local backup.
12. Registry is set to service mode.
13. Logins of test entries are attempted; these should fail.

13.1.1.3 ‘‘backup_restore_rgy_checklist’’ Default Values

All values are supplied by the test user, based on his or her DCE configuration.

13.1.1.4 Performing ‘‘backup_restore_rgy_checklist’’

Perform the **backup_restore_rgy_checklist** steps as follows:

cd *systest-root*/**admin/sec/tests**

Execute the steps in

backup_restore_rgy_checklist

as specified.

13.1.2 Registry Replica Checklist

The purpose of **replica_checklist** is to verify that the replication of a registry functions properly.

13.1.2.1 Prerequisites for Performing ‘‘replica_checklist’’

The checklist must be performed as root, and the machine the checklist is being executed on must have root access via **.rhosts** to all machines in the cell.

Note: You should not execute this checklist in any DCE cell which you cannot afford to corrupt as a result of performing the test steps.

13.1.2.2 ‘‘replica_checklist’’ Logic Flow

When performed, the steps in **replica_checklist** accomplish the following series of operations:

1. The initial security and CDS servers and at least two DCE clients are installed and configured.
2. The state of the master registry is changed to maintenance mode, and the master registry is backed up.
3. Disabling of write access to the master registry is verified.
4. The state of the master registry is changed to service mode.
5. Enabling of read access to the master registry is verified.
6. A slave registry is configured.
7. Listings of master and slave registries are obtained and compared to verify that propagation occurred.
8. Read access to the registry from a non-registry machine is verified.
9. Five users are added, and their presence in the slave registry is verified.
10. The master registry is disabled using the **sec_admin stop** command.
11. Login is attempted from a non-registry machine.
12. An attempt is made to add a principal on a non-registry machine.
13. The master registry is enabled by starting **secd**.
14. Two accounts are deleted.
15. The two deleted accounts are verified to no longer be present in the slave registry.

16. The master registry is disabled using the **sec_admin stop** command.
17. Read access to the registry from a non-registry machine is verified.
18. The original master registry which was backed up before beginning the test is restored.
19. The master registry is enabled by starting **secd** with the **-restore_master** option.

13.1.2.3 Default Values for ‘‘replica_checklist’’

All values are supplied by the test user, based on his or her DCE configuration.

13.1.2.4 Performing ‘‘replica_checklist’’

Perform the **replica_checklist** steps as follows:

cd *systest-root*/**admin/sec/tests**

Execute the steps in

replica_checklist

as specified.

13.2 CDS Administrative Tests and Checklists

The following sections describe automated tests and checklists for system testing DCE CDS administrative functions.

A checklist is a series of instructions and manually-entered commands, together with a description of the expected results of executing the commands. Checklists are used to document test functions for which no automated test exists.

The DCE Administration tests are installed with **dcetest_config**. See ‘‘Installing the DCE System Tests’’, in Chapter 11, for instructions on running **dcetest_config**.

13.2.1 Backup and Restore Clearinghouse Automated Test

The purpose of the **backup_restore_ch.ksh** test is to show that clearinghouses can be backed up and restored locally, i.e. that a clearinghouse can be successfully replaced with a backup version of the clearinghouse.

13.2.1.1 Prerequisites for Running ‘‘backup_restore_ch.ksh’’

The following things must be true in order to successfully run the **backup_restore_ch.ksh** system test:

- The test must be run as root, and the machine the test is being executed on must have root access via **.rhosts** to all machines in the DCE cell.
- The test must be executed on a CDS server machine.
- There can be no CDS clients running DCE during the test except for a Security server in a split server configuration.

Note: You should not execute this test on any CDS server which you cannot afford to corrupt as a result of running the test. In some instances the CDS clearinghouse can be corrupted if the test fails.

13.2.1.2 ‘‘backup_restore_ch.ksh’’ Logic Flow

When invoked, **backup_restore_ch.ksh** performs the following series of operations:

1. The master clearinghouse is backed up locally.
2. The master clearinghouse is checked to make sure the test directory and object entries do not already exist.
3. A test directory and object are created in the master clearinghouse; the master clearinghouse has now become a test clearinghouse.
4. The test clearinghouse is checked to make sure the test directory and object entries exist.
5. The test clearinghouse is backed up locally.
6. The master clearinghouse is restored.
7. The master clearinghouse is checked to make sure the test directory and object entries do not exist.
8. The test clearinghouse is restored.
9. The test clearinghouse is checked to make sure the test directory and object entries do exist.
10. The master clearinghouse is restored.

13.2.1.3 Default Values for ‘‘backup_restore_ch.ksh’’

backup_restore_ch.ksh runs with the following default values:

- DCE Administration login

  cell_admin

- DCE Administration password

  -dce-

- Test Name

  bkrsch

- CDS Test Directory

  bkrsch_*hostname*

- Clearinghouse Name

  cellname#hostname

Note that all the defaults can be changed by editing the test scripts and changing the variable values.

13.2.1.4 Objects Created by ‘‘backup_restore_ch.ksh’’

Directories Created (in the current directory):

- **./tmp**
- **./backup**
- **./log**

Temporary Files Created (in the current directory):

- **./tmp/bkrsch_***hostname_pid***_STDOUT**
- **./tmp/bkrsch_***hostname_pid***_STDERR**

Log Files Created (in the current directory):

- **./log/bkrsch_***hostname_pid***_ERRORS**
- **./log/bkrsch_***hostname_pid***_SHORT**
- **./log/bkrsch_***hostname_pid***_FULL**

13.2.1.5 Running ‘‘backup_restore_ch.ksh’’

backup_restore_ch.ksh is invoked as follows:

```
cd /dcetest/dcelocal/test/systest/admin/cds

tests/backup_restore_ch.ksh
```

13.2.2 Backup Clearinghouse Automated Test

The purpose of the **backup_ch.ksh** test is to show that clearinghouses can be backed up locally.

13.2.2.1 Prerequisites for Running ‘‘backup_ch.ksh’’

The following things must be true in order to successfully run the **backup_ch.ksh** system test:

- The test must be run as root, and the machine the test is being executed on must have root access via **.rhosts** to all machines in the DCE cell.
- The test must be executed on a CDS server machine.
- There can be no CDS clients running DCE during the test except for a Security server in a split server configuration.
- You must source the system test profile file:

  /dcetest/dcelocal/test/systest/profile.dcest

Note: You should not execute this test on any CDS server which you cannot afford to corrupt as a result of running the test. In some instances the CDS clearinghouse can be corrupted if the test fails.

13.2.2.2 ‘‘backup_ch.ksh’’ Logic Flow

When invoked, **backup_ch.ksh** performs the following series of operations:

1. DCE is stopped.
2. The master clearinghouse is backed up locally.
3. DCE is restarted.

13.2.2.3 Default Values for ‘‘backup_ch.ksh’’

backup_ch.ksh runs with the following default values:

- DCE Administration login

  cell_admin

- DCE Administration password

-dce-

- Test Name

  bkch

- Clearinghouse Name

  cellname#**hostname**

Note that all the defaults can be changed by editing the test scripts and changing the variable values.

13.2.2.4 Objects Created by ‘‘backup_ch.ksh’’

Directories Created (in the current directory):

- **./tmp**
- **./backup**
- **./log**

Temporary Files Created (in the current directory):

- **./tmp/bkch_***hostname_pid*_**STDOUT**
- **./tmp/bkch_***hostname_pid*_**STDERR**

Log Files Created (in the current directory):

- **./log/bkch_***hostname_pid*_**ERRORS**
- **./log/bkch_***hostname_pid*_**SHORT**
- **./log/bkch_***hostname_pid*_**FULL**

13.2.2.5 Running ‘‘backup_ch.ksh’’

backup__ch.ksh is invoked as follows:

cd /dcetest/dcelocal/test/systest/admin/cds

tests/backup_ch.ksh

or:

tests/backup_ch.ksh *directory_to_save_CDS_clearinghouse_in*

13.2.3 Restore Clearinghouse Automated Test

The purpose of the **restore_ch.ksh** test is to show that clearinghouses can be restored from a local backup.

13.2.3.1 Prerequisites for Running ‘‘restore_ch.ksh’’

The following things must be true in order to successfully run the **restore_ch.ksh** system test:

- The test must be run as root, and the machine the test is being executed on must have root access via **.rhosts** to all machines in the DCE cell.
- The test must be executed on a CDS server machine.
- There can be no CDS clients running DCE during the test except for a Security server in a split server configuration.

Note: You should not execute this test on any CDS server which you cannot afford to corrupt as a result of running the test. In some instances the CDS clearinghouse can be corrupted if the test fails.

13.2.3.2 ‘‘restore_ch.ksh’’ Logic Flow

When invoked, **restore_ch.ksh** performs the following series of operations:

1. DCE is stopped.
2. The master clearinghouse is backed up locally.
3. The backed up clearinghouse is restored.
4. DCE is started.

13.2.3.3 Default Values for ‘‘restore_ch.ksh’’

restore_ch.ksh runs with the following default values:

- DCE Administration login

  cell_admin

- DCE Administration password

  -dce-

- Test Name

  rsch

- Clearinghouse Name

  cellname#hostname

Note that all the defaults can be changed by editing the test scripts and changing the variable values.

13.2.3.4 Objects Created by ‘‘restore_ch.ksh’’

Directories Created (in the current directory):

- **./tmp**
- **./backup**
- **./log**

Temporary Files Created (in the current directory):

- **./tmp/rsch_***hostname_pid***_STDOUT**
- **./tmp/rsch_***hostname_pid***_STDERR**

Log Files Created (in the current directory):

- **./log/rsch_***hostname_pid***_ERRORS**
- **./log/rsch_***hostname_pid***_SHORT**
- **./log/rsch_***hostname_pid***_FULL**

13.2.3.5 Running ‘‘restore_ch.ksh’’

restore_ch.ksh is invoked as follows:

cd /dcetest/dcelocal/test/systest/admin/cds

tests/restore_ch.ksh *directory_to_restore_CDS_clearinghouse_from*

Note: The *directory_to_restore_CDS_clearinghouse_from* should contain the files of a previously successfully backed up clearinghouse.

This script does not verify the presence of the files it is to restore, and it does not recover the file to its original clearinghouse if there is a failure during the test.

13.2.4 Clearinghouse and Replica Checklist 1

The purpose of the **rep_ch_1_checklist** test is to do manipulations of CDS replicas and clearinghouses.

13.2.4.1 Prerequisites for Performing ‘‘rep_ch_1_checklist’’

The following things must be true in order to successfully perform the **rep_ch_1_checklist** system test checklist steps:

- The checklist must be performed as root, and the machine the checklist is being executed on must have root access via **.rhosts** to all machines in the DCE cell.
- The checklist must be executed on the initial CDS server machine.
- An additional CDS server is required on which to perform the checklist steps. There can be no CDS clients running DCE while the steps are being performed, except for a Security server in a split server configuration.
- The test must have access to the **expect** command.

Note: You should not execute this checklist on any CDS server which you cannot afford to corrupt as a result of running the test steps. In some instances the CDS clearinghouse can be corrupted if the test fails.

13.2.4.2 ‘‘rep_ch_1_checklist’’ Logic Flow

When performed, the steps in **rep_ch_1_checklist** result in the following series of operations:

1. The master clearinghouse is verified
2. DCE is stopped on the remote and master machines
3. Clearinghouses on the remote and master machines are backed up
4. DCE is started on the master and remote machines
5. A test directory is created in the master clearinghouse
6. The test directory is validated
7. A test clearinghouse is created on the remote machine
8. A replica of the test directory is created and verified in the test clearinghouse
9. The test clearinghouse is verified on the remote machine
10. The test clearinghouse is verified on the master machine

11. The master clearinghouse is disabled, and the switch to the test clearinghouse is verified on the master machine
12. The master clearinghouse is restarted and verified
13. The test clearinghouse is disabled on the remote machine
14. An attempt to skulk the test clearinghouse is made on the master machine
15. The switch from the test to the master clearinghouse is verified on the remote machine
16. The test clearinghouse is restarted and verified on the remote machine
17. A new directory and object are added and verified on the master machine
18. The new directory and object are verified on the remote machine
19. Convergence is tested on the master and remote machines
20. The master replica is changed to the one located on the remote machine
21. The test directory replica is deleted on the local machine
22. DCE is stopped on the remote and master machines
23. The test clearinghouses are deleted
24. DCE is restarted on the master and remote machines

13.2.4.3 Default Values for ‘‘rep_ch_1_checklist’’

rep_ch_1_checklist runs with the following default values:

- DCE Administration login

  cell_admin

- DCE Administration password

  -dce-

- Root (superuser) password on additional server

  systest1

- Test Name

  reps

- Clearinghouse Name

  cellname#hostname

13.2.4.4 Objects Created by ‘‘rep_ch_1_checklist’’

Directories Created (in the current directory):

- **./tmp**
- **./backup**
- **./log**

Temporary Files Created (in the current directory):

- **./tmp/reps_***hostname_pid***_STDOUT**
- **./tmp/reps_***hostname_pid***_STDERR**

Log Files Created (in the current directory):

- **./log/reps_***hostname_pid***_ERRORS**
- **./log/reps_***hostname_pid***_SHORT**
- **./log/reps_***hostname_pid***_FULL**

13.2.4.5 Performing ‘‘rep_ch_1_checklist’’

Perform the **rep_ch_1_checklist** steps as follows:

cd *systest-root*/**admin/cds/tests**

Execute the steps in

rep_ch_1_checklist

as specified.

13.2.5 Clearinghouse and Replica Checklist 2

The purpose of the **rep_ch_2_checklist** is to perform manipulations of CDS replicas and clearinghouses.

13.2.5.1 Prerequisites for Performing ‘‘rep_ch_2_checklist’’

The following things must be true in order to successfully perform the **rep_ch_2_checklist** system test checklist steps:

- The checklist must be performed as root, and the machine the checklist is being executed on must have root access via **.rhosts** to all machines in the DCE cell.

- The checklist must be executed on the initial CDS server machine.
- An additional CDS server is required on which to perform the checklist steps. There can be no CDS clients running DCE while the steps are being performed, except for a Security server in a split server configuration.

Note: You should not execute this checklist on any CDS server which you cannot afford to corrupt as a result of performing the test steps. In some instances the CDS clearinghouse can be corrupted if the test fails.

13.2.5.2 ‘‘rep_ch_2_checklist’’ Logic Flow

When performed, the steps in **rep_ch_2_checklist** result in the following series of operations:

1. A test directory and object are created and verified in the master clearinghouse
2. A readonly replica of the test directory is created in a clearinghouse located on a second machine
3. The test directory is excluded from the master clearinghouse
4. The test directory is verified as accessible in the clearinghouse on the second machine
5. The test directory on the master machine is made readonly
6. The clearinghouse on the second machine is disabled
7. The clearinghouse on the second machine is relocated to a third machine, enabled, and verified

13.2.5.3 Performing ‘‘rep_ch_2_checklist’’

Perform the **rep_ch_2_checklist** steps as follows:

cd *systest-root*/**admin/cds/tests**

Execute the steps in

rep_ch_2_checklist

as specified.

13.2.6 Intercell GDA Checklist

The purpose of the **intercell_gda_checklist** is to validate the response of servers and clients when the GDA exits unexpectedly.

13.2.6.1 Prerequisites for Performing ‘‘intercell_gda_checklist’’

The following things must be true in order to successfully perform the **intercell_gda_checklist** system test checklist steps:

- The checklist steps must be performed as root, and the machine the checklist is being executed on must have root access via **.rhosts** to all machines in the DCE cell.
- The checklist must be executed on the initial CDS server machine.
- The **gdad** must be configured in both cells.
- The names given to the cells during configuration should be expressed in their full DNS form (e.g., **mycell.osf.org**, not **mycell**).
- An additional CDS server is required on which to perform the checklist steps. There can be no CDS clients running DCE while the steps are being performed, except for a Security server in a split server configuration.

Note: You should not execute this checklist on any CDS server which you cannot afford to corrupt as a result of performing the test steps. In some instances the CDS clearinghouse can be corrupted if the test fails.

13.2.6.2 ‘‘intercell_gda_checklist’’ Logic Flow

When performed, the steps in **intercell_gda_checklist** result in the following series of operations:

1. Information to enable configuration for intercell testing is generated
2. The DNS database is updated with intercell information
3. The intercell configuration is setup and verified using **rgy_edit**
4. CDS intercell access is performed
5. The Global Directory Agent (GDA) (**gdad**) is stopped
6. CDS intercell access is again performed
7. The Global Directory Agent (GDA) (**gdad**) is restarted
8. CDS intercell access is performed once again

13.2.6.3 Performing ‘‘intercell_gda_checklist’’

Perform the **intercell_gda_checklist** steps as follows:

cd *systest-root*/**admin/cds/tests**

Execute the steps in

intercell_gda_checklist

as specified.

13.2.7 dcecp System Tests

The **dcecp** system tests are implemented as a set of four **dcecp** scripts:

DCPSAN001 Implements the procedure to backup and restore the CDS namespace, using a local directory.

DCPSAN002 Implements the procedure to backup the CDS name space to a local directory.

DCPSAN003 Implements the procedure to restore the CDS files that were backed up by **DCPSAN002**. It expects to find all the namespace files that were backed up by **DCPSAN002**.

DCPSAN004 Implements the procedure to unconfigure a DCE client.

The first three scripts are installed at:

dce-root-dir/**install**/*platform*/**dcetest/dce1.2.2/test/systest/admin/cds/tests**

For information on how to run these tests, see the

dce-root-dir/**install**/*platform*/**dcetest/dce1.2.2/test/systest/admin/cds/tests/README**

file.

The fourth script, **DCPSAN004**, is installed in:

dce-root-dir/**install**/*platform*/**dcetest/dce1.2.2/test/systest/admin/config**

For information on how to run this test, see the

dce-root-dir/**install**/*platform*/**dcetest/dce1.2.2/test/systest/admin/config/README**

file.

13.2.8 DFS Administrative Checklist

The purpose of the **admin_checklist** is to exercise many of the administrative functions associated with DFS. Note that this checklist does *not* include testing of the backup system.

13.2.8.1 Prerequisites for Performing ‘‘admin_checklist’’

In order to successfully perform the **admin_checklist** system test checklist steps, you must have a multi-**flserver** DFS cell configured with both native and LFS aggregates ready for configuring.

13.2.8.2 ‘‘admin_checklist’’ Logic Flow

When performed, the steps in **admin_checklist** result in the following series of operations:

1. Configure native filesystem into DFS
2. Create LFS aggregates/filesets
3. Create ACLs
4. Disable/Enable:
   - aggregates
   - filesets
   - servers
   - **setuid** capability
5. Update/Modify:
   - aggregates
   - filesets
   - server keys
   - cache
   - ACLs
6. Start/Stop servers
7. Cleanup cache
8. Monitoring
9. Dump/Restore

13.2.8.3 Performing ‘‘admin_checklist’’

Perform the **admin_checklist** steps as follows:

cd *systest-root*/**admin/file/tests**

Execute the steps in

admin_checklist

as specified.

13.3 Global Directory System Tests

Testcase **dcegdshd** tests the shadowing functions of the GDS component of DCE. Testcase **gds_xds_str_001** tests the operation of the threadsafe XDS, threadsafe XOM, and threaded DUA subsystems of the DCE Global Directory Service (GDS).

13.3.1 dcegdshd

dcegdshd tests the shadowing functions of GDS. Specifically, it tests the ability of GDS to maintain replicas (called ''shadows'' in GDS) of an object with a single, simple attribute, in some number of DSAs in a GDS administrative domain, with updates being done with what GDS considers to be ''high'' frequency (every 5, 10, 15, or 30 minutes). **dcegdshd** does not test the ability of GDS to shadow either subtrees or objects with more complex attributes, nor does it test at any other update frequencies than those mentioned above.

The syntax for **dcegdshd** is:

dcegdshd -d *master_dsa* **-s** *shadow_dsa [* **-u** *update_time]*

or

dcegdshd *[*-**h** | -**H***]*

where:

-c	Specifies that the workstation logfiles be cleaned up (the default is that this option is specified).
-e	Specifies that the testcases listed on the command line be excluded.
-h	Specifies that a detailed usage message be displayed.
-H	Specifies that input parameters be validated only.
-l *loops*	Specifies that test be executed for *loops* number of iterations.
-t *hours*	Specifies that test be executed for *hours* number of hours.
-d *master_dsa*	Specifies the DSA that will house the master copy of the object. This DSA must already exist.

-s *shadow_dsa*	Specifies DSA to shadow the object in. This DSA must already exist (user can specify multiple **-s** options).
-u *update_time*	Specifies (in minutes) the amount of time to allow to elapse before propagating updates to first shadows. Default is 10 minutes.

Note: The granularity in update time is one of: 5, 10, 15, or 30 (minutes).

This test can be used for stress testing by specifying many **-s** options.

13.3.1.1 Running the dcegdshd Driver

To run the **dcegdshd** system test, follow these steps:

1. Make sure that the following are available on each system involved in the test:

   - *systest-root*/**tools**

     This is the directory containing the test tools.

   - *systest-root*/**profile.dcest**

   On the machine that will contain the master DSA, the following must be available in addition to what is mentioned above:

   - *systest-root*/**directory/gds/dcegdshd**

     Directory the contains the test scripts and files.

   Note that if you install the system tests using **dcetest_config**, all of the items mentioned above should be automatically installed in their correct locations.

2. Modify the file:

   systest-root/**directory/gds/dcegdshd/sTest.gds**

   to reflect the environment for the test. Change the strings *machine [1-n]* in the assignments of values to the variables **mach_1** through **mach_***n* to be the names of the machines at your site that are to participate in the system test. Note that the machine assigned to variable **mach_1** is considered the master for the GDS administrative domain that is created by running the **worldSetup.gds** script. (This master DSA is the DSA most stressed during the test run.) After these assignments have been changed, you may wish also to change the names used in setting up the test directory service, though this is not necessary. These names are stored in the **GDS_DSADNPREFIX** variable (in **sTest.gds**) in the following format:

   GDS_DSADNPREFIX="*country_name org_name org_unit_name* **dsa"**

   Change this assignment, if you wish, to reflect the country name, organization name, and organizational unit name you prefer.

   Make sure that an entry appears in the list assigned to the **GDS_HOSTCONFIG** variable for each **mach_1** through **mach_***n* variable to which you have assigned a system name. See the example version of **sTest.gds**, given below, to see how this is done.

3. Copy the

   systest-root/**directory/gds/dcegdshd/sTest.gds**

   file to all machines that will be involved in the test.

4. Source the

   systest-root/**profile.dcest**

   environment file.

5. On the machine assigned to the variable **mach_1** in the **sTest.gds** file, enter the following command:

   ksh *systest-root*/**tools/worldSetup.gds** *systest-root*/**directory/gds/dcegdshd/sTest.gds**

   —When executed, this script will first remove any existing GDS configuration information on each system test machine for directory ID 2, and then configure GDS on each machine, setting up a GDS administrative domain, with the master or initial DSA on the machine specified by the variable **mach_1** in the **sTest.gds** file. The script will report on its progress, telling you what files are logging the progress of setting up each machine involved in the test (this is in case you want to monitor the progress directly). When all machines indicate setup is complete, the script will display a ''SETUP OK'' message and exit. All the test machines are now ready for testing. Note that all information on remote DSAs and objects is created in the DIT and cache of directory ID 2.

6. Make sure that the following files exist on the machine specified by the variable **mach_1** in the **sTest.gds** file:

   - *systest-root*/**directory/gds/dcegdshd/dcegdsh1**
   - *systest-root*/**directory/gds/dcegdshd/Alter_object.sv**
   - *systest-root*/**directory/gds/dcegdshd/Create_object.sv**
   - *systest-root*/**directory/gds/dcegdshd/Create_shadow.sv**
   - *systest-root*/**directory/gds/dcegdshd/Delete_object.sv**
   - *systest-root*/**directory/gds/dcegdshd/Remove_object.sv**

7. Make sure that the following program exists on the machine specified by the variable **mach_1** in the **sTest.gds** file:

   systest-root/**directory/gds/dcegdshd/view_obj**

8. On the machine containing the initial DSA, enter the following command:

   systest-root/**directory/gds/dcegdshd/dcegdshd** *options*

   — where *options* are as specified for **dcegdshd** at the beginning of this section. The script will report the environment in which it is running, where it is logging, and so on. It will then start running the **dcegdsh1** script, reporting on success or failure at the end of each run, storing the log files in the **pass** and **fail** directories, and updating the **pass-fail-log** file.

Note that when **dcegdshd** is run, the object:

/C=us/O=osf/OU=dce/CN=Mark

must not be present in the Directory Information Tree.

Note also that the format for specifying a DSA to the program is:

*/country_name/org_name/org_unit_name/***dsa***/dsa_name*

For the following 4-machine configuration:

mach_1 **dceqa1** (initial DSA)

mach_2 **dceqa2**

mach_3 **dceqa3**

mach_4 **dceqa4**

dcegdshd would be invoked as follows:

```
dcegdshd -t 48 -d /us/osf/dce/dsa/dceqa1 -s /us/osf/dce/dsa/dceqa2 \
        -s /us/osf/dce/dsa/dceqa3 -s /us/osf/dce/dsa/dceqa4
```

13.3.1.2 Example Configuration File

Following are the contents of a configuration file for **dcegdshd**:

```
mach_1=dceqa1
mach_2=dceqa2
mach_3=dceqa3
mach_4=dceqa4

HOURS=12.0
TIME_SERVERS=
TIME_CLERKS=
NTP_SERVER=
NTP_CLIENTS=
CDS_SERVERS="mach_1"
AUTH_SERVER="mach_1"
CELLNAME=NO_CELL
GATEWAYS=

GDS_REQVARS="GDS_DSADNPREFIX GDS_DUAPORTBASE GDS_DSAPORTBASE GDS_NCLIENTS GDS_HOSTCONFIG"
GDS_DSADNPREFIX="us osf dce dsa"
GDS_INITIALDSA="2,mach_1"
GDS_DIR_ID=2
GDS_DUAPORTBASE="2200"
GDS_DSAPORTBASE="2100"
GDS_NCLIENTS=16
```

```
GDS_HOSTCONFIG="mach_1:name=same:dir=1,Cli-Srv,mach_1:dir=2,Cli-Srv,mach_1"
GDS_HOSTCONFIG="$GDS_HOSTCONFIG mach_2:name=same:dir=1,Cli-Srv,mach_2:dir=2,Cli-Srv,mach_1"
GDS_HOSTCONFIG="$GDS_HOSTCONFIG mach_3:name=same:dir=1,Cli-Srv,mach_3:dir=2,Cli-Srv,mach_1"
GDS_HOSTCONFIG="$GDS_HOSTCONFIG mach_4:name=same:dir=1,Cli-Srv,mach_4:dir=2,Cli-Srv,mach_1"

export mach_1 mach_2 mach_3 mach_4
export ALL_MACHINES
export HOURS TIME_SERVERS TIME_CLERKS NTP_SERVER NTP_CLIENTS CDS_SERVERS
export AUTH_SERVER CELLNAME GATEWAYS GDS_DSADNPREFIX  GDS_INITIALDSA
export GDS_DIR_ID GDS_DUAPORT GDS_DSAPORTBASE GDS_DSAPORT1 GDS_DSAPORT2
export GDS_HOSTCONFIG GDS_DUAPORTBASE
```

13.3.1.3 dcegdshd and DSA Processes

An active GDS on any given machine consists of from three to five processes which cooperate as a unit. From time to time, one or more of these processes may die (due to system problems, network difficulties, or whatever), rendering GDS on the machine on which this happens inoperative. Usually in such situations, deactivating and then reactivating all the GDS processes will restore GDS to full functionality. **dcegdshd**, in order to avoid curtailed or useless test runs caused by instances of service failure, parses the log from each run of the **dcegdsh1** script and attempts to reactivate GDS in this way on any machine that could not be reached during the **dcegdsh1** run.

dcegdshd reports these restart attempts in the file **restart_info**. This file is copied, along with the **JOURNAL** and **spoolfile** files from the **dcegdshd** run, to the **fail** directory named for the current iteration of the test. Thus the contents of this file can help you to determine the reasons for a test failure, and also provide a measure of the stability of GDS on the machines being tested.

13.3.1.4 Success Criterion for dcegdshd

The success criterion for **dcegdsh1** is: no failed updates to any of the DSAs containing shadows. If you consider this too rigorous, you can modify the **dcegdsh1** script to use other criteria. To find the section of code where success or failure is currently decided, edit **dcegdsh1** and search for the second occurrence of the string **TEST_FAILED**.

13.3.2 gds_xds_str_001

The **gds_xds_str_001** test provides a means to verify the operation of the threadsafe XDS, threadsafe XOM, and threaded DUA subsystems of the DCE Global Directory Service (GDS). The **gds_xds_str_001** test uses the **xt_test** test driver (from the XDS

functional tests; see Chapter 6 for a description of the XDS functional tests) to execute a specially constructed set of testcases that exercise the threadsafe features of XDS. The **gds_xds_str_001** test files are located in the directory

systest-root/**directory/gds/gds_xds_str_001**

in the source tree, and in the installed test tree.

The **gds_xds_str_001** test supports three levels of stress:

- **HIGH**
- **MEDIUM**
- **LOW**

The meaning of each of these levels is user-specified. The duration of the test run can be specified as a number of hours or as a number of passes.

13.3.2.1 Prerequisites for Running the Test

The **gds_xds_str_001** test requires the following to be run successfully:

- **rsh** (or the equivalent) and **rcp** access to all of the test machines
- installation of GDS on all test machines
- installation of the system test tools on all test machines
- installation of the GDS system tests on all machines
- installation of the GDS functional tests on all test machines
- installation of the system test profile file (**profile.dcest**) on all test machines
- modifications to the **gds_xds_str_001** configuration file to make it correspond to the local test environment.

13.3.2.2 Test Input

The test derives all of its runtime parameters from a datafile, and is scalable to any number of machines and client processes. A sample datafile can be found at:

systest-root/**directory/gds/gds_xds_str_001/gds_xds_str_001.data**

13.3.2.3 Test Output

The **gds_xds_str_001** system test produces the following output:

runlog.*pid***.***date*	This file contains output describing all of the parameters for the test run, including the command line used to invoke the test, output about progress in verifying the input to and environment for the **gds_xds_str_001** test run, output about progress in setting up the GDS configuration and testcase files for the test, output showing when each test iteration started, and a one line summary of the result from each iteration. This log also contains the final statistics for the test run.
*pid.iteration***.passlog**	This file contains detailed information about the progress of a test iteration. It contains output on progress in setting up the GDS test tree for the test iteration, progress in starting the test clients, whether clients exited, the results of the client runs, and progress in cleaning up the GDS test tree.
***.suxtlog**	This is the **xt_test** standard output from the creation of the test tree.
***.sud2log**	This is the **xt_test** D2 logging output from the creation of the test tree.
***.xtlog**	These contain the **xt_test** standard output from the test clients.
***.d2log**	These contain the **xt_test** D2 logging output from the test clients.

The ***xtlog** and ***d2log** files have names in the following format:

*host***.***iteration***.***client_num***.***driver_PID***.***cli_OS***.***log_type***log**

where:

host	is the name of the client machine
iteration	is the test iteration for which the client is being started
client_num	is the number assigned to this client
driver_PID	is the PID of the **gds_xds_str_001** driver invoking this client
cli_O/S	is the operating system on the client machine as reported by **uname(1)**
log_type	is one of **sux**t, **sud2**, **xt**, or **d2**

For example, the client creating the test tree on an HP/UX machine for test iteration 3 might send its **xt_test** standard output to the file:

hp1.2.3.4434.HP-UX.suxtlog

The *runlog* is created in the directory specified by the variable **LOG_DIR** in the configuration file. The *per-iteration* logs (i.e., all logs except for the runlog) are also written in this directory during the iteration. After each iteration, the log files from that iteration are either deleted or moved. The logs are deleted if the variable **CLEANUP** is set to 1 in the configuration file *and* the iteration was successful. If **CLEANUP** is set to 0 *or* if the iteration was not successful, the logs are moved to a directory named *iteration*

under one of the following directories, which are created under the directory specified by the variable **LOG_BASE** in the configuration file:

config_only	contains logs from configuration only runs
error	contains logs from iterations where errors occurred
failed	contains logs from iterations that ran normally, but had client failures
killed	contains the log from the cleanup of the client machines and GDS if a signal was received
passed	contains logs from successful iterations

13.3.2.4 Execution Flow of Test

The **gds_xds_str_001** test execution flow is as follows:

1. Read the command line

   This step gives the script the name of the data file which describes the test run. The command line can also optionally specify that GDS is to be configured. Note that normally GDS is configured only once (on the first invocation of the test), and that this configuration is then used by subsequent invocations of the test.

2. Check variable settings

   The variable **VARLIST** in the **gds_xds_str_001** configuration file specifies a list of variables that must be defined in order for the test to run. Each variable in this list is checked to see if it has a value. The value of each variable that is set is recorded in the runlog file. If the variable **VARLIST**, or any of the variables in the list are undefined, a message indicating this is printed to the runlog and to the standard output, and the script exits.

3. Set variables and create directories

   The variables used to run the test are derived from the values of the variables set in the configuration file, and the log and temporary directories are created if they do not yet exist.

4. Print the ‘‘Hi’’ message

   A message is printed to the runlog and the standard output that shows the time the test started, all of the operation parameters, where logs will be written, what they will be named, and where the programs the test uses are expected to be.

5. Setup the trap handler

   The **trap()** function is called to setup a handler for **SIGHUP**, **SIGINT**, and **SIGQUIT**.

6. Build the client information file

A file is built that describes the logical clients to be run for each iteration. First, access to each client machine is verified. Second, the presence of the **xt_test** program in the expected directory is verified. Third, a line for each logical client assigned to each machine is generated in the client information file. This file contains a line for each client of the following form:

client_numb:*machine_name*:*client_OS*

7. Set up GDS (if specified)

   If the **-G** command line switch is specified, or the variable **CONFIG_GDS** is set to 1 in the configuration file, GDS will be configured for the test run on all the client machines. The first machine in the list of machines assigned to the variable **CLI_MACHINES** will be configured as the DSA that will be the server for the test run, unless the variable **mach_1** is assigned a machine name in the configuration file.

   Note: The **xt_test** program has hardcoded dependencies on the names of the DSAs involved and the directory id that is used. This means that the values of the variables **GDS_DSADNPREFIX**, **GDS_INITIALDSA**, and **GDS_HOSTCONFIG** in the configuration file must *not* be changed.

8. Setup testcases

   The testcases to be executed are setup on all of the test machines. Testcase setup involves creating testcase files that cause the proper number of threads for the specified stress level to be created by the **xt_test** program, and then propagating these files to the client machines. See the section below on configuration variables for more information on specifying the testcases to run.

9. Check for a GDS Configuration only run

   If GDS configuration was specified, and the number of passes was specified as 0, then the test is being run to configure GDS, but not to run any testcases. If this is the case, just report, cleanup, and exit

10. Loop, executing testcases until finished

    For each iteration the following steps are executed:

    a. Check to see if test loop should exit.

    b. Pick the ‘‘next’’ testcase to be run from the list of testcases to be executed.

    c. Pick the ‘‘next’’ client to create and cleanup the GDS test tree from the list of clients.

    d. Create the new passlog name.

    e. Print the ‘‘BEGINNING iteration’’ message to the runlog and passlog.

    f. Run the selected client to create the GDS test tree, and report on success or failure in the passlog.

    g. Start all the clients, reporting the start of each in the passlog.

    h. Verify client startups.

i. Verify client exits.

j. Verify client exit status.

k. Cleanup the GDS test tree.

l. Cleanup the log files from the iteration.

m. Report the success or failure of the iteration.

n. Update the passed, failed, and error counters.

11. Remove the testcase files for this run and report statistics from the run

13.3.2.5 Test Options

The syntax of the **gds_xds_str_001** system test command line is:

gds_xds_str_001 -h | **-f** *config [-***G***]*

where:

-h	print a usage message. This works for the regular or enhanced command-line modes.
-f *config*	specifies the path to the **gds_xds_str_001** configuration file.
-G	specifies that GDS is to be configured on all the test machines.

The **-h** option cannot be specified with any other option. If **-h** is not specified, the **-f** *config* option is required.

There is also an enhanced command line interface to **gds_xds_str_001**. This interface allows some of the parameters for the test to be specified on the command line; however, specifying parameters in this way is not recommended as usual practice. The enhanced command-line interface is enabled by creating a link (named **gds_xds_str_001_cl**) to the **gds_xds_str_001** script, and then invoking the test using this link. For further information on the enhanced interface, create the link and run the test with the **-h** (usage message) option.

13.3.2.6 Data and Configuration Variables

This section describes in a general way the configuration variables that control the execution of the **gds_xds_str_001** system test. For more details refer to the configuration file at:

systest-root/**directory/gds/gds_xds_str_001/gds_xds_str_001.data**

- Required variables

  As stated above, the **VARLIST** variable describes all of the variables that must be defined in order for the test to run. This list should be updated if required variables

are added to the configuration file, and *must* be updated if required variables are deleted from the file.

- Test machines and GDS configuration

  The test machines should be specified by shell variables **mach_1** to **mach_***n* (where *n* is the number of machines participating in the test). The shell variable **CLI_MACHINES** also should be coded with the names of all the test machines; these can be hard-coded in the variable, or specified via the values of the single machine variables (**$mach_1**, and so on). The variables containing the string ''**GDS_**'' in their names define the GDS configuration. In general, the only changes you will probably want to make to the GDS configuration will be to specify the names of the test machines; specify a different machine as the DSA server for the test by assigning the name of that mchine to the variable **mach_1**; or specify a different number of client processes; by changing the value of the *stress_level***_GDS_NCLIENTS** variables near the end of the file.

- Testcase available and testcases to execute

  Which testcases are to be executed is specified by the variable **TESTCASES** in the configuration file. The testcases that are available to be executed is specified by the variable **TESTCASE_LIST**. The value of the **TESTCASES** variable is a list of one or more of the testcase names specified by the variable **TESTCASE_LIST** in the configuration file, or the string ''variant''. If ''variant'' is specified, the test driver cycles through all of the testcases specified by the **TESTCASE_LIST** variable, executing a different testcase in each iteration. If ''variant'' is not specified, the driver will cycle through the testcases specified by the **TESTCASES** variable.

- Directories

  The following variables specify the paths to the directories needed by the test:

TESTCASEDIR	directory where the testcase files should be located, and where the per-run testcase files will be created
BINDIR	directory where the **xt_test** binary should be located
TMP_DIR	directory where temporary files will be created
LOG_ROOT	directory under which the **gds_xds_str_001** test results directory will be created
LOG_BASE	directory under which all of the **gds_xds_str_001** results directories (**pass**, **failed**, **error**, etc.) will be created
LOG_DIR	directory in which the runlog will be created and in which the per-iteration files will be written during the course of the iteration
STTOOLS_DIR	directory where the system test tools are located
D2_LOGDIR	directory (on both the local and the remote machines) in which the **D2** log output of the **xt_test** program will be written

- Wait Values

  The following variables specify the amount of time to allow for certain operations to complete, or to wait at some point in the test:

GDSSETUP_WAIT The amount of time, in minutes, to allow for the configuration of GDS to complete. This value can be adjusted to correspond to the number of machines you are configuring. For example, a combination of two HP/9000-700's, one RISC System/6000 and one DECpc 450ST will take about 15 - 20 minutes to configure.

READY_WAIT Amount of time, in minutes, to wait for the client building or removing the GDS test tree to complete, and to wait for the clients to report ready.

EXIT_WAIT Amount of time, in minutes, to wait for the clients to report that they are exiting.

ITERATION_SLEEP Amount of time, in seconds, to wait between test iterations.

The ***_WAIT** variables specify the upper bounds on how long some phase of the test should take. If the phase is not complete by the end of the specified wait period, something is most likely hung. The **ITERATION_SLEEP** value can be used to exert more stress on the server, since if it is set high enough (i.e., at approximately 3 minutes or higher), the DSAs from each iteration will exit due to lack of activity. This will in turn force the S-stub on the server to spawn a new group of DSAs for each iteration, instead of allowing it to continue to reuse the DSAs from the previous iteration(s). Note however that running the test in this fashion has been noted to decrease the load on the DSAs.

- Duration and Log Handling

  The **HOURS** variable sets the time of the test run in hours, and the **CLEANUP** variable specifies whether to save or remove logs from successful iterations. If the test is to run for some number of passes, the number of passes is specified via the **PASSES** variable (which supersedes **HOURS** if both are specified).

- Stress Level Semantics

  The **LOW_***, **MEDIUM_***, and **HIGH_*** variables specify the meaning of the stress levels **LOW**, **MEDIUM**, and **HIGH** respectively. For each stress level, the number of client processes and number of threads per client process can be specified. The ***stress_level*_GDS_NCLIENTS** variables allow specification of the number of client processes specified when GDS in configured. This number *must* be greater than the number of threads per client, multiplied by the number of clients per system.

13.3.2.7 Example GDS Configuration

An example GDS configuration for the test is shown in the following table.

TABLE 13-1. Example Cell Configuration for gds_xds_str_001

Machine	DSA-name	Dir-id	GDS type	default DSA	initial DSA
mach_1	dsa-m1	1 2	Client/Server Client	dsa-m1 dsa-m2	dsa-m1 none
mach_2	dsa-m2	1 2	Client/Server Client/Server	dsa-m1 dsa-m2	dsa-m1 none
mach_3	hostname	1 2	Client/Server Client	dsa-m1 dsa-m2	dsa-m1 none
. . .					
mach_n	hostname	1 2	Client/Server Client	dsa-m1 dsa-m2	dsa-m1 none

Note: ‘‘hostname’’ in the above table means that the DSA name is the name of the machine.

13.3.2.8 Running gds_xds_str_001

After all test prerequisites have been satisfied, the test can be executed with the command:

gds_xds_str_001 -f *configuration_file* **-G**

—which means to configure GDS before starting the test itself.

When running **gds_xds_str_001**, you should keep the following information in mind:

- If tracing is turned on (via **gdssysadm**) for the DSA server, DSA log files will accumulate in the

  dcelocal/**var/directory/gds/adm/dsa/dir1**

  directory over the course of the test run. For long runs, if tracing is turned on, the logs can consume large amounts of disk space. For example, over a 48 hour run, the logs have been observed to consume approximately 70 megabytes of disk space. If you are planning a long run with tracing turned on, make sure there is plenty of space available for the log files. The directory can be a symbolic link to a partition with more disk space.

- If large numbers of clients and threads are to be used, you may experience problems with limits on process table size and processes per id on the server side. This may happen because the S-Stub must spawn a new DSA process for each client request it receives when all DSAs are busy.

13.3.2.9 Analyzing Test Results

The results of running **gds_xds_str_001** can be determined during the test run by examining the runlog file for messages indicating that iterations failed or that errors were encountered. Another method is to look for a directory named **error** or **failed** in the **LOG_BASE** directory. The presence of either of these directories indicates that some iterations either failed or encountered errors. When the test run is complete, the number of iterations that passed, failed, or encountered errors is printed in the runlog. To determine which iterations failed or encountered errors, examine the **error** and **failed** directories in the **LOG_BASE** directory.

13.3.2.10 Sample Configuration File

A sample **gds_xds_str_001** configuration file is located in the source tree at:

systest-root/**directory/gds/gds_xds_str_001/gds_xds_str_001.data**

13.4 DFS System Tests

The following sections describe the automated tests and checklists used for system testing the DCE Distributed File Service, and how to set them up and run them.

A checklist is a series of instructions and manually-entered commands, together with a description of the expected results of executing the commands. Checklists are used to document test functions for which no automated test exists.

13.4.1 DFS System Test Cell Requirements

The following list shows the minimum cell requirements for running each of the DFS system tests. The configurations and optional data file settings used are recorded in the

dce-root-dir/**project/test.plans**

subdirectory for each DCE release. Data files and log files for automated tests can be found in the

dce-root-dir/**project/test.results**

directory.

dfs.maxfile and

dfs.maxdir Require a single DFS (server and client) machine. These tests should be run both for **dfsexport**ed native

	filesystems and for LFS filesystems.
dfs.glue	Requires 2 or more DFS machines (1 combined server and client; the others may be simply clients). The test must be run on the server machine, and the server machine must have **dfsexport**ed a native filesystem.
dfs.lock	Requires 2 or more DFS machines (1 combined server and client; the others may be simply clients). The test can be run on any DFS machine and can use either native or LFS filesystems.
dfs.read_write_all.main	Requires 2 or more DFS machines (1 combined server and client; the others may be simply clients). The test can be run on any DFS machine, but there must be at least one LFS aggregate **dfsexport**ed.
dfs.block_frag	Requires a single DFS (combined server and client) machine with an *expendable* LFS aggregate. This aggregate will be **newaggr**'d a number of times during the test, so it must not contain needed filesets.
dfs.repfldb_checklist	Requires 3 DFS (**fldb** server) machines and 1 core server machine. The test involves rebooting the machine serving as the **fldb sync** site, so this machine *must not* be providing the DCE core services (unless sufficient core replication is in place).
dfs.repfs_checklist	Requires 2 DFS (server and client) machines and 3 LFS aggregates.
dfs.sec.cross_bind_checklist	Requires 2 DFS (server and client) machines in separate cells.
dfs.wan_checklist	Requires 2 DFS (server and client) machines in separate cells across a WAN connection, *and* an additional DFS (client) machine across a WAN connection to a server in the same cell.

13.4.2 Installing the DFS System Tests and Checklists

The DFS system tests are installed with **dcetest_config**. See ''Installing the DCE System Tests'', in Chapter 11, for instructions on running **dcetest_config**.

13.4.3 dfs.glue

The **dfs.glue** test tests the DFS glue code by accessing directories and files via their UFS and DFS paths.

13.4.3.1 Syntax

The **dfs.glue** system test is invoked as follows:

dfs.glue [**-f**] *datafile* [**-t**] *hours_of_operation*

Where:

datafile

Specifies the name of a datafile. A sample can be found at:

systest-root/**file/glue.data**

You should create one datafile per **dfsexport**ed UFS partition.

hours_of_operation

An integer value that specifies the number of hours of continuous operation desired.

13.4.3.2 Prerequisites for Running the ‘‘dfs.glue’’ System Test

In order for you to successfully run the **dfs.glue** test, the following things must be true:

- The local machine is both the file server for the UFS/DFS path variables in the data file *and* a DFS client.
- You are currently logged in as UNIX user and DCE principal with ‘‘root’’ read and write access to the UFS/DFS paths and **rsh** (remote shell) permission to all machines specified by the **MACHINES** datafile variable.
- There is sufficient space to run the test (see below).
- Any data written via the DFS path is visible to all **MACHINES** after **MAXTIME_DFSUPDATE** seconds.
- Unauthenticated users have read and execute permission to the DFS path.

13.4.3.3 Space Required for Running the ‘‘dfs.glue’’ Test

The significant space requirements for running **dfs.glue** are as follows.

- Each test file created by **filewnr** (the program called by **dfs.glue** to write and read files) will be:

  8193 bytes * **NUMFILEWRITES**

  large (where **NUMFILEWRITES** is a variable whose value is specified in the **dfs.glue** datafiles). Note that the value 8193 can be overridden by specifying a

different value via the **-b** parameter to **filewnr**.

- Each test directory created by **dirwrite.sh** (the script called by **dfs.glue** to write test directory entries) will contain a copy of the host kernel. Note that you can override this by specifying an alternate ‘‘large’’ file to **dirwrite.sh** via the **-l** parameter, or by specifying a different value for the **LARGE_FILE** datafile variable.
- The maximum number of test files and test directories that will exist at any given time during the test can be determined by multiplying the value of the datafile variable **MACHINES** by the value of the datafile variable **NUMPROCPERMACH**.

13.4.3.4 Components of ‘‘dfs.glue’’

The **dfs.glue** test alternates between reading and writing files and directories locally and remotely via their UFS and DFS paths; the test components for writing and reading (i.e., verifying what was written) are:

- *systest-root*/**file/filewnr**
- *systest-root*/**file/dirwrite.sh**
- *systest-root*/**file/dirread**
- *dce-root-dir*/**dce/src/test/file/cache_mgr/spoke**
- *dce-root-dir*/**dce/src/test/file/cache_mgr/hub**

13.4.4 dfs.lock

The **dfs.lock** test script uses the **filewnr** program to test if whole file shared read locks and exclusive write locks can be obtained and honored correctly in DFS using **fcntl( )** calls. That is, if **NCLIENTS** processes per machine all concurrently request an exclusive write lock to the same DFS file, does only one succeed? And if **NCLIENTS** processes per machine all concurrently request a shared read lock to the same DFS file, do all succeed?

Concurrency is achieved by starting all the processes sequentially but having them all wait for the existence of a file in DFS before attempting to access the test file.

13.4.4.1 Syntax

dfs.lock is invoked as follows:

dfs.lock [**-f**] *datafile* [**-t**] *hours_of_operation*

Where:

datafile

Specifies the name of a datafile. This script can be executed from *any* DFS client machine. A sample datafile can be found at:

systest-root/**file/lock.data**

hours_of_operation

Specifies number of hours test is to run.

13.4.4.2 Prerequisites for Running the ‘‘dfs.lock’’ System Test

In order for you to successfully run the **dfs.lock** test, the following things must be true:

- You are currently logged in as a UNIX user and DCE principal with read and write access to the DFS path and **rsh** (remote shell) permission to all machines specified by the **CLI_MACHINES** datafile variable.
- The DCE principal specified by the **PRINC** datafile variable (see below) is valid and has read and write access to the DFS path.
- There is sufficient space to run the test (see below).

13.4.4.3 Space Required for Running the ‘‘dfs.lock’’ Test

The only significant space requirement for running **dfs.lock** is that the host machine must contain:

8193 * **NUMFILEWRITES** bytes

(where **NUMFILEWRITES** is a variable whose value is specified in the **dfs.lock** datafile).

13.4.4.4 Components of ‘‘dfs.lock’’

The **dfs.lock** test uses:

systest-root/**file/filewnr**

to perform writes, reads and lock operations.

13.4.5 dfs.maxdir

This test creates, reads and deletes a large directory with various entries (symbolic links, hard links, fifo file, etc). The bulk of the entries are simple ASCII files. Directory size and location are specified in a data file. The test verifies that the various entries can be created and read successfully.

The test uses the **dirwrite.sh** and **dirread** test components. An example data file can be found at:

systest-root/**file/maxdir.data**

The test is invoked as follows:

./dfs.maxdir [**-f**] *config_file* > *log_file* **2>&1 &**

If the test runs successfully to completion, the last line of the **maxdir.log** file will contain the string ''PASSED''.

13.4.6 dfs.maxfile

This test creates, reads and deletes a large file. The test invokes the **filewnr** program with switches based on the contents of a data file. Note that files created by **filewnr** consist of ''blocks'' of bytes that are empty except for the specified pattern, and that these blocks are read randomly. An example data file can be found at:

systest-root/**file/maxfile.data**

The test can be invoked as follows:

./dfs.maxfile [**-f**] *config_file* > *log_file* **2>&1 &**

If the test runs successfully to completion, the last line of the *log_file* will contain the string ''PASSED''.

13.4.7 dfs.block_frag

This test exercises all the block-fragment combinations by performing the following operations on an LFS aggregate:

- **newaggr**
- **dfsexport**
- **fts create**
- **dfsexport -detach**

- **salvage**

Block-fragment combinations used are based on ranges specified in a data file. The data file also specifies log sizes and fileset sizes. Future versions of the test may include fileset and replication operations. An example data file can be found at:

systest-root/**file/block_frag.data**

The test can be invoked as follows:

./dfs.block_frag [**-f**] *config_file*

If the test runs successfully to completion, the last line of the test output will contain the string ‘‘PASSED’’.

13.4.8 dfs.read_write_all.main

This test emulates concurrent but independent end user activity in LFS filesets in DFS. The test establishes DCE principals and ‘‘home’’ filesets for each principal, containing a work script. The test then logs in as each DCE principal on client machines and runs the work script for the specified number of hours. The data file specifies aggregates, aggregate sizes, server machines, client machines, principal names, uids and passwords. An example data file can be found at:

systest-root/**file/dfs.read.write.data**

An example work script can be found at:

systest-root/**file/do.ksh**

The test can be invoked as follows:

./dfs.read_write_all.main [**-f**] *config_file* [**-t**] *hours_of_operation* **> 2>&1 &**

If the test runs successfully to completion, the last line of the test output will contain the string ‘‘PASSED’’.

13.4.9 filewnr.c

The **filewnr.c** program is the basic file write-and-read module for the DFS system tests.

filewnr simply opens the specified file, and then writes and/or reads a specified pattern, at a specified offset in the file, in ‘‘blocks’’ of bytes for all blocks in the file. When allowed to run with all defaults, **filewnr** will create an 8 kilobyte+ file containing ‘‘holes’’. i.e., unwritten bytes.

The program operates on a single file. It is called by the **dfs.glue** and **dfs.lock** scripts to verify the ability to write, read and lock a file. Its unique characteristics are that it does not write every byte of the file and that it performs random rather than sequential reads.

13.4.9.1 Syntax

filewnr accepts the following parameters and options:

TABLE 13-2. filewnr.c Parameters and Values

Parameter	Values	Description
-b size	integer	Size (in bytes) of blocks to read and/or write
-d	none	Delete file when finished
-f filename	valid pathname	Name of file to read and/or write
-l locktype	EW SR	Exclusive write lock Shared read lock
-h	none	Print help message
-c	integer	ID number assigned to client by driver
-C	character string	Continuation message string
-T	character string	Termination message for wait file
-n bnum	integer	Number of blocks to read and/or write
-o offset	integer	Byte offset at which to read and/or write
-p pattern	character string	Data to read and/or write
-r	none	Read only flag
-s range	integer	Number of seconds to sleep while waiting
-v	none	Verbose output flag If specified, following data is logged: Parameters used Test successes and failures Output is to stdout; test failure messages to stderr
-w filename	valid pathname	Do not start until specified file exists

13.4.9.2 Logic Flow of ‘‘filewnr.c’’

When invoked, **filewnr** performs the following series of actions:

if (**-w** flag was specified)
loop until *file_to_wait_for* exists

if (**-r** flag was *not* specified)
open file (with *locktype* specified with **-l** flag if specified)
exit with failure if unable to obtain lock (i.e., don't block)

if (continuation message was specified)
inspect the wait file for specified string:

if (termination message is found)
exit

for each block:
write (*pattern* specified with **-p** flag) at (*offset* specified by **-o** flag)

close file

open file (with *locktype* specified with **-l** flag if specified)

exit with failure if unable to obtain lock (i.e., don't block)

if (continuation message was specified)
inspect the wait file for specified string:

if (termination message is found)
exit

for each block (randomly chosen):

read *pattern* length number of bytes at *offset*

compare what was read to *pattern*

close file

13.4.9.3 ''filewnr'' Program Output

The output of **filewnr** is one of the following exit values:

Value	*Meaning*
0	**SUCCESS**
1	**FAILURE**
2	**USAGE**

3 BAD_OPTIONS

4 BAD_WAIT

5 BAD_LOCK

If the **-v** (verbose) flag is specified, **filewnr**'s invocation parameters and operation success/failure messages will be logged to standard output. If the program is compiled with the **PERROR_is_perror** flag, error output will be sent to **stderr**; otherwise it will be sent to **stdout** (the default, and necessary for the operation of **dfs.lock**).

13.4.10 dirread.c

The **dirread.c** program is the read module for DFS directory integrity system testing.

dirread is passed an input file of directory entries (one entry per line) and the name of a test directory in which to find the entries. The program then verifies, using the **readdir( )** call, that all the entries do in fact exist in the test directory, and that no other entries exist in the test directory.

The program can be used in conjunction with the script **dirwrite.sh** to verify directory contents. It will accept as input the output file of **dirwrite.sh** and verify that the supposedly just-written entries do exist.

13.4.10.1 Syntax

dirread accepts the following parameters and options:

TABLE 13-3. dirread.c Parameters and Values

Parameter	Values	Description
-i inputfile	Valid file pathname	File from which to read
-p parentdir	Valid directory pathname	Directory in which to find testdir
-n nr_entries	integer	Number of directory entries to read
-t testdir	Valid directory pathname	Directory which contains file entries to read
-d	none	If specified, entries and testdir are deleted when program completes execution
-v	none	Verbose output flag

13.4.10.2 Logic Flow of ‘‘dirread.c’’

When invoked, **dirread** performs the following series of operations:

read (*inputfile* specified with **-i** flag) into an array

readdir the test directory, marking array entries as found:

if there is a directory entry that is not in the array

report an error

if there is an array entry that is not in the directory

report an error

if an entry is found in the directory more than once

report an error

if deleting test directory:

readdir the test directory, marking array entries as deleted

stat entry

rmdir directory entry

unlink non-directory entry

13.4.10.3 ‘‘dirread’’ Program Output

The normal output of **dirread** is one of the following exit values:

Value	*Meaning*
0	**SUCCESS**
1	**FAILURE**
2	**USAGE**
3	**chdir** to *parentdir* failed
4	**chdir** to *testdir* failed
5	*inputfile* error
6	file close error
7	**malloc** error
8	directory open error
9	close *testdir* error
10	**rmdir** *testdir* error
11	**stat** error

If the **-v** (verbose) flag is specified, **dirread**'s invocation parameters and operation success/failure messages will be logged to standard output (failure messages are logged to standard error).

13.4.11 dirwrite.sh

The **dirwrite.sh** script is the write module for the DFS directory integrity system test.

dirwrite.sh simply creates a test directory at a specified or default path and fills it with the specified (or default) number of entries. The minimum number of entries is six (6). These are:

- a ''large'' file (by default, the kernel)
- an empty file
- a directory
- a hard link
- a symbolic link
- a special file (mkfifo)

Any subsequent files created (up to **NUMDIRENTRIES**) are all small ASCII files.

In addition to filling the test directory, **dirwrite.sh** also performs the following operations on the directory:

- **cp**

- **mv**
- **chown**
- **chgrp**
- **chmod**

The return status of each operation is checked and **dirwrite.sh** exits immediately after any detected failure.

13.4.11.1 Syntax

dirwrite.sh accepts the following parameters and options:

TABLE 13-4. dirwrite.sh Parameters and Values

Parameter	Values	Description
-r	none	Remove test directory when finished
-h	none	Help flag: Display a usage message
-p	valid pathname	Specifies name of parent directory in which to create test directory Default is current directory.
-t	valid pathname	Specifies name of test directory to create Default name is: <hostname_PID>_dir
-n	integer	Specifies number of entries to create in test directory Default is 5000
-l	valid pathname	Specifies pathname of a ‘‘large file’’ to place in test directory
-o	valid pathname	Specifies name of file in which to output a listing of contents of test directory Default is <test_directory>/CONTENTS

13.4.11.2 Logic Flow of ‘‘dirwrite.sh’’

When invoked, **dirwrite.sh** performs the following series of operations:

1. create a directory
2. **chmod** the directory
3. **chgrp** the directory

4. **chown** the directory
5. create links to the directory
6. copy the directory
7. move the directory
8. fill the directory (includes using **cp**, **ln**, **touch**, **mkdir**, **mkfifo**, **rm**, **rmdir**)
9. (if specified) remove the directory (**rm -rf**)

13.4.11.3 ‘‘dirwrite.sh’’ Usage Example

Following is an example of calling **dirwrite.sh** directly:

```
dirwrite.sh  -p /:/ctd -t test_dir -n 500 -l /vmunix -o /:/ctd/test_dir_ls
```

This command line specifies the following:

- The test directory’s parent directory has the following pathname:

  /:/ctd

  Note that the parent directory must exist *before* you run **dirwrite.sh**, and you must have write permission for this directory.

- Create the test directory with the following pathname:

  /:/ctd/test_dir

- Create 500 entries in the test directory.
- Use **/vmunix** as the ‘‘large’’ file.
- Output a listing of the test directory’s contents to:

  /:/ctd/test_dir_ls

13.4.11.4 ‘‘dirwrite.sh’’ Output

If the test directory was not specified to be removed, the specified (by **-o**) output file will contain a listing of the test directory’s contents.

If a command fails, **dirwrite.sh** exits with a message to standard output announcing the failure.

13.4.12 dfs.fmul

The **dfs.fmul** test currently tests partial file locking, blocking while locked, and unlocking by using RPC from the client to the servers.

The test consists of three modules:

- **dfs.fmul** (Top level script not yet implemented)

  Tests fileset move under load.

- **fmul.client**

  Client module for **dfs.fmul**.

- **fmul.server**

  Server module for **dfs.fmul**.

13.4.12.1 Syntax

The **dfs.fmul** system test is invoked as follows:

For each server:

fmul.server [**-d**]

For the client:

fmul.client -f *datafile* **-s** *number_of_servers* **-n** *lockfile* **-p** *lockfile_path* [**-d**]

Where:

-d Specifies additional output useful for debugging.

-f *datafile* Specifies the name of a datafile.

s *number_of_servers* Specifies the total number of servers required.

-n *lockfile* Specifies the name of the file created and locked by test.

-p *lockfile_path* Specifies the path to the lockfile.

13.4.12.2 Prerequisites for Running the dfs.fmul

In order for you to successfully run the **dfs.fmul** test, the following things must be true:

- All machines used for the client and servers must be able to read and write the lockfile specified in the arguments to **fmul.client**.

- You are currently logged in as UNIX user root and DCE principal **cell_admin**.
- The appropriate number of servers must be started before the client. More than one server may run on an individual machine.

13.4.13 DFS System Testing Checklists

The present section describes checklists used for system testing DCE administrative and distributed file system functions.

A checklist is a series of instructions and the expected results of following those instructions. Checklists are used to document how to test functions for which no automated test currently exists.

13.4.13.1 dfs.repfs_checklist

Steps to follow for setting up and testing replicated filesets. At the minimum, 2 machines are required, both DFS servers, one as both client and server.

The testing includes:

- multiple read/write access to a LFS fileset that is replicated
- verifying both release and scheduled replication
- disabling and re-enabling the primary (r/w) fileset
- disabling and re-enabling a secondary (read-only) fileset

13.4.13.2 dfs.repfldb_checklist

Steps to follow for setting up and testing replicated fileset location database servers. At the minimum, 3 machines are required, each configured as a fileset location database server.

The testing includes:

- multiple read/write access to both native and LFS filesets
- fileset manipulation (cloning, renaming, moving)
- disabling and re-enabling one or more fileset location database servers.

13.4.13.3 dfs.wan_checklist

Steps to follow for setting up and testing wide-area network access to DFS. A minimum of 2 machines, one at each end of a wide-area network, is required for the test.

The testing includes:

- intra-cell access
- cross-cell access

13.4.13.4 dfs.sec.cross_bind_checklist

Steps to follow for setting up and testing cross-cell DFS access with ACLs. A minimum of 2 machines, each configured in a different cell, is required for the test.

The testing includes:

- cross-cell write access (denied/granted)
- cross-cell read access (denied/granted)

13.5 Security Delegation Tests

There are two security delegation system tests that are not run under TET. They are described in the following two subsections.

13.5.1 dlgstr001

dlgstr001 is a multi-delegate test of delegation. See the

dce-root-dir/**dce/src/test/systest/security/dlgstr001/README**

file for details on how to run it.

13.5.2 dlgcf002

dlgcf002 is an ACL and compatibility delegation system test. See the

dce-root-dir/**dce/src/test/systest/security/dlgcfg002/README**

file for details on how to run it.

13.6 RPC-CDS System Test

The **rpc.cds.3** system test is designed, as its name suggests, to stress the RPC and CDS components of the DCE software.

The test first determines the maximum number of concurrent calls that the server can handle, using no authentication. The maximum number of concurrent calls has been reached when clients start receiving the status **rpc_s_server_too_busy** (if the client is using a connection-oriented protocol) or **rpc_s_comm_failure** (if the client is using a connectionless protocol) in response to calls to the server.

After the maximum for concurrent calls has been determined, the test loops, importing a server binding from a different CDS object on each loop, and using that binding to request data from the server (the data consists of a conformant structure containing an array of strings, modelled as a personal calendar). During this looping, the CDS cache data maintained on behalf of the clients is frequently invalidated in order to force the CDS clerk to obtain new information from the cell clearinghouse.

The **rpc.cds.3** system test exerts stress on the RPC component by making many remote procedure calls passing a complex data type at some specified level of authentication. The test exerts stress on the CDS component by executing many namespace lookups and binding import operations, forcing the use of group attributes to resolve binding searches, and forcing many namespace searches to resolve names by frequently invalidating the contents of the CDS cache.

13.6.1 Features of the RPC-CDS System Test

Some special features of the **rpc.cds.3** system test are:

- Instant status reports on receipt of **SIGQUIT**
- Toggling of debug output on receipt of **SIGHUP**
- Graceful shutdown on receipt of **SIGINT**

13.6.2 Logic Flow of RPC-CDS System Test Setup

In outline, the operation of **rpc.cds.3_setup.sh** is as follows:

1. Checks to make sure that the user has a valid identity as **cell_admin**.
2. Checks to make sure that all of the variables used by the setup script are assigned values in the configuration file.

3. Creates the client and server principals.
4. Creates the client and server accounts and keytab files. If the path to the keytab file does not exist, the script attempts to create it. Note that you will be prompted for the **cell_admin** password twice during this part of the setup.
5. Creates the CDS directory into which the server interface entry will be exported.
6. Attempts to create a client keytab file on any systems named in the configuration file (or via the **-r** command line option) as client machines in the test.
7. Attempts to create a server keytab file on any systems named in the configuration file (or via the **-R** command line option) as server machines in the test.

The **rpc.cds.3** server binary (**rpc.cds.3_srv**) exports to the CDS namespace a number of objects that refer, via the object UUID in each entry, to one of the calendars that the server has data for.

In order to make use of the server easier, the object names are of the form:

rpccds3_cal*N*

—that is, the string **rpccds3_cal** with a numeric suffix.

13.6.3 Server Side Logic Flow

The **rpc.cds.3_srv** binary implements the server side of the **rpc.cds.3** system test. The flow of logic in the server is as follows:

1. Parse the command line.
2. Read the configuration file specified by the **-f** command line parameter.
3. Register authorization information.

The following step is executed only if the **rpc.cds.3_smain.o** object was compiled with the **DRPCCDS3_DO_LOGIN** switch:

4. Establish the server identity.

If the **rpc.cds.3_smain.o** object was not compiled with the **DRPCCDS3_DO_LOGIN** switch then the following step is executed:

4. Get the login context for the current identity.

The main line of the test logic flow resumes with step 5:

5. Initialize the mutex and condition variables for the **pthread_cond_timedwait( )** call that controls the duration of the RPCs.

If the **rpc.cds.3_smain.o** object was compiled with the **DRPCCDS3_AUTO_REFRESH** switch, then the following four steps are executed:

6. Initialize the mutex and condition variables for the **pthread_cond_timedwait( )** call that is used to time identity refreshes.

7. Get the expiration time of the server's current identity, and from it calculate the ticket lifetime.

8. Save the encrypted key from the key returned by **sec_key_mgmt_get_key()** in order to use it when refreshing the server identity.

9. Spawn the thread that will maintain the server identity.

The main line of the test logic flow resumes with Step 10:

10. Spawn the thread that will catch and handle signals for the process.

11. Read the calendar data files specified on the command line or in the configuration file, and load the calendar data into an internal array. Note that the number of calendar data files does not have to be the same as the number of calendars. If the number of data files is smaller then the number of calendar objects to be exported to the namespace, then in some cases more than one namespace entry will refer to a single calendar. This convention allows the user to specify that many objects are to be created without having to specify many calendar data files.

12. Loop through the range of numbers specified by the sequence start and number of calendars (specified respectively with **RPCCDS3_SRV_CALSEQ** and **RPCCDS3_SRV_NCALS** in the configuration file) to be managed by this server.

For each number in the range of numbers mentioned in the previous step above, the test now does the following:

1. Construct the CDS name of the calendar object that will be associated with that number. This name is of the form:

   CDS_PATH/**rpccds3_cal***number*

2. Get a UUID for the calendar object. If the calendar object already exists in the CDS namespace, then the UUID from the existing entry is used; if the entry does not exist, or if it exists but has no UUID in it, then one of two things can happen:

   - If the calendar that is to be associated with the CDS entry already has a UUID associated with it, then that UUID is used.
   - If the calendar that is to be associated with the CDS entry has not yet had a UUID associated with it, then a new UUID is generated.

   If the CDS entry exists and has a UUID in it and the calendar has a UUID associated with it, then if the UUIDs match, that UUID is used; if the UUIDs don’t match, then the old UUID is removed from the object, and the UUID from the calendar is used.

3. If the UUID is not from the namespace entry, then the UUID is exported to the namespace entry (this has the side effect of creating the namespace entry if does not already exist).

4. If the server object UUID vector does not yet contain the UUID, then the UUID is added to the server object UUID vector.

5. The CDS name of the CDS server entry is added to the group attribute of the CDS object.

6. The server obtains binding handles and exports them to the namespace entry specified in the configuration file.

7. The server listens for client requests for calendar data.

8. When a client call request is received, the server extracts the object UUID from the client binding and searches the internal array of calendars for a calendar associated with that UUID. If such a calendar is found, then the calendar data is returned to the client.

Note that during the server's run, information on total calls handled, calls that passed and failed, number of id refreshes, and the last time the id was refreshed can be obtained by sending **SIGQUIT** to the server process. This report is also generated if the server is killed with **SIGINT**.

13.6.4 Client Side Logic Flow

Following is a detailed list of the steps the client performs:

1. Gets values for operational parameters by reading the configuration file, the name of which by default is **rpc.cds.3.conf**; or it can be passed in the command line via the **-f** option.

2. Assumes the client principal identity specified in the configuration file.

3. Sets various strings for reporting, such as hostname and operating system.

4. Looks in the CDS namespace for an existing entry for the **rpc.cds.3** interface (defined in the configuration file). If a CDS namespace entry is found, then all the binding handles that **rpc_binding_import_next( )** will return are sequentially imported, and an **rpc_mgmt_is_server_listening( )** call is made to verify that the binding is usable. This step ensures that the later steps will find some usable bindings at the server, and that the server is alive.

5. If a protocol is specified, converts the binding handle to a string binding and parses it to find the protocol type.

6. Looks for the synchronization file and sleeps after finding it, in order to synchronize startup of its RPCs.

7. Makes the RPC. If the **-b** flag was specified, the client checks the return status from the call and exits with the appropriate value, as described above. If the client is running in stress mode, the status is checked, counts of successes, failures, total calls and call times are updated, and the next RPC is made.

13.6.5 Parameters and Options for the RPC-CDS System Test

The **rpc.cds.3_setup.sh** script accepts the following command line switches.

TABLE 13-5. Command Line Switches for rpc.cds.3_setup.sh

Parameter	Function	Default value
-B path	Sets the path to the rpc.cds.3 binaries on the remote machine or machines.	The value of RPCCDS3_BIN_DIRPATH in the configuration file.
-f path	Sets the path to the rpc.cds.3 configuration file on the	The path to the rpc.cds.3 binaries on the remote machine(s).
-r mach	Adds a machine to be configured for running the test's client side.	The value of RPCCDS3_CLI_MACHINES in the configuration file.
-R mach	Adds a machine to be configured for running the test's server side.	The value of RPCCDS3_SRV_MACHINES in the configuration file.
-l	Specifies local setup only.	None.

The **rpc.cds.3_srv** executable accepts the following command line switches.

TABLE 13-6. Parameters for rpc.cds.3_srv

Parameter	Option	Specification in Configuration File	Values
Directory for calendar data files	**-D**	**not specified**	Default is "."
Calendar data file list	**-c**	**RPCCDS3_SRV_CAL_DATA**	List of file names separated with spaces.
Configuration file pathname	**-f**	**not specified**	path
Protection level for RPCs	**-1**	**RPCCDS3_PROT_LEVEL**	conn, call, pkt, integ, priv Default is priv
Number of calendars to be exported to the namespace by server	**-n**	**RPCCDS3_SRV_NCALS**	any number Default is 200
Initial sequence number of cds calendars	**-I**	**RPCCDS3_SRV_CALSEQ**	any number Default is 1

TABLE 13-7. Flags for rpc.cds.3_srv

Parameter	Option
Debug on	**-d**
Let epv default	**-e**
Replace any existing uuids use only one of -r or -n	**-r**

The **rpc.cds.3_cli** executable accepts the following command line switches.

TABLE 13-8. Parameters for rpc.cds.3_cli

Parameter	Option	Specification in Configuration File	Values
Directory for calendar data files	**-D**	**not specified**	Default is "."
Calendar data file list with spaces	**-c**	**RPCCDS3_SRV_CAL_DATA**	list of file names separated
Protocol to use	**-P**	**RPCCDS3_PROT_LEVEL**	datagram or connection
Sync file name	**-S**	**not specified**	file name
Configuration file pathname	**-f**	**not specified**	path
Last client flag	**-L**		
Protection level for RPCs Default is priv	**-1**	**RPCCDS3_PROT_LEVEL**	conn, call, pkt, integ, priv
Number of passes (cannot be used with time interval or boundary mode	**-p**	**not specified**	any number
Hours to execute (plus minutes if -m specified)	**-h**	**RPCCDS3_CLI_DEF_RUN_INTVL**	any number Default is 48
Minutes to execute (plus hours if -h specified)	**-m**	**not specified**	any number
Report interval (in passes if -p specified, in calendar imports if -r specified, in minutes if -m or -h specified)	**-i**	**RPCCDS3_CLI_DEF_REP_INTVL**	any number Default is 60 minutes or passes
Number of calendars to be exported to the namespace by server	**-n**	**RPCCDS3_SRV_NCALS**	any number Default is 200
Age in minutes for cds cache data	**-a**	**RPCCDS3_CLI_CDS_CACHE_AGE**	any number Default is 5
Initial sequence number of cds calendars	**-I**	**RPCCDS3_SRV_CALSEQ**	any number Default is 1
Starting sequence number for calendars exported by the server	**-s**	**RPCCDS3_SRV_CALSEQ**	any number Default is 1

TABLE 13-9. Flags for rpc.cds.3_cli

Parameter	Option
Boundary mode (not allowed to do boundary mode with -h -i -p or -m options)	**-b**
Debug on	**-d**
Test all bindings	**-t**

13.6.6 Compile-Time Switches for Optional Functionality

There are several areas of optional functionality available in the **rpc.cds.3** system test that can be used to expand the scope of the test or to provide additional runtime information. These areas of optional functionality are compiled into the program via the definition of tags which can be specified in either of two ways:

- On the **build** command line; for example:

```
% build -DRPCCDS3_ALL_OPTS
```

- In the environment variable **CENV**; for example (in a C shell):

```
% setenv CENV RPCCDS3_ALL_DEBUGS
% build
```

The default **build** flag value is **RPCCDS3_ALL_OPTS**.

The table below lists the supported compiler flags, the functionality that they control, and the operation of the test depending on whether the flag is or is not specified.

TABLE 13-10. Compile-Time Switches for rpc.cds.3

Flag	Functionality	Test Operation
RPCCDS3_KEEP_SYMBOLS	Keeps debugging symbols in compiled objects	If defined, code is compiled with debugging symbols; else debugging symbols usually stripped from objects.
RPCCDS3_ID_REFR_DEBUG	Print ID refresh messages	If defined, code is compiled to cause messages about identity maintenance activity to be printed.
RPCCDS3_ADD_DUMP_ROUTINES	Dump data structures	If defined, server code is compiled to dump contents of data structures.
RPCCDS3_SRV_REPORTING	Turns on server status reporting	If defined, server reports on call requests received, calls passed and failed, id refreshes, and time of last id refresh at an interval specified by RPCCDS3_CLI_DEF_REP_INTVL in configuration file. If not defined, server reports only upon receipt of SIGQUIT.
RPCCDS3_AUTO_REFRESH	Turns on automatic identity refreshing	If defined, server spawns a thread that will maintain the authentication ticket by waking up prior to the ticket's expiration time, and refresh the ticket. If not defined, server will lose its network credentials when its tickets expire as dictated by cell security policy.
RPCCDS3_DO_LOGIN	Causes server to assume its own identity	If defined, server will make security calls to establish network credentials. If not defined, server will run with invoker's credentials.
RPCCDS3_ALL_OPTS	turns on all optional code	Has the same effect as specifying RPCCDS3_SRV_REPORTING, RPCCDS3_AUTO_REFRESH, and RPCCDS3_DO_LOGIN.
RPCCDS3_ALL_DEBUGS	turns on all debugging options	Has the same effect as specifying RPCCDS3_ID_REFR_DEBUG, RPCCDS3_KEEP_SYMBOLS, and RPCCDS3_ADD_DUMP_ROUTINES.

Specifying server reporting can provide useful information about the server side of the test. The login and auto refresh flags allow the scope of the test to be expanded to include the Security component, especially if the policy for the test run is set to expire tickets frequently, and a high protection level is used on RPC calls. The login and auto refresh options are also useful if the test is intended to run for extended durations.

13.6.7 Customizing the Configuration File

Setting up to run the setup script for the **rpc.cds.3** system test consists of one step, namely customizing the

/dcetest/dcelocal/test/tet/system/rpc/ts/rpc.cds.3/rpc.cds.3.conf

configuration file. The present section describes this step.

The scripts and programs that make up the **rpc.cds.3** system test get most of the information they need from a single configuration file whose default name is **rpc.cds.3.conf**. If the file is named something other than the default, the name can be specified to the test via the **-f** command line option (see below) or via the environment variable **RPCCDS3_CONF**.

The information normally contained in this file can be split up roughly into two categories: default runtime parameters, and environment information.

Examples of default runtime information in the file are: the time duration a test should run; the names of machines on which clients will be run; etc.

Examples of environment information stored in this file are: the name of the CDS namespace entry to which the server exports its bindings; the name of the client and server principals; etc.

Before running the test, it is important to inspect the configuration file to see if any changes should be made for the site at which the test is to be performed. This is particularly important in regard to the environment configuration information. For example, you may wish to use a different client or server principal, a different CDS entry name, etc. All of these things, if they are to be changed, must be changed in the configuration file before running the test.

Note that all machines that the test is to be run on must have identical **rpc.cds.3.conf** files.

13.6.8 Format of the Configuration File

The contents of the test configuration file consist of text lines conforming to normal Bourne shell syntax.

Note, however, the following restriction. The configuration file, as implied above, is read by shell scripts, and by the **rpc.cds.3_cli** and **rpc.cds.3_srv** binary programs. In order to simplify the routine used by these programs to read the file, lines that set values for the **rpc.cds.3_cli** and **rpc.cds.3_srv** programs *must* be in one of the two followingformats:

```
<string>=<string1>  # NOTE: in this case, string1 cannot
                    #       contain any spaces.
```

or:

```
<string>="<string1>"  # NOTE: in this case string1 may
```

```
#       contain spaces.
```

Any lines that are not in this format will either be ignored by the routine (**rd_conf( )**, in the file **rdconf.c**) that the client and server use to read the configuration file, or will generate an error. Comments are begun by a ‘‘#’’ character anywhere on a line, as shown above, and continue to the end of the line.

13.6.9 Contents of the Configuration File

The assignments in the configuration file as it is shipped represent the minimum set required to run the tests scripts and programs. You may add to the configuration file, but you should not remove any of the original assignments.

The information in the configuration file determines the way that your Security and CDS namespaces are set up. This being the case, you may want to modify the configuration information to tailor the namespace to your preferences. If you do not want to use the default values in the configuration file for the client or server principal name, CDS directory, CDS name, or for any of the other configuration file variables, you will have to modify the configuration file in accordance with your preferences before running the setup script.

TABLE 13-11. Contents of Configuration File

Variable in Config File	Description	Default Value as Shipped
RPCCDS3_PROT_LEVEL	Default protection level	none
RPCCDS3_CLI_PRINC_NAME	Client principal name	rpc.cds.3_cli
RPCCDS3_CLI_INIT_PW	Client initial password	"rpc&cdsC"
RPCCDS3_CLI_KEYTAB_DIRPATH	Directory for client keytab	/tmp file
RPCCDS3_CLI_KEYTAB_FN	Client keytab file name	rpc.cds.3_cli.keytab
RPCCDS3_CLI_MACHINES	Client machine names	"machine1 machine2"
RPCCDS3_CLI_DEF_RUN_INTVL	Client interval to run	48 (hours)
RPCCDS3_CLI_DEF_REP_INTVL	Client report interval	60 (minutes)
RPCCDS3_CLI_SYNC_DELAY	Client start delay time after finding sync file	60 (seconds)
RPCCDS3_CLI_START_DELAY	Startup delay	180 (seconds)
RPCCDS3_CLI_CDS_CACHE_AGE	Maximum time that data can cached	5 (minutes)
RPCCDS3_SRV_PRINC_NAME	Server principal name	rpc.cds.3_srv
RPCCDS3_SRV_GROUP_NAME	Server Group	subsys/dce/cds-test-group

Variable in Config File	Description	Default Value as Shipped
RPCCDS3_SRV_INIT_PW	Server initial password	"rpc&cdsS"
RPCCDS3_SRV_KEYTAB_DIRPATH	Directory for server keytab file	/tmp
RPCCDS3_SRV_KEYTAB_FN	Server keytab file name	rpc.cds.3_srv.keytab
RPCCDS3_SRV_MACHINES	Server machine names	"machine1"
RPCCDS3_SRV_CDS_NAME	Server interface name	/.:/test/systest/srv_ifs /rpccds3_if
RPCCDS3_SRV_OBJ_DIR	Directory for server objects	/.:/test/systest /srv_objs/rpccds3
RPCCDS3_SRV_CAL_DATA	Calendar data file names	‘‘rpc.cds.3_cal1.data rpc.cds.3_cal2.data rpc.cds.3_cal3.data’’
RPCCDS3_SRV_CALSEQ	Starting calendar sequence number	1
RPCCDS3_SRV_NCALS	Number of calendar objects	200
RPCCDS3_SRV_CDS_DIR_ACL	Directory ACL	rwditca
RPCCDS3_SRV_CALL_DELAY	Server call duration	2 (seconds)
RPCCDS3_SRV_CDS_IO_ACL	Initial object ACL	rwdtc--
RPCCDS3_SRV_MAX_CALLS	Maximum concurrent calls for server	5
RPCCDS3_SRV_MAX_EXEC	Maximum concurrent execs for server	1
RPCCDS3_BIN_DIRPATH	Directory for binaries	/dcetest/dcelocal/test /tet/system/rpc /ts/rpc.cds.3
RPCCDS3_TMP_DIRPATH	Directory for tmp files	/dcetest/dcelocal/tmp
RPCCDS3_LOG_DIRPATH	Directory for log files	/dcetest/dcelocal/status

13.6.10 Setting Up to Run the RPC-CDS System Test

Before you can run the **rpc.cds.3** system test, certain objects in the CDS namespace and certain accounts in the Security registry must exist. The following table describes these necessary items, as well as the variables in the configuration file relevant to the creation of these objects, and the default values of these variables (i.e. the values in the file as shipped). Note that the required objects are created from the specified values automatically by the **rpc.cds.3_setup.sh** script described below.

The configuration file describes the parameters and environment for running the **rpc.cds.3** system test. Customization of this file for your site characteristics and testing requirements is the only prerequisite for running the **rpc.cds.3_setup.sh** script, which will setup your DCE cell to run the **rpc.cds.3** system test.

TABLE 13-12. Objects Required by the rpc.cds.3 System Test

DCE Object Needed	Variable in config file	Default value as shipped
Server principal and account	**RPCCDS3_SRV_PRINC_NAME**	rpc.cds.3_srv
Client principal and account	**RPCCDS3_CLI_PRINC_NAME**	rpc.cds.3_cli
Group for the server test	**RPCCDS3_SRV_GROUP_NAME**	subsys/systest/cds_test
Server key file	**RPCCDS3_SRV_KEYTAB_FN**	rpc.cds.3_srv.keytab
Server key file directory	**RPCCDS3_SRV_KEYTAB_DIRPATH**	/tmp
Client key file	**RPCCDS3_CLI_KEYTAB_FN**	rpc.cds.3_cli.keytab
Client key file directory	**RPCCDS3_CLI_KEYTAB_DIRPATH**	/tmp
CDS directory for server interface object	**RPCCDS3_SRV_CDS_NAME**	/.:/test/systest/srv_ifs/rpccds3_if
CDS directory for calendar objects exported by server	**RPCCDS3_SRV_OBJ_DIR**	/.:/test/systest/srv_objs/rpccds3

13.6.11 Running the rpc.cds.3_setup.sh Setup Script

Make sure that the machine on which **rpc.cds.3_setup.sh** will be run can **rsh** to the client machines for the test.

Note that both the setup script and the test assume that you have a DCE cell up and running.

If you wish to use a configuration file with a name other than **rpc.cds.3.conf**, then you can specify the desired name by assigning it to the environment variable **RPCCDS3_CONF** before running the setup script, or the name can be specified on the command line with the **-f** option.

In order to run the **rpc.cds.3_setup.sh** script, you must **dce_login** as the **cell_admin** principal. This is necessary because you will be creating DCE accounts during the setup, and this requires special privileges. During execution of the setup script you will be prompted twice for the **cell_admin** password. If you want to skip these prompts, you must modify the **rpc.cds.3_sec_util.sh** script; see the comments to the shell function **rpccds3_sec_add_account** there for details on the modifications required. After you are **dce_login**ed, make sure that the path to the directory containing the **rpc.cds.3** scripts and the configuration file is in your execution path.

Now you can simply type:

rpc.cds.3_setup.sh -B *path*

(where *path* is the path to the **rpc.cds.3** binaries and scripts on the client machines). The setup script assumes that the configuration file is in the same directory as are the **rpc.cds.3** binaries on each client test machine. If this is not the case, then the path to the configuration file (which *must* be the same on all test machines) must be specified with the **-f** option.

Enter the **cell_admin** password when prompted, and, if no errors are reported, your cell will be set up to run the **rpc.cds.3** system test. Note that the setup can be repeated as many times as necessary without adverse effect.

13.6.12 Starting the Servers

Once the setup script has been successfully executed, the servers must be started. This is done as follows.

On the machines specified in the configuration file or through the **-R** switch on the command line, you must run **rpc.cds.srv** using the appropriate server options described in the ‘‘Test Options’’ section. The output from **rpc.cds.3_srv** should be redirected into a file for future reference.

For example:

rpc.cds.3_srv -I 1 -n 20 > /dcetest/dcelocal/tmp/rpc.cds.3_srv.log

The above command specifies 20 calendars, starting with the sequence number 1. The rest of the parameters have been specified in the example configuration file.

13.6.13 Starting the Clients

Starting the clients is done similarly to the servers.

On the machines specified as clients, you must run **rpc.cds.3_cli** using the client options described in the ‘‘Test Options’’ section. You can start multiple clients on the samemachine. Again, you should redirect the output to a file for future reference.

For example:

rpc.cds.3_cli -I 1 -n 20 -P datagram > /dcetest/dcelocal/tmp/cli_log*pid***.1**

The above command specifies 20 calendars, starting with sequence number 1. The **ncadg_ip_udp** protocol is also specified on this command line.

13.6.14 Analyzing the Results

If you are not running the test in boundary mode, then after all the clients have exited, you can generate a report of the results of the test by executing the following commands in a Bourne or Korn shell:

```
$ cd logdir

$ for i in 'ls cli_logpid.*'
> do
> grep -v READY $i | awk -f bindir/rpc.cds.3_gen_summ.awk >> runpid.summ
> done

$ awk -f bindir/rpc.cds.3_gen_rep.awk runpid.summ > runpid.results
```

where *pid* is the process id of the driver script.

Note that this sequence of commands can be run at any time during the test run to obtain a report on the current status of the clients as of the last time that reports were generated. If up-to-the-minute status is desired, then executing:

```
kill -3 <pids>
```

(where <*pids*> is the process ids of all the clients running on a particular machine) should be run on each machine in the test to cause the clients on that machine to generate a current status line in the log file. Then the **for** loop and **awk** command combination described above can be used to generate a current status report.

13.6.15 Implementation Notes

As shipped, the **rpc.cds.3** test exerts stress on the CDS clerk and (indirectly) on the CDS clearinghouse. Other stresses can be induced by running the test in a manner different from the way it runs as shipped. For example:

- If you want to stress the system and the CDS clearinghouse by running multiple CDS clerk processes, you can invoke different test clients with different UIDs on the same machine. The CDS advertiser process will start a new CDS clerk for each different UID for which a CDS operation is requested.
- If you want to stress the CDS clerk caching and ACL mechanisms, you can run a number of test clients with different UIDs accessing the same object or objects.

If you wish to have more than one server exporting objects to the namespace for this test, it is a good idea to use a different configuration file for each server, each specifying a different server CDS name. This makes administration of the namespace easier because the RPC API does not (for a number of reasons) provide a way to remove some bindings from a CDS entry; all or none must be removed. This means that if two servers export bindings to the same namespace entry, and one of the servers later terminates, you

cannot remove that server's bindings from the entry while preserving the other server's bindings. On the other hand, leaving the entry as it is means that clients can still import (and attempt to use) the invalid bindings. The only thing that can be done in such a situation is remove, and then re-export, all of the bindings.

13.6.16 Runtime Error Handling

The spurious test failure scenario described earlier for **rpc.sec.2** can also occur with **rpc.cds.3**, for the same reason: the **rpc.cds.3** clients do not currently perform any error handling of the communication status value returned from a remote call. This scenario can probably be avoided if you add code to handle the three following errors:

- **rpc_s_server_too_busy**

  (Returned only by TCP clients.) The server does not have a thread available to service the client request, nor does it have space in any call request buffer to queue the request. When a test client receives this error, it will go into a tight loop as described in the previous section, making RPCs and continuing to receive this same status, until sufficient resources are freed at the server to permit the call to be serviced or queued. While testing did not prove this looping to have a significant impact on the overall success rate of the TCP clients, it is wasteful of CPU cycles. One way to avoid the tight looping would be to have the TCP clients wait for a few seconds if they receive this status before doing anything. Another approach would be to allocate more server threads to begin with, and thus avoid the situation altogether.

- **rpc_s_connection_closed**

  A protocol error has occurred in the connection to the server. This means (with a connection-oriented protocol) that the binding to the server has become permanently useless, and the thread in the server runtime that listens for connection-oriented protocol requests is probably unavailable, so that no connection-oriented protocol calls will succeed. The only remedy for this condition is for the server to re-export its binding handles.

- **rpc_s_auth_tkt_expired**

  The client's network credentials (i.e., ticket) have expired. The client thread receiving this error can recover from the situation by notifying the ticket maintainance thread that it should now refresh the ticket.

Appendix A. File and Path Names Cross-Reference

This appendix lists the pathnames of many files mentioned in the DCE documentation.

A.1 Threads Files

Filename	Default Location
exc_handling.h	*dceshared/***share/include**
pthread.h	*dceshared/***include**
cma_stdio.h	*dceshared/***share/include**

A.2 RPC Files

Filename	Default Location
dce.rc	*dcelocal*/**etc**
dcecds.cat	*dceshared/***nls/msg/${LANG}**
dce_error.h	*dceshared*/**share/include**
dcerp.cat	*dceshared/***nls/msg/${LANG}**
ep.idl	**dce/ep.idl**
file.ext	*dceshared*/**share/include**
idl	**/pbin/idl**
id_base.h	*dceshared*/**share/include**

idlbase	*dceshared*/**share/include**
idlbase.h	*dceshared*/**share/include**
idl.cat	*dceshared*/**nls/msg/${LANG}**
nbase.acf	*dceshared*/**include**
nbase.idl	*dceshared*/**include**
nidl_to_idl	*dceshared*/**bin**
rpccp	*dceshared*/**bin**
rpcexc.h	*dceshared*/**share/include/dce**
rpc.h	*dceshared*/**share/include**
sec_login.h	*dceshared*/**share/include**
uuidgen	*dceshared*/**bin**
uuidgen.cat	*dceshared*/**nls/msg/${LANG}**
uuid.h	*dceshared*/**share/include**

A.3 CDS Files

Filename	**Default Location**
cds_attributes	*dcelocal*/**etc**
cdsadv	*dceshared*/**bin**
cds_cache.*nnnnnnnn*	*dcelocal*/**var/adm/directory/cds**
cds_cache.version	*dcelocal*/**var/adm/directory/cds**
cdsclerk	*dceshared*/**bin**
cdscp	*dceshared*/**bin**
cdsd	*dceshared*/**bin**
cds_files	*dcelocal*/**var/directory/cds**
cds_globalnames	*dcelocal*/**etc**
clearinghouse-name.checkpoint*nnnnnnnn*	*dcelocal*/**var/directory/cds**
clearinghouse-name.tlog*nnnnnnnn*	*dcelocal*/**var/directory/cds**
clearinghouse-name.version	*dcelocal*/**var/directory/cds**

A.4 GDA Files

Filename	Default Location
gda_child	*dceshared/***bin**
gdad	*dceshared/***bin**

A.5 GDS Files

Filename	Default Location
gdscache	*dceshared/***bin**
gdscacheadm	*dceshared/***bin**
gdscmxl	*dceshared/***bin**
gdscstub	*dceshared/***bin**
gdsditadm	*dceshared/***bin**
gdsdsa	*dceshared/***bin**
gdsipcchk	*dceshared/***bin**
gdsstep	*dceshared/***bin**
gdsstub	*dceshared/***bin**
gdssysadm	*dceshared/***bin**
osiforminfo	*dcelocal/***var/adm/directory/gds/conf**
nsapmacros	*dcelocal/***var/adm/directory/gds/adm**

A.6 DTS Files

Filename	Default Location
dce	**/usr/include/dce**
utc.h	*dceshared***/share/include**
dts	*dcelocal***/usr/examples**
dtscp	*dceshared***/bin**
dtsd	*dceshared***/bin**
dtsprovider.idl	*dceshared***/examples/dts**

dts-servers	**/.:/subsys/dce**

A.7 Security Files

Filename	Default Location
acct.h	*dceshared*/**share/include/dce**
aclbase.h	*dceshared*/**share/include/dce**
acl_edit	*dcelocal*/**bin**
binding.h	*dceshared*/**share/include/dce**
daclif.h	*dceshared*/**share/include/dce**
group	**/etc**
keymgmt.h	*dceshared*/**share/include/dce**
kdestroy	*dcelocal*/**bin**
klist	*dcelocal*/**bin**
kinit	*dcelocal*/**bin**
krb5cc_*unix_id*	**/tmp**
misc.h	*dceshared*/**share/include/dce**
passwd	**/etc**
passwd_export	*dceshared*/**bin**
pe_site	*dcelocal*/**etc/security**
pgo.h	*dceshared*/**share/include/dce**
policy.h	*dceshared*/**share/include/dce**
rdaclif.h	*dceshared*/**share/include/dce**
rgybase.h	*dceshared*/**share/include/dce**
rgy_data	*dcelocal*/**var/security**
rgy_edit	*dcelocal*/**bin**
sec_admin	*dceshared*/**bin**
sec_create_db	*dcelocal*/**bin**
secd	*dcelocal*/**bin**
secidmap.h	*dceshared*/**share/include/dce**
sec_login.h	*dceshared*/**share/include/dce**
su	*dcelocal*/**bin**

v5srvtab	**/krb5**

A.8 DFS Files

Filename	Default Location
admin.bak	*dcelocal*/**var/dfs**
admin.bos	*dcelocal*/**var/dfs**
admin.fl	*dcelocal*/**var/dfs**
admin.ft	*dcelocal*/**var/dfs**
admin.up	*dcelocal*/**var/dfs**
bak	*dceshared*/**bin**
BakLog	*dcelocal*/**var/dfs/adm**
bakserver	*dcelocal*/**bin** and *dceshared*/**bin**
bkdb.*	*dcelocal*/**var/dfs/backup**
bos	*dcelocal*/**bin** and *dceshared*/**bin**
BosConfig	*dcelocal*/**var/dfs**
BosLog	*dcelocal*/**var/dfs/adm**
bosserver	*dcelocal*/**bin** and *dceshared*/**bin**
butc	*dceshared*/**bin**
CacheInfo	*dcelocal*/**etc**
CacheItems	*dcelocal*/**var/adm/dfs/cache**
cm	*dcelocal*/**bin** and *dceshared*/**bin**
core.*	*dcelocal*/**var/dfs/adm**
dfsatab	*dcelocal*/**var/dfs**
dfsbind	*dcelocal*/**bin** and *dceshared*/**bin**
dfsd	*dcelocal*/**bin** and *dceshared*/**bin**
dfsexport	*dcelocal*/**bin** and *dceshared*/**bin**
DFSLog	*dcelocal*/**var/adm/dfs/cache**
dfstab	*dcelocal*/**var/dfs**
FileLog	*dcelocal***var/dfs/adm**
FilesetItems	*dcelocal*/**var/adm/dfs/cache**
fldb.*	*dcelocal*/**var/dfs**

FlLog	*dcelocal*/**var/dfs/adm**
flserver	*dcelocal*/**bin** and *dceshared*/**bin**
fms	*dceshared*/**bin**
FtLog	*dcelocal*/**var/dfs/adm**
fts	*dcelocal*/**bin** and *dceshared*/**bin**
ftserver	*dcelocal*/**bin** and *dceshared*/**bin**
fxd	*dcelocal*/**bin** and *dceshared*/**bin**
newaggr	*dcelocal*/**bin** and *dceshared*/**bin**
NoAuth	*dcelocal*/**var/dfs**
RepLog	*dcelocal*/**var/dfs/adm**
repserver	*dcelocal*/**bin** and *dceshared*/**bin**
salvage	*dcelocal*/**bin** and *dceshared*/**bin**
SalvageLog	*dcelocal*/**var/dfs/adm**
scout	*dceshared*/**bin**
TapeConfig	*dcelocal*/**var/dfs/backup**
TE_*device_name*	*dcelocal*/**var/dfs/backup**
TL_*device_name*	*dcelocal*/**var/dfs/backup**
upclient	*dcelocal*/**bin** and *dceshared*/**bin**
UpLog	*dcelocal*/**var/dfs/adm**
upserver	*dcelocal*/**bin** and *dceshared*/**bin**
V*n*	*dcelocal*/**var/adm/dfs/cache**

Appendix B. DCE Abbreviations List

This appendix contains a list of abbreviations and acronyms used in DCE, both in the DCE source code and in the documentation.

Note that the distinction in many abbreviations and acronyms between the upper- and lower-case version is arbitrary. For example, the abbreviation ‘‘XOM’’ is spelled thus when cited in documentation as the component name; the same abbreviation appears in lowercase in library routine and constant names in source code (it has the same meaning, however, in both forms). In other words, although an attempt has been made to preserve the customary case of all abbreviations, the list below should be regarded as being case-insensitive.

Note also that the following list is of abbreviations only; it is not a general DCE glossary.

B.1 A

acb	association control block (RPC internal)
acf	attribute configuration file (RPC)
acl	Access Control List (Security)
acct	account
ACSE	Association Control Service Element
addr	address
admin_gd	OSF DCE Administration Guide
command_ref	OSF DCE Command Reference
AEP	Application Environment Profile (see ISP & IEEE 1003.10, .11)
AES	OSF Application Environment Specification
afl	aggregate fileset list (DFS LFS)

AFNOR Association Francaise de Normalisation. French ISO member body

afs Andrew filesystem (both Carnegie and Mellon had first names Andrew) (DFS)

agfs aggregate filesystem (DFS)

AIX Trademark name for IBM OS, derived from UNIX System V

alloc allocate

ANSI American National Standards Institute, US member of ISO

API application programming interface

APP Application Portability Profile. NIST environment for application portability

app_gd OSF DCE Application Development Guide

app_ref OSF DCE Application Development Reference

ASN (ASN.1) abstract syntax notation: ISO/ANSI Std. 8824/8825 Data format for various data types

assoc association

attr attribute

auth authentication (Security)

authn authentication (Security)

authz authorization (Security)

AVA attribute value assertion (XDS/XOM/GDS)

B.2 B

BIND Berkeley Internet Naming Daemon (DNS)

bos Basic Overseer Server (BOS Server) (DFS)

BOSS Basic Overseer Server (BOS Server) (DFS)

bosserver Basic Overseer Server (BOS Server) (DFS)

butc back up tape coordinator (DFS) (‘‘backup tape controller’’ in some specs)

butm back up tape manager (DFS)

B.3 C

ccall	client call
CCITT	International Telegraph & Telephone Consultative Committee (of ITU)
CDS	Cell Directory Service
cdsadv	the CDS advertiser
cdspi	CDS's (DCE-private) programming interface
cdsta	CDS transaction agent protocol; a DCE-private protocol between the CDS server and clerk. Also used among CDS servers.
cf	configuration
C-ISAM	C-based index sequential access method database; used to hold the GDS DIB
CLNS	Connectionless network service (OSI Layer 3 Protocol)
cm	cache manager (DFS)
cma	Concert Multithread Architecture (name for earlier DCE Threads interface)
cn	connection (connection-oriented RPC protocol)
com	common
cond	condition variable (Threads)
CONS	Connection oriented network service (OSI Layer 3 Protocol)
CPIO	Tape I/O format, interchange format Std. in IEEE 1003.1 (POSIX)
CPU	central processing unit
crc	cyclic redundancy check (RPC internal)
cred	credentials (Security) (RPC internal)
cs	character set or code set
CSMA/CD	Carrier Sense, Multi-access/Collision Detection (see IEEE 802.3)
ctl	control
ctx	context

B.4 D

dap	directory access protocol; used between the GDS DUA and DSA
db	database

DBMS	Data Base Management System
dcache	data cache
DCE	Distributed Computing Environment
dced	DCE Host Daemon
DECdns	Digital Distributed Naming Service
DECdts	Digital Distributed Time Synchronization Service
DES	Digital Encryption Standard (Security)
DFS	Distributed File Service
dg	datagram (connectionless RPC protocol)
DIB	directory information base; the GDS database
Dir-X	the Siemens/Nixdorf implementation of X.500 that serves as a base for GDS
DIS	ISO Draft International Standard (DP accepted, second technical ballot)
DIT	directory information tree; the logical structure of the GDS database
dn	DECnet network address family services
DN	Distinguished name (GDS)
DNS	Domain Name Service DEC DNA Name Server; the base technology for CDS
dnspi	original name of cdspi
dnsta	DNS transaction agent interface; the original name of CDSTA
DP	ISO Draft Proposed Standard (has started first technical ballot)
ds	XOM directory service
DSA	directory service agent; the GDS name for the directory server code
dsP	ds private extension
DSP	directory service agent protocol; a DSA/DSA protocol in GDS
dsm	distributed storage manager (underlies the epdb)
DTS	Distributed Time Service
DUA	directory user agent; the GDS name for the directory client code

B.5 E

elt	element
ep	endpoint

epdb	endpoint database
epv	endpoint vector; entry point vector
exc	exception
exp	expiration

B.6 F

fd	file descriptor
fifo	first-in, first-out (the standard model of a queue)
FIPS	Federal Information Processing Standard (US Government)
fldb	Fileset Location Database (or FLDB) (DFS)
flserver	Fileset Location server (DFS)
FL server	Fileset Location server (DFS)
fptgt	foreign privilege ticket-granting ticket
FTP	File transfer protocol (DDN- TCP/IP application) Functional Standards
ftserver	Fileset Server (DFS)
fxd	File Exporter (formerly known as ‘‘Protocol Exporter’’, px) (DFS)

B.7 G

GDA	Global Directory Agent
GDS	Global Directory Service
gen	generate
GOSIP	Government OSI Profile (US FIPS 146, UK, EC versions) Gateway System that interfaces one network to another
grp	group

B.8 H

HP/UX	Trademark name for Hewlett-Packard OS, derived from UNIX

B.9 I

iapl*	interface application programming language; interface used between XDS/XOM and GDS
icl	in core logging
id	identifier
IDL	Interface Definition Language (RPC)
IDU	interface data unit
IEEE	Institute of Electrical and Electronics Engineers. Professional organization
IEEE	1003.0 Guide to POSIX Open Systems Environment. POSIX suite
IEEE	1003.1 Operating System interface Std. (ISO 9945). POSIX suite
IEEE	1003.2 Shell and Utilities document. POSIX suite
IEEE	1003.3 Test Methods. POSIX Suite (see also PCTS)
IEEE	1003.4 Real Time extensions to 1003.1. POSIX suite
IEEE	1003.4a Threads Extension to 1003.1. POSIX suite
IEEE	1003.5 Ada API for IEEE 1003.1 Std.
IEEE	1003.6 Security extensions for POSIX
IEEE	1003.7 System Administration services for POSIX
IEEE	1003.8 POSIX Distribution Services (RPC, XTI, TFS, FTAM API)
IEEE	1003.9 FORTRAN API for IEEE 1003.1 Std.
IEEE	1003.10 Supercomputing AEP (Application Environment Profile)
IEEE	1003.11 Transaction Processing AEP (Application Environment Profile)
IEEE	1003.14 Multiprocessor AEP (Application Environment Profile)
IEEE	1201.1 High level (toolkit) windowing project
IEEE	1201.2 Windowing drivability guide
IEEE	802.3 ISO/ANSI Std.LAN OSI layer 1 CSMA/CD (Ethernet)
IEEE	802.4 ANSI/IEEE Std.Token Bus LAN OSI layer 1 (a la MAP)
IEEE	802.5 ANSI/IEEE Std.Token Ring LAN OSI layer 1 (a la IBM)
IEEE	Standards Board. Authorized by ANSI as a Standards development organization
if	interface
info	information
init	initialize

inq	inquire
intro	Introduction to OSF DCE (book)
IP	IP network address family services
IPC	Interprocess Communications (function in IEEE 1003.4)
IS	ISO International Standard (DIS accepted)
ISAM	Indexed Sequential Access Method. No standards to date, except COBOL
ISO	International Organization for Standards (see also JTC1)

B.10 K

kdc	Key Distribution Center (Security)
kutils	kernel utilities

B.11 L

LAN	Local Area Network (such as ISO/IEEE 802.3...)
LFS	Local Filesystem (DFS)
lifo	last-in, first-out (the standard model of a stack)

B.12 M

mepv	manager entry point vector (RPC)
mgmt	management
mgt	management services
MHS	Message Handling Service (X.400 name for mail service)
msg	message
mutex	mutual exclusion lock (Threads)

B.13 N

NAF	network address family
NAS	Network Application Support
NDR	network data representation (RPC)
NFS	Network File System (SUN specification)
NIST	National Institute of Standards and Technology (formerly NBS)
np	non portable (Threads routine name suffix)
ns	nameservice; naming service
NSAP	Network Service Access Point (OSI)
NSI	Name Service Interface (RPC)
NTP	Network Time Protocol

B.14 O

obj	object
OID	object identifier (GDS, CDS)
om	X/Open object management (XOM)
op	operation
org	organization
OS	Operating System
OS	Interface ISO DIS9945. IEEE 1003.1 Std.operating system service API (POSIX)
OSF	Open Software Foundation. Consortium developing AES, OSF/1 and tests
OSF/1	First release of OSF's system implementation
OSF/Motif	OSF's Windowing environment: toolkit and style guide
osi	operating system independent (DFS)
OSI	Open System Interconnect network address family services (communication protocols) (GDS). ISO 7498-1984
OSS	OSI Session Service

B.15 P

PAC	Privilege Attribute Certificate (Security)
pag	process authentication group (DFS)
PDU	protocol data unit
perm	Permission
pgo	principal/group/organization (Security)
pkt	packet (RPC)
pmax	DECstation 3100 platform
port_gd	OSF DCE Build Environment, Porting, and Testing Guide
POSIX	Suite of API standards (see IEEE 1003, OS interface, shell, admin., UPE)
prin	principal (Security)
protseq	protocol sequence (RPC)
psap	presentation service access point; the address of a GDS DUA
ptgt	privilege ticket-granting ticket (Security)
pthread	DCE Threads (POSIX 1003.4a conformant)
pvt	private (Security)
px	protocol exporter (alternatively fxd) (DFS)

B.16 R

rcx	recovery tests
RDN	relative distinguished name; the GDS name for an attribute/value pair
relnotes	OSF DCE Release Notes
repl	replica/replication
repserver	Replication Server
rgy	registry (Security)
rios	IBM RISC System/6000 platform
RISC	Reduced Instruction Set Computer (as opposed to CISC)
ROS	remote operation service layer; a collection of networking support routines used to implement GDS
ROSE	Remote Operation Service Elements

RPC Remote Procedure Call

rpcd remote procedure call daemon (also known as the ‘‘endpoint mapper’’) (not supported in DCE 1.2.1) (RPC)

rpc_ss RPC stub support

B.17 S

SAP service access point

sautils stand alone utilities (DFS LFS)

scache status cache

scall server call

sec security; Security service

SGML Std.Generalized Markup Language. ISO 8879-1986 - page formatting

sm state machine

SQL Structured Query Language. ISO/ANSI Std.X3.135-1986. Relational DBMS API

ssr stub support routine (RPC)

svc service; serviceability

SVID System V Interface Definition. Specification for AT&T’s UNIX System V

sys system

s5 System V (a popular implementation of UNIX)

B.18 T

tar Tape archive format, interchange format std. in IEEE 1003.1 (POSIX)

tcb task control block; thread control block (Threads)

TCP transmission control protocol - used by the RPC CN protocol (RPC)

TCP/IP Transmission control protocol, Internet protocol: US DoD network (DDN)

TDF Time Differential Factor

tech_supp OSF DCE Technical Supplement

TET Test Environment Toolkit

tgs	ticket-granting service (Security)
tgt	ticket granting ticket (Security)
thr	threads (do not use thd) (Threads)
tkc	token cache (DFS)
tkm	token manager (DFS)
tkt	ticket (Security)
tlr	(auth) trailer (Security)
tpq	thread pool queue
tsap	transport service access point (GDS)
twr	tower
twrfl	tower floor
T1	Standard for high bandwidth WAN connection

B.19 U

ubik	the library of routines used to implement the FLDB
UDP/IP	User Datagram Protocol/Internet Protocol - used by the RPC DG protocol (RPC)
UFS	UNIX file system (also known as the ‘‘Berkeley file system’’ and the ‘‘fast file system’’)
ULTRIX	Trademark name for Digital OS, derived from Berkeley
UNIX	Trademark name for AT&T operating system product (System V)
users_gdref	OSF DCE User’s Guide and Reference
utc	coordinated universal time (DTS)
util	utility
UUID	Universal Unique Identifier

B.20 V

VFS	virtual file system
VFS+	OSF’s extension to VFS (necessary for DFS)
vnops	vnode operations

volreg volume registry

B.21 W

WAN Wide area network (as in world-wide, usually synchronous)

way ‘‘who are you’’ (RPC protocol)

B.22 X

xaggr extended aggregate (DFS)

XDS X/Open Directory Service

X lib Low level windowing API, linked to X-11. X Window System, X3H3.6 Std.

XOM X/Open OSI-Abstract-Data Manipulation

XPG4 X/Open Portability Guide 4

XTI X/Open Transport Interface (network stack independent)API. Also IEEE 1003.8 project

xvnode extended vnode

xvolume extended volume

B.23 Z

zlc zero link count