**OSF DCE SIG** **H. Melman (OSF)**
**Request For Comments: 42.1** **January 1994**

# DCECP FUNCTIONAL SPECIFICATION

## 1. INTRODUCTION

A goal of DCE 1.1 is to make DCE easier to administer. One of the current problems with DCE is the fact that there are many different control programs for different components, with different syntaxes. Often, more than one must be used to perform certain operations, such as adding a user to a cell. This Functional Specification describes a new program, known as the *DCE Control Program*, abbreviated `dcecp`, providing an administrative user interface that will augment the use of the existing programs, at least for commonly performed tasks.

While the licensees and users of DCE 1.0 have found it to be a useful distributed computing platform, they have also found it to be a difficult environment to administer. The feedback we have heard in this area has made the simplification of DCE administration an important goal of DCE 1.1.

The fact that DCE is composed of several different components originating from different vendors, is readily apparent when viewing the administrative tools. Many of the difficulties in administering DCE arise from the need to use many different tools that behave inconsistently and operate at a very low level relative to administrative **tasks**[1] (e.g., add a user to a cell, configure a client machine, backup, etc.). This functional specification describes a new administrative tool known as `dcecp` which can perform many of the operations of existing control programs via a new consistent command-line interface. It is expected that DCE **administrators** will use `dcecp` instead of the existing **control programs** to perform most daily operations.

In DCE 1.1, `dcecp` will include functionality from the following control programs: `rpccp`, `cdscp`, `rgy_edit`, `acl_edit`, `dtscp`, and `sec_admin`. That is, it will be able to manipulate data (e.g., directories, entries, groups, principals, accounts, ACL's, etc.) stored in the various databases (e.g., the registry, clearinghouses, ACL managers, etc.). It will also be able to perform certain management operations (e.g, server shutdown, ping, get status, etc.). An important feature of `dcecp` is that it will provide a script language including but not limited to: variables, control structures, and the ability to create new procedures. This will allow administrators to create their own task **commands**. The first version of `dcecp` will include several such task commands implemented as `dcecp` scripts.

The `dcecp` Language will be based on **Tcl** (pronounced ''tickle''), the embeddable Tool Control Language developed at the University of California at Berkeley. The `dcecp` commands will be implemented as Tcl commands in a Tcl interpreter. This document assumes knowledge of Tcl and to a limited extent, its syntax.

### 1.1. Changes Since Last Publication

A number of changes have been made since this document was originally published in [RFC 42.0]. The changes that were made are:

(a) Various name changes, including changing the name of the program from DCE Shell to the DCE Control Program. The name was changed due to confusion with typical Unix shells such as `sh`, `ksh`, and `csh`. Here is a list of other name changes:

```
nsi-entry              rpcentry
nsi-group              rpcgroup
nsi-profile            rpcprofile
cds-cache              cdscache
```

1. This **boldface** convention is used to introduce terminology the first time it appears in this paper. See the *Terminology* section, below, for definitions.

| | |
|---|---|
| org | organization |
| addto | add |
| rmfrom | remove |
| mapping | endpoint |
| update | synchronize |
| -recurse | -recursive |
| major-only | majoronly |
| config | configure |
| dtsd | dts |

The names of all attributes have been reviewed and some have been changed to ensure a consistent naming scheme. Below are standard abbreviations used in naming all attributes:

| | |
|---|---|
| account | acct |
| clearinghouse | ch |
| database | db |
| default | def |
| expiration | exp |
| length | len |
| lifetime | life |
| message | msg |
| observed | obs |
| password | pwd |
| project | proj |
| standard | std |
| synchronization | sync |
| ticket | tkt |
| update | upd |

All the CDS attributes have different names than in the last version of this specification. That version tried to rename those attributes to keep them consistent with `dcecp` attribute naming conventions. However, since CDS allows new attributes to be created and uses local files (i.e., `cds_attributes`) to generate their names, `dcecp` should do the same, and will. Below is a listing of all non-CDS attributes whose names have changed:

| | |
|---|---|
| alifetime | acctlife |
| avalid | acctvalid |
| badrepversions | badtimereps |
| courier | courierrole |
| creation | creationtime |
| defticketlife | deftktlife |
| disable | disables |
| enable | enables |
| expires | expdate |
| forwardableticket | forwardabletkt |
| inplist | inprojlist |
| invalids | invalidmsgs |
| lowgid | mingid |
| loworgid | minorgid |
| lowuid | minuid |
| maxticketlife | maxtktlife |
| maxticketrenew | maxtktrenew |
| minticketlife | mintktlife |
| nomemory | nomemories |
| notrespond | noresponses |
| palpha | pwdalpha |
| pexpiration | pwdexpdate |

| | |
|---|---|
| plifetime | pwdlife |
| pminlen | pwdminlen |
| postdatedticket | postdatedtkt |
| providererrors | providerfailures |
| proxiableticket | proxiabletkt |
| pspaces | pwdspaces |
| pvalid | pwdvalid |
| pwhidden | hidepwd |
| renewableticket | renewabletkt |
| rootlost | rootlosts |
| synchs | syncs |
| tgtauth | stdtgtauth |
| updates | synchronizations |

Hyphens have been removed from command names. This more closely matches the names of attributes used in `dcecp`. Also there was the minor issue that the dollar-sign notation of Tcl doesn't recognize them as being part of variable names. This affected four objects: `nsi-entry`, `nsi-group`, `nsi-profile`, `cds-cache`.

(b) A description of the abbreviation capabilities of `dcecp` was added to the *User Interfaces* section. All commands, verbs, and switches can be abbreviated to the shortest unique name.

(c) The type of help available has not changed, but the commands used to access them has. A description of the on-line help facilities was added to the *User Interfaces* section.

(d) The `dcecp` will support the use of several convenience variables to save typing for the interactive user. These variables store the last entered object type and name and the last return value among other things.

(e) The value of `dcecp_library` is now specified as *`dceshared`*`/dcecp`.

(f) The `dcecp` will include a script that supports the operation-object syntax. This script will define new commands named for the verbs that take the first argument, an expected object name, and reverse them to call the object-verb command.

(g) The `$c` convenience variable was removed as there is no syntaxic way to determine the cell in a name. While guesses can be used based on DNS and X.500 syntaxes, hierarchical cells will make this impossible.

(h) Arguments are identified as being either a single argument or a list of arguments in each command. Commands such as `directory create` will accept either a single argument or a list of arguments.

(i) Commands that were defined to return `0` on success (e.g., `create`) are now defined to return an empty string.

(j) A description of attribute lists and attribute switches was added including when they are allowed to be used.

(k) Attribute values that were listed as ''an internationalized yes or no'' have been changed to just `yes` or `no`. This is so scripts can be portable. Users can write scripts that need to do comparisons against values (e.g., `if {`*`attribute`*` == yes}`).

(l) Descriptions of the `list` and `show` operations were made more precise. The `-simplename` switch was added to the `list` command. The `-all` was removed from the `list` command and its meaning was changed in the `show` command.

(m) A new `catalog` operation was added. It takes no arguments and returns a list of all instances of an object. It is only implemented for those objects that support this functionality, e.g., the registry when referring to pgo's or accounts. The functionality was originally intended to be available by using the `-all` switch on the `show` operation, but was often described in the previous version of this specification as being on the `list` operation.

(n) A new `rename` operation was added. It takes a required `-to` switch and is implemented for `principal`, `group` and `organization`.

(o) The description of the `show` operation was made more precise regarding what it returns. It returns any information associated with an object including attributes, counters, policies, child pointer information, etc. Various switches determine the output. The old `status` operation has been removed since `show -counters` does what it used to. A brief description of the order of attributes returned by a `show` operation was added.

(p) There is a slightly new syntax for the `modify` command regarding multiple `add` switches. Also, the names of the switches changed.

(q) Changed the more general `-in` switch to `-clearinghouse` in CDS commands.

(r) The old `policy` operation has been removed, and the functionality replaced by the `show -policies` and `modify` commands.

(s) The `-fullname` switch on the pgo `create` commands is no longer required. It can now be defaulted which results in a fullname which is the null string. This matches existing `rgy_edit` functionality.

(t) A new read-only attribute has been added to the `principal`, `group`, and `organization` objects. The new attribute is called `uuid` in all three objects, and it represents the UUID of the respective object. It cannot be modified, but is returned by the `show` operation.

(u) The `rgy_edit adopt` command will be implemented by new switches on the `create` commands of the `principal`, `group`, and `organization` objects. This will allow a new principal to be created with a specified UUID. Normally the UUID is assigned by the system when a new PGO is created. The `adopt` functionality is important when re-configuring a cell and trying to retain old cell data, when moving data from one cell to another, potentially when restoring backups that contain data owned by users no longer in the system, etc. A `-uuid` switch has been added to each of the `create` commands.

(v) The `members` attribute for the `group` and `organization` attribute that was displayed on a `show` operation was removed. This information is available via the `list` operations on both of these objects. Including the attribute violated the distinction between the `list` and `show` operations.

(w) The account object changed in the way it handles the associated group and organization. Accounts are no longer named with a principal, group, *and* organization. They are now named with just a principal name. There are now two new attributes to an account: `group` and `organization`, that are required and have no default value. That is, they must be input on the `create` command via either at attribute list (via the `-attribute` switch), or with attribute switches.

Also two other attributes were added: `created` and `lastchange`.

(x) The `keytab` object has been redesigned based on the `dced` keymgmt service.

(y) The `readonly` attribute on the `registry` object was removed as that property in `rgy_edit` was obsoleted in DCE 1.0.2.

(z) The `acl` object has been significantly redesigned. The `acl access` command has been renamed `acl check`. The `acl` command's argument has been specified to be either a name or string binding. A new attribute was added to an `acl` which represents the default cell. Switches were added to the `modify` and `show` operations to allow manipulation of this attribute. The `cell` operation was removed. The `modify` command now has a `-purge` and `-mask` switch.

(aa) There is a new object called `clock` which understands three commands which were formally `dts` commands. The time of a machine may be set and displayed via this object.

(ab) The `dts` attributes were changed. Removed the old status attributes and made them merely attributes (two new attributes `localservers` and `globalservers`). Separated the attributes that are shown on a server only from those that are shown for both a clerk or a server.

As per CR's 7973 and 8229 (both fixed in DCE 1.0.3), removed the `toomanyservers` counter from the `dtsd` object and added a `providertimeouts` counter. There was a bug in the implementation of the `toomanyservers` counter in `dtsd` itself, such that this counter was never incremented. Due to this, there is no backward compatibility issue of dcecp having to recognize this attribute when communicating

with a pre-DCE 1.0.3 server. Removed the `invalidmsgs` counter, as it was a documentation anomaly and didn't appear in the code.

Removed the `localservers` attribute and added the `actcourierrole` attribute of the `dts` object. Added the `badlocalservers` and `badservers` counters.

(ac) Changed the `state` attribute of `dts` to `status` and changed `on` and `off` to `enabled` and `disabled`.

(ad) The `dtscp` commands `enable` and `disable` are now mapped to the `dcecp activate` and `deactivate` commands. The `activate` command takes a `-abruptly` switch.

(ae) New commands for auditing were added. The following new objects were added for auditing support: `aud`, `audfilter`, `audevents`, and `audtrail`.

(af) New commands were added to the `registry` object to support security replication. These new commands add `sec_admin` functionality to `dcecp`.

(ag) The open issue regarding if undo functionality is required has been removed. The resolution is that DCE 1.1 will not contain undo functionality in `dcecp` mostly due to time constraints.

(ah) The open issue regarding server starting has been closed. This functionality will be supplied by `dced` and `dcecp` will have commands to address it.

(ai) The open issue regarding an API to get serviceability routing information has been resolved. The routine `dce_svc_inq_routings` has been added to the library.

(aj) The appendix containing the Tcl man page was removed. Tcl 7.0 was released and incorporation of the man pages seems unnecessary. Refer to either [TCL MAN] or [TCL BOOK] for details on Tcl.

(ak) An appendix giving a list of section names and numbers was added. This is as much of a Table of Contents that can be added in a DCE RFC due to limitations in the tools used.

(al) Numerous editing corrections.

### 1.2. Changes For the Next Version

These changes will be incorporated into the next version of this RFC. All of this information is subject to change.

(a) Commands to support `dced` will be added (see [RFC 42.1]). There are five new objects: `server`, `daemon`, `hostdata`, `keytab`, and `secval`.

(b) The audit commands `aud`, `audevents`, and `audfilter` will be updated to allow remote operation. This will be accomplished by making use of a `s(aud)` convenience variable.

(c) The task scripts will be described.

(d) CDS subtree commands will be described.

(e) Support for extended registry attributes will be added.

(f) Support for hierarchical cells will be added.

## 2. TERMINOLOGY

The following are the terms used in this document.

(a) **Administrator**

The person (or persons) responsible for the installation and maintenance of a DCE Cell. They perform various administrative operations and **tasks**. While performing these duties they must have the proper permissions, so they must be members of the appropriate administrative groups.

(b) **Argument**

A parameter given to a command. **Commands** take zero or more arguments followed by zero or more **switches**.

(c) **Command**

Used in this document to mean a Tcl command or `dcecp` script that will be executed if entered as a command (Tcl provides a $PATH-like facility). Almost all `dcecp` commands are named for the type of **object** that they operate on (e.g., ''the `acl` command''). This word is also used to refer to object-operation pairs (e.g., ''the `acl show` command'').

(d) **Control Programs**

All of the administrative user interfaces provided with DCE 1.0. This includes but is not limited to: `rpccp`, `cdscp`, `rgy_edit`, `acl_edit`, and `dtscp`.

(e) **Element**

Existing DCE terminology uses this term to refer to elements of RPC profiles. The `dcecp` uses **member** for this purpose. In `dcecp` (and in Tcl), element is used to refer to the contents of lists.

(f) **Member**

Existing DCE terminology uses this term to refer solely to members of groups (both RPC groups and registry groups). Other terms are used to refer to objects in other types of containers (e.g., RPC profiles contain **elements**). The arbitrary choice of words has caused confusion for some users. The `dcecp` uses the word `member` to refer to anything stored in container.

(g) **Operation**

The commands in `dcecp` are usually named for the type of object they act on. The first argument to the command is usually the operation to perform on that object.

(h) **String Syntax**

A format of a data structure that is understood by `dcecp` and existing control programs. In some cases, data structures have two syntaxes, one that is used in existing control programs, and one that is more natural for `dcecp`. Both are supported by `dcecp`.

(i) **Switch**

Tagged arguments to commands. All switches begin with a `-` (dash) and are found after all **arguments** in `dcecp` command lines. Some switches are optional and some are required. Some switches take values and some do not.

(j) **Task**

A high level operation that an administrator performs, such as add a new user, or configure a new machine. A task is above the level of DCE components (e.g, add principal and account to registry), and usually involves operations on several different components. Usually implemented as dceep task scripts.

(k) **Tcl**

The embeddable Tool Command Language. Tcl refers to both the language and to a library (`libtcl.a`) that can be linked into a program which provides it. Commands can be added to the language via both C code and Tcl scripts. The `dcecp` language can be considered a dialect of Tcl, i.e., it is Tcl plus some DCE-specific commands.

(l) **Tcl Syntax**

A format of a data structure that is understood by `dcecp`. In some cases, data structures have two syntaxes, one that is used in existing control programs, and one that is more natural for `dcecp`. Both are supported by `dcecp`.

## 3. TARGET

The `dcecp` and is meant to be used by DCE administrators. The emphasis of the first release will be on a commonly performed tasks of DCE cell administrators. It will have a command-line interface only.

Developers will probably find an advantage to using `dcecp`, however they are not the primary target user.[2] For example, the most commonly performed task of a DCE developer is to configure a new cell. However, it is not expected that this will be the same for DCE administrators.

## 4. GOALS AND NON-GOALS

The following are goals for `dcecp`:

(a) The major goal is to simplify the administration of a DCE cell by creating a single powerful and consistent administrative user interface.

(b) Emphasis in scheduling development resources must be placed on common everyday administrative duties rather than more esoteric configuration management. Administrators will get more use from a simpler way to add a new user or to modify a CDS entry and its ACL, rather than a new way to create a registry or to initialize a new CDS namespace.

(c) The `dcecp` will augment use of existing control programs, it will *not* replace them. Existing control programs will continue to be installed with DCE, and their use will probably remain required for some administrative operations. See Appendix A for a list of existing control programs commands and the `dcecp` commands that replace them.

(d) Any new administrative interfaces required by new DCE 1.1 functionality should be provided via `dcecp` commands rather than new separate control programs.

(e) Commands should be remotely invocable. This is currently true for most existing control program operations. The extension of those commands that are not remotely invocable is desired.

(f) The `dcecp` must be portable to a variety of different platforms and management situations. Vendors must be able to integrate it into their own management systems.

The following are not goals of `dcecp`:

(a) The `dcecp` will only be used to administer DCE itself, it will *not* be used to administer applications that are written with DCE.

(b) Extended registry attributes will not be supported by `dcecp`.

(c) A graphical user interface (GUI) is desirable, but a command line interface is: more portable to a wider variety of platforms, scriptable, and usable over `telnet` lines and on machines not equipped with GUI's. Therefore, the command line interface is more important for the first release of the `dcecp`.

(d) This is not DME. The `dcecp` is not meant to replace the need for DME, but rather to substitute for its absence in a very limited manner (but better than the existing control programs have done). Nothing in the design of the `dcecp` should preclude the use of DME in the future.

## 5. REQUIREMENTS

The following are the requirements of `dcecp` as specified in [RFC30] and how they are met by the program described in this functional specification:

(a) Existing Control Program Functionality

  (i) Stop servers — Met with `stop` operation.
  (ii) Show/add/modify/delete data — Met with the `show`, `create`, `modify`, `delete`, `add`, and `remove` operations on various commands.

---

2. It is expected that during the course of DCE 1.1 development, developers who have access to early versions of `dcecp`, will produce task-oriented scripts that may be of use to DCE administrators. As many of these as is possible and practical will be included in the first release of `dcecp`.

(iii) Show status — Met with various `show` operations
(iv) Remote operations — Met by most commands, see the *User Interface* section for details.

(b) Usability

(i) Common syntax for all functions — Met with Tcl and designed syntax.
(ii) Help — Met with `help` operations on all commands.
(iii) Provide command line recall/editing — Met with command-line editing facility.
(iv) Scriptability — Met with Tcl's `source` command and with the interpretation of an argument as a script file to execute.

(c) New Functionality

(i) Administer serviceability — Met with the `log` command.
(ii) Administer auditing — Met with new commands: `aud`, `audevents`, `audfilter`, and `audtrail`.
(iii) Programmability — Met with use of Tcl.
(iv) Extensibility — Met with Tcl's `proc` command.
(v) Bind to different servers or data — Met with separate commands, the `s` convenience variable, the `-clearinghouse`, etc.
(vi) Shell escape — Met with the `shell` command and Tcl's `exec` command.
(vii) Administration of little files — Will be met by `dced` commands.
(viii) DCE ping — Met with the `ping` operation.
(ix) Security — Will be done by existing ACL's.

(d) Meet Other DCE 1.1 Goals

(i) Code Cleanup — Will follow the DCE Coding Style Guide [RFC 34].
(ii) Internationalization — The new `dcecp` code will follow proper internationalization coding practices. Tcl will be internationalized.
(iii) Serviceability — Will use `sams` for message catalogs and messaging API's. The three letter serviceability component name of `dcecp` will be `dcp`. The routine `dce_sprintf` will be used for all messages. See [RFC 23], [RFC 24], and [RFC 34] for details.

## 6. FUNCTIONAL DEFINITION

Many people seem to get confused trying to understand the relationship between `dcecp` and Tcl. It is a close relationship, but understanding which commands are Tcl commands and which are DCE commands is unimportant.

A user of `dcecp` starts the control program and enters interactive commands at the `dcecp>` prompt. There are other use models (e.g., writing scripts and entering commands from the shell command line) but they are less common. An important usability feature of `dcecp` is that it provides command-line recall and editing similar to `ksh`'s Emacs mode. At the prompt the user can enter base Tcl commands such as:

```
dcecp> echo foo
foo
dcecp> lsort {a c e b d}
a b c d e
dcecp> foreach i [lsort {a c e b d}] {puts stdout $i}
a
b
c
d
e
```

The Tcl library implements an interpreter that understands all the base Tcl commands. Any program using this library has access to these same commands.[3] A typical Tcl program usually implements new project-specific

commands. Not surprisingly, dcecp implements commands that manipulate DCE data. Users can at the same dcecp> prompt enter DCE commands such as:

```
dcecp> object delete /.:/hosts/absolut
dcecp> acl modify /.:/hosts -add {user melman -rwx}
dcecp> principal create melman -fullname {Howard Melman}
```

To the user of dcecp there is no visible difference between a base Tcl command and a command that manipulates DCE data. They might see the base Tcl commands when running other programs that use Tcl, but they will only see the DCE commands in dcecp. The Tcl and DCE commands can be combined (output not shown for space reasons):

```
dcecp> foreach i [dir list /.: -directories] {dir show $i}
```

This allows an advanced user to write extensible scripts, a feature that we deem very important. For a list of base Tcl commands, see [TCL]. See the *User Interfaces* section for all other details about the usage of dcecp.

## 7. DATA STRUCTURES

This section provides a brief introduction to Tcl data types and then presents several dcecp data structures. For a complete reference of Tcl data types, see [TCL MAN].

In Tcl, there is only one data type, strings. All commands, expressions, variable values, and procedure return values are strings. Some Tcl commands interpret the strings in different manners. For example, eval treats its **argument** as an expression. The argument it receives is a string. If the argument is 4 + 5 then the returned result is the string 9.

Other Tcl commands interpret their string arguments as lists. In Tcl lists consist of elements separated by white space (space, tab, or newline). The following are three examples of lists with three elements:

```
tom dick harry

a b c

1 2 !
```

There are three meta-characters that are treated specially when parsing lists. They are:

(a) Braces are used for grouping. If a left brace is encountered, then the element is terminated by the matching right brace, instead of the next whitespace. Braces nest. The following are also examples of lists with three elements:

```
tom {dick harry} {sue mary jane}

a {b {c d e}} f
```

(b) Double quotes are also used for grouping, however, there are two differences between them and braces. First they don't nest. Second Tcl will perform command, variable, and backslash substitutions for items within double quotes. Tcl does not perform these substitutions when braces are used.

(c) Backslashes are used to escape meta-characters, and to insert non-printing characters. Standard substitutions occur such as \t is a tab character, and \n is a newline. Braces and double quotes can be inserted in lists with backslashes as follows, again these example lists all have three elements:

---

3. This is orthogonal to DCE. The only part of DCE that will use libtcl is dcecp. The Tcl library will *not* be part of libdce. The goal of DCE is to ship dcecp not Tcl, as such if we will not install libtcl and the headers for general availability. This does not prevent vendor's from doing this. IT affects only the reference implementation of dce_config. The AES mentions neither dcecp nor Tcl. Tcl is freely available to all from the internet.

```
a b\ c d

a b \}
```

The net effect of using these meta-characters and Tcl commands on a command line is that lists appear as elements surrounded by braces, and the top level of braces is not present on output. This last point is tricky; it has to do with how commands are parsed. The best way to explain it is with an example. The Tcl command `lindex` takes two arguments, a list and an integer (remember, both of these are just strings and it is the `lindex` command that interprets these string arguments specially). It returns the element from the list indicated by the second argument. Elements in lists are numbered from zero. So the following command:

```
lindex {a b c} 1
```

returns the string `b`. The braces in the example are used to group the elements of the list so that they are passed as one argument to the lindex command. Now, the following example returns a list:

```
lindex {a {b c} d} 1
```

The list returned is `b c`, there are no braces around it, but it is a perfectly valid list. To prove this, pass it to the `llength` command which takes one argument, a list, and returns the length as an integer (in Tcl, square brackets perform command substitution, much like backquotes do in command shells):

```
llength [lindex {a {b c} d} 1]
```

This returns the value `2` since the list `b c` has two elements.

In `dcecp`, there are several data structures used to represent DCE objects such as attributes, bindings, and ACL's. Most of these had some representation in the existing control programs. The `dcecp` often supports these representations for backward compatibility, as they will be recognized by the administrator. This is known as the **string syntax** of the data structure. Often `dcecp` provides a representation that is more easily handled by Tcl for these data structures, usually they are lists. This representation is known as the **Tcl syntax** of the data structure.

### 7.1. Attributes

In `dcecp`, attributes are represented as a list. The first element is the attribute type. This is usually some string name, but in some cases, OID's or UUID's are acceptable. The following elements in the list are the values. If the attribute is single-valued then the attribute list will have two elements, multi-valued attributes will have more. Attribute lists are lists of attributes. The following is an attribute list of two single-valued attributes:

```
{acctlife 2} {pwdlife none}
```

### 7.2. ACL's

An ACL is an Access Control List which consists of a set of ACL Entries. Each ACL Entry is defined as a type, an optional key or value, and a required set of permissions. In access checks, the permissions granted to a user are established by checking the type and key of each ACL Entry and or'ing the permissions described therein.

Each application is free to define new permission bits. The defined types are listed below:[4]

(a) `user_obj`
(b) `group_obj`
(c) `other_obj`
(d) `user`
(e) `group`
(f) `foreign_user`
(g) `foreign_group`

4. The new ''delegate'' types are to be added during DCE 1.1 development to support delegation.

(h) `foreign_other`
(i) `any_other`
(j) `mask_obj`
(k) `unauthenticated`
(l) `extended`
(m) `user_obj_delegate`
(n) `group_obj_delegate`
(o) `other_delegate`
(p) `user_delegate`
(q) `group_delegate`
(r) `foreign_user_delegate`
(s) `foreign_group_delegate`
(t) `foreign_other_delegate`
(u) `any_other_delegate`

All ACL type names are case insensitive. The following are all legal: `user`, `USER`, and `uSer`.

### 7.2.1. String syntax

The string syntax of ACL Entries is as follows:

*type*[:*key*]:*permissions*

The permission bits need not include –'s for permissions not granted and may be given in any order. Examples of nine ACL Entries follow (the last appears on two lines for space reasons[5]):

```
unauthenticated:-r-----
user_obj:crwx---
user:britten:crwx---
user:mahler:-rwx---
foreign_user:/.../C=US/O=OSF/OU=dce/pro/bach:crwxidt
group_obj:-rwx---
group:dds:-rwx---
any_other:-r-----
extended:c417faf8-8340-11c9-ace3-\
08001e5559bb.a.b.c.a1.4.0a0b0c0d:-rwx---
```

### 7.2.2. Tcl syntax

The core Tcl Syntax of ACL Entries is a list of two or three elements. The first element is the type, the optional second element is the key, and the last element is the set of permission bits. The permission bits are represented as a single character if the permission is granted and as a – (dash) if it is not. An ACL is a list of ACL Entries. An example of an ACL is as follows:

```
{unauthenticated -r-----}
{user_obj crwx---}
{user britten crwx---}
{user mahler -rwx---}
{foreign_user /.../C=US/O=OSF/OU=dce/pro/bach crwxidt}
{group_obj -rwx---}
{group dds -rwx---}
```

5. Line-terminating ''\'' denotes line-folding of long lines, sometimes as a display meta-character and sometimes as a Tcl meta-character (the distinction will always be clear from context).

```
{any_other -r-----},
{extended c417faf8-8340-11c9-ace3-\
08001e5559bb.a.b.c.a1.4.0a0b0c0d -rwx---}
```

On output the above Tcl Syntax will be used, with one addition. If due to masking[6] there are ineffective bits in an ACL Entry, the entry will be extended with two elements. The first is the string `effective` and the second is the set of effective permissions. This will only be added for those ACL Entries that have ineffective bits. For example:

```
{mask_obj -r-----}
{user_obj crwx---}
{user britten crwx--- effective -r-----}
```

On input, these two elements are ignored. Also on input, the permission bits need not include `-`'s for permissions not granted and may be given in any order. For example, if the user wanted to enter the above ACL to a command, all that they need to type is:

```
{mask_obj r}
{user_obj crwx}
{user britten wcrx}
```

### 7.3. Bindings

In RPC, all bindings are represented as string bindings which contain four pieces of information: a protocol sequence, a network address, an endpoint, and an optional object UUID.

#### 7.3.1. String syntax

The string syntax of a string binding is as follows:

*object-UUID*`@`*protseq*`:`*address*`[`*endpoint*`]`

The *endpoint* can be preceded by the string `endpoint=`. There can be no whitespace in the string binding. Examples follow:

```
ncadg_udp_ip:130.105.96.3[1234]

001c8234-74ff-1be9-b11c-08002b0f59bb@ncadg_udp_ip:\
130.105.96.3[1234]
```

Note that in Tcl, square brackets are used to indicate command substitution. So, if a string binding in the string syntax is entered, the brackets must be preceded by a backslash (i.e., escaped) or enclosed in braces (i.e., quoted). Therefore, the above example would not always be accepted by the Tcl parser, two corrected examples of the first string binding follow:

```
{ncadg_udp_ip:130.105.96.3[1234]}

ncadg_udp_ip:130.105.96.3\[1234\]
```

#### 7.3.2. Tcl syntax

The Tcl Syntax is a list of 3 or 4 elements as follows:

*protseq address endpoint*

or

*protseq address endpoint object-UUID*

---

6. See [POSIX.6] and [USERGD] for details on masks in ACL's.

If an object UUID is to be entered but an endpoint is not, then there must be a null element in the list (specified as `{}`). Examples follow:

```
ncadg_udp_ip 130.105.96.3 1234

ncadg_udp_ip 130.105.96.3 1234 001c8234-74ff-1be9-\
b11c-08002b0f59bb

ncadg_udp_ip 130.105.96.3 {} 001c8234-74ff-1be9-b11c-\
08002b0f59bb
```

### 7.4. Interface Identifiers

An interface identifier describes an interface. Specifically, it contains the UUID, major version number, and minor version number of an interface.

#### 7.4.1. String syntax

The string syntax is as follows, where *UUID* is the string syntax of a UUID.

*UUID*,*major*.*minor*

An example follows:

```
0067b39e-67ac-1be9-abb5-08002b0f59bb,1.2
```

#### 7.4.2. Tcl syntax

The Tcl Syntax is a list of two elements: the UUID in its string syntax and the major and minor version numbers separated by period. An example follows:

```
0067b39e-67ac-1be9-abb5-08002b0f59bb 1.2
```

### 7.5. Routing Specifiers

Serviceability routing information [RFC 24] is specified as a list of routing elements. Each routing element contains three fields containing PCS data:

(a) The first field specifies a message severity. It must be one of the following:

(i) `FATAL` Fatal error, about to exit.

(ii) `EXIT` Normal exit.

(iii) `WARNING` Error detected, program proceeding.

(iv) `NOTICE` Informational notice.

(v) `VERBOSE` Verbose informational notice.

(b) The second field specifies how the messages should be processed. It must be one of the following:

(i) `BINFILE` Write binary log entry.

(ii) `TEXTFILE` Write human-readable text.

(iii) `FILE` Equivalent to `TEXTFILE`.

(iv) `DISCARD` Do not record this level.

(v) `STDOUT` Write human-readable text to standard output.

(vi) `STDERR` Write human-readable text to standard error.

A limited form of log-rolling is available. Any of the file specifiers, above, may be followed by `.`*gens*`.`*count* where *gens* is an integer less then 100 indicating the number of files to be kept and *count* is an integer indicating the number of entries per file. No more then *count* entries will be written to a file, before a new file is started. After *gens* files have been written, the files wrap back around to the

beginning. The file names are formed by appending `.gen` to the end of the filename, where `gen` runs from one to the last generation number.

Note that any debug messages (calls to `DCE_SVC_DEBUG`) that are routed through a `BINFILE` route will lose the types of any arguments.

(c) The third field specifies where the messages should be sent. Presently, this will always be a filename, although it can be blank for the last three specifiers given above. This field may contain a `%ld` string which will be replaced by the process-ID of the currently-running process.

The Tcl syntax of a routing specification is a list of three elements. The first element is the severity, the second is the how specification, and the third is the where specification.

```
{severity how where}
```

For example, to send fatal errors to the console, ignore normal exit reports, and send informational messages to a temporary binary log, use the following routing specification:

```
{{FATAL TEXTFILE /dev/console}
 {EXIT DISCARD}
 {NOTICE BINFILE /tmp/log%ld}}
```

### 7.6. Debug Routing Specifiers

A debug routing specification is a list of debug routing elements. Each debug routing element contains four fields:

(a) The first field is the name of a serviceability component.

(b) The second field is a list of subcomponents and levels separated by a period. If the subcomponent is an asterisk then the level is set for all subcomponents of that component. The level should be a number between zero and nine. This list is parsed in order, so the asterisk may be used to set the initial level for all subcomponents; later entries override earlier ones.

(c) The third field specifies how the messages should be processed. It is identical to the second field of a serviceability routing element.

(d) The fourth field specifies where the messages should be sent. It is identical to the third field of a serviceability routing element.

The Tcl syntax of a debug routing specification is a list of four elements, one for each of the above fields. The second element may itself be a list. For example, to set RPC debugging at a minimal level so that all messages go to standard error, use the following:

```
{rpc *.1 STDERR}
```

The following example sets DTS debugging messages for the general, events, and msgs subcomponents (at the levels indicated for each) to go to a logfile:

```
{dts {general.1 events.0 msgs.3} TEXTFILE /tmp/log}
```

### 7.7. Other Data Types

There are three other data types mentioned in this document: UUID's, UTC times, and Names. They have no Tcl Syntax. For a description of these string syntaxes, see the DCE Documentation.

## 8. USER INTERFACES

The `dcecp` will present a command-line interface to the user with command-line recall and editing capabilities. The commands and command-line editing interfaces are described below.

This section describes the following in various sub-sections:

(a) *Invocation* — How `dcecp` can be started and its modes of operation.

(b) *Initialization* — The various scripts that are read when `dcecp` is started.

(c) *Command Syntax* — A description of the design of the command syntax. Also how to invoke external commands, and how `dcecp` deals with commands it does not know.

(d) *Abbreviations* — The `dcecp` allows objects, operations, and switches to be abbreviated to the shortest unique string.

(e) *Convenience Variables* — Describes builtin variables to make interactive use easier.

(f) *Error Handling* — The overall approach to return values and error handling.

(g) *Help System* — Describes the available on-line help.

(h) *Common Operations* — These are operations that are common to many but not all objects.

(i) *Common Switches* — These are switches that are implemented in more than one object.

(j) *Objects* — These are the object commands that `dcecp` will support. Each sub-section describes one object, including the operations that it supports. Variations in the operations of a particular object versus the common operations are described in this section, as well as any non-common operations supported by the object.

(k) *Tasks* — These are the administrative tasks that `dcecp` will support. They are implemented as object commands so as to have similar syntaxes as other `dcecp` commands. They are more complex and are typically implemented in Tcl code.

(l) *Command-line Editing* — A description of the command-line editing interface.

### 8.1. Invocation

The `dcecp` is a command-line user interface for administrative commands. There are several methods of invocation:

(a) The user starts `dcecp` and then sees the `dcecp` prompt:

```
dcecp>
```

The above is the default prompt. This can be changed using the standard Tcl `tcl_prompt1` and `tcl_prompt2` mechanisms. See [TCL MAN] for details.

(b) The user invokes `dcecp` with one argument which is a filename of a `dcecp` script. The script is run and then `dcecp` exits. This method of invocation allows interpreter files (those with `#!/bin/dcecp` as the first line) to work.

Tcl sets the `argc`, `argv`, and `arg0` variables to provide access to command-line arguments.

(c) The user invokes `dcecp` with the `-c` argument followed by commands (all on one line separated by `;`'s which must be quoted in the shell). The commands are executed and then `dcecp` terminates. For example:

```
$ dcecp -c directory create /.:/foo
```

### 8.2. Initialization

When `dcecp` is invoked, it executes the following scripts in the order shown. Note the syntax used for the script locations is Tcl.

(a) `[info library]/init.tcl`

Contains standard Tcl initialization commands on a per-host basis. Contains definitions for the `unknown` command and the `auto_load` facility.

(b) `$dcecp_library/init.dcecp`

Contains `dcecp`-specific startup information. The `dcecp` scripts implementing commands and tasks are stored in the directory identified by the Tcl variable `$dcecp_library`. The reference implementation will set `dcecp_library` to *`dceshared`*`/dcecp` by default.

(c) `$env(HOME)/.dcecprc`

This file stores user customizations.

### 8.3. Command Syntax

The `dcecp` commands are implemented as procedures in a Tcl interpreter. Commands return strings, and the main interpreter loop displays the results to the user. Errors are displayed as well. In most cases a Tcl object-oriented approach has been used to define the commands. In this approach commands represent a type of object, and have the same name as the object.

*object operation name* [*-switch* [*value*]]...

The first argument to the command specifies an operation to perform on an object, the second argument identifies the particular object to operate on (usually by name) and the third and following arguments are switches and values. Some commands might require one or more switch. Switches have a leading - and are a full word, not a single letter, but may be abbreviated (see below for details). Some switches take a value which immediately follows the switch. All arguments are strings, and all commands return strings as values. Some examples of Tcl commands are:

(a) `file exists` *filename*
(b) `history add` *command*
(c) `string compare` *string1 string2*
(d) `string toupper` *string*

The `dcecp` will define commands in a similar manner. Commands will be named for a type of object to operate on, and the operations will be implemented for each object. Operation names will be consistent whenever appropriate (e.g., `create` will always be used to create a new instance of the object, `delete` will always be used to delete an instance). This syntax will be used whenever possible, but in some cases (such as `quit`, `exit` and `shell`) such a mapping would be too contrived and will be avoided. The syntax is described below.

The commands available to the user are those known by the Tcl interpreter. In the `dcecp`, the interpreter will know about the Tcl built-in commands and the additional `dcecp` commands described below. If a command name that Tcl does not know about is invoked, then Tcl calls the Tcl built-in command `unknown` with the entered command and arguments. The user is free to define this command as they wish to have a variety of things happen. By default, Tcl defines the `unknown` command to use the following heuristic:

(a) See if the Tcl autoload facility can locate the command in a Tcl script file. If so, load it and execute it.

(b) If running interactively see if the command exists as an executable UNIX program. If so, `exec` the command.

(c) If the command was invoked at the top-level:

(i) See if the command requests `csh`-like history substitution in one of the common forms `!!`, `!`*number*, or `^`*old*`^`*new*. If so, emulate `csh`'s history substitution.

(ii) See if the command is a unique abbreviation for another command. If so, invoke the command.

(d) Otherwise, fail with an error indicated an unrecognized command.

This means that from the `dcecp>` prompt, a user can type `ls` to see a listing of the files in the current directory, or `rm foo` to delete the file `foo` (see about filename completion in the *Command-line Editing Commands* section).

#### 8.3.1. Object-operation vs. operation-object

The choice of using an object-operation command order has been controversial. Many parties have stated that an operation-object ordering is more intuitive and is more common in other user interfaces. The major advantage to using an object-operation model is extensibility. To add a new object type to `dcecp`, that object type merely needs to be added with all supported operations. If an operation-object order was used, then each operation that was supported by the object would need to be taught about the new object. This would be a big problem as operations are expected to be common among objects, e.g., `create`, `delete`, and `show`.

In deference to the operation-object faction, `dcecp` will include a script that can be loaded by any user, or configured to be loaded for an entire system. This script will define commands named for operations, e.g., `create`, `delete`, and `show`. The commands will expect their first argument to be the name of an operation, and will try to execute that operation on that object using the object-operation syntax. This will provide an operation-object syntax to the user *in addition to* the object-operation order. The commands will be based on the Tcl `info commands` command and the `operations` command supported by each object. Therefore, as new objects are added to the system, these commands will learn about them automatically. A sample (and incomplete) implementation of the `show` command is shown below:

```
proc show {obj args} {
  if {[llength [info commands $obj*]] < 1} {
    error "Object $obj does not exist"
  } elseif {[llength [info commands $obj*]] > 1} {
    error "$obj not unique"
  } else {
    if  {[llength [lsearch [$obj operations] show]] < 1} {
      error "show command not supported by $obj object"
    } else {
      $obj show $args
    }
  }
}
```

This example will be modified to do somewhat better error checking. Note that the user of these commands will not have to understand the above script. All they will need to do is have the following command in their `.dcecprc` file or in the system's `init.dcecp` file:

```
source verb-object.dcecp
```

Also note that the object-operation order is the supported syntax. The documentation will show sample commands using that order. There will be a description of how to use the operation-object order. Since the difference is just transposing the first two words, this should not be a problem for an administrator of DCE choosing to use an operation-object syntax.

### 8.3.2. Attribute lists and switches

Many of the commands need to specify attributes to operate upon. For example, the `modify` command allows attributes to be changed and the `create` command often allows attributes to be created along with the object. In all cases, a mechanism exists so that an attribute list is used to specify the attributes and their values. This makes passing information from one command to another very easy. For example, an ACL copy operation could be written as follows:

```
# copy acl name1 to acl name2
# no error checking
proc acl_copy {name1 name2} {
  acl replace $name2 -acl [acl show $name1]
}
```

While attribute lists are useful for writing scripts, they are often not user-friendly. For those objects that have a fixed list of attributes (e.g., `principal` and `dts`, but not `object`), wherever an attribute list is allowed, switches for each attribute that are named the same as the attribute are allowed followed by their values. For example, the following are equivalent:

```
principal create melman -attribute {{quota 5} {uid 123}}
principal create melman -quota 5 -uid 123
```

### 8.3.3. Lists of lists

Since Tcl lists are confusing enough, it's worth noting here a common usage when lists contain lists. See the *Data Structures* section above for a brief description of Tcl lists, or see [TCL BOOK] for more detail. The following

example is the the command to remove some ACL's from a the object `/.:/foo`:

```
acl modify /.:/foo -remove {user melman}
```

The argument to the `-remove` switch is an ACL Entry. The ACL Entry happens to be a list where the first element describes the ACL Type, in this case `user`, and the second is the key for which user, in this case `melman`. However, the `-remove` switch may take a list of ACL Entries, so the following is valid as well:

```
acl modify /.:/foo -remove {{user melman} {user salamone}}
```

All of this seems pretty clear. The point to remember is that lists of one value that do not contain spaces, do not need braces. The string syntax of an ACL Entry allows the type and key to be separated by a `:`, so the following are valid:

```
acl modify /.:/foo -remove user:melman
acl modify /.:/foo -remove {user:melman user:salamone}
```

A point of confusion is if there is only one ACL Entry given, i.e., the `-remove` switch's value has only one element (and that element does not contain spaces), then braces are not needed to delimit the list. The following are all valid, but are all examples with unnecessary braces:

```
acl modify /.:/foo -remove {{user melman}}
acl modify /.:/foo -remove {{{user melman}}}
acl modify /.:/foo -remove {user:melman}
acl modify /.:/foo -remove {{user:melman} {user:salamone}}
```

### 8.4. Abbreviations

The `dcecp` makes use of two mechanisms to allow all object names, operation names, and switches to be abbreviated to the shortest unique string. The first is a standard Tcl mechanism builtin to the Tcl `unknown` command. See the *Command Syntax* section above.

Note that this mechanism only works if the command is entered interactively. If the command is found in a script, abbreviation checking is not performed by the standard implementation of `unknown`. This is to discourage the practice of using abbreviations of commands in scripts.

The other mechanism used for abbreviations is builtin to the individual `dcecp` commands themselves. They all share the same parser code which is used for both the operation names and the switch names. This allows the operation name to be abbreviated to the shortest unique operation supported for that object, and the switches to be abbreviated to the shortest unique string representing a switch supported by that object and operation. This fact has had a bearing on the names of operations and switches. For example, in most cases `-member` can be abbreviated `-m` since few other switches begin with ''m''. Note that this form of abbreviation is always available, whether invoked interactively or via a script.

### 8.5. Convenience Variables

All `dcecp` commands will set several variables on execution. The variables will contain the name of the object operated on, the return value of the last command, the cell name of the last object operated on, etc. Users can substitute the value of these variables into the next command to save typing.

These are regular Tcl variables, so any mechanism to perform the variable substitution will be supported. The most common method is to prepend a `$` before the name of the variable, though the `set` command can be used as well. Note that since `dcecp` uses these variables the user should not change the values of these variables. Tcl traces will be implemented to prevent this.

The `dcecp` defines the following variables:

(a) `n` — holds a list of the names entered to the last command.[7] These names are the names that the command

7. Note that this means the last object command, not any Tcl command or miscellaneous `dcecp` commands like `login`.

operated on, typically entered as the third argument. Examples follow:

```
dcecp> dir list /.: -simplename
hosts subsys absolut_ch cell-profile fs lan-profile sec \
sec-v1
dcecp> echo $n
/.:
dcecp> dir create {/.:/x /.:/y}
dcecp> echo $n
/.:/x /.:/y
```

(b) `o` — holds the object used in the last operation. For example, if the last command was `dir show /.:`, then `$o` is `directory`.

(c) `p` — holds the parent of `$n`. If `$n` is a list, then this is a list of the same length, where each element is the parent of the corresponding element in `$n`.

(d) `r` — holds the return value of the last executed command.

(e) `s` — holds the name of the server bound to for the last command. This is actually a Tcl array where the indexes are used to identify the service (e.g., `sec`, `cds`). The value specifies the name of a server in whatever manner the service finds useful. This could be the name of an RPC server entry in the namespace, or a string binding, or the name of a cell. This variable may be set by the user.

The values of this variable (array) can be treated differently per service. For example, the security service uses this variable to display the registry bound to for the last command, and is used as a default for the next registry operation. If bound to a read-only replica and an update is requested, `dcecp` will try to bind to the master registry to perform the change. The CDS service uses this variable in the same way as the `cdscp` concept of a `preferred clearinghouse`. If set, CDS will only attempt to communicate with the specified server (the word ''preferred'' was a poor choice).

If any objects set or read this value, their behavior is explicitly mentioned in the description of the object in following sections.

(f) `u` — holds the current principal name. This variable can be set by the user to change the current login context. See the `login` command.

### 8.6. Error handling

All commands in `dcecp` return either a list of some information (e.g., the `show` and `list` operations), or an empty string on success.

The `dcecp` will make use of Tcl's native error handling facilities. At the C code level, all Tcl commands return a code and a string. The code is usually either `TCL_OK` or `TCL_ERROR`. The returned string is either the normal result of the command or the error message. Tcl stores the stack-trace of nested errors, and this is accessible to the user. Tcl also provides facilities for scripts to catch errors and invoke error handlers (errors are one of four types of exceptions that Tcl supports).

Also there are two global Tcl variables visible in `dcecp`, `errorInfo` and `errorCode`. The former contains the stack-trace of the error messages, and the later is meant to be machine readable information. The `errorCode` variable is a relatively new addition to Tcl so its use is limited. Currently the commands that do I/O and such set `errorCode` to be a list where the first element is `POSIX` and the next two elements are the Posix name for the error, such as `ENOENT` and a human-readable message describing the error.

On error, `dcecp` commands will return the message string of the error, the Tcl Error exception (which can be caught with the Tcl `catch` command), and will set `errorCode` as appropriate to a list where the first element is `DCE` and the second is the numeric value of the status code. A `dcecp` script which traps an error is shown below:

```
# This routine returns 1 if the argument is the name
# of an rpc group, 0 if it is not.  It uses the show operation
# to determine if it is a group.  If the command works then
# it is a valid group, if not, an error is generated and
# caught by the Tcl catch command and the routine returns 0.

proc isrpcgroup {name} {
  if {[catch {rpcgroup show $name}]==0} {return 1} {return 0}
}
```

In cases where an argument list is given to a command indicating that an operation is to be performed on more than one object, the operations are usually performed iteratively. If there is an error, the command will abort at the time of error with the Tcl `TCL_ERROR` exception. Some operations will have finished and others will not have. The operations are always performed in the order entered and the error message should make it clear on which object the command failed.

### 8.7. Help System

There will be several different kinds of help available to the user of `dcecp`. All will return help strings obtained from message catalogs as appropriate.

Each `dcecp` object will support the `operations` command. This returns a list of the operations supported by the object. It is similar to the Tcl builtin command `info commands`. An example is:

```
dcecp> principal operations
create delete modify show catalog operations help
```

This provides only very limited help, but it is important that the output is machine readable (i.e., a well formed Tcl list). It is analogous to a simple usage message found on many systems. Users unsure of an operation name or if an operation is supported by an object can use this command to find the answer.

Each `dcecp` object will also support a `help` command. This command has three different modes. If called with no arguments or switches, it returns a one line per operation help message. Operations are listed in alphabetical order, with the `operations` and `help` commands listed last since they are supported by all `dcecp` objects. An example is:

```
dcecp> principal help
catalog     Return the names of all principals in the registry
create      Creates a new principal
delete      Deletes the named principal
modify      Changes attributes of a principal
show        Return all the attributes of a principal
help        Provides help on principals
operations  Returns names of operations supported by principal
```

The `help` command takes one argument which is interpreted as the name of one operation[8], (not a list of operations). If invoked in this manner, a one line per switch help message is returned with information on each switch supported by the command. Attribute switches are listed in alphabetical order, other switches follow in alphabetical order as well. If no switches are supported then an empty string is returned. An example is:

```
dcecp> principal help create
-alias      If present principal is an alias (not a primary)
```

8. The word order is a bit strange here, but the rationale is as follows. This order creates a real `help` operation which behaves similarly to other operations. It takes the name of an operation to act on, much as most commands take the name of some object instance to act on. Alternative orderings such as `principal create help` have the problem that it is then impossible to create a principal of the name `help`. Also the specified order allows a `help` verb operation (see the *Object-operation vs. operation-object* section) to be created which supports the `help principal create` syntax.

```
-fullname   PCS limited description of principal
-quota      Maximum number of principals this principal can
            create
-uid        The User ID of the principal (can be generated)
-uuid       Automatically generated internal UUID of the
            principal
-change     Give a list of attributes and new values for
            them
```

Finally, if the `help` command is called with the `-verbose` switch, then a one paragraph description of the object is returned. This command will return text explaining what the object represents and how to use it. The level of information returned will be on the order of the first paragraph of the DESCRIPTION section of a manual page. This description should include a definition of what the object command represents, the possible operations and attributes (or at least the commonly used ones), perhaps permission information, side effects, etc.

```
dcecp> principal help -verbose
The principal object represents Registry principals.  All of
the principal operations take one argument which is a list
of names of principals to act on.  It must be a principal
name, not the name of the database object that contains the
registry information about the principal (i.e., it should
not begin with /.:/sec/principal/).  After this command
executes, the s(sec) convenience variable is set to the name
of the server that was bound to for the command.  The value
of the variable before the command is treated as a hint.
In the registry, each principal has a primary name and a
value for each of the following attributes: alias, fullname,
quota, uid, uuid.
```

Also, due to the fact that the Tcl `unknown` command does a `PATH` search for commands that it does not know about (see the *Command Syntax* section), a user can get detailed help on an object by invoking the `man` command on a [POSIX.2] system specifying an object name as the argument to `man`. For example, the user can get detailed help on the `principal` command by entering:

```
dcecp> man principal
```

### 8.8. Internationalization

Areas that need to be addressed by internationalization include:

(a) `dcecp` scripts will be limited to run in the locale in which they were written. All scripts shipped by OSF will be in the C Locale.

(b) Portions of Tcl will need to be internationalized:

  (i) Basic parsing routines, since the only datatype in Tcl is a string.

  (ii) Most output strings (errors but not prompts) will need to be isolated into message catalogs.

  (iii) The `scan`, `string`, `lsort`, `lsearch`, and `lreplace` commands.

  (iv) The regular expression handling.

  (v) The `glob` command.

### 8.9. Common operations

This section gives a description of operations that are common to more than one object. Some operations presented here are implemented in all objects, some in only a few, and some only for specific types of objects such as containers. Each object is described fully in a following section. Each of those sections lists each operation that the objects supports. Unless otherwise mentioned in those descriptions, those given here apply for the operations.

This section also defines the arguments for the operations. Often the phrase ''a list of names'' is used. Please note that one name is the degenerate case of a list of names, and that in Tcl, braces are only required on lists of more than one item. For example the `delete` operation is described below as taking a list of names as arguments. Both of the following are valid:

```
object delete {foo bar}
object delete foo
```

The descriptions in the sections on individual objects may override some of the information presented here. Usually this is only in the form of an operation accepting more switches, but other changes are possible.

(a) `add`

Adds an object to a container. It is implemented for all objects that represent containers. Returns an empty string on success. The argument is a list of names of containers. The required `-member` switch is used to specify the name of the member to be added to the containers. It's value is a list of members to be added. If lists are specified for both the `-member` switch and as the argument, then each member name is added to each container.

For example it is used to add a member to an RPC group, and is used to add an element to an RPC profile.

(b) `catalog`

Returns the names of all instances of an object. It takes no arguments. For example, the `principal catalog` command returns a list of all principals in the registry. Only implemented by those objects for which this is possible. By default, fullnames are returned. Some objects will support a `-simplename` switch which will return names in a shorter form (either relative or not fully qualified). The order of the returned list is dependent on the service.

(c) `create`

Creates a new instance of an object. Takes one argument which a list of names of instances to be created. Returns an empty string on success. Returns an error if the object already exists. For some objects this command takes a `-attribute` switch or a set of attribute switches to create attributes on the new object.

(d) `delete`

Destroys an instance of the object. Takes one argument which a list of names of instances to be deleted. Returns an empty string on success. If the object does not exist, an error is returned.

(e) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(f) `list`

Returns a list of the names of all the members of a container. This operation only returns names of members, never any other (e.g., attribute) information about the members. It is to be implemented for all objects that represent containers. The argument is a list of names of containers to return the children of. The order of the returned list is dependent on the service. If more than one container name is given as an argument, all the member names are returned in one list. The user may always use the output of this command as a list in a `foreach` statement. For example:

```
foreach i [dir list /.: -objects] {object show $i}
```

The names returned are fully qualified names by default. In some objects, the member's names are the same as the parent's with another simple name (RDN in X.500 terminology, name component in other terminology, everything following the last `/` in others) appended to the end. In others, such as RPC Profiles, the members must be fully qualified global names. Since some names *must* be fully qualified, the `list` operation by default returns all names as fully qualified.

Some object implement the following switches:

(i) `-recursive`

Returns the names of all descendants (not just immediate children) of the specified instance. All the names are returned as one list without any sub-levels. This means that some information can be lost.

(ii) `-simplename`

This switch indicates to return simple names, i.e., names relative to the parent. It will be implemented for all objects that can meaningfully return simple names; e.g., `rpcprofile`'s will not implement this, but `directory` will.

(g) `modify`

This operation is used to modify attributes, policies, counters, or any other information in an object. This fact means that all attributes, policies, counters, etc. in any one object must have unique names. It will not be available to all objects. Returns an empty string on success. The argument is a list of names of objects to modify. All objects are modified in the same way.

The specific modification is described by the use of one or more of the following switches. If more than one is used, then the whole modify operation is treated atomically in that either it all will work, or none of it will. The order of the switches does not matter. Each switch may only be used once per command invocation. If more then one attribute is to be added, then the value of that switch should be an attribute list.

(i) `-add`

Used to add an attribute to an object or merely to add values to an existing attribute. The value of this switch is an attribute list. If only an attribute type is specified after the switch, then it is added to the object with no values. If an attribute type and value(s) are given, then they are added to the object, with the attribute being created if necessary. If the attribute or value is already present an error is returned.

(ii) `-remove`

Used to remove an entire attribute or merely some values from an object. The value of this switch is an attribute list. If only the attribute type is specified after the switch, then the entire attribute is removed. If an attribute type and value is specified then only that value is removed. If it is the only value present in the attribute then the entire attribute is removed. If the attribute or value is not present an error is returned.

(iii) `-change`

Used to change one attribute value to another. The value of this switch is an attribute list. Each attribute in the list has its existing value replaced by the new value given in the attribute list. For multi-valued attributes, *all* existing values are removed, and replaced by all the values listed for the attribute in the specified attribute list. For some objects this function is equivalent to doing a `-remove` followed by an `-add` for the same attribute. If the attribute or value is not present an error is returned.

Some objects implement attribute switches in addition to attribute lists as described above in the section *Attribute Lists and Switches*. When this is done either attribute lists or attribute switches may be used on a single command line, but not both.

(h) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last. If the user wants them sorted they should use:

```
lsort [object operations]
```

All operations valid for the object are returned, including the `operations` operation.

(i) `remove`

Removes an object from a container. For example, it is used to remove a member from an RPC group or an element from an RPC profile. It is implemented for all objects that represent containers. The argument is a list of names of containers.

Requires one switch, `-member`, which is used to specify the name of the member to be removed from the container. The value is a list of names of members of the containers. If the value of this switch and the argument to the command are both lists, then each listed member is removed from each specified container. If the members do not exist an error is returned.

(j) `rename`

This operation changes the name of a specified object. The argument is a single name of an object to be renamed, i.e., it cannot be a list. Takes a required `-to` switch with a value of the new name. The value may not be a list. Returns an empty string on success.

(k) `show`

Returns information about an object instance. Objects can have various types of information such as attributes, counters, policies, etc. The `show` command is used to return any of this information. Switches are passed to the command to specify what information is to be returned. All of the switches used for this purpose are in the plural form. Each object defines a default switch, defining what information is to be displayed when no switches are passed. In almost all cases this will be `-attributes`, and unless specified otherwise, this should be assumed.

Unlike the `list` command which returns information about the members of a container, the `show` command only looks at the named object instance. If it is a container, it does *not* return information about the members, only the container itself. However, sometimes a container contains more than just the name of a member. For example, an RPC profile is a container that has no attributes, counters, or anything else. However, for each member there is additional information such as an annotation and a priority. This information is not in the member objects, only in the container stored with the reference to the member. This member information (or child pointer information in CDS parlance) can be displayed with the `show` command.

This command takes one argument which is a list of names of instances to be shown. If more than one name is included, the output is organized as if the command was invoked separately for each instance. The attributes (or other information) for each instance is displayed in a list, all captured as one element in a larger list. The common switches to the `show` command are:

(i) `-all` — Return a combination of all showable information.

(ii) `-attributes` — Return information about attributes.

(iii) `-counters` — Return information about counters.

(iv) `-members` *`name`* — Return information about the member identified by *`name`* that is stored in the container. In some cases *`name`* need not be specified in which case all information that is stored in the container about all members is returned.

(v) `-policies` — Return information about policies.

The output of the command is in the form of an attribute list. If different types of information are specified to be returned by the use of multiple switches or the `-all` switch, then the attribute lists are concatenated together. All attributes, counters, and policies are named from the same namespace and must be unique. For example, the `organization show` *`orgname`* `-all` command would return a list of the form:n

```
{fullname xxxx}
{orgid xxxx}
{uuid   xxxx}
{acctlife xxxx}
{pwdalpha xxxx}
{pwdexpdate xxxx}
{pwdlife xxxx}
{pwdminlen xxxx}
```

```
{pwdspaces xxxx}
```

Attributes (or other data) returned by the `show` operation are returned in an order defined by the object. For example, CDS attributes are returned in lexical order of the string representations of OID's of the attribute types, while registry commands return the data in fixed order.

(l) `synchronize`

Tells the instance to synchronize with any replicas of itself. In CDS terminology this will perform a skulk on a directory, in DTS it will cause a server to synchronize. It is to be implemented for all objects that support replication. Returns an empty string on success. The argument is a list of instance names to synchronize. If more than one instance name is given then each instance synchronizes, there is no relationship such as synchronize with each other, as this doesn't make sense for many objects.

### 8.10. Common switches

The following are switches that are implemented for some commands. When used, they behave in a common manner, although there may be slight differences in the implementations.

(a) `-annotation`

Used to specify an annotation string.

(b) `-attribute`

Used to specify an attribute consisting of a type and one or more values. Attributes are represented as a list where the first element is the type and the following elements are the values. Where applicable (e.g., in CDS) attribute types are specified, with either OID's or their string representation.

(c) `-binding`

Used to specify a string binding.

(d) `-default`

Used to specify that an operation is to be invoked on the default profile member (element). When this switch is specified other switches used to identify a particular profile member are illegal.

(e) `-interface`

Used to specify an interface identifier.

(f) `-member`

Used to identify an item in a container.

(g) `-object`

Used to specify an object UUID.

(h) `-recursive`

Used to have an operation recursively evaluate results. Some commands allow this switch to take a numeric value specifying the maximum depth of the recursion.

(i) `-syntax`

Specifies the naming syntax to use. DCE (and `dcecp`) will only support one syntax (the `dce` syntax) for DCE 1.1. This syntax will be the default for all commands so this switch will rarely be used.

(j) `-version`

Used to specify how the version specification of the interface identifier given in the `-interface` switch is used. Requires a value which is one of the following:

(i) `all`

The interface version is ignored, all values are acceptable. This is the default value.

(ii) `exact`

Both the major and minor versions must match the specified versions.

(iii) `compatible`

The major version must match the specified version, and the minor version must be greater than or equal to the specified version.

(iv) `majoronly`

The major version must match the specified version; the minor version is ignored.

(v) `upto`

The major and minor versions must be less than or equal to those specified.

### 8.11. Objects

The following are the object commands supported by `dcecp`. The supported operations are listed for each command. Users can add more objects by using the Tcl `proc` command to define a new command. The tasks described in the next section are examples of doing this.

#### 8.11.1. rpcentry

The `rpcentry` object represents an RPC server entry. The argument is a list of names of server entries. Each server entry contains a set of object UUID's, and a set of interface identifier/string binding pairs. Supports the following operations:

(a) `create`

Creates an empty entry. Since an empty entry is the same as an empty RPC group or RPC profile, calling `rpcentry create` is the same as calling `rpcgroup create` or `rpcprofile create`. The argument is a list of names of RPC entries to be created. Recognizes one switch: `-syntax` which specifies the syntax of the names in the argument. Returns an empty string on success. If the RPC entry already exists, an error is returned.

(b) `delete`

Deletes the specified entry. The argument is a list of names of server entries to be deleted. Recognizes the `-syntax` switch which specifies the syntax of the names in the argument. Returns an empty string on success. If the entry does not exist an error is returned.

(c) `export`

Exports binding information to the specified entry. The argument is a list of names of server entries to be exported to. If an entry does not exist, it is created. Uses the `-interface`, `-binding`, `-object` and `-syntax` switches to specify what to export. Returns an empty string on success.

This operation will also be able to export codeset and other attributes as specified in the RPC Internationalization specifications.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `import`

Returns a string binding from the specified RPC entry. The argument is a list of names of RPC entries (not a list of RPC entries) to import from. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify matching bindings. Each of these takes only one value not a list of values. Also accepts the `-max` switch to specify a number of string bindings to return. If the value is greater than one, then a list of as many matching bindings less than or equal to the value is returned. The order of bindings returned

is arbitrary.

This operation will also be able to distinguish acceptable entries based on codeset and other attributes as specified in the RPC Internationalization specifications.

(f) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(g) `show`

Returns a list containing the binding information in the RPC server entry. The list is composed of two lists. The first list is also a list, where the first element is an interface identifier, and the rest of the elements are string bindings. The second list is a list of Object UUID's exported by the server. The order of the data returned is arbitrary. The argument is a list of names of server entries to return information about.

This operation recognizes the `-interface` switch (its value is a list of interface id's) so that only string bindings matching that interface identifier are returned. The match is exact. The `-object` switch is used to specify a list of object UUID's to match. The `-noupdate` switch is used to avoid updating a local cache before performing the operation. Also takes a `-syntax` switch to specify the syntax of the name given as an argument.

This operation will also return codeset and other attributes as specified in the RPC Internationalization specifications.

(h) `unexport`

Removes binding information from an entry. The argument is a list of names of server entries to have information unexported from. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify what to unexport. Each can have only one value, not a list of values. Returns an empty string on success. An error is generated if the information is not found.

### 8.11.2. rpcgroup

The `rpcgroup` object represents an RPC group. Each RPC group is named in the DCE namespace; therefore each operation described below takes one argument which is the name of the group to act upon. An RPC group is a container of names of either RPC server entries or other RPC groups; there is no data in the group other than the name of the members. Supports the following operations:

(a) `add`

Adds a member to the specified group. The argument is a list of names of RPC groups to have members added to. The value of the required `-member` switch is a list of names which are references to an RPC entry, RPC group, or RPC profile (i.e., they do not have to actually exist). Also takes the `-syntax` switch to identify the naming syntax used of the argument. All members are added to each group mentioned in the argument list. Returns an empty string on success. If the new members already exist, an error is returned.

(b) `create`

Creates an empty group. Since an empty group is the same as an empty RPC entry or RPC profile, calling `rpcgroup create` is the same as calling `rpcentry create` or `rpcprofile create`. The argument is a list of names of RPC groups to be created. Recognizes one switch: `-syntax` which specifies the syntax of the names in the argument. Returns an empty string on success. If the RPC group already exists, an error is returned.

(c) `delete`

Deletes the specified group. The argument is a list of names of RPC groups to be deleted. Recognizes the `-syntax` switch to specify the name syntax of the arguments. The `-recursive` switch can be used to delete the group and all members; it does not take a numeric argument limiting the levels of recursion. When using this switch members are removed whether they are RPC entries, RPC groups, or RPC profiles.

Returns an empty string on success. If the RPC group does not exist an error is generated.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `import`

Returns a string binding from the specified RPC group. The argument is a list of names of RPC groups (not a list of RPC groups) to import from. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify matching bindings. Each of these takes only one value not a list of values. Also accepts the `-max` switch to specify a number of string bindings to return. If the value is greater than one, then a list of as many matching bindings less than or equal to the value is returned. The order of bindings returned is arbitrary.

This operation will also be able to distinguish acceptable entries based on codeset and other attributes as specified in the RPC Internationalization specifications.

(f) `list`

Returns a list of the names of all members of the specified group. The returned names are fully qualified names and are returned in an arbitrary order. The argument is a list of names of RPC groups to have their members names returned. The `-recursive` switch is used to list all the names in sub-groups as well. If called so that more than one RPC group is listed (either via the `-recursive` switch or due to more than one RPC group name being given in the argument) then the members are concatenated on output into one list.

(g) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(h) `remove`

Removes one or more members from the specified group. The argument is a list of names of RPC groups to have members removed from. The value of the required `-member` switch is a list of names of RPC entries, RPC groups, or RPC profiles; they are only references stored in the RPC group and does not have to actually exist outside of the group. All members specified are removed from all RPC groups specified in the argument. Also takes the `-syntax` switch to specify the name syntax of the arguments. Returns an empty string on success. If a specified member does not exist in an RPC group an error is returned.

### 8.11.3. rpcprofile

The `rpcprofile` object represents an RPC profile. Each operation described below takes one argument which names the RPC profiles to be acted upon. An RPC Profile consists of members (also known as elements in other DCE documentation) which can be either RPC server entries, RPC groups, or other RPC profiles; therefore each member of a profile has a name in the DCE namespace. Each profile can have one member which is the default member of the profile (a.k.a. the default profile element).

A profile contains no attributes, but it does contain information about each member, which is not contained in the member itself. This information is an interface identifier, a priority, and an optional annotation. They are identified by the following keywords: `interface`, `priority`, and `annotation`. The keyword `member` is used to identify the member name. Also note that each name available via the NSI, can have a different syntax. Currently there is only one supported syntax, the `dce` syntax, but most commands take a `-syntax` switch to allow specification of another syntax.

Supports the following operations:

(a) `add`

Adds a member to the specified profile. The argument is a list of names of RPC profiles to have members added to. The value of the required `-member` switch is a list of names which are references to an RPC entry, RPC group, or RPC profile (i.e., they do not have to actually exist). Accepts the `-interface`, `-priority`, and `-annotation` switches with one value (not a list) each. Also takes the `-syntax` switch to identify the naming syntax used of the argument. All members are added to each profile mentioned in the argument list. Also accepts a `-default` switch to indicate that the add operation is performed on the default profile member (other switches are illegal if the `-default` switch is used). Returns an empty string on success. If the new members already exist, an error is returned.

(b) `create`

Creates an empty profile. Since an empty profile is the same as an empty RPC entry or RPC group, calling `rpcprofile create` is the same as calling `rpcentry create` or `rpcgroup create`. The argument is a list of names of RPC profiles to be created. Recognizes one switch: `-syntax` which specifies the syntax of the names in the argument. Returns an empty string on success. If the RPC profile already exists, an error is returned.

(c) `delete`

Deletes the specified profile. The argument is a list of names of RPC profiles to be deleted. Recognizes the `-syntax` switch to specify the name syntax of the arguments. The `-recursive` switch can be used to delete the profile and all members; it does not take a numeric argument limiting the levels of recursion. When using this switch members are removed whether they are RPC entries, RPC groups, or RPC profiles. Returns an empty string on success. If the RPC profile does not exist an error is generated.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `import`

Returns a string binding from the specified RPC profile. The argument is a list of names of RPC profiles (not a list of RPC profiles) to import from. Uses the `-interface`, `-version`, `-object` and `-syntax` switches to specify matching bindings. Each of these takes only one value not a list of values. Also accepts the `-max` switch to specify a number of string bindings to return. If the value is greater than one, then a list of as many matching bindings less than or equal to the value is returned. The order of bindings returned is arbitrary.

This operation will also be able to distinguish acceptable entries based on codeset and other attributes as specified in the RPC Internationalization specifications.

(f) `list`

Returns a list of the names of all members of the specified profile. The returned names are fully qualified names and are returned in an arbitrary order. The argument is a list of names of RPC profiles to have their members names returned. The `-recursive` switch is used to list all the names in sub-groups and sub-profiles as well. If called so that more than one RPC profile is listed (either via the `-recursive` switch or due to more than one RPC profile name being given in the argument) then the members are concatenated on output into one list.

(g) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(h) `remove`

Removes one or more members from the specified profile. The argument is a list of names of RPC profiles to have members removed from. The members to be removed are those that match the values given in the following switches: `-member`, `-interface`, `-annotation`, `-syntax`. The values of each of

these switches is a list, any members that match any of these values are removed from all RPC profiles specified in the argument. Also takes the `-syntax` switch to specify the name syntax of the arguments. Also accepts a `-default` switch, in which case, the above switches are illegal and the default profile member is removed. Returns an empty string on success. If a specified member does not exist in an RPC profile an error is returned.

(i) `show`

Returns a list of all members of one or more profiles. The argument is a list of names of RPC profiles to have members of returned. An attribute list is returned for each member with all of the entered information. It is always in the order: `interface`, `member`, `priority`, `annotation`. If any of the items is not given, they are not included in the output, i.e., no place holder is included.

Only those members that match the values specified by the following optional switches are returned: `-interface`, `-version`, `-annotation`, `-member`, `-syntax`. Each switch may have only one value (i.e., the value may not be a list). Also accepts a `-default` switch, in which case, the above switches are illegal and the default profile member is returned. The `-recursive` switch (with a value) can be used to recursively display the elements of the profile.

### 8.11.4. endpoint

The `endpoint` object represents RPC endpoint mappings. Since endpoints have no names, none of these operations take an instance argument. Endpoints contain an interface identifier and one or more string bindings; optionally they contain object UUID's and an annotation. Supports the following operations:

(a) `create`

Creates new endpoints in the endpoint map database. This command takes no arguments. Requires the `-interface` and `-binding` switches. Also accepts the `-object` and `-annotation` switches. The value of the `-binding` and `-object` switches can be a list, but the others must be a single value. If the mapping already exists it is replaced unless the `-noreplace` switch is included. Returns an empty string on success.

(b) `delete`

Deletes the specified endpoints from the endpoint map database. This command takes no arguments. Requires the `-interface` and `-binding` switches; and accepts the `-object` switch. The values of all but the `-interface` switch may be lists. If the mappings don't exist an error is generated. Also supports a `-host` switch to specify the use of the endpoint map on another host. Returns an empty string on success.

(c) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(d) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(e) `show`

Returns a list of information about endpoints. With no switches returns all the endpoint mappings. The `-interface`, `-version`, and `-object` switches can be used so that only those endpoint mappings matching the values of the switches are returned. The `-object` switch accepts a list as a value the others do not. This command takes no arguments. Also supports a `-host` switch to specify the use of the endpoint map on another host.

### 8.11.5. object

The `object` object represents CDS objects. An object in the CDS directory service, has a name in the DCE Cell namespace, and a collection of attributes associated with it. Attributes consist of a type and one or more values. Every object is the child of a directory. See the `directory list` command for information on how to get a list of objects beneath a directory.

If the `s(cds)` convenience variable is set, it is treated as the name of a clearinghouse to contact for this operation. This is the only clearinghouse that will be contacted in an attempt to complete the operation. These commands do *not* set the value of this variable after completion.

There is no concept of current working directory, or cd'ing through the CDS namespace. Certainly this is a possibility for future expansion.

The following are the CDS defined attributes that may be present in CDS Objects.

(a) `CDS_Class`

Specifies the class to which an object belongs.

(b) `CDS_ClassVersion`

Contains the version number of the object's class. This allows applications to build in compatibility with entries created by earlier versions.

(c) `CDS_CTS`

Specifies the creation timestamp (CTS) of this object entry.

(d) `CDS_ObjectUUID`

Specifies a unique identifier for the object being referenced.

(e) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the object entry.

Supports the following operations:

(a) `create`

Creates a new object. The argument is a list of names of objects to be created. Takes an optional `-attribute` switch to specify a list of attributes to be included in each created object. Returns an empty string on success.

(b) `delete`

Deletes an object. The argument is a list of names of objects to be deleted. Returns an empty string on success. An error is returned if any of the objects do not exist.

(c) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(d) `modify`

Used to add or remove attributes and their values from the object. The argument is a list of names of objects to be modified. Accepts the `-add`, `-remove`, and `-change` switches which each can have an attribute list as the value. See the *Common Operations* section for details. Attribute switches are *not* supported for this command. Returns an empty string on success.

(e) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(f) `show`

Returns an attribute list of the attributes in the specified objects. The argument is a list of names of objects to be shown. If more than one object is to be shown, the attributes of all the objects are concatenated into one list. The order of the returned attributes is the lexical order of the OID's of each attribute for each object.

### 8.11.6. directory

The `directory` object represents CDS directories. CDS directories are containers for other objects, links, and other directories (as well as clearinghouses). If any of these items reside in a directory, they are said to be called children of that directory. Directories also contain attributes that may be viewed or modified.

This object also represents the CDS concept of replicas. Replicas are read-only copies of directories stored in other clearinghouses. Several of the supported operations take switches to indicate that the command is to operate on a specific replica.

If the `s(cds)` convenience variable is set, it is treated as the name of a clearinghouse to contact for this operation. This is the only clearinghouse that will be contacted in an attempt to complete the operation. These commands do *not* set the value of this variable after completion.

The following are the CDS defined attributes that may be present in directories and replicas in CDS:

(a) `CDS_AllUpTo`

Indicates the date and time of the last successful skulk on the directory. All replicas of the directory are guaranteed to receive all updates whose timestamps are less than the value of this attribute.

(b) `CDS_Convergence`

Specifies the degree of consistency among replicas. This attribute's value is defined as one of the following:

(i) `low`

CDS does not immediately propagate an update. The next skulk distributes all updates that occurred since the previous skulk. Skulks occur at least once every 24 hours.

(ii) `medium`

CDS attempts to immediately propagate an update to all replicas. If the attempt fails, the next scheduled skulk makes the replicas consistent. Skulks occur at least once every 12 hours.

(iii) `high`

CDS attempts to immediately propagate an update to all replicas. If the attempt fails (for example, if one of the replicas is unavailable), a skulk is scheduled for within one hour. Skulks usually occur at least once every 12 hours. Use this setting temporarily and briefly, because it uses extensive system resources.

By default, every directory inherits the convergence setting of its parent at creation time. The default setting on the root directory is `medium`.

(c) `CDS_CTS`

Specifies the creation timestamp (CTS) of the CDS directory.

(d) `CDS_DirectoryVersion`

Specifies the current version of the directory (derived from the `CDS_DirectoryVersion` attribute of the clearinghouse in which the directory was created). Multiple directory versions are supported in a cell.

(e) `CDS_Epoch`

A UUID that identifies a particular incarnation of the directory.

(f) `CDS_GDAPointers`

A set-valued attribute which is only present in the root directory of a cell. This attribute contains location information about registered Global Directory Agents for that cell, similar to the CDS_Replicas attribute. It is created and only used by a GDA.

(g) `CDS_InCHName`

Indicates whether a directory or any of its descendants can store clearinghouse names. If this value is `true`, the directory can store clearinghouse names. If it is `false`, the directory cannot store clearinghouse names.

(h) `CDS_LastSkulk`

Records the timestamp of the last skulk performed on this directory.

(i) `CDS_LastUpdate`

Records the timestamp of the most recent change to any attribute of a directory replica, or any change to an entry in the replica.

(j) `CDS_ObjectUUID`

Specifies the unique identifier of the directory.

(k) `CDS_ParentPointer`

Contains a pointer to this directory's parent in the namespace.

(l) `CDS_Replicas`

Specifies the address, UUID, and name of every clearinghouse where a copy of this directory is located. This attribute also specifies whether the replica in a particular clearinghouse is a master or read-only replica.

(m) `CDS_ReplicaState`

Specifies whether a directory replica can be accessed.

(n) `CDS_ReplicaType`

Indicates whether a directory replica is a master or read-only replica.

(o) `CDS_ReplicaVersion`

Specifies the version of a replica of the directory.

(p) `CDS_RingPointer`

Specifies the UUID of a clearinghouse containing another replica of this directory.

(q) `CDS_UpgradeTo`

A single-valued attribute used to control the upgrading of a directory from one version of CDS to another. By modifying this attribute, the process of upgrading a directory to a newer version of CDS may be initiated.

(r) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the directory.

Supports the following operations:

(a) `add`

This operation creates a child pointer in the parent directory.

The argument is a list of names of parent directories to have child pointers added to them. The value of the required `-member` switch is a list of names of child pointers to be added to each of the directories listed in the argument. Each child pointer name entered should only contain the last RDN of the name. The full

name of a clearinghouse that holds a replica of the child directory is given as the value to the required `-clearinghouse` switch. This switch may only have one value and is used for each of the values of the `-member` switch. Returns an empty string on success. If a child pointer of the same name already exists an error is returned.

This command is only needed to recreate a child pointer that was accidently deleted, i.e., in a troubleshooting situation. Normally child pointers are created internally by CDS when creating directories with the `directory create` command.

(b) `create`

Creates a new directory of the specified name. The argument is a list of names of directories to be created. Takes an optional `-clearinghouse` switch to specify a clearinghouse to create all the directories in. The value of this switch is not a list, it is only one clearinghouse name to create all the directories named in the argument in. If this switch is not specified, the new directories are created in the same clearinghouse as the parent directory. Also takes a `-replica` switch which indicates that a directory replica is created; when this switch is used, the `-clearinghouse` switch is required. Returns an empty string on success.

(c) `delete`

Used to delete a directory. The argument is a list of names of directories to be deleted. An optional `-recursive` switch (takes no value) is supported to remove the directory and everything (all directories, objects, links, and clearinghouses) beneath it. Also there is a `-replica` and a `-clearinghouse` switch which must be used together to delete a replica instead of a directory. The clearinghouse the replica is in is specified as the value the `-clearinghouse` switch; only one value can be specified, not a list. Returns an empty string on success. If a specified directory does not exist, an error is generated.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `list`

Returns a list of the names of all the descendants of a directory. The argument is a list of names of directories to be operated on. Descendants includes all directories, objects, links, and clearinghouses of the directory. Since only the names are returned there is no way to tell the class of each name unless by convention (e.g., most clearinghouses end with `_ch`). The following switches can be used to specify the types of descendents to return: `-directories`, `-objects`, `-links`. They take no values and can be used in combination. By default fullnames are returned, the `-simplename` switch can be used to return merely the last RDN of the name.

(f) `modify`

Used to add, remove or change attributes and their values from the directory. The argument is a list of names of directories to be operated on. Attribute switches are not supported; use one or more of `-add`, `-remove` or `-change` which each take an attribute list as an argument. All modifications are made to each directory listed in the argument. An error in any one causes the command to immediately abort and generate an error. Returns an empty string on success.

(g) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(h) `remove`

This operation deletes a child pointer from the directories specified. The argument is a list of names of directories to be operated on. The value of the required `-member` switch is a list of child pointers (specified as only one RDN each) to be removed from each directory in the argument. Returns an empty string on success.

This command is only needed to delete a child pointer that accidently remains after the child directory is deleted. Normally child pointers are removed internally by CDS when deleting directories with the `directory delete` command.

(i) `show`

Returns a list of attributes in the directories specified. The argument is a list of names of directories to be operated on. If more than one directory is specified then all the arguments are grouped together in one list. The order of the returned arguments is the lexical order of the OID's of each attribute for each directory.

Takes a `-replica` and a `-clearinghouse` switch to specify that the contents of a replica are to be returned. This will return the directory specific and replica specific attributes for the replica. The `-replica` switch takes no value. The value of the `-clearinghouse` switch is the name of one clearinghouse (not a list of names) to be used for all replicas. If one of the directories mentioned in the argument list is not in the specified clearinghouse then an error is returned.

Also takes an optional `-member` switch with one required value which is the last RDN of child pointer in this directory. In this case the returned list describes the child pointer information for the specified member stored in the specified directories. This switch may not be combined with the `-replica` or `-clearinghouse` switch.

(j) `synchronize`

Initiates a directory skulk of the directories specified. The argument is a list of names of directories to be operated on. Skulks are performed serially and begun immediately and the command does not return until all are finished. Returns an empty string on success.

### 8.11.7. link

The `link` object represents CDS softlinks. A softlink in the CDS directory service contains an attribute that has a name which is the name that the softlink points to. The softlink contains several built-in attributes, but users not free to add their own. Softlinks can point to objects, directories, and other softlinks.

If the `s(cds)` convenience variable is set, it is treated as the name of a clearinghouse to contact for this operation. This is the only clearinghouse that will be contacted in an attempt to complete the operation. These commands do *not* set the value of this variable after completion.

The following are the CDS defined attributes that may be present in a CDS softlink entry.

(a) `CDS_CTS`

Specifies the creation timestamp (CTS) of this softlink.

(b) `CDS_LinkTarget`

Specifies the full name of the directory, object entry, or other softlink to which the softlink points.

(c) `CDS_LinkTimeout`

Specifies a timeout value after which the softlink is either renewed or deleted. Its value is a list of two elements:

(i) The expiration-time. A date and time after which CDS checks for the existence of the softlink's target and either extends or deletes the soft link. The value is specified as *yyyy-mm-dd-hh:mm:ss*, and portions of it can be defaulted.

(ii) The extension-time. A period of time by which to extend the softlink's expiration time (if the server has validated that the target still exists). The value is specified as *ddd-hh:mm:ss*, and portions of it can be defaulted.

(d) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the softlink.

Supports the following operations:

(a) `create`

Creates a new softlink. The argument is a list of names of softlinks to be created. There is one required switch, `-linkto` which takes a single value, a name that these new links point to. Additionally, the `-timeout` switch can be used to specify a value for the `CDS_LinkTimeout` attribute in each softline (i.e., this switch takes a single value, not a list, which is applied to all softlinks mentioned in the argument). If this attribute is not included in the command line then the new softlinks are permanent and must be explicitly deleted. Returns an empty string on success.

(b) `delete`

Deletes a softlink. The argument is a list of names of softlinks to be deleted. Returns an empty string on success. An error is returned if any of the objects do not exist.

(c) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(d) `modify`

The `modify` operation can be used to change two attributes of a softlink: `CDS_LinkTarget` and `CDS_LinkTimeout`. The argument is a list of names of softlinks to be operated on. Takes the `-add`, `-remove`, and `-change` switches to specify an attribute list to describe the changes. All the changes are performed on each softlink named in the argument. Returns an empty string on success.

(e) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(f) `show`

Returns an attribute list of the attributes in the specified softlinks. The argument is a list of names of softlinks to be operated on. If more than one softlink is to be shown, the attributes of all the softlinks are concatenated into one list. The order of the returned attributes is the lexical order of the OID's of each attribute for each softlink.

### 8.11.8. clearinghouse

The `clearinghouse` object represents CDS clearinghouses. Clearinghouses are databases located on CDS server machines that store the data in the CDS directory service. On the server machines are files that contain the actual clearinghouse data. Clearinghouses are also represented in the CDS namespace by an entry that contains information about the clearinghouse.

If the `s(cds)` convenience variable is set, it is treated as the name of a clearinghouse to contact for this operation. This is the only clearinghouse that will be contacted in an attempt to complete the operation. These commands do *not* set the value of this variable after completion.

The following are the CDS defined attributes that may be present in a clearinghouse:

(a) `CDS_AllUpTo`

Indicates the date and time the clearinghouse object has been updated to reflect the `CDS_CHDirectories` attribute.

(b) `CDS_CHDirectories`

Specifies the full name and unique identifier (UUID) of every directory that has a replica in this clearinghouse.

(c) `CDS_CHLastAddress`

Specifies the current reported network address of the clearinghouse.

(d) `CDS_CHName`

Specifies the full name of the clearinghouse.

(e) `CDS_CHState`

Specifies the state of the clearinghouse. The state `on` indicates the clearinghouse is running and available.

(f) `CDS_CTS`

Specifies the creation timestamp (CTS) of the clearinghouse.

(g) `CDS_DirectoryVersion`

Specifies the current version of the directory in the clearinghouse in which the directory was created.

(h) `CDS_NSCellname`

Specifies the name of the cell in which the clearinghouse resides.

(i) `CDS_ObjectUUID`

Specifies the unique identifier of the clearinghouse.

(j) `CDS_ReplicaVersion`

Specifies the current version of the replica in which the directory was created.

(k) `CDS_UpgradeTo`

A single-valued attribute used to control the upgrading of a directory from one version of CDS to another. By modifying this attribute, the process of upgrading a directory to a newer version of CDS may be initiated.

(l) `CDS_UTS`

Specifies the timestamp of the most recent update to an attribute of the clearinghouse.

The counters are:

(a) `badchs`

Specifies the number of times the `clearinghouse entry missing` event was generated.

(b) `baddata`

Specifies the number of times that the `data corruption` event was generated.

(c) `badskulks`

Specifies the number of times that a skulk of a directory, initiated from this clearinghouse, failed to complete— usually because one of the replicas in the replica set was unreachable.

(d) `badupgrades`

Specifies the number of times that upgrades failed.

(e) `disables`

Specifies the number of times that the clearinghouse was disabled since it was last started.

(f) `enables`

Specifies the number of times that the clearinghouse was enabled since it was last started.

(g) `partials`

Specifies the number of requests directed to this clearinghouse that resulted in the return of a partial answer instead of satisfying the client's request.

(h) `reads`

Specifies the number of read operations directed to this clearinghouse.

(i) `rootlosts`

Specifies the number of times the `root lost` event was generated.

(j) `writes`

Specifies the number of write operations directed to this clearinghouse.

Supports the following operations:

(a) `create`

Creates a new clearinghouse on the local machine. Takes one argument which is the name of a new clearinghouse (it is not a list). Clearinghouses should only be named in the root directory (i.e., `/.:`). When this operation is invoked, a read-only replica of the root directory is stored in the new clearinghouse. Because the the process that creates the new clearinghouse initiates a skulk of the root directory, all replicas of the root should be reachable when you enter the command. To ensure this, perform an immediate skulk of `/.:` prior to invoking this command, using the `directory synchronize /.:` command. Returns an empty string on success.

(b) `delete`

This operation deletes the clearinghouse given as an argument from the *local* machine. The argument is a list of names of clearinghouses to be deleted. Clearinghouses that contain master replicas of directories may not be deleted. This command also automatically deletes all read-only replicas from the clearinghouse, however it is recommended that all read-only replicas be deleted by hand *before* invoking this command since invoking many skulks will cause the command to execute slower. Returns an empty string on success.

(c) `disable`

This operation removes knowledge of the specified clearinghouse from the server running on the *local* machine. The argument is a list of names of clearinghouses to be operated on. It is useful when relocating a clearinghouse. Specifically the name of the prefix of the clearinghouse files is removed from the `/opt/dcelocal/var/directory/cds/cds_files` file and the local server is contacted and notified that the clearinghouse is disabled. The clearinghouse entry is *not* removed from the namespace, nor are the datafiles associated with the clearinghouse removed. Returns an empty string on success.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(f) `show`

Returns an attribute list showing the attributes of the clearinghouse named as the argument. The argument is a list of names of clearinghouses to be operated on. If more than one clearinghouse is given as an argument, all the attributes from each are concatenated together into one list. The order of the returned attributes is the lexical order of the OID's of each attribute for each clearinghouse.

If given a `-counters` switch returns an attribute list of the counters in the clearinghouse. The `-all` switch can be used to return both attribute and counter information. The order is the attributes first then the counters as if the two commands were done one after the other. Counters are displayed in the order returned

by the server, for example:

```
dcecp> clearinghouse show /.:/foo_ch -counters
{read_accesses 2336}
{write_accesses 68}
{references_returned 2}
{root_not_reachable 0}
{data_corruption 0}
{skulk_failures 0}
{entry_missing 0}
{upgrade_fail 0}
{enables 1}
{disables 0}
```

### 8.11.9. cdscache

The `cdscache` object represents the cds cache on the local node. The CDS cache contains information about servers and clearinghouses known to the local machine, and also contains user data about CDS entries that have been read. The `create` and `delete` operations only apply to the server information. The `show` and `dump` commands can display other information. Supports the following operations:

(a) `create`

This operation adds information about a CDS server to the local CDS cache. The argument is a list of names of servers to be added to the cache. These are the simple names of the machine, not the name of entry in the DCE namespace for the server (e.g. `absolut` not `/.:/hosts/absolut/self`). There is also one required switch, `-binding`. This is the binding information for the server to be added to the CDS cache. The value must be a single partial string binding, not a list. Returns an empty string on success.

(b) `delete`

This operation is used to remove information about a CDS server from the local CDS cache. The argument is a list of names of servers to be deleted from the cache. It can only be used to remove information added with the `cdscache create` command. Returns an empty string on success.

(c) `dump`

Returns all information stored in the local CDS cache. It takes no argument. Returns an empty string on success. This command is meant to be used when debugging CDS problems. As such the output is not in a form easily parsed by Tcl. This command may be removed from a production environment.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(f) `show`

Returns information stored in the CDS cache. The argument is a list of names of either clearinghouses or (simple names of) servers about whose information from the cache is returned. Must be invoked with only one of the following switches: `-server` or `-clearinghouse` to identify the type of the argument.

### 8.11.10. principal

The `principal` object represents Registry principals. Unless otherwise noted, all of the operations of this object take one argument which is a list of names of principals to act on. They must be principal names, not the names of

the database objects that contains registry information about principals (i.e., they should not begin with `/.:/sec/principal/`).

After this command executes, the `s(sec)` convenience variable is set to the name of the server that was bound to for the command. The value of the variable before the command is treated as a hint, i.e., the server specified will be contacted if it can service the request. A case where it can't service the request is if a read-only registry was bound to, and the next command is a `create` command. In this case, the master registry will be bound to automatically and the `s(sec)` variable updated appropriately. The value of the variable is the name of registry bound to in one of the formats specified as valid for the argument to the `registry` object.

In the registry, each principal has a primary name and a value for each of the following attributes:

(a) `alias`

The value of this attribute is either `yes` or `no`. Each principal can have only one name, but may have one or more alias names. All these names refer to the same principal, and therefore the same UUID and uid. While aliases refer to the same principal they are separate entries in the registry database. Therefore the instance name given to a `principal` command can refer to either the primary name or an alias name of a principal. The value of this attribute determines this.

(b) `fullname`

The full name of the principal, it is for information purposes only. It typically describes or expands a primary name to allow easy recognition by users. For example, a principal could have a primary name of `jsbach` and a fullname of `Johann S. Bach`. The value is a string, if it contains spaces, it is displayed in quotes, and on entry must be in quotes or braces (as per Tcl quoting rules). If not entered, the fullname defaults to the null string (that is, blank).

(c) `quota`

Specifies the principal's object creation quota, the total number of registry objects that can be created by the principal. It is either a non-negative number or the string `unlimited`. A value of `0` prohibits the principal from creating any registry objects. Each time a principal creates a registry object, this value is decremented for that principal.

(d) `uid`

The User Identifier for the principal. No two principals can have the same uid. However, aliases can share one uid. It is often called the Unix ID and is an integer.

(e) `uuid`

An internal identifier for a principal. No two principals can have the same UUID.

Supports the following operations:

(a) `catalog`

Returns a list of the names of all principals in the registry. It takes no arguments. By default, fully qualified names are returned in the form *`cellname/principalname`*. If the `-simplename` switch is given, then the cellname is not prepended to the front of the principal names. Names are returned in lexical order.

(b) `create`

Creates a new principal. The argument is a list of names of principals to be created. Returns an empty string on success. Switches are used to specify the attributes of the newly created principal. All switches are applied to all principals in the argument.[9]

9. Therefore do not use the `-uid` switch when creating two principals with the same command, the second will fail since the uid is already in use after the first is created. However, the `-alias` and `-uid` switches can be used to create several aliases to an existing principal with one command.

Accepts a `-attribute` switch to specify the attributes as an attribute list. The attribute switches are:

(i) `-alias`

Takes a value of either `yes` or `no` to specify if the newly created principal is an alias or not. The value of this attribute defaults to `no`[10] When creating an alias, the `-uid` switch is required to specify the uid of the primary principal to alias. For example, the following command creates an alias called `postmaster` for the principal with uid 1234:

```
principal create postmaster -uid 1234 -alias
```

(ii) `-fullname`

The full name of the principal to be added to the registry. The value is a string. If it contains spaces, it must be in quotes or braces. If not entered, the fullname defaults to the null string (that is, blank).

(iii) `-quota`

Specifies the principal's object creation quota, the total number of registry objects that can be created by the principal. It is either a non-negative number or the string `unlimited`. If you do not specify this switch the object creation quota defaults to `unlimited`.

(iv) `-uid`

The User Identifier for the principal. No two principals can have the same uid, however, aliases can share one uid with a principal. If this switch is not present, then a uid will be assigned to the principal automatically. The value of this switch is an integer.

(v) `-uuid`

This switch is only used to ''adopt'' an orphaned UUID. The value of this switch is a UUID. Normally the UUID for a new principal is generated by the registry. In cases where data exists tagged with a UUID of a principal that has been deleted from the registry, this switch can be used on the `create` command to specify the old UUID for a new principal. The UUID specified *must* be an orphan, i.e., a UUID for which no name exists in the registry. An error occurs if you specify a name (or uid, see below) that is already defined in the registry.

The `-alias` switch may *not* be used with this switch. Both the `-fullname` and `-quota` switches may be.

If the principal is a cell principal, the `-uid` switch can be used to specify the UNIX number to be associated with the principal. (If you do not enter this switch for a cell principal, the next sequential UNIX number is supplied as a default by the registry.) For all principals other than cell principals, the uid is extracted from information embedded in the principal's UUID and cannot be specified here.

(c) `delete`

This operation deletes principals from the registry. When a principal is deleted, the principal's account is deleted as well. The argument is a list of names of principals to be deleted. Note that these names can be either a primary or alias names. In either case, an account associated with that name is deleted. If a named principal does not exist an error is generated. Returns an empty string on success.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

---

10. So specifying `-alias no` is never required.

(e) `modify`

This operation is used to change attributes of principals. The argument is a list of names of principals to be operated on. All modifications are applied to all principals named in the argument. Principals are modified in the order they are listed and all modifications to an *individual* principal are atomic. Modifications to multiple principals are not atomic. A failure for any one principal in a list causes an error to be generated and the rest of the operation to be aborted. Returns an empty string on success.

The `-change` switch can be used to modify the value of any one of the attributes.[11] Its value of the `-change` switch is an attribute list describing the new values for the specified attributes. Also supports the following attribute switches: `-alias`, `-quota`, and `-fullname`.

(f) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(g) `rename`

This operation changes the name of a specified group. The argument is a single name of a group to be renamed. Takes a required `-to` switch with a value of the new name. The value may not be a list. Returns an empty string on success.

(h) `show`

This operation returns an attribute list describing the specified principals. The argument is a list of names of principals to be operated on. If more than one principal is given, then the attributes are concatenated together. There is one attribute in addition to the attributes described above. It is called `groups` and its value is a list of group names that the principal is a member of. Attributes are returned in the following order: `fullname`, `uid`, `uuid`, `alias` and `quota`.

### 8.11.11. group

The `group` object represents Registry groups. Groups are collections of principal names. Unless otherwise noted, all of the operations of this object take one argument which is the name of the group to act on. It must be a group name, not the name of the database object that contains the registry information about the group (i.e., it should not begin with `/.:/sec/group/`).

After this command executes, the `s(sec)` convenience variable is set to the name of the server that was bound to for the command. The value of the variable before the command is treated as a hint, i.e., the server specified will be contacted if it can service the request. A case where it can't service the request is if a read-only registry was bound to, and the next command is a `create` command. In this case, the master registry will be bound to automatically and the `s(sec)` variable updated appropriately. The value of the variable is the name of registry bound to in one of the formats specified as valid for the argument to the `registry` object.

In the registry, each group has a primary name and a value for each of the following attributes:

(a) `alias`

The value of this attribute is either `yes` or `no`. Each group can have only one name, but may have one or more alias names. All these names refer to the same group, and therefore the same UUID and gid. While aliases refer to the same group they are separate entries in the registry database. Therefore the instance name given to a `group` command can refer to either the primary name or an alias name of a group. The value of this attribute determines this.

---

11. The `-add` and `-remove` switches are not supported, as each attribute associated with a principal must always have a value.

(b) `fullname`

The full name of the group, it is for information purposes only. It typically describes or expands a primary name to allow easy recognition by users. For example, a group could have a primary name of `dce` and a fullname of `DCE Development`. The value is a string, if it contains spaces, it is displayed in quotes, and on entry must be in quotes or braces (as per Tcl quoting rules). If not entered, the fullname defaults to the null string (that is, blank).

(c) `gid`

The User Identifier for the group. No two groups can have the same gid, however, aliases can share one gid. It is often called the Unix ID and is an integer.

(d) `inprojlist`

The value of this attribute is either `yes` or `no`. If it is `no` then members of this group will not accrue the access rights of this group.

(e) `uuid`

An internal identifier for a group. No two groups can have the same UUID.

Supports the following operations:

(a) `add`

This command is used to add a members to a group. The argument is a list of names of groups to be have members added to. The value of the required `-member` switch is a list of names of principals to be added to each group in the argument. The principals must exist or the command will return an error. Returns an empty string on success.

(b) `catalog`

Returns a list of the names of all groups in the registry. It takes no arguments. By default, fully qualified names are returned in the form *`cellname/groupname`*. If the `-simplename` switch is given then the cellname is not prepended to the front of the group names. The groups are returned in lexical order.

(c) `create`

Creates a new group. The argument is a list of names of groups to be created. Returns an empty string on success. Switches are used to specify the attributes of the newly created group. All switches are applied to all groups in the argument.[12]

Accepts a `-attribute` switch to specify the attributes as an attribute list. The attribute switches are:

(i) `-alias`

Takes a value of either `yes` or `no` to specify if the newly created group is an alias or not. The value of this attribute defaults to `no`[13] When creating an alias, the `-gid` switch is required to specify the gid of the primary group to alias.

(ii) `-fullname`

The full name of the group to be added to the registry. The value is a string. If it contains spaces, it must be in quotes or braces. If not entered, the fullname defaults to the null string (that is, blank).

---

12. Therefore do not use the `-gid` switch when creating two groups with the same command, the second will fail since the gid is already in use after the first is created. However, the `-alias` and `-gid` switches can be used to create several aliases to an existing group with one command.

13. So specifying `-alias no` is never required.

(iii) `-gid`

The Group Identifier for the group. If this switch is not present, then a gid will be assigned to the group automatically.

(iv) `-inprojlist`

Specifies if the group is to be included in the principal's project list. The value of this field is either `yes` or `no`. If this switch is not included in the command, then the `yes` value is used as a default.

(v) `-uuid`

This switch is only used to ''adopt'' an orphaned UUID. Normally the UUID for a new group is generated by the registry. In cases where data exists tagged with a UUID of a group that has been deleted from the registry, this switch can be used on the `create` command to specify the old UUID for a new group. The UUID specified *must* be an orphan, i.e., a UUID for which no name exists in the registry. An error occurs if you specify a name that is already defined in the registry.

Neither the `-alias` or `-gid` switches may be used with this switch. Both the `-fullname` and `-inprojlist` switches may be.

(d) `delete`

This operation deletes a group from the registry. When a group is deleted, any accounts associated with that group are deleted as well. The argument is a list of names of groups to be deleted. If the groups do not exist an error is generated. Returns an empty string on success.

(e) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(f) `list`

Returns a list of the names of all members of a group. The argument is a list of names of groups to be operated on. If more than one group is listed, the names are concatenated together on output. By default, fully qualified names are returned in the form *`cellname/name`*. If the `-simplename` switch is given then the cellname is not prepended to the front of the member names. The members of each group are listed in lexical order. Returns an empty string on success.

(g) `modify`

This operation is used to change attributes of groups. The argument is a list of names of groups to be operated on. All modifications are applied to all groups named in the argument. Groups are modified in the order they are listed and all modifications to an *individual* group are atomic. Modifications to multiple groups are not atomic. A failure for any one group in a list causes an error to be generated and the rest of the operation to be aborted. Returns an empty string on success.

The `-change` switch can be used to modify the value of any one of the attributes.[14] Its value is an attribute list describing the new values for the specified attributes. Also supports the following attribute switches: `-alias`, `-inprojlist`, and `-fullname`.

(h) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

---

14. The `-add` and `-remove` switches are not supported, as each attribute associated with a group must always have a value.

(i) `remove`

This command is used to remove a member from a group. The argument is a list of names of groups to have members removed from. The value of the required `-member` switch is a list of names of principals which will be removed from the groups listed in the argument. When a member is removed from a group, any accounts associated with that principal and group are deleted.[15] Returns an empty string on success.

(j) `rename`

This operation changes the name of a specified group. The argument is a single name of a principal to be renamed. Takes a required `-to` switch with a value of the new name. The value may not be a list. Returns an empty string on success.

(k) `show`

This operation returns an attribute list describing the specified groups. The argument is a list of names of groups to be operated on. If more than one group is given, then the attributes are concatenated together. Attributes are returned in the following order: `fullname`, `gid`, `uuid`, `alias` and `inprojlist`.

### 8.11.12. organization

The `organization` object represents Registry organizations. Organizations are collections of principal names. Unless otherwise noted, all of the operations of this object take one argument which is the name of the organization to act on. It must be an organization name, not the name of the database object that contains the registry information about the organization (i.e., it should not begin with `/.:/sec/org/`).

After this command executes, the `s(sec)` convenience variable is set to the name of the server that was bound to for the command. The value of the variable before the command is treated as a hint, i.e., the server specified will be contacted if it can service the request. A case where it can't service the request is if a read-only registry was bound to, and the next command is a `create` command. In this case, the master registry will be bound to automatically and the `s(sec)` variable updated appropriately. The value of the variable is the name of registry bound to in one of the formats specified as valid for the argument to the `registry` object.

Each organization has both attributes and policies associated with them. Policies regulate such things as account and password lifetimes for all accounts associated with a particular organization. In fact, these policies are the main feature that organizations provide. There are two types of policies referred to here, those associated with a particular organization, and those that are registry wide. Please note that there are other policies that are associated with accounts (and they also can be registry-wide).

The attributes are:

(a) `fullname`

The full name of the organization, it is for information purposes only. It typically describes or expands a primary name to allow easy recognition by users. For example, an organization could have a primary name of `osf` and a fullname of `Open Software Foundation`. The value is a string, if it contains spaces, it is displayed in quotes, and on entry must be in quotes or braces (as per Tcl quoting rules). If not entered, the fullname defaults to the null string (that is, blank).

(b) `orgid`

The User Identifier for the organization. No two organizations can have the same orgid. However, aliases can share one orgid. It is often called the Unix ID and is an integer.

(c) `uuid`

---

15. Remember that accounts are associated with a principal, a group, and an organization; so the statement means that any accounts where *both* the principal name and group name match those given to this command are removed, but accounts where only one matches are untouched.

An internal identifier for a organization. No two organizations can have the same UUID. In normal use, this attribute will be set by the system and will be displayed for administrators, in some unusual cases an admin might want to set this value explicitly.

The policies are:

(a) `acctlife`

This policy defines the lifespan of accounts. It is specified as either an integer (for which the units are days) or the string `unlimited`.

(b) `pwdalpha`

This policy defines whether or not passwords can consist entirely of alphanumeric characters. Its value is either `yes` or `no`.

(c) `pwdexpdate`

This policy defines a date on which a password will expire. The date is entered as an internationalized date string or the string `none` in which case, there is no expiration date for the password.

(d) `pwdlife`

This policy defines the lifespan of passwords. It is specified as either an integer number of days or the string `unlimited`. The tag `days` can follow the integer on output and is ignored on input, for example:

```
{pwdlife 22 days}
```

(e) `pwdminlen`

This policy defines the minimum number of characters in a password. Its value is a positive integer or the integer `0` which means there is no minimum length.

(f) `pwdspaces`

This policy defines whether or not passwords can consist entirely of spaces. Its value is either `yes` or `no`.

Supports the following operations:

(a) `add`

This command is used to add a members to an organization. The argument is a list of names of organizations to be have members added to. The value of the required `-member` switch is a list of names of principals to be added to each organization in the argument. The principals must exist or the command will return an error. Returns an empty string on success.

(b) `catalog`

Returns a list of the names of all organizations in the registry. It takes no arguments. By default, fully qualified names are returned in the form *`cellname/name`*. If the `-simplename` switch is given then the cellname is not prepended to the front of the organization names. The organizations are returned in lexical order.

(c) `create`

Creates a new organization. The argument is a list of names of organizations to be created. Returns an empty string on success. Switches are used to specify the attributes of the newly created organization. All switches are applied to all organizations in the argument.[16]

16. Therefore do not use the `-orgid` switch when creating two organizations with the same command, the second will fail since the orgid is already in use after the first is created. However, the `-alias` and `-orgid` switches can be used to create several aliases to an existing organization with one command.

Supports both attribute lists (with a `-attribute` switch) and attribute switches named after each attribute and policy. If an orgid is not entered, then one will be assigned to the organization automatically. A UUID should only be specified to ''adopt'' an orphaned UUID. Normally the UUID for a new organization is generated by the registry. In cases where data exists tagged with a UUID of an organization that has been deleted from the registry, it can be entered on the `create` command line to specify the old UUID for a new organization. The UUID specified *must* be an orphan, i.e., a UUID for which no name exists in the registry. An error occurs if you specify a name that is already defined in the registry. The `orgid` attribute may *not* be specified if the `uuid` is, but the `fullname` attribute may be. Examples:

```
org create dce -attr {{fullname {Dist Comp Env}} \
{orgid 101}}
org create dce -fullname {Dist Comp Env} -orgid 101
org create dce -fullname {Dist Comp Env} \
-uuid c2aac790-dc6c-11cc-a6f8-080009251352
```

(d) `delete`

This operation deletes the organization from the registry. When an organization is deleted, any accounts associated with that organization are deleted as well. The argument is a list of names of organizations to be deleted. If the organizations do not exist an error is generated. Returns an empty string on success.

(e) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(f) `list`

Returns a list of the names of all members of a organization. The argument is a list of names of organizations to be operated on. If more than one organization is listed, the names are concatenated together on output. By default, fully qualified names are returned in the form *`cellname/name`*. If the `-simplename` switch is given then the cellname is not prepended to the front of the member names. The members of each organization are listed in lexical order. Returns an empty string on success.

(g) `modify`

This operation is used to change attributes and policies of organizations. To change registry-wide policies, see the `registry` command. The argument is a list of names of organizations to be operated on. All modifications are applied to all organizations named in the argument. Organizations are modified in the order they are listed and all modifications to an *individual* organization are atomic. Modifications to multiple organizations are not atomic. A failure for any one organization in a list causes an error to be generated and the rest of the operation to be aborted. Returns an empty string on success.

The `-change` switch can be used to modify the value of any one of the attributes.[17] Its value is an attribute list describing the new values for the specified attributes or policies. Also supports the following attribute switches: `-orgid`, `-uuid`, `-fullname`, `acctlife`, `pwdalpha`, `pwdexpdate`, `pwdlife`, `pwdminlen`, and `pwdspaces`. Examples:

```
org modify dce -attr {{fullname {Dist Comp Env}} \
{orgid 101}}
org modify dce -fullname {Dist Comp Env} -orgid 101
```

(h) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that

17. The `-add` and `-remove` switches are not supported, as each attribute associated with a group must always have a value.

`help` and `operations` are listed last.

(i) `remove`

This command is used to remove a member from a organization. The argument is a list of names of organizations to have members removed from. The value of the required `-member` switch is a list of names of principals which will be removed from the organizations listed in the argument. When a member is removed from a organization, any accounts associated with that principal and organization are deleted.[18] Returns an empty string on success.

(j) `rename`

This operation changes the name of a specified organization. The argument is a single name of a principal to be renamed. Takes a required `-to` switch with a value of the new name. The value may not be a list. Returns an empty string on success.

(k) `show`

This operation returns an attribute list describing the specified organizations. The argument is a list of names of organizations to be operated on. If more than one organization is given, then the attributes and policies are concatenated together. Accepts a `-policies` switch that returns the polices of the organization instead of the attributes. A `-all` switch can be used to return the attributes followed by the policies.

Attributes are returned in the following order: `fullname`, `orgid`, `uuid` Policies are returned in the following order: `acctlife`, `pwdalpha`, `pwdexpdate`, `pwdlife`, `pwdminlen`, and `pwdspaces`. If the organization does not have any policies then `nopolicies` is returned.

The policies that are actually in effect can be different from the organization policies due to conflicts with registry wide policies. If this is the case, the show command will alter the attribute structure on output to include an `effective` tag and the effective value, much in the same way ACL's handle masks (see *ACL's* in the *Data Structures* section for details). For example:

```
dcecp> org show foo -policies
{acctlife 30 days}
{pwdalpha no}
{pwdexpdate none}
{pwdlife unlimited effective 5 days}
{pwdminlen 6}
{pwdspaces no}
```

### 8.11.13. account

The `account` object represents Registry accounts. Each account is associated with one principal, one group and one organization. However, accounts are named by just the principal name. Aliases are differentiated for principals, so one principal can have multiple accounts under different alias names. The account has attributes to specify the group and organization of the account, these must be specified on account creation.

After this command executes, the `s(sec)` convenience variable is set to the name of the server that was bound to for the command. The value of the variable before the command is treated as a hint, i.e., the server specified will be contacted if it can service the request. A case where it can't service the request is if a read-only registry was bound to, and the next command is a `create` command. In this case, the master registry will be bound to automatically and the `s(sec)` variable updated appropriately. The value of the variable is the name of registry bound to in one of the formats specified as valid for the argument to the `registry` object. Each account also has a password associated with it.

---

18. Remember that accounts are associated with a principal, a group, and an organization; so the statement means that any accounts where *both* the principal name and organization name match those given to this command are removed, but accounts where only one matches are untouched.

Accounts contain the following attributes:

(a) `acctvalid`

A flag set to determine account validity. Possible values are either `yes` or `no`. An account with an `acctvalid` attribute set to `no` is invalid and cannot be logged into. The default is `yes`.

(b) `client`

A flag set to indicate whether or not the account is for a principal that can act as a client. Possible values are either `yes` or `no`. If you set this flag to `yes`, the principal is able to log in to the account and acquire tickets for authentication. The default is `yes`.

(c) `created`

A list of two items. The first is the principal name of the creator of the account, the second is an ISO timestamp showing the time of creation. This attribute is set by the system and may not be modified.

(d) `description`

A text string (limited to PCS) typically used to describe the use of the account.

(e) `dupkey`

A flag set to determine if tickets issued to the account's principal can have duplicate keys. Possible values are either `yes` or `no`. The default is `no`.

(f) `expdate`

The date on which the account expires. To renew the account, change the date in this field. The value is an ISO timestamp or the string `none`.. The default is `none`.

(g) `forwardabletkt`

A flag set to determine whether a new ticket-granting ticket with a network address that differs from the present ticket-granting ticket network address can be issued to the account's principal. The `proxiabletkt` attribute performs the same function for service tickets. Possible values are either `yes` or `no`. The default is `yes`.

(h) `goodsince`

The date and time the account was last known to be in an uncompromised state. Any tickets granted before this date are in valid. The value is an ISO timestamp. When the account is initially created, the `goodsince` date is set to the current date.

Control over this date is especially useful if you know that an account's password was compromised. Changing the password can prevent the unauthorized principal from accessing the system again using that password, but it does not prevent the principal from accessing the system components for which tickets were obtained fraudulently before the password was changed. To eliminate the principal's access to the system, the tickets must be cancelled. If you set the `goodsince` date to the date and time the compromised password was changed, all tickets issued before that time become invalid and the unauthorized principal's system access is eliminated.

(i) `group`

The name of the group associated with the account. The value is a single group name of an existing group in the registry. This attribute must be specified on an `account create` command, it does not have a default value. If this group is deleted from the registry, then the account is as well.

(j) `home`

The filesystem directory in which the principal is placed in at login.

(k) `lastchange`

A list of two items. The first is the principal name of the last modifier of the account, the second is an ISO timestamp showing the time of the last modification. This attribute is set by the system and may not be modified directly.

(l) `organization`

The name of the organization associated with the account. The value is a single organization name of an existing organization in the registry. This attribute must be specified on an `account create` command, it does not have a default value. If this organization is deleted from the registry, then the account is as well.

(m) `postdatedtkt`

A flag set to determine if tickets with a start time some time in the future can be issued to the account's principal. Possible values are either `yes` or `no`. The default is `no`.

(n) `proxiabletkt`

A flag set to determine whether or not a new ticket with a different network address than the present ticket can be issued to the account's principal. The `forwardabletkt` attribute performs the same function for ticket-granting tickets. Possible values are either `yes` or `no`. The default is `no`.

(o) `pwdvalid`

A flag set to determine whether the current password is valid. If this flag is set to `no`, the next time a principal logs in to the account, the system prompts the principal to change the password. (Note that this flag is separate from the `pwdexpdate` policy, which sets time limits on password validity.) Possible values are either `yes` or `no`. The default is `yes`.

(p) `renewabletkt`

A flag set to determine if the ticket-granting ticket issued to the account's principal can be renewed. If this flag is set to `yes`, the Authentication service renews the ticket-granting ticket if its lifetime is valid. Possible values are either `yes` or `no`. The default is `yes`.

(q) `server`

A flag set to indicate whether or not the account is for a principal that can act as a server. If the account is for a server that engages in authenticated communications, set this flag to `yes`. Possible values are either `yes` or `no`. The default is `yes`.

(r) `shell`

The path of the shell that is executed when a principal logs in.

(s) `stdtgtauth`

A flag set to determine whether or not service tickets issued to the account's principal use the standard DCE ticket-granting ticket authentication mechanism. Possible values are either `yes` or `no`. The default is `yes`.

The policies are:

(a) `maxtktlife`

The maximum amount of time in hours that a ticket can be valid. When a client requests a ticket to a server, the lifetime granted to the ticket takes into account the `maxtktlife` set for both the server and the client. In other words, the lifetime, cannot exceed the shorter of the server's or client's `maxtktlife`. If you do not specify a `maxtktlife` for an account, the `maxtktlife` defined as registry authorization policy is used.

(b) `maxtktrenew`

The amount of time in hours before a principal's ticket-granting ticket expires and that principal must log in again to the system to reauthenticate and obtain another ticket-granting ticket. The lifetime of the principal's service tickets can never exceed the lifetime of the principal's ticket-granting ticket. The shorter

you make this, the greater the security of the system. However, since principals must log in again to renew their ticket-granting ticket, the time needs to take into consideration user convenience and the level of security required. If you do not specify this for an account, the `maxtktrenew` lifetime defined as registry authorization policy is used.

Supports the following operations:

(a) `catalog`

Returns a list of the names of all accounts in the registry. It takes no arguments. By default, fully qualified names are returned in the form *`cellname/acctname`*. If the `-simplename` switch is given then the cellname is not prepended to the front of the account names. The accounts are returned in lexical order.

(b) `create`

Creates a new account. The argument is a list of names of accounts to be created. Returns an empty string on success. Switches are used to specify the attributes of the newly created account. The `group` and `organization` attributes *must* be specified on the command line (either in an attribute list or with attribute switches). All switches are applied to all accounts in the argument. Supports both attribute lists (with a `-attribute` switch) and attribute switches named after each attribute and policy.

(c) `delete`

This operation deletes the account from the registry. The argument is a list of names of accounts to be deleted. If the accounts do not exist an error is generated. Returns an empty string on success.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `modify`

This command is used to change information about an account. The `-add` and `-remove` switches are not supported, as each attribute associated with an account must always have a value, however, the `-change` switch can be used to modify the value of any one of the attributes. The value of the `-change` switch is an attribute list describing the new values for the specified attributes.

(f) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(g) `show`

This operation returns an attribute list describing the specified accounts. The argument is a list of names of accounts to be operated on. If more than one account is given, then the attributes and policies are concatenated together. Accepts a `-policies` switch that returns the polices of the account instead of the attributes. A `-all` switch can be used to return the attributes followed by the policies.

Attributes and policies are returned in lexical order. If the account does not have any policies then `nopolicies` is returned.

The policies that are actually in effect can be different from the account policies due to conflicts with registry wide policies. If this is the case, the show command will alter the attribute structure on output to include an `effective` tag and the effective value, much in the same way `organization show` will.

### 8.11.14. registry

The `registry` object represents the security service database of account information. The registry is a replicated database with a single master copy capable of processing both update and read requests, and zero or more slave copies capable of processing only read requests. The term replica is used to refer to either a master or slave copy of the registry. The commands in `dcecp` automatically bind to a registry server capable of performing a

specific operation. For most operations, the user does not care which registry server is bound to, as long as the request is fulfilled. However, administrators modifying or checking the replication configuration need to be able to bind to a specific registry. The `registry` command can do this.

After this command executes, the `s(sec)` convenience variable is set to the name of the server that was bound to for the command. The value of the variable before the command is treated as a hint, i.e., the server specified will be contacted if it can service the request. A case where it can't service the request is if a read-only registry was bound to, and the next command is a `create` command. In this case, the master registry will be bound to automatically and the `s(sec)` variable updated appropriately. The value of the variable is the name of registry bound to in one of the formats specified below as valid for the argument to the `registry` object. If a registry name is specified as an argument to the command it will be used and the `s(sec)` variable updated appropriately.

Other `dcecp` objects such as: `principal`, `group`, `organization`, and `account` are used to represent most of the data in the registry. The `registry` object is only used for security replication configuration (i.e., `sec_admin` functionality) and the manipulation of registry-wide attributes (a.k.a. properties) and policies. Registry-wide policies refer to policies for organizations (standard policies in `rgy_edit` terminology) and accounts (see the `rgy_edit auth policies` command). For a list of the policies supported, see the descriptions of both the `organization` and the `account` objects. The list of attributes is as follows:

(a) `deftktlife`

The default lifetime (in hours) for tickets issued to principals in this cell's registry. The value is an integer.

(b) `mingid`

The starting point for gid's automatically generated by the Security Service when a group is created. You can explicitly enter a lower gid than this number; it applies only to automatically generated numbers. The value is an integer.

(c) `minorgid`

The starting point for orgid's automatically generated by the Security Service when an organization is created. You can explicitly enter a lower orgid than this number; it applies only to automatically generated numbers. The value is an integer.

(d) `minuid`

The starting point for uid's automatically generated by the Security Service when a principal is created. You can explicitly enter a lower uid than this number; it applies only to automatically generated numbers. The value is an integer.

(e) `maxuid`

The highest number that can be supplied as a uid when principals are created. This maximum applies to both the system-generated and user-entered uid's. The value is an integer.

(f) `mintktlife`

The minimum amount of time (in minutes) before the principal's ticket must be renewed. The value is an integer. This renewal is performed automatically with no intervention on the part of the user. The shorter this time is, the greater the security of the system. However, extremely frequent renewal can degrade system performance. Both system performance and the level of security required by the cell should be taken into consideration when selecting the value of this attribute. This is a registry-wide value only, it cannot be set for individual accounts.

(g) `hidepwd`

Determines whether encrypted passwords are displayed or not. If this attribute is set to `yes`, an asterisk is displayed in place of the encrypted password in command output and files where passwords are displayed. Possible values are either `yes` or `no`.

Each registry replica keeps the following propagation information concerning registry updates:

(a) `name`

The name of the replica. It is in the form of a fully qualified CDS name.

(b) `cell`

The name of the cell that the replica is in. It is a fully qualified cellname.

(c) `uuid`

The UUID of the replica.

(d) `type`

Indicates if the replica is a `master` or a `slave`.

(e) `status`

The state of the replica, one of:

(i) `changingkey`

The replica is in the process of having its master key changed.

(ii) `closed`

The replica is in the process of stopping.

(iii) `copyingdb`

The replica is in the process of initializing (copying its database to) another replica.

(iv) `deleted`

the replica is in the process of deleting itself.

(v) `disabled`

The replica is unavailable for updates but will accept queries. This was called ''in maintenance'' by `sec_admin`.

(vi) `enabled`

The replica is available for use. This was called ''in service'' by `sec_admin`.

(vii) `initializing`

the replica is in the process of being initialized by the master replica or another up-to-date replica.

(viii) `savingdb`

The replica is in the process of saving its database to disk.

(ix) `uninitialized`

The database is a stub database that has not been initialized by the master replica or another up-to-date replica.

(f) `lastupdtime`

The localized date and time the replica was last updated.

(g) `lastupdseq`

The sequence number of the last update the replica received. The value is an integer.

(h) `addresses`

A list of the network addresses of the replica. Can be more than one for connectionless and connection-oriented protocols for example.

(i) `masteraddrs`

The network address of the master registry replica of the cell. This is what the replica believes, it is not necessarily correct. Can be more than one for connectionless and connection-oriented protocols for example.

(j) `masterseqnum`

The master sequence number, which is the sequence number of the event that made the replica the master. This is what the replica believes, it is not necessarily correct.

(k) `masteruuid`

The UUID of the master registry replica of the cell. This is what the replica believes, it is not necessarily correct. The value is an integer.

(l) `version`

The version of the security server software.

(m) `updseqqueue`

The update sequence numbers that are still in the propagation queue and have yet to be propagated. This is only present in the master replica. The value is an integer.

The master replica keeps the following for *each* replica:

(a) `propstatus`

The status of the propagation, possible values:

(i) `initmarked`

The replica is marked for initialization, that is getting an up-to-date copy of the registry.

(ii) `initing`

The replica is in the process of initialization, that is getting an up-to-date copy of the registry.

(iii) `update`

The replica is ready to receive propagation updates.

(iv) `delete`

The replica is marked for deletion.

(b) `lastupdtime`

The localized time the last update was sent to the replica.

(c) `lastupdseqsent`

The last sequence number of the last update sent to this replica. The value is an integer.

(d) `numupdtogo`

The number of outstanding updates. The value is an integer.

(e) `lastcommstatus`

The status of the last communication with the replica.

The argument to any `registry` command identifies the registry server to use. It can be one of the following:

(a) A specific cell name (or `/.:` for the local cell) to bind to any replica in the named cell.

(b) The global name of a replica to bind to that specific replica in that specific cell.

(c) The name of a replica as it appears on the replica list to bind to that replica in the local cell.

(d) A string binding to a specific replica (e.g., `ncadg_ip_udp:15.22.144.163`), see the *Bindings* section for the two available syntaxes for string bindings. This form is used primarily for debugging or if the Cell Directory Service is not available.

(e) If no argument is given, then the value of `s(sec)` is used. If the variable is not set, then the default argument of `/.:` is assumed.

Supports the following operations:

(a) `catalog`

Returns a list of the names of the security servers (i.e., each copy of the registry) running in the cell. This is also known as the replica list. It takes no arguments. The order is arbitrary.

(b) `delete`

Called with no switches this command performs an orderly deletion of a security replica specified as the argument. That is, it binds to the master registry, marks the specified replica as deleted, propagates this deletion to the other replicas, sends a message to the specified replica to delete itself, and finally removes the replica from the masters replica list.

If the specified replica is unreachable, the `-force` switch can be used. This switch causes the master to delete the specified replica from the replica list and then propagates that deletion to the other replicas on the list. The specified replica is not contacted. When that replica is again accessible, the `registry delete` command should be called, specifying the replica and using the `-only` switch. This causes that replica to delete its copy of the registry database and stop running.

The argument is a list of names of registry replicas to be deleted. Returns an empty string on success.

(c) `disable`

Disables the *master* registry for updates. This is called `maintenance mode` in `sec_admin`. The argument is a list of names of registry replicas to be disabled. Returns an empty string on success.

(d) `enable`

Enables the *master* registry for updates. This is called `server mode` in `sec_admin`. The argument is a list of names of registry replicas to be enabled. Read-only copies are not allowed, but the name of other cells can be used. Returns an empty string on success.

(e) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(f) `modify`

This command is used to change attributes of the registry. The argument is a list of names of registry replicas to be operated on. The argument is a list of names of accounts to be operated on. All modifications are applied to all accounts named in the argument. Accounts are modified in the order they are listed and all modifications to an *individual* account are atomic. Modifications to multiple accounts are not atomic. A failure for any one account in a list causes an error to be generated and the rest of the operation to be aborted. The `-change` switch can be used to modify the value of any one of the attributes.[19] Its value is an attribute list describing the new values for the specified attributes or policies. Returns an empty string on success.

(g) `operations`

19. The `-add` and `-remove` switches are not supported, as each attribute associated with a group must always have a value.

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(h) `show`

Returns information about the registry and its replicas. Takes no argument. Returns a variety of different information based on the switch given. If called with no switches or with the `-attributes` switch, returns an attribute list of all the registry-wide attributes. If called with the `-policies` switch, returns an attribute list of all the registry-wide polices.

If called with the `-replica` switch, returns the propagation information that is kept by the specified replica. The `-replica` switch can be called with an argument that specifies a slave replica. If called in this manner, then the master is contacted and the propagation information that the master keeps for the specified slave is returned.

(i) `stop`

Stops the specified security server process.

(j) `synchronize`

Causes the specified replica to reinitialize itself with an up-to-date copy of the database. The argument is a list of names of registry replicas to be operated on. This command binds to the master, marks the specified replica for reinitialization, sends a message to the replica informing it to reinitialize itself and providing a list of other replicas with up-to-date copies of the registry. The specified replica then selects a replica from the list and asks for a copy of the database.

This command is generally not used under normal circumstances. Mainly because the registries should remain synchronized. Also there will be functionality in the DCE 1.1 `secd` such that if the master loses contact with a slave for a protracted period of time, the master reinitializes the slave automatically by giving it a fresh copy of the entire database when communication is restored. In pre-1.1 versions of DCE, if the master loses contact with the slave you must explicitly use the `registry synchronize` or the `sec_admin initrep` command to perform this function.

### 8.11.15. acl

The `acl` object represents an ACL which may be on any object. The `acl_edit` use model is a batched one. Each `acl_edit` session modifies the ACL on one object. The commands are structured so that individual ACL Entries are modified locally, but nothing is written back to the server until either a `commit` or `exit` command is performed. This model will not be followed in `dcecp`. Each `acl` command will write to the server, however, commands will allow more than one modification at once.

ACL's are composed of ACL Entries. ACL Entries are visible only as members of ACL's. There is no object that represents ACL Entries, only the `acl` object representing an entire ACL. Most of the `acl` operations deal directly with the ACL. See the *Data Structures* section for a description of the syntax of ACL's and ACL Entries. An ACL does have one attribute which represents the default cell of the ACL; it is called `cell`. Manipulation of this attribute is possible only through the `modify` and `show` commands.

An object can have more than one ACL. If the object is a container it has three ACL's: the ACL for the container object itself, an ACL specifying the default ACL on new objects added to the container (the initial object ACL), and an ACL specifying the default ACL on new containers added to the container (the initial container ACL). By default the `acl` command will bind to the ACL of the container. The other ACL's can be operated upon by using switches on the commands.

In addition there is another ambiguity that can occur in identifying the ACL to operate on. Since entries in the namespace often identify resources that have ACL's (e.g. servers sometimes have ACL's and they export their binding information into an RPC Entry) the name of a object can mean either the object itself or the namespace entry that identifies the object. This ambiguity is also solved by using a switch. The switches that specify the ACL to edit can be given on any of the operations and are as follows:

(a) `-ic`

Specifies that the command is to operate on the initial container ACL of the named object.

(b) `-io`

Specifies that the command is to operate on the initial object ACL of the named object.

(c) `-entry`

Specifies that the command is to operate on the ACL of the namespace entry of the object.

The `acl` command is called with one argument which is a list of items specifying the ACL's to be modified. Each item can be either the name or string binding (see the *Bindings* section) of the object. Any switches used to specify an ACL to operate upon apply to all items entered in the list.

Supports the following operations:

(a) `check`

Returns the permissions granted in the specified object's ACL to the principal that invoked the command. The argument is a list of names of ACL's to be operated on.

```
dcecp> acl check /.:/hosts
rwdtcia
dcecp> acl check /.:/subsys /.:/hosts
r--t-ia
rwdtcia
```

(b) `delete`

Removes all ACL Entries from the object, except the `user_obj` entry, if it exists. The argument is a list of names of ACL's to be operated on. Generates an error if the ACL does not exist. Returns an empty string on success.

(c) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(d) `modify`

This operation is used to change one or more individual ACL entries. The argument is a list of names of ACL's to be modified. They are processed in the order they are entered. The specific operation to perform is described with one or more of the following switches:

(i) `-add`

Adds the ACL Entries to the ACL. The value of this switch is a list of ACL Entries with permissions filled in.

(ii) `-remove`

Removes the ACL Entries from the ACL. The value of this switch is a list of ACL Entries without permissions filled in.

(iii) `-change`

Changes existing ACL Entries in the ACL. The value of this switch is a list of ACL Entries with permissions filled in. The permissions will be the new permissions placed on the specified ACL Entries. The ACL Entries must exist in the ACL or an error occurs.

(iv) `-purge`

Purges all masked permissions (before any other modifications are made), in all ACL Entries except: `user_obj`, `other_obj`, `mask_obj`, and `unauthenticated` if they exist. This switch is useful only for ACL's that contain an entry of type `mask_obj`.

(v) `-mask`

If a `modify` operation causes a mask recalculation that unintentionally adds permissions to an existing acl entry, the modify operation will abort with an error unless you specify the `-mask` switch with a value of either `calc` or `nocalc`, or a unique abbreviation of one of these values.

Specifying `calc` creates or modifies the object's `mask_obj` type entry with permissions equal to the union of all entries other than type `user_obj`, `other_obj`, `mask_obj` and `unauthenticated`. This creation or modification is done after all other modifications to the ACL are performed. The new mask is set even if it grants permissions previously masked out. It is recommended that you use this switch only if not specifying it results in an error. If you specify the `calc` switch for an ACL that does not support `mask_obj` entry type, an error is returned.

Specifying `nocalc` means that a new mask should not be calculated.

This switch is only legal if the `-add` or `-change` switch is present on the modify command. The ACL must support the `mask_obj` ACL type, or else an error is returned. If the user explicitly includes a `mask_obj` ACL entry in the command (i.e., in the value of the `-add` or `-change` switches), the use of the `-mask` switch is illegal. The reason is that the `-mask` switch is meant to resolve unintentional results, the presence of a `mask_obj` ACL entry implies intent.

Either all changes are made to an *individual* object, or none are. Multiple actions can be specified on the command line, they will be processed in a fixed order to guarantee proper processing of the ACL's. See [POSIX.6] for a description of this processing order. The command returns an empty string on success. Examples are:

```
dcecp> acl modify /.:/hosts -add {user melman rwcia}
dcecp> acl modify /.:/hosts -change {user melman rwdtcia}
dcecp> acl modify /.:/hosts -add {group dce rwdtcia} \
-remove {user melman rwdtcia}
```

The `modify` operation can change the value of the `cell` attribute using an attribute switch called `-cell` and giving the new default cell as the value (it must be one value, not a list):

```
dcecp> acl modify /.:/hosts/ -cell /.../newcell
dcecp> acl modify /.:/hosts/ -add {user melman rwcia} \
-cell /.../newcell
```

The `-cell` switch may be combined with other switches on a single `modify` command. It is always applied before the other switches.

(e) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(f) `permissions`

Returns a list describing the permissions associated with an object. The argument is a list of names of ACL's to be operated on. If more than one is entered the output is concatenated together. Each element of the list is a list of two items, the first is the permission token, the second is a description of the permission. For example:

```
dcecp> acl permissions /.:/hosts
{r read}
{w write}
{d delete}
{t test}
{c control}
```

```
{i insert}
{a administer}
```

(g) `replace`

Replaces the entire ACL on the object specified by the argument, with the value of the `-acl` switch. The argument is a list of names of ACL's to be operated on. If more than one is given the output is concatenated together. The syntax of the value of the `-acl` switch is a list of ACL entries. Returns an empty string on success.

(h) `show`

Returns a list of the ACL Entries for the specified object. If they exist, the `mask_obj` and `unauthenticated` ACL Entries are displayed first. For example:

```
dcecp> acl show /.:/hosts
{unauthenticated r--t---}
{user cell_admin rwdtcia}
{user hosts/absolut/cds-server rwdtcia}
{user hosts/absolut/self rwdtcia}
{user root rwdtcia}
{group subsys/dce/cds-admin rwdtcia}
{group subsys/dce/cds-server rwdtcia}
{any_other r--t---}
```

If called with the `-cell` switch returns the value of the default cell for the ACL:

```
dcecp> acl show /.:/hosts -cell
/.../absolut_cell
```

There is no `-all` switch.

### 8.11.16. dts

The `dts` object represents the `dtsd` process running on the local host. Since DTS is designed not to have stored data, this object represents the information in and about a process rather than some data stored somewhere as other objects do. Also this is not implemented as an instance of the `server` object (described below) since objects, not instances, have operations associated with them. These commands all affect the local `dtsd` entity, they take no arguments.

There are a number of attributes associated with the `dts` object. All of these can be viewed with the `show` operation, and many can be changed with the `modify` operation. Attribute arguments can contain a maximum of 80 characters and are recalculated to a normalized date format. For example if the input value is `0-0025:10:99.99999999`, the result is `1-01:11:39.990`.

The attributes are:

(a) `autotdfchange`

Specifies whether automatic changes to the time differential factor are enabled or disabled. The value is either `yes` or `no`. The value is determined by the operating system (i.e., it cannot be changed with the `modify` operation).

(b) `clockadjrate`

Specifies the rate at which the DTS server or clerk entity adjusts the node's clock during a synchronization. This may not be set by a user, but is built into `dtsd`.

(c) `clockresolution`

Specifies the amount of time between system clock ticks. The value is determined by the operating system (i.e., it cannot be changed with the `modify` operation).

(d) `globalservers`

Specifies the set of global servers known by the node. The information returned for each server is as follows: the DCE name of the host followed by `/self`, the last time polled, the last observed time, the last observed skew, a binary value of whether the server was used in the last last synchronization, and the transport time. The sub-attributes will be respectively be called: `name`, `timelastpolled`, `lastobstime`, `lastobsskew`, `inlastsync`, and `transport`.

(e) `globaltimeout`

Specifies the amount of time the node waits for a response to a WAN synchronization request before sending another request or declaring a global server to be unavailable. The number of attempts made to reach the server is controlled by the `queryattempts` attribute. The default value is `0-00:00:15.000`, and the range of possible values is `0-00:00:00.000 - 0-00:10:00.000`.

(f) `localservers`

Specifies the set of local servers known by the node. The information returned for each server is as follows: the DCE name of the host followed by `/self`, the last time polled, the last observed time, the last observed skew, a binary value of whether the server was used in the last last synchronization, and the transport time. The sub-attributes will be respectively be called: `name`, `timelastpolled`, `lastobstime`, `lastobsskew`, `inlastsync`, and `transport`.

(g) `localtimeout`

Specifies the amount of time the node waits for a response to a synchronization request before sending another request or declaring a server to be unavailable. The number of attempts made to reach the server is controlled by the `queryattempts` attribute. The default is `0-00:00:05.000`, and the range of possible values is `0-00:00:00.000 - 0-00:01:00.000`.

Note that this attribute controls only the initial contact with a time provider. During this initial contact, the time provider itself determines the timeout value for actually reporting back times. This allows a time provider attached to a slow source like a modem to request that `dtsd` wait for a longer interval.

(h) `maxdriftrate`

Specifies the worst-case drift rate of the node's clock, in nanoseconds per second, as determined by the manufacturer's specifications (i.e., it cannot be changed with the `modify` operation).

(i) `maxinaccuracy`

Specifies the inaccuracy limit for the node. When the node exceeds the maximum inaccuracy setting, it attempts to synchronize. The default is `0-00:00:00.100`, and the range of possible values is `0-00:00:00.0 - 10675199-02:48:05.478`.[20]

(j) `minservers`

Specifies the minimum number of servers required for a synchronization. Settings of `1` or `2` may cause unreliable computed times. The default is `3` and the range of possible values is `1-10`.

(k) `nexttdfchange`

Specifies the future time at which the time differential factor is automatically changed. The value is determined by the operating system (i.e., it cannot be changed with the `modify` operation).

(l) `queryattempts`

---

20. This has been changed to be a max of 24 hours as per DCE OT 7357. It really makes no sense for this value to be anything other than seconds.

Specifies the number of attempts that a node makes to contact a server before the node considers the server unavailable. The default is `3`, and the range of possible values is `1-10`.

(m) `syncinterval`

Specifies the interval a node must wait to synchronize. Also specifies synchronization frequency when a node reaches the value specified by the `maxinaccuracy` attribute. For clerks the default is `0-00:10:00.0`, and the range of possible values is `0-00:00:30.0 - 01 00:00:00.00`. For servers the default is `0-00:02.00.0`, and the range of possible values `0-00:00:30.0 - 01 00:00:00.00`.

(n) `tdf`

Specifies the Time Differential Factor (TDF), which is the amount of time the server varies from Greenwich mean time (GMT) or Coordinated Universal Time (UTC). The default is `0-00:00:00.000`, and the range of possible values is `-13-00:00:00 - 13-00:00:00`. This may not be set by a user, but rather is obtained from various timezone information repositories (e.g., the `TZ` environment variable, kernel structures, etc.).

(o) `timerep`

Specifies the internal timestamp format used by the node. This is *not* related to the format that is used to display the current time to the user (see the `clock show` command below). Currently DTS only uses `V1.0` timestamps. This attribute may not be set by the user, its value is built into a `dtsd`.

(p) `tolerance`

Specifies the maximum separation allowed between the local clock and the computed time before synchronizations become abrupt rather than gradual (monotonic). The default is `0-00:10:00.000`, and the range of possible values is `0-00:00:00.500 - 10675199-02:48:05.478`.

(q) `type`

Specifies whether the node is a DTS `server` or `clerk`.

(r) `version`

Specifies the DTS software version installed on the node. This attribute cannot be changed with the `modify` operation.

Additional attributes on a server are:

(a) `actcourierrole`

Specifies a server's acting interaction with the set of global servers. The values are the same as for the `courierrole` attribute below. The difference between this attribute and that, is when `courierrole` is `backup`, the courier could be actually acting as a courier or not.

(b) `checkinterval`

Specifies the amount of time between checks for faulty servers. Applicable only to servers that have external time providers. The default is `0-01:30:00.00`, and the range of the possible values is `0-00:00:30.000 - 10675199-02:48:05.478`.

(c) `courierrole`

Specifies a server's interaction with the set of global servers. Possible values are:

(i) `backup`

The local server becomes a courier if none are available on the LAN.

(ii) `courier`

The local server synchronizes with the global set of servers.

(iii) `noncourier`

The local server does not synchronize with the global set of servers. This is the default.

(d) `epoch`

Specifies the server's epoch number. The default is `0`, and the range of possible values is `0-255`. This value may not be changed via the `modify` command; use the `clock set` command with the `-abruptly` and `-epoch` switches to change its value.

(e) `lastsync`

Specifies the computed time at the last synchronization.

(f) `provider`

Specifies whether or not the entity used an external time-provider at the last successful synchronization. This attribute applies to servers only and may not be set by a user.

(g) `serverentry`

Specifies a server's ACL entry name. The default setting is the following recommended value: `/.:/hosts/`*`hostname`*`/dts-entity`.

(h) `servergroup`

Specifies the security group name for the time servers within the cell. The default is `/.:/subsys/dce/dts-servers`.

(i) `serverprincipal`

Specifies a server's principal name for authentication purposes. The default setting is the following recommended value: `/.:/hosts/`*`hostname`*`/self`.

(j) `status`

Specifies the state of the DTS entity. This is a read-only attribute and its possible values are:

(i) `disabled` — The DTS entity is disabled.

(ii) `enabled` — The DTS entity is enabled.

(iii) `syncing` — The DTS entity is synchronizing.

(iv) `updating` — The DTS entity is updating the time.

(k) `uuid`

Specifies the entity's unique identifier, which is generated when the entity is created.

The counters are:

(a) `abrupts`

Specifies the number of times the node clock has been set non-monotonically (abruptly).

(b) `badlocalservers`

Specifies the number of times that a local server was contacted, but it was not in the dts security group.

(c) `badprotocols`

Specifies the number of times the local node failed to process a received message containing an incompatible protocol version.

(d) `badtimereps`

Specifies the number of times the local node failed to process a received message containing an incompatible timestamp format.

(e) `creationtime`

Specifies the time at which the DTS entity was created and the counters were initialized.

(f) `diffepochs`

Specifies the number of times the node received time response messages from servers or clerks that had epoch numbers different from its own.

(g) `disables`

Specifies the number of times the DTS has been disabled.

(h) `enables`

Specifies the number of times the DTS has been enabled.

(i) `nointersections`

Specifies the number of times the node's time interval failed to intersect with the computed interval of the servers.

(j) `nomemories`

Specifies the number of times the node has been unable to allocate virtual memory.

(k) `providertimeouts`

Specifies the number of times a dtsd server process initiated contact with a time provider and did not receive the initial response within the interval specified by the `localtimeout` attribute.

(l) `syncs`

Specifies the number of times the node successfully synchronized.

(m) `syserrors`

Specifies the number of times a DTS operation detected a system error.

(n) `toofewservers`

Specifies the number of times a node failed to synchronize because it could not contact the required minimum number of servers.

Additional counters on a server are:

(a) `badservers`

Specifies the number of times that a non-local server was contacted, but it was not in the dts security group.

(b) `diffepochs`

Specifies the number of times the node received time response messages from servers or clerks that had epoch numbers different from its own.

(c) `epochchanges`

Specifies the number of times the server's epoch has changed.

(d) `faultyservers`

Specifies the number of times a server has detected faulty servers (other than itself).

(e) `noglobals`

Specifies the number of times the courier server could not contact any global servers.

(f) `noresponses`

Specifies the number of times the courier server could not contact a specific global server.

(g) `providerfailures`

Specifies the number of times the external time-provider signaled a failure or the node was unable to access the time-provider.

(h) `synchronizations`

Specifies the number of times a server has attempted to synchronize its clock.

Supports the following operations:

(a) `activate`

Changes a DTS entity from an inactive state to an active state. The `status` attribute is changed to `enabled`. This tells the DTS entity to begin synchronizing. Takes a `-abruptly` switch to determine if the first clock adjustment due to synchronization is an abrupt or gradual one. Returns an empty string on success.

(b) `configure`

Requires one of two switches `-global` or `-notglobal` to configure the local `dtsd` as a global server or not. The difference is whether they are listed in `/.:/cell-profile`. Returns the string `global` or `notglobal` to indicate the current (new) state of the `dtsd`.

(c) `deactivate`

Changes a DTS entity from an active state to a deactive state. The `status` attribute is changed to `disabled`. This tells the DTS entity to stop synchronizing. Returns an empty string on success.

(d) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(e) `modify`

Allows attributes to be changed with the `-change` switch. Attribute switches are also support for all modifiable attributes. Returns an empty string on success.

(f) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(g) `show`

When called with the `-attributes` returns an attribute list giving the values of the attributes listed above. If called with the `-counters` switch then counter information is returned. If called with the `-all` or with both the `-attributes` and `-counters` switches then both attribute and counter information is returned. The default behavior (invoked by using no switches) is the same as if the `-attributes` switch was used. Attributes and counters are listed in the order they are returned by the server.

(h) `stop`

This stops the `dtsd` process. Returns an empty string on success.

(i) `synchronize`

Causes the local `dtsd` to synchronize with DTS servers. The machine's clock is adjusted accordingly. By default the clock is adjusted gradually. This command also takes the optional `-abruptly` switch to have the clock set abruptly. Returns an empty string on success.

### 8.11.17. clock

The `clock` object represents the clock on the local system, and the time that it tells. This object has commands to display and set the time. The time setting commands are done via DTS.

Supports the following operations:

(a) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(b) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(c) `set`

Sets the clock to the specified time. The argument is a DTS style timestamp. The clock is adjusted gradually by DTS to the specified time. This command takes a `-abruptly` switch which specifies that the time is to be immediately changed to the specified time. When this switch is present the `-epoch` switch must also be specified to indicate a new epoch. Returns an empty string on success.

(d) `show`

Returns a DTS style timestamp with the TDF indicated. Takes no arguments.

(e) `synchronize`

This is a synonym for the `dts synchronize` command. Takes no arguments. Returns an empty string on success.

### 8.11.18. aud

The `aud` object represents the audit daemon (called `auditd` in the reference implementation) on a host.

The attributes are:

(a) `stostrategy`

The audit trail storage strategy of the daemon. This attribute defines what the daemon will do if the audit trail storage is full. Its possible values are: `stop` in which case the daemon will no longer service logging requests (the `state` attribute will be changed to `disabled`) or `wrap` where the daemon will overwrite the old audit trails.

(b) `state`

Tells whether the audit daemon is accepting audit log requests or not. The values are `enabled` or `disabled`.

The commands only affect the audit daemon on the local host, so none of these commands take arguments. Supports the following operations:

(a) `disable`

Disables an audit daemon. The `state` attribute is changed to `disabled`. The control (`c`) permission on the audit daemon's ACL is required for this command. Returns an empty string on success.

(b) `enable`

Enables an audit daemon. The `state` attribute is changed to `enabled`. The control (`c`) permission on the audit daemon's ACL is required for this command. Returns an empty string on success.

(c) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(d) `modify`

Allows modification of the attributes above. Accepts the `-change` switch which takes an attribute list as a value. Also accepts the attribute switches `-stostrategy` and `-state`. The control (`c`) permission on the audit daemon's ACL is required for this command. Returns an empty string on success.

(e) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(f) `show`

Returns an attribute list for the audit daemon. The attributes are returned in lexical order. The read (`r`) permission on the audit daemon's ACL is required for this command.

(g) `stop`

Stops the audit daemon process. The control (`c`) permission on the audit daemon's ACL is required for this command. Returns an empty string on success.

### 8.11.19. audfilter

This object represents audit filters (a.k.a. event filters) which are kept by the audit daemon to determine if an auditable event should be logged. An audit filter consists of a list of guides. The name of the audit filter is a filter type and possibly a key (dependent on the type). The audit filter types are as follows:

(a) `principal` — key is a principal_name
(b) `foreign_principal` — key is a /.../cellname/principal_name
(c) `group` — key is a group_name
(d) `foreign_group` — key is a /.../cellname/group_name
(e) `cell` — key is a cell_name
(f) `cell_overridable` — key is a cell_name
(g) `world` — has no key
(h) `world_overridable` — has no key

So the following are examples of audit filter names: `principal melman`, `group dce`, `world`.

A guide contains three elements: audit conditions, audit actions, and event classes. Each of these three elements is itself a list. Event classes are definable by the administrator. The possible audit conditions are: `success`, `failure`, `denial`, and `all`. The possible audit actions are: `none`, `log`, `alarm`, and `all`.

Supported operations are:

(a) `catalog`

Returns a list of names of all filters in the audit daemon. This command takes no arguments. The names are a list of a type and if necessary a key. They are returned in an arbitrary order. The read (`r`) permission on the audit daemon's ACL is required for this command. In the following example, there are four filters in the audit daemon:

```
dcecp> audfilter catalog
{principal melman}
{foreign_principal /.../cell_X/kevins}
{group dce}
world
```

(b) `delete`

Deletes the filter including all filter guides. The argument is a list of names of audit filters to be deleted. The write (`w`) permission on the audit daemon's ACL is required for this command. Returns an empty string on success.

(c) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(d) `modify`

This operation is used to add or remove one or more guides of a filter. The argument is a list of names of audit filters to be deleted. In addition, the specific operation to perform is described with one or more of the following switches: `-add` and `-remove`. The values of these is a list of guides. Returns an empty string on success. The write (`w`) permission on the audit daemon's ACL is required for this command. For example:

```
dcecp> audfilter modify \
{foreign_principal /.../cell_X/foo} \
-remove {denial alarm IBM_Confidential_Restricted}
dcecp> audfilter modify world -add {
denial log Monetary_Transfers}
```

Atomicity of multiple actions is not guaranteed. The effect of removing a guide from a filter which does not have an identical guide is to ''change'' the existing guide(s). Similarly, the effect of adding a guide which partially exists in the specified filter is to ''change'' the existing guide(s). These changes guarantee that the semantics of the removal/addition is maintained.

(e) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(f) `show`

Returns a list of guides in a specified filter. The argument is a list of filter names (a filter type, and if needed a key) to be shown. If more than one is entered, the output is concatenated together. The read (`r`) permission on the audit daemon's ACL is required for this command. For example:

```
dcecp> audfilter show {foreign_principal /.../cell_X/foo}
{denial log IBM_Confidential}
{denial {alarm log} IBM_Confidential_Restricted}
dcecp> audfilter show world
{success log Monetary_Transfers}
```

### 8.11.20. audevents

Supports the following operations:

(a) `catalog`

Returns a list of the names of all event classes. This command takes no arguments. The order returned is arbitrary. The read (`r`) permission on the audit daemon's ACL is required for this command.

(b) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(c) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

In the future `create` and `delete` operations could be added to this object to allow for the creation of new event classes. These could be implemented as base Tcl scripts as all that is necessary is the manipulation of files.

### 8.11.21. audtrail

The `audtrail` objects represents an audit trail file. This object currently supports only one operation which converts the audit trial into a human readable format. Future extensions can add fancier conversion operations, or more likely new switches. Supported operations are:

(a) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(b) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(c) `show`

Returns the audit trail in a human readable format. This command takes as an argument a list of names of audit trial files. If more than one name is given then the output of each audit trail is concatenated together. Takes no switches. Read permission to the audit files is required for this operation.

### 8.11.22. log

The `log` object represents the current state of routing for DCE serviceability messages of a process. There are two types of routing supported, serviceability routing and debug routing. Debug routing may be removed from a production environment.

The commands take one argument which is either a fully bound string binding or the name of a RPC server entry. See the *Data Structures* section for a description of the syntax of serviceability routing specifications and debug routing specifications.

Supports the following operations:

(a) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(b) `list`

Returns a list of serviceability components registered by the server. Takes one argument which a list of either a fully bound string binding or the name of a RPC server entry (they can be mixed in the same list). If more than one is entered the output is concatenated together.

Also takes an optional `-comp` switch which is used to specify a component name. In this case, a list of sub-components is returned where each element of the list is a list of four elements: the subcomponent name, its level, the message id of the long description of the subcomponent, and the text of the long description of the subcomponent. The order is arbitrary.

Example:

```
dcecp> log list /.:/hosts/hostname/cds-server
{svc cds dts rpc sec}
dcecp> log list /.:/hosts/hostname/cds-server -comp dts
{{general 0 0x12345671 "General server administration"}
 {events  0 0x12345672 "Events received and acted upon"}
 {arith   0 0x12345673 "Math operations"}
 {ctlmsgs 0 0x12345674 "Control messages received"}
 {msgs    0 0x12345675 "Messages received"}
 {states  0 0x12345676 "Server state transitions"}
 {threads 0 0x12345677 "Thread interactions"}
```

```
 {config  0 0x12345678 "Server/cell configuration"}
 {sync    0 0x12345679 "Server/synchronization \
interactions"}}
```

(c) `modify`

Is used to change the routing specifications of the specified servers. Takes one argument which a list of either a fully bound string binding or the name of a RPC server entry (they can be mixed in the same list). The `-add` and `-change` switches are synonymous. The `-remove` switch can be used to specify that messages are to be handled with `DISCARD`. Each switch takes a list of either serviceability or debug routing specifications. Returns an empty string on success.

(d) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(e) `show`

Returns a list describing the current serviceability routing settings for a server. If more than one is entered the output is concatenated together. The order of the returned routing settings is arbitrary. Takes an optional `-debug` switch which indicates that debug routing settings are to be returned instead of normal serviceability routing settings.

### 8.11.23. server

The `server` object represents instances of running servers. The commands take one argument which is either a fully bound string binding or the name of a RPC server entry. Supports the following operations:

(a) `help`

Returns help information on the object as described in the *Help System* section. Takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

(b) `operations`

Returns a list of the operations supported by the object. It takes no arguments, and always returns a Tcl list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that `help` and `operations` are listed last.

(c) `ping`

Returns `1` if the server specified by the argument is listening for RPC requests, `0` if not. The argument must be a fully bound string binding.

### 8.11.24. Miscellaneous commands

(a) `dcecp_initInterp`

This command will initialize a base Tcl interpreter with all the `dcecp` commands.

(b) `errtext`

Takes a DCE status code as an argument and returns the text of the associated message as found in the message catalogs. The argument can be in decimal, octal (leading `0`), or hexadecimal (leading `0x`) notation.

(c) `expand`

Takes a DCE name as an argument and returns the canonical form of the name.

(d) `login`

Creates a new login context to be used for the rest of the `dcecp` session. Login contexts are stacked. Takes an account name as an argument. The password is prompted for and not echoed to the screen. Also takes the `-password` switch to enter a password.

(e) `logout`

Logs you out of the current login context as established with a previous `login` command. Only contexts created with `dcecp`'s `login` can be logged out of. Trying to logout of an inherited context results in an error. Leaving `dcecp` will do a `logout` for all contexts created in the session.

(f) `quit`

Exits from `dcecp`. A synonym of the Tcl builtin command `exit`.

(g) `resolve`

Takes a partial string binding and returns a fully bound string binding. Takes a `-interface` switch with an interface identifier as an argument to provide enough information for the mapping to occur.

(h) `shell`

Spawns a command shell for the user. The value of the `SHELL` environment variable is used to obtain the name of the shell to spawn. When the command shell terminates, control is returned to `dcecp`. If called with arguments, they are passed to the shell and executed. Control is returned upon completion. Always returns an emptry string, though an error exception is generated if the shell exists abnormally.

### 8.12. Tasks

The following are administrator tasks to be implemented as `dcecp` scripts. To the user, they will appear exactly like commands.

*NOTE: I need to fill this section in with details of what these commands will do. For example,* `user create` *will say something like: adds a principal to registry (and optionally a group and and organization as specified via switches) creates an account in the registry for the user, adds a entry to CDS with the name /.:/blah/blah/user, etc.*

(a) `cell show`

(b) `cell ping`

(c) `cell backup`

(d) `user create`

(e) `user delete`

(f) `user show`

(g) `host configure -client`

(h) `host configure -server`

(i) `host unconfigure`

(j) `host start`

(k) `host stop`

(l) `host show`

(m) `host ping`

### 8.13. Command-line Editing Commands

A line may be edited before it is sent to the `dcecp` by typing control characters or escape sequences. A control character, shown as a ''C-'' followed by a letter, is typed by holding down the ''control'' key while the letter is typed. For example, `C-A` is a control-A. An escape sequence is entered by typing the ''escape'' key followed by one or more characters. The escape key is abbreviated as `ESC`. Note that unlike control keys, case matters in escape sequences; `ESC F` is not the same as `ESC f`.

An editing command may be typed anywhere on the line, not just at the beginning. In addition, a return may also be typed anywhere on the line, not just at the end.

Most editing commands may be given a repeat count, `n`, where `n` is a number. To enter a repeat count, type the escape key, the number, and then the command to execute. For example, `ESC 4 C-f` moves forward four characters. If a command may be given a repeat count then the text ''[n]'' is given at the end of its description.

The following control characters are accepted:

(a) C-A Move to the beginning of the line
(b) C-B Move left (backwards) [n]
(c) C-D Delete character [n]
(d) C-E Move to end of line
(e) C-F Move right (forwards) [n]
(f) C-G Ring the bell
(g) C-H Delete character before cursor (backspace key) [n]
(h) C-I Complete filename (tab key); see below
(i) C-J Done with line (return key)
(j) C-K Kill to end of line (or column [n])
(k) C-L Redisplay line
(l) C-M Done with line (alternate return key)
(m) C-N Get next line from history [n]
(n) C-P Get previous line from history [n]
(o) C-R Search backward (forward if [n]) through history for text; must start line if text begins with an uparrow
(p) C-T Transpose characters
(q) C-V Insert next character, even if it is an edit command
(r) C-W Wipe to the mark
(s) C-XC-X Exchange current location and mark
(t) C-Y Yank back last killed text
(u) C-[ Start an escape sequence (escape key)
(v) C-]*c* Move forward to next character *c*
(w) C-? Delete character before cursor (delete key) [n]

The following escape sequences are provided.

(a) ESC C-H Delete previous word (backspace key) [n]
(b) ESC DEL Delete previous word (delete key) [n]
(c) ESC SPC Set the mark (space key); see C-XC-X and C-Y above
(d) ESC . Get the last (or [n]'th) word from previous line
(e) ESC ? Show possible completions; see below
(f) ESC < Move to start of history
(g) ESC > Move to end of history
(h) ESC b Move backward a word [n]
(i) ESC d Delete word under cursor [n]
(j) ESC f Move forward a word [n]
(k) ESC l Make word lowercase [n]
(l) ESC u Make word uppercase [n]
(m) ESC y Yank back last killed text
(n) ESC w Make area up to mark yankable
(o) ESC *nn* Set repeat count to the number *nn*

There is filename completion. Suppose the root directory has the following files in it:

```
bin     vmunix
core    vmunix.old
```

If you type `rm /v` and then the tab key, as much of the name as possible will be finished off by adding `munix`. Because the name is not unique, it will then beep. If you type the escape key and a question mark, it will display the two choices. If you then type a period and a tab, the filename will be finished for you:

```
rm /v[TAB]munix.[TAB]old
```

The tab key is shown by `[TAB]` and the automatically entered text is shown in italics.

### 9. API'S

All commands in `dcecp` will be implemented as Tcl commands. Therefore, they are user-visible and can be used in `dcecp` scripts to implement new commands. Some commands are implemented in C since they make DCE library calls; they are also visible to users. However, often there will be another command written in Tcl that will wrap the C command to present a better interface to the user. In this case both commands are available to the user, but only the higher-level command will be documented.

The lower-level commands can be found in the Tcl source, and can be considered in some sense to be `dcecp` API's. They will not typically be called directly by users, but will be visible to the ''Tcl-knowledgeable'' (i.e., they will be listed if an `info commands` is performed).

For example, there will be one `create` command implemented in Tcl, it will read its first argument to determine the type of object being created (e.g., a group, a principal, a directory, etc.); it will then call another routine (usually implemented in C) for the lower-level functionality. This lower-level routine will be named the same as the upper-level command with an underscore and the type of object appended to it (e.g., `create_group`, `create_principal`, `create_directory`, etc.).

### 10. REMOTE INTERFACES

Not applicable

### 11. MANAGEMENT INTERFACES

Not applicable

### 12. RESTRICTIONS AND LIMITATIONS

Not applicable

### 13. OTHER COMPONENT DEPENDENCIES

The `dcecp` will be dependent on the following DCE components:

(a) NSI

(b) CDSPI

(c) The `rgy` interface

(d) The `sec_acl` interface

(e) The `sec_login` interface

(f) DTS interfaces

(g) Serviceability interface

(h) The `dced` interfaces

(i) Auditing interface

(j) The `editline` functionality from `libdceutils.a`.

### 14. COMPATIBILITY

The existing control programs will still be released, `dcecp` will be released in addition to them. As this is the first release of `dcecp` there is nothing for it to be compatible with. It makes use of DCE API's, so as long as they remain compatible with older versions of DCE servers, `dcecp` should be able to contact pre-1.1 DCE services.

## 15. STANDARDS

Not applicable except in that the `dcecp` code will follow the DCE Coding Style Guide [RFC 34].

## 16. OPEN ISSUES

None.

## APPENDIX A. MAPPING OF EXISTING CONTROL PROGRAM COMMANDS

In the mappings lists of this appendix, the commands in the left column map to the commands in the right column, unless specified otherwise.

### A.1. rpccp

These are commands that are mapped directly:

| | |
|---|---|
| (a) `add element` | `rpcprofile add` |
| (b) `add entry` | `rpcentry create` |
| (c) `add member` | `rpcgroup add` |
| (d) `export` | `rpcentry export` |
| (e) `import` | `rpc-{entry,group,profile} \ import` |
| (f) `remove element` | `rpcprofile remove` |
| (g) `remove entry` | `rpcentry delete` |
| (h) `remove group` | `rpcgroup delete` |
| (i) `remove mapping` | `endpoint delete` |
| (j) `remove member` | `rpcgroup remove` |
| (k) `remove profile` | `rpcprofile delete` |
| (l) `show entry` | `rpcentry show` |
| (m) `show group` | `rpcgroup show` |
| (n) `show mapping` | `endpoint show -all` |
| (o) `show profile` | `rpcprofile show` |
| (p) `show server` | `rpcentry show` |
| (q) `unexport` | `rpcentry unexport` |

### A.2. cdscp

These are commands that are mapped directly:

| | |
|---|---|
| (a) `add directory` | `directory modify` |
| (b) `add object` | `object modify` |
| (c) `clear cached server` | `cdscache delete` |
| (d) `clear clearinghouse` | `clearinghouse disable` |
| (e) `create child` | `directory add` |
| (f) `create clearinghouse` | `clearinghouse create` |
| (g) `create directory` | `directory create` |
| (h) `create link` | `link create` |
| (i) `create object` | `object create` |
| (j) `create replica` | `directory create -replica\ -clearinghouse` |
| (k) `define cached server` | `cdscache create` |
| (l) `delete child` | `directory remove` |
| (m) `delete clearinghouse` | `clearinghouse delete` |
| (n) `delete directory` | `directory delete` |
| (o) `delete link` | `link delete` |
| (p) `delete object` | `object delete` |
| (q) `delete replica` | `directory delete -replica\ -clear` |

| | |
|---|---|
| (r) dump clerk cache | cdscache dump |
| (s) list child | directory list |
| (t) list clearinghouse | directory list |
| (u) list directory | directory list |
| (v) list link | directory list |
| (w) list object | directory list |
| (x) remove directory | directory modify |
| (y) remove link | link modify |
| (z) remove object | object modify |
| (aa) set cdscp preferred \ clearinghouse | set s(cds) |
| (ab) set directory | directory modify |
| (ac) set directory to skulk | directory synchronize |
| (ad) set link | link modify |
| (ae) set object | object modify |
| (af) show cached clearinghouse | cdscache show -clear |
| (ag) show cached server | cdscache show -server |
| (ah) show cdscp preferred \ clearinghouse | set s(cds) |
| (ai) show child | directory show -child |
| (aj) show clearinghouse | clearinghouse show |
| (ak) show directory | directory show |
| (al) show link | link show |
| (am) show object | object show |
| (an) show replica | directory show -replica -clear |

These are commands which don't have a mapping:

(a) disable clerk
(b) disable server
(c) set directory to new epoch
(d) show cdscp confidence
(e) show cell
(f) show clerk
(g) show server

### A.3. acl_edit

These are commands that are mapped directly:

| | |
|---|---|
| (a) assign | acl replace [cat *filename*] |
| (b) cell | acl modify -cell |
| (c) delete | acl modify -remove |
| (d) get_access | acl check |
| (e) kill_entries | acl delete |
| (f) list | acl show |
| (g) modify | acl modify {-add \| -change} |
| (h) permissions | acl permissions |
| (i) purge | acl purge |
| (j) substitute | acl replace |

These are commands which don't have a mapping:

| | |
|---|---|
| (a) abort | not applicable |
| (b) commit | not applicable, use RET |
| (c) test_access | use acl check instead |

### A.4. rgy_edit

#### A.4.1. Principal, group, and organization commands

These are commands that are mapped directly. In the following list *pgo* is used to mean one of: `principal`, `group`, or `organization`. Also, *go* is used to mean one of: `group` or `organization`.

| | |
|---|---|
| (a) `add` | *pgo* `create` |
| (b) `change` | *pgo* `modify` |
| (c) `delete` | *pgo* `delete` |
| (d) `member` | *go* `[add \| remove]` |
| (e) `view` | *pgo* `show/catalog` |
| (f) `adopt` | *pgo* `create -uuid` |

#### A.4.2. Account commands

These are commands that are mapped directly:

| | |
|---|---|
| (a) `add` | `account create` |
| (b) `change` | `account modify` |
| (c) `delete` | `account delete` |
| (d) `view` | `account show/catalog` |

These are commands which don't have a mapping:

(a) `cell`

#### A.4.3. Key management commands

These are commands that are mapped directly (via a new `dced` related `dcecp` command):

| | |
|---|---|
| (a) `ktadd` | `keytab create` |
| (b) `ktlist` | `keytab show` |
| (c) `ktdelete` | `keytab delete` |

#### A.4.4. Miscellaneous commands

These are commands that are mapped directly:

| | |
|---|---|
| (a) `auth_policy` | `registry modify/show -policies` |
| (b) `login` | `login` |
| (c) `policy` | `registry modify/show -policies` |
| (d) `properties` | `registry modify` |

These are commands which don't have a mapping:

(a) `defaults`
(b) `domain`
(c) `scope`

#### A.4.5. Local registry commands

These are commands which don't have a mapping:

(a) `delete`
(b) `purge`
(c) `view`

### A.5. dtscp

These are commands that are mapped directly:

| | |
|---|---|
| (a) `advertise` | `dts configure` |
| (b) `change` | `clock set -abruptly` |
| (c) `delete` | `dts stop` |

| | |
|---|---|
| (d) `disable` | `dts deactivate` |
| (e) `enable` | `dts activate` |
| (f) `set` | `dts modify` |
| (g) `show` | `dts/clock show` |
| (h) `synchronize` | `dts/clock synchronize` |
| (i) `unadvertise` | `dts configure` |
| (j) `update` | `clock set` |

These are commands which don't have a mapping:

| | |
|---|---|
| (a) `create` | should be a switch on `dts`[21] |

### A.6. sec_admin

These are commands that are mapped directly:

| | |
|---|---|
| (a) `delrep` | `registry delete` |
| (b) `destroy` | `registry delete -`*`something`* |
| (c) `info` | `registry show -attributes` |
| (d) `initrep` | `registry create` |
| (e) `lrep` | `registry catalog` |
| (f) `master_key` | `registry modify -key` |
| (g) `site` | `set s(sec)` |
| (h) `state` | `registry enable/disable` |
| (i) `stop registry stop` | |

These are commands which don't have a mapping:

(a) `become`
(b) `change_master`
(c) `monitor`

## APPENDIX B. LIST OF SECTION TITLES

1. INTRODUCTION
   1.1. Changes Since Last Publication
   1.2. Changes For Later Drafts
   1.3. Not To Be Included

2. TERMINOLOGY

3. TARGET

4. GOALS AND NON-GOALS

5. REQUIREMENTS

6. FUNCTIONAL DEFINITION

7. DATA STRUCTURES
   7.1. Attributes
   7.2. ACL's
      7.2.1. String syntax

21. See DCE CR 8353.

7.2.2. Tcl syntax
7.3. Bindings
7.3.1. String syntax
7.3.2. Tcl syntax
7.4. Interface Identifiers
7.4.1. String syntax
7.4.2. Tcl syntax
7.5. Routing Specifiers
7.6. Debug Routing Specifiers
7.7. Other Data Types

8. USER INTERFACES
8.1. Invocation
8.2. Initialization
8.3. Command Syntax
8.3.1. Object-operation vs. operation-object
8.3.2. Attribute Lists and Switches
8.3.3. Lists of Lists
8.4. Abbreviations
8.5. Convenience Variables
8.6. Error handling
8.7. Help System
8.8. Internationalization
8.9. Common operations
8.10. Common switches
8.11. Objects
8.11.1. rpcentry
8.11.2. rpcgroup
8.11.3. rpcprofile
8.11.4. endpoint
8.11.5. object
8.11.6. directory
8.11.7. link
8.11.8. clearinghouse
8.11.9. cdscache
8.11.10. principal
8.11.11. group
8.11.12. organization
8.11.13. account
8.11.14. registry
8.11.15. acl
8.11.16. dts
8.11.17. clock
8.11.18. aud
8.11.19. audfilter
8.11.20. audevents
8.11.21. audtrail
8.11.22. log
8.11.23. server
8.11.24. Miscellaneous commands
8.12. Tasks
8.13. Command-line Editing Commands

9. API'S

10. REMOTE INTERFACES

11. MANAGEMENT INTERFACES

12. RESTRICTIONS AND LIMITATIONS

13. OTHER COMPONENT DEPENDENCIES

14. COMPATIBILITY

15. STANDARDS

16. OPEN ISSUES

## REFERENCES

[PORTG] DCE Porting and Testing Guide, Open Software Foundation, 1992.

[PORTG] DCE User's Guide and Reference, Open Software Foundation, 1992.

[POSIX.2] IEEE Std 1003.2 — 1988, Portable Operating System Interface — Shells and Utilities.

[POSIX.6] IEEE Std 1003.6 — 1991, Portable Operating System Interface — Security Interface, Draft 12. and Utilities.

[RFC 23] R. Mackey, ''DCE 1.1 Internationalization Guide'', January 1993.

[RFC 24] R. Salz, ''DCE 1.1 Serviceability Proposal'', November 1992.

[RFC 28] S. Luan, R. Weisz, ''A DCE Server Auditable-Event Identification and a Proposed Audit Logging API'', December 1992.

[RFC 29] S. Luan, R. Weisz, ''A Design of the DCE Audit Subsystem — An Implementation Specification'', December 1992.

[RFC 30] H. Melman, ''DCE Administrative Improvement: Requirements for DCE Shell'', December 1992.

[RFC 34] H. Melman, ''DCE 1.1 Coding Style Guide'', April 1993.

[RFC 42.1] R. Salz, P. Wang, ''DCED: The DCE Host Daemon — Command Set'', January 1994.

[TCL] J. Ousterhout, ''Tcl: An Embeddable command Language'', Proc. USENIX Winter Conference, January 1990, pp. 133-146.

[TCL MAN] J. Ousterhout, ''tcl(3) — overview of tool command language facilities'', unpublished manual page, University of California at Berkeley, December 1992.

[TCL BOOK] J. Ousterhout, ''Tcl and the Tk Toolkit'', unpublished draft of book to be published by Addison-Wesley in early 1994.

## AUTHOR'S ADDRESS

Howard Melman
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: melman@osf.org
Telephone: +1-617-621-8989