OSF DCE SIG
Request For Comments: 47.3

J. Bowe (OSF)
D. Mackey (OSF)
R. Salz (OSF)
P. Wang (OSF)
April 1994

DCED: THE DCE HOST DAEMON — FUNCTIONAL SPECIFICATION

1. INTRODUCTION

One of the major thrusts of DCE 1.1 is to improve the lot of the DCE administrator. DCE currently has some omissions that make it difficult to configure or administer a DCE cell from a single login session. These include the following:

(a) There is no DCE mechanism to start a server on a remote host.

(b) There is no DCE mechanism to specify which servers should be started at boot time, nor let administrators easily configure which servers should be running in the face of varying loads.

(c) The current end-point mapper, `rpcd`, has no security.

(d) While DCE has many servers that export multiple interfaces, there is no abstraction or formal definition of what a ''server'' is.

(e) A DCE host has a handful of ''little config files'' such as DCE host name and server key (password) files; there is no remote access to any of these files.

This document describes a new server we will provide with DCE 1.1. It will be a backward-compatible replacement for `rpcd`. It will be able to directly start servers under a variety of circumstances and provide better control over and information about services running on a host. The new server will also be used to maintain the configuration files required by current DCE programs. Finally, the new server will be organized so that other per-host processes can be integrated into it. For example, we will merge in the functionality of `sec_clientd`, removing the need for a separate program.

While the new server will still be used for RPC endpoint management, to emphasize its enhanced role we are calling it `dced`, the DCE daemon.

2. CHANGES SINCE LAST PUBLICATION

The following list identifies the major changes that have been made in this version of the specification, in no particular order:

(a) The concept of ''repositories'', and support for them, has been removed.

(b) The common server operations interface, provided primarily to support repositories, has been removed.

(c) The DCED API RFC [RFC 47.2] has been integrated.

(d) The attribute datatype, `dced_attr_t`, has been replaced by the attributes and interface defined by the Extended Registry Attributes work [RFC 6, ERA].

3. TERMINOLOGY

A ''service'' is a portion of a server that implements related functionality, typically an IDL interface. Conceptually, it could be split off into a separate program, such as the registry service implemented in `secd`. A service is also a collection of peers that supports common interfaces (e.g., several CDS servers within a cell). In this document we always mean the first sense of the word.

This document uses the terms ‘‘daemon’’ and ‘‘server’’ interchangeably to refer to a single executable program running on a host. More specifically, a ‘‘server’’ is a program that accepts RPC requests for one or more interfaces.

4. TARGET

The DCE daemon will be used by the RPC runtime to resolve bindings to servers on the local host.

Application programmers will use `dced`, through the existing RPC API, to register server endpoints. There will be an additional API so that programmers can write servers that require more advanced start-up methods. For example, a server may wish to synchronize with its peers before accepting client requests.

DCE administrators will use `dced` (via `dcecp` commands [RFC 42]) to determine what servers are configured to run on a host, to make those configurations, to start and stop servers, and to determine the state of the active servers. Administrators will use the same access methods to securely manipulate the configuration data currently required by the various DCE programs.

5. GOALS AND NON-GOALS

We want the DCE daemon to provide the following capabilities:

(a) ACL’s associated with RPC endpoints.

(b) Stable storage of server configuration data, such as program pathname, program arguments, and dependencies on other servers.

(c) Automatic startup of registered DCE servers upon receipt of an RPC request. (This is similar to the capabilities provided by `inetd` on UNIX systems.)

(d) The ability to allow clients to manually start registered servers.

(e) The ability to monitor (via both RPC and Posix mechanisms) the execution state of servers and to restart them if they crash.

(f) More useful and convenient information about endpoints and server status by using names in addition to UUIDs and providing a server abstraction in addition to an unrelated list of endpoints.

(g) Integrated code that removes the need for a separate `sec_clientd` program.

(h) Secure remote management of server keys.

(i) Better mechanisms to synchronize clients and servers by allowing servers to indicate their operational state.

(j) The ability to request tickets to talk to servers so that only one process needs to update the machine’s credentials. We will propose an interface for this, but the functionality may not be implemented in the initial release.

In general, the DCE daemon should provide full support for remote, secure configuration of DCE components.

There are other goals that, strictly speaking, are outside the domain of DCE. They include providing the following additional capabilities:

(a) Basic functionality that would make it easier to build a CORBA environment on top of DCE.

(b) The ability to spawn and monitor non-DCE servers.

We would like to meet these goals, subject to resource constraints.

The following are not goals of the initial `dced` release:

(a) To provide a CORBA-compliant Object Adaptor.

(b) To remove the need for a separate `cdsadv` program.

(c) To remove the need for a separate `bosserver` program.

(d) The ability to garbage-collect old keys from all known key tables.

6. STRUCTURE AND REQUIREMENTS

The DCE daemon provides seven independent services. The rest of this section introduces each service and explains the requirements imposed on each. Subsequent sections of this document review each service in the same order.

The `dced` code must conform to the standard DCE 1.1 coding guidelines [RFC 34], including serviceability and internationalization requirements.

The current remote interfaces and API must not be invalidated, and full interoperability with previous release of DCE must be maintained. The end-point mapping rules must be preserved, although additional rules for finding a match can be added. There should be no performance degradation of current DCE functionality. The DCE daemon should be able to handle dozens of servers, each with several interfaces that could control several hundred objects each.

The objects maintained by `dced` will be protected by ACL's using the ACL Library [RFC 46]. All stable storage needed by `dced` will be maintained by using the Backing Store Library [RFC 45].

The DCE daemon will have client interfaces for the new services it provides. This will be in two parts: a client library that calls the wire protocol (analogous to the relationship of the `sec_acl_`*xxx* routines to the RDACL interface), and a set of commands to be part of the DCE control program, `dcecp`. The API is defined below; the commands are defined in [RFC 42].

6.1. Endpoint Mapper (ep)

The endpoint mapper is used by the RPC runtime to resolve bindings and route RPC requests to the appropriate server. This functionality is currently performed by `rpcd`.

Servers on a local host register endpoints, interfaces, and a list of object UUIDs with the endpoint mapper. Servers can remove their endpoints; the mapper also tries to flush stale entries by periodically ''pinging'' servers[1]. The endpoint database currently has no ACL's. This means that anyone in a DCE cell can cause a great deal of damage, such as by removing the security endpoints from a ''locked room'' security server host. DCE 1.0.3 addresses part of this issue by limiting the `ept_delete` operation to clients on the same host. The `rpcd` also maintains a backing store of the endpoint database so that if it crashes, it can be restarted without requiring running servers to re-register.

The DCE daemon must include backwards-compatible endpoint map management. This includes support for current libraries and client programs such as `rpccp`. While this service will remain for compatibility with existing code, we expect that new code will use the higher-level abstractions provided by other (new) parts of `dced`, and that very little code will use the `ept_`*xxx* operations.

Each endpoint will have an ACL associated with it. We use the term ''associated'' because an endpoint will be a part of a larger data item that will include an ACL. The existing operations could now fail because of permission problems; this could not happen with `rpcd` because it had no ACL's. It will be possible to run `dced` in ''compatibility mode'' to maintain this same insecure behavior.

Earlier unpublished `dced` designs required that names and ACL's be directly attached to the data in the current endpoint database. We now believe that it is better to leave the current interface alone and add the new features to new services. The `ep` interface will become a ''viewport'' into the new service data, modifying and retrieving the endpoint fields within the new data items.

6.2. Server Configuration (srvrconf)

This service supports the adminstration of data needed to define and invoke servers. It contains static configuration information about the registered servers that may run on the host. This data will be maintained primarily by configuration or installation scripts (such as `dce_config`) with some maintenance by the DCE administrator.

---

1. The ping is performed using the `rpc_mgmt_is_server_listening` call. All servers should allow any client to invoke this operation.

The `srvrconf` service must be able to start all DCE servers. It must be able to start a server when explicitly requested to do so by a client. It must also be able to start a server implicitly, upon receipt of an RPC operation that would be directed to that server if it were running; this is called dynamic start-up. We will restrict dynamic startup to newly-developed servers, and not provide it for existing servers that have not been registered with the `srvrconf` service.

Server Configuration records may be identified by either name or UUID. They will include a standard set of information specified by OSF, yet must be extensible to allow for vendor- and application-specific needs and future modifications.

A Server Configuration record must include enough information to start servers on a variety of platforms (e.g., UNIX, VMS, MVS). It must have all security-sensitive data, including program pathname, DCE identity, key table identities, local execution identity, and similar information.

6.3. Server Execution (srvrexec)

This service monitors running server processes. The data is maintained by the DCE daemon.

The `srvrexec` service replaces the existing endpoint management interface and API, the `ept_`*xxx* operations provided by `rpcd` and the `rpc_ep_`*xxx* routines in the RPC runtime. Once a server has been started, the Service Configuration service ‘‘hands off’’ supervision of the process to the Server Execution service. This service maintains one object for each configured executing server process. The dynamic information maintained by this service will be stored in a Backing Store database so that if `dced` crashes connectivity will not be lost when it restarts. This database will replace the database used by `rpcd`.

The `srvrexec` service will use the data maintained by the `srvrconf` service to determine if a server should be restarted if it crashes. It will do this by synthesizing a start-up request to the `srvrconf` service.

This service will be invoked by the `ep` service to synthesize a `srvrexec` object for older servers that do not use the registration or configuration aspects of the DCE daemon. It will also be used by management clients to determine what servers are running. Using another definition of the word ‘‘execute’’ the `srvrexec` service can be used to shut down servers in an orderly, as well as abrupt, manner.

Changes in server execution state must be recorded as serviceability events.

6.4. Host Data (hostdata)

This service manages the configuration data for the local host. The data is ‘‘owned’’ by the administrator, and perhaps specific servers or their installation procedures. Much of this information is currently accessible only to local users through the `dce_cf_`*xxx* routines.

This service must provide a remote interface to allow clients to query or change the name of the host, as well as the primary and alias names of the cell. It must be extensible so that additional vendor- and application-specific data can be retrieved and modified. There will be a separate ACL for each item.

The data storage and management must be backward compatible with the existing API. It must be compatible at the file storage level so that existing programs will continue to operate.

The interface must be able to meet the needs of other DCE 1.1 features, such as the hierarchical cells work [RFC 7]. For example, this will include the ability to change the primary name of the cell or host, as well as modify the cell alias list.

6.5. Key Management (rkeytab)

This service provides remote administration of server keys and key tables. The data is ‘‘owned’’ by the administrator, and perhaps specific servers or their installation procedures. A key table is stable storage for a set of encryption keys. A table can hold the keys for multiple principals so that, for example, several servers may share the same key table. Keys also have a version number and an expiration time so that a key can be changed without invalidating existing client sessions that are using the old key.

Each key table will be protected by an ACL.

We do not want to transmit keys in cleartext, but because of United States regulations we cannot export encryption software. As a compromise, we will call the DCE-private routine `rca_pwd_gen_transmit_rep`. This is used by the security runtime to send keys to the security server using one key (the client's current key) to encrypt another key by calling routines in the `sec_crypt_`*xxx* API. (See the ''Open Issues'' section, below.) We expect that foreign licensees will find it easier to add the necessary pieces for security encryption than it will be for them to provide full DCE Privacy.

Key table storage and local access must be backward compatible with the current key management facility.

6.6. Host Security Validation (secval)

This service performs two functions. First, it maintains a login context for the ''self'' identity for the host. The DCE security run-time will inherit this context if the process has no DCE credentials and is a local privileged (e.g., Unix root) process. Second, other programs (on that host) can do an authenticated ''ping'' to this service. If the operation succeeds, the client program can trust the user credentials that it received from the security service since it trusts the `secval` service to not have given away its key. This process builds a ''trust chain'' that prevents a client and fake security server from conspiring to defraud a sensitive program such as `login`. (We have deliberately glossed over some of the more complicated aspects of this issue.)

In DCE 1.0, both functions are provided by `sec_clientd`, a separate process which can be independently stopped and started. It must therefore be possible to enable and disable this service, independently of the other services provided by `dced`.

Recent DCE Change Requests have shown that we must provide a better way for clients to synchronize the use of the credentials and tickets across multiple processes running on the local host. We would like to be able to address this issue.

6.7. Attribute Schema (attr_schema)

Several `dced` services need to store an extensible set of data with their objects. For example, `dtsd` could store its configuration information in its Server Configuration record. As another example, a `hostdata` item can be retrieved as either a set of lines (in the DCE Portable Character Set), or an uninterpreted array of bytes.

In order to provide this flexibility and allow for host-specific enhancements without changing the protocol and breaking interoperability, the DCE daemon supports ''attributes'', as defined in the Extended Registry Attributes (ERA) work being done for the security server [RFC 6, ERA]. Note that ERA was designed primarily for the security database, and that many of the fields and semantics will not be supported by `dced`.

For readers not familiar with the ERA specification, the following synopsis (taken from the functional specification and the `sec_attr_base.idl` file) should be provide enough information.

An ''attribute instance'' is a structure containing an identifying UUID and a value:

```
typedef struct {
                uuid_t                  attr_id;
                sec_attr_value_t        attr_value;
} sec_attr_t;
```

Instances are typically stored as fields within a server's objects. For example, the Server Configuration record will contain a set of of `sec_attr_t`'s that could be used by different (non-Unix) implementations of `dced` or by the servers themselves. An attribute value is a union that supports several types of values:

```
typedef union sec_attr_u
    switch (sec_attr_encoding_t attr_encoding) tagged_union {
                uuid_t                  attr_id;
    case sec_attr_enc_integer:
                signed32                signed_int;
    case sec_attr_enc_printstring_array:
        [ptr]   sec_attr_enc_str_array_t      *string_array;
    case sec_attr_enc_bytes:
        [ptr]   sec_attr_enc_bytes_t  *bytes;
    case sec_attr_enc_uuid:
                uuid_t                  uuid;
    /* ... etc ... */
} sec_attr_value_t;
```

An ‘‘attribute type definition’’ contains the UUID, name, and other identifying information about all instances of a particular attribute. For example, it will identify whether the attribute value must be unique, and what types of objects may have the attribute. The collection of attribution definitions maintained by a server is called the ‘‘attribute schema’’. The `attr_schema` service maintains `dced`’s schema.

7. DEFINITION AND SPECIFICATION

In this section we expand on the formal definition of a DCE service and server that were defined in the ‘‘Terminology’’ section, above. We also define the remote interface for the services to be provided by `dced`. Some of the datatypes shown below have fields marked as ‘‘advisory’’. Such fields will have defined semantics, but will not be implemented in the first release. We are providing them so that later release will not require protocol changes.

The following datatypes introduce some of the basic types used by `dced`, building up to the definition of a server. We first present some low-level ‘‘list’’ datatypes:

```
typedef [unique, string] char           *dced_string_t;

typedef struct dced_string_list_s_t {
                unsigned32              count;
    [size_is (count)]
                dced_string_t           *list;
} dced_string_list_t;

typedef struct uuid_list_s_t {
                unsigned32              count;
    [size_is (count)]
                uuid_t                  *list;
} uuid_list_t;

typedef struct dced_attr_list_s_t {
            unsigned32                  count;
    [ptr,size_is(count)]
            sec_attr_t                  *list;
} dced_attr_list_t;

typedef struct dced_opnum_list_s_t {
            unsigned32                  count;
    [ptr,size_is (count)]
            unsigned32                  *list;
} dced_opnum_list_t;

typedef struct dced_tower_vector_list_s_t {
            unsigned32                  count;
```

```
    [ptr,size_is (count)]
            rpc_tower_vector_p_t        *list;
} dced_tower_vector_list_t;
```

A service contains an interface that may have a name. For compatibility with the current EP and as a (minor) aide to DCE administrators, a service also has an annotation; the annotation is not used by any DCE software, other than to display it to the user. It also has a set of bindings, a (possibly zero-length) list of objects it manages, a set of flag bits, an optional indication of where it is registered with CDS (the `entryname` field is advisory). A service may only implement a subset of the operations defined in an interface, this is recorded in the `operations` field. This field will be advisory in the initial release.

```
typedef struct service_s_t {
            rpc_if_id_t                 ifspec;
            dced_string_t               ifname;
            dced_string_t               annotation;
            dced_tower_vector_list_t    towers;
            uuid_list_t                 objects;
            unsigned32                  flags;
            dced_string_t               entryname;
            dced_opnum_list_t           operations;
} service_t;

typedef struct service_list_s_t {
                unsigned32              count;
    [size_is (count)]
                service_t               *list;
} service_list_t;
```

The `flags` field in the `service_t` datatype should be chosen from the following set[2]:

```
service_c_disabled
```

The `disabled` flag indicates that this service within a server is not currently available. The DCE daemon will not map to an endpoint of a disabled service. Flag bits may be combined; several bits will be reserved for licensee use. The `entryname` field is a ‘‘hint’’ as to where this service appears in the DCE namespace.

A server is specified primarily by the set of services that it provides. It also has a set of attributes that are related to the program. Some of these, such as program name and arguments, are common to all DCE implementations. Others, such as VMS privilege set, will vary depending upon the host operating system. The DCE daemon must run on all types of hosts and must be interoperable across all of them, allowing any client to modify any attribute of any server (subject to access control). To do this, we use a list of attribute instances.

In addition to the variable set of attributes, there is a fixed set constant across all `dced` implementations. This may be pushed down into a union to allow for future changes:

2. You can choose any color you want, as long as it’s black.

```
typedef struct server_fixedattr_s_t {
                unsigned32              startupflags;
                unsigned32              flags;
                dced_string_t           program;
                dced_string_list_t      arguments;
                uuid_list_t             prerequisites;
                uuid_list_t             keytables;

                /* Posix-oriented execution attributes. */
                unsigned32              posix_uid;
                unsigned32              posix_gid;
                dced_string_t           posix_dir;
} server_fixedattr_t;
```

The `startupflags` give the rules for when the server should be started. They may be chosen from the following set:

```
server_c_startup_at_boot
server_c_startup_auto
server_c_startup_explicit
server_c_startup_on_failure
server_c_startup_on_schedule
```

The `at_boot` bit means that `dced` should start the server when `dced` itself is first started. The `auto` bit means that the server can be started automatically upon receipt of an RPC request if `dced` determines that it needs to. The `explicit` bit means that the server can be started if `dced` receives an explicit command to start the server. The `on_failure` bit means that the server should be restarted when `dced` detects that it exited with non-succesful exit code. The `on_schedule` bit is a hook so that an administrator can specify things like ‘‘start this server at 6am Sunday morning’’; it is advisory. Flag bits may be combined; several bits will be reserved for licensee use. The `prerequisites` field is also advisory, and is intended to indicate that the server is normally a client of the specified servers and requires them to be operating before it can function.

Using the above definitions we can define a server:

```
typedef struct server_s_t {
                uuid_t                  id;
                dced_string_t           name;
                service_list_t          services;
                server_fixedattr_t      fixed;
                dced_attr_list_t        attributes;
                dced_string_list_t      prin_names;

                /* If server is running there will be data
                 * about the executing process. */
                union switch (unsigned32 execstate) {
                case server_c_exec_notrunning:
                    ;
                case server_c_exec_running:
                    srvrexec_data_t     data;
                } exec_data;
} server_t;
```

The `srvrexec_data_t` datatype is defined in the Server Execution’’ section below.

The `srvrconf`, `srvrexec`, `hostdata`, and `rkeytab` services provide a uniform way of listing the objects each maintains. This is called the service’s ‘‘entry list’’ and is described using the datatypes given below. Note that for some services the entry list may be explicitly modified, while for others the list is maintained internally and may only be retrieved.

```
typedef struct dced_entry_s_t {
                uuid_t                  id;
                dced_string_t           name;
                dced_string_t           description;
                dced_string_t           storage_tag;
} dced_entry_t;

typedef struct dced_entry_list_s_t {
                unsigned32              count;
    [size_is (count)]
                dced_entry_t            *list;
} dced_entry_list_t;
```

The id and name fields identify each item. The description is an annotation for end-user use. The storage_tag field is interpreted in a host-specific manner; in the reference implementation it is typically the name of the file where the data is stored.

The ep and attr_schema services implement interfaces defined elsewhere; we do not want to add new operations to those services. The secval service does not maintain any objects.

7.1. Endpoint Mapper

The ep service is used to connect clients to servers. Most servers do not use well-known endpoints and the RPC runtime does not export the dynamic part of an endpoint (e.g., the IP port number) to the namespace. As a result, after importing a namespace entry a client has partial endpoint that connects it to the right host. The RPC runtime will then connect to the endpoint mapper on that host at its well-known endpoint and ask it to resolve the endpoint, connecting the client to the proper server.

The endpoint is resolved using a complex algorithm that takes into account the interface uuid and version, and the object uuid specified in the binding handle. Both items are optional. The algorithm is described in the DCE documentation and the DCE RPC AES.

The connection-oriented protocol makes an explicit connection to the endpoint mapper and performs an ept_map operation. The connectionless protocol sends partially-bound operations directly to the endpoint mapper, which uses an internal forwarding mechanism in the DCE runtime to forward the message packet to the correct server which will return the fully-bound handle back to the client.

The full set of endpoint operations are defined in ep.idl. We will support this interface, but do not believe that many clients call these routines directly. Instead, client and server code use the rpc_ep_*xxx* API, which is layered on top of the endpoint wire protocol.

7.2. Server Configuration

The srvrconf service allows administrators to define the servers that can be run on a host. (Of course, an unregistered server may still start and register endpoints without using this service.)

The following operations create and delete srvrconf objects:

```
void
srvrconf_create(
    [in]        handle_t                h,
    [in]        server_t                *server,
    [out]       error_status_t          *st
);
```

```
void
srvrconf_delete(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [out]       error_status_t          *st
);
```

The following operations may be used to modify the attributes of a defined server. These operations completely replace the current contents with the new contents.

```
void
srvrconf_attr_replace(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        dced_attr_list_t        *attributes,
    [out]       error_status_t          *st
);

void
srvrconf_fixattr_replace(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in,unique] server_fixattr_t        *fixed,
    [out]       error_status_t          *st
);
```

The list of `srvrconf` objects maintained by `dced` can be obtained by using the following operation:

```
void
srvrconf_get_list(
    [in]        handle_t                h,
    [out]       dced_entry_list_t       *items,
    [out]       error_status_t          *st
);
```

Once the UUID of a particular object is known, the following operation will return the object:

```
void
srvrconf_get(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [out]       server_t                *value,
    [out]       error_status_t          *st
);
```

The following operation will start a configured server, returning the UUID of the created `srvrexec` object:

```
void
srvrconf_start(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in,unique] server_fixattr_t        *fixed,
    [in,unique] dced_attr_list_t        *attributes,
    [in,out]    uuid_t                  *instance,
    [out]       error_status_t          *st
);
```

If the `fixed` or `attributes` parameters are NULL, then the attributes stored with the server configuration object are used. If they are not NULL, then they will be used to start the server if the ACL allows it, otherwise an error will be returned. The instance UUID may be provided by the client; if a NIL UUID is given dced will create one.

7.3. Server Execution

The srvrexec service allows administrators to query the status of DCE servers running on a host. It also allows them to perform orderly shutdowns of DCE services.

Once a server has been started, this service will maintain a set of execution information about the server using the following datatype:

```
typedef struct srvrexec_data_s_t {
                uuid_t                          instance;
                unsigned32                      posix_pid;
} srvrexec_data_t;
```

If a server is completely defined by its srvrconf entry, it can tell dced that it is running, and register all of its endpoints and object UUIDs with the following datatype and operation:

```
typedef struct srvrexec_ep_s_t {
                rpc_if_id_t                     ifspec;
                dced_tower_vector_list_t        towers;
} srvrexec_ep_t;

typedef struct srvrexec_ep_list_s_t {
                unsigned32              count;
    [ptr,size_is (count)]
                srvrexec_ep_t           *list;
} srvrexec_ep_list_t;

void
srvrexec_started(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        srvrexec_ep_list_t      *eps,
    [in,out]    uuid_t                  *instance,
    [out]       error_status_t          *st
);
```

This operation does not spawn a program, it just changed the dced state.

An unconfigured server registers itself with dced by invoking the following operation:

```
void
srvrexec_create(
    [in]        handle_t                h,
    [in]        server_t                *server,
    [in,out]    uuid_t                  *instance,
    [out]       error_status_t          *st
);
```

The following operation may be invoked to stop a running server:

```
void
srvrexec_stop(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        unsigned32              method,
    [out]       error_status_t          *st
);
```

The id parameter refers to an instance UUID of a running server. The method parameter indicates how dced should stop the service and should be chosen from the following set:

```
srvrexec_c_stop_rpc
srvrexec_c_stop_soft
srvrexec_c_stop_hard
srvrexec_c_stop_error
```

The rpc value indicates that an rpc_mgmt_server_stop_listening call should be done. The other values indicate host-specific ways of doing ‘‘process stop’’ requests at different levels of urgency; error indicates that an attempt should be made to capture the process state before it exists. In the DCE reference implementation, they correspond to sending the TERM, KILL, and ABRT signals, respectively. Some values will be reserved for licensee use.

The following operation can be invoked to effect an orderly shutdown of all DCE servers on the local host:

```
void
srvrexec_shutdown(
    [in]        handle_t                h,
    [out]       error_status_t          *st
);
```

If the client does not have permission to stop all the servers then no action is taken.

It is also possible to modify the set of interfaces for which a server will accept operations:

```
void
srvrexec_enable_service(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        rpc_if_id_t             ifspec,
    [out]       error_status_t          *st
);

void
srvrexec_disable_service(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        rpc_if_id_t             ifspec,
    [out]       error_status_t          *st
);
```

These operations clear and set the service_c_disabled bit in the server’s server_t structure for the specified interface. By using these operations a server can start, allow peer requests on its management interface. When its initialization is done, it can quickly enable its client interface. When shutting down, the inverse set of actions can be performed. This can also be useful for load-balancing, state recovery, and so on.

The list of srvrexec objects and the objects themselves can be retrieved by invoking the srvrexec_get_list and srvrexec_get operations, which have the same calling sequence as the srvrconf operations described above.

7.4. Host Data

The hostdata service provides remote access to view and modify various DCE configuration data. In the reference implementation, this data is stored in separate files that must be maintained on each DCE host. For compatibility we must keep the existing files and format. We also want to provide an extensible mechanism that does not impose any particular storage requirements. We will use the attribute datatypes defined above to access this data:

```
void
hostdata_get(
    [in]        handle_t            h,
    [in]        uuid_t              *id,
    [in]        uuid_t              *attr_id,
    [out]       sec_attr_t          *value,
    [out]       error_status_t      *st
);
```

The attr_id specifies the desired attribute of the object to return. Two attributes will be supported by the reference implementation, both with well-known UUID's: hostdata/textcontents which represents the file as an array of DCE text strings, and hostdata/bincontents which represents the file as an opaque array of bytes.

The following operation can be used to modify a host data item:

```
void
hostdata_set(
    [in]        handle_t            h,
    [in]        uuid_t              *id,
    [in]        sec_attr_t          *value,
    [out]       error_status_t      *st
);
```

Each configuration item will have its own ACL. The following operation can be used to retrieve the list of configuration items:

```
void
hostdata_get_list(
    [in]        handle_t            h,
    [out]       dced_entry_list_t   *items,
    [out]       error_status_t      *st
);
```

The following operation can be used to add a configuration item maintained by the service:

```
void
hostdata_create(
    [in]        handle_t            h,
    [in]        dced_entry_t        *id,
    [in]        boolean32           just_import,
    [in]        sec_attr_t          *value,
    [out]       error_status_t      *st
);
```

If just_import is true, then the id is taken to refer to an existing item on the local host and the contents of the value are ignored. This distinguishes between creating an empty item and adding an existing item to be maintained.

The following operation removes a configuration item from the service:

```
void
hostdata_delete(
    [in]        handle_t              h,
    [in]        uuid_t                *id,
    [in]        boolean32             remove_object,
    [out]       error_status_t        *st
);
```

The remove_object parameter indicates whether or not the stable storage for the item (e.g., the actual data file) should be removed, or if the hostdata should just ‘‘forget’’ about this configuration item by removing it from the entry list.

We will include the following well-known hostdata items:

```
host_name
cell_name
cell_aliases
post_processors
```

These items have special semantics. Changing any of the first three will result in an automatic update of the dce_cf.db file so changes will be reflected through the local dce_cf_XXX API. The post_processors item provides a great deal of flexibility. This item contains set of UUID/program pairs. The UUID specifies a hostdata object; if the UUID of the object is found in the list, then the associated program is run. An example of this is given in the ‘‘Host Security Validation’’ section below.

7.5. Key Management

The rkeytab service provides remote access to view and modify key tables. Key tables are accessed through the sec_key_mgmt_*xxx* API. In the DCE reference implementation key tables are stored as files.

The following datatypes define a key and a list of keys:

```
typedef struct dced_key_s_t {
                dced_string_t           principal;
                unsigned32              version;
                unsigned32              authn_service;
    [unique]
                sec_passwd_rec_t        *passwd;
    [unique]
                rs_acct_key_transmit_t  *transmit_rep;
} dced_key_t;

typedef struct dced_key_list_s_t {
                unsigned32              count;
    [ptr,size_is(count)]
                dced_key_t              *list;
} dced_key_list_t;
```

The authn_service field identifies the authentication protocol; for now, it should always be rpc_c_authn_dce_secret. The rs_acct_key_transmit_t datatype is a key encoded using the application-level session key as described above, and is used only for RPC transmission; in local use it is decrypted into the passwd field. The two fields should be put into a union, but are not because of current IDL limitations.

There are operations to administer the list of key tables. The rkeytab_get_list and rkeytab_delete operations have the same calling sequence and semantics as their counterparts in the other services. The calling sequence to create a key table is slightly different but the semantics are the same as hostdata_create:

```
void
rkeytab_create(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        boolean32               *just_import,
    [in]        dced_key_list_t         *value,
    [out]       error_status_t          *st
);
```

The following operation will retrieve the contents of a key table. It provides the functionality of `sec_key_mgmt_initialize_cursor`, `sec_key_mgmt_get_next_key`, and `sec_key_mgmt_release_cursor` by returning the entire contents of a key table all at once:

```
void
rkeytab_get_keys(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [out]       dced_key_list_t         *keys,
    [out]       error_status_t          *st
);
```

The following operations add or remove a key from a key table, analogous to `sec_key_mgmt_set_key` and `sec_key_mgmt_delete_key_type`:

```
void
rkeytab_add_key(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        dced_key_t              *key,
    [out]       error_status_t          *st
);

void
rkeytab_delete_key(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        dced_key_t              *key,
    [out]       error_status_t          *st
);
```

The following operation changes a key stored in the security server and, if that succeeds, it changes the key in the key table. The action is performed by calling `sec_key_mgmt_change_key`:

```
void
rkeytab_change_key(
    [in]        handle_t                h,
    [in]        uuid_t                  *id,
    [in]        dced_key_t              *key,
    [out]       error_status_t          *st
);
```

This will probably be done by having `dced` create a login context from the old key and use that identity to change the new key.

The existing `sec_key_mgmt_`*xxx* API will remain unchanged. If a server is going to manage its own key table (e.g., by calling `sec_key_mgmt_manage_key`) then it should probably ensure that the corresponding key table object has a read-only ACL.

7.6. Host Security Validation

The secval service exists to validate the DCE security service for programs on the local host, and to verify the cell's names. It does this by exporting the rsec_cert interface, which provides a single operation, rsec_login_validate_cert_auth. All the ''real work'' of this operation is handled by the security run-time; the implementation consists of a single line that set an error_status_ok return code. This service also exports the password override interface, roverride.idl. The operations defined in this file are used by DCE-aware login programs to supercede cell-wide Unix login attributes. For example, a local override file may be used to prevent anyone from logging in as root.

This functionality, currently provided by sec_clientd, will be provided by the secval service. In addition, the following operations will enable or disable the service, analogous to starting or stopping a program:

```
void
secval_enable(
    [in]        handle_t              h,
    [out]       error_status_t        *st
);

void
secval_disable(
    [in]        handle_t              h,
    [out]       error_status_t        *st
);
```

We expect that the sec_login_*xxx* API will be enhanced to return a timestamp indicating when the cell's name(s) have changed. If this timestamp is presented to secval it can update the local host's values. This may either be done implicitly as a result of the pre-authentication work [RFC 26], or it can be done implicitly by modifying the sample login program to invoke an operation like the following:

```
[maybe] void
secval_check_cell_names(
    [in]        handle_t              h,
    [in]        rs_cache_data_t       *cache
);
```

If this operation is invoked and the timestamp is more recent then the cell_aliases hostdata item, then secval will invoke a hostdata_set operation to change the aliases. This will be registered in the post_processors item so that a script is run that takes the aliases and updates the dce_cf.db file automatically.

The following operation will request a ticket to a DCE server:

```
void
secval_request_ticket(
    [in]        handle_t              h,
    [in]        idl_char              *server_princ_name,
    [in]        sec_timeval_period_t  how_long,
    [out]       error_status_t        *st
);
```

The how_long parameter indicates how long the (presumably local) client is willing to wait for a reply. This operation only updates the machine principal's credentials, which in the DCE reference implementation are stored in a file. It may not be implemented in the initial release.

7.7. Attribute Schema

The operations for this service are defined by the rs_attr_schema interface in the ERA work. In the DCE daemon, attribute triggers, multi-valued attributes, default values, inter-cell actions, and the various permission sets are not supported.

Within an attribute value, confidential bytes and attribute sets are not supported.

For complete definitions of these terms, see [ERA].

8. ACCESS CONTROL

Every object in `dced` (except the pseudo-objects provided by the `ep` service) has an ACL. All `dced` objects have names that may be used with the `sec_acl_bind` routine to provide a junction-like naming scheme for their ACLs. The ACL for any object may be specified as:

```
/.:/<hostname>/config/<service>/<name>
```

where `<service>` is one of `srvrconf`, `srvrexec`, `hostdata`, `keytab`[3], or `attr_schema` and `<name>` is the value in the `name` field of the appropriate datatype. Each service will ensure that the names of all its objects are unique. For example, the name of the ACL for the DCE security server configuration record is[4]:

```
/.:/hosts/sulphur/config/srvrconf/secd
```

In the text below, all relative pathnames are based from the `config` entry. Note that these are the names of ACL objects and are not true CDS names. When federated naming [XFN] becomes available, we expect to enhance `dced` so that it can be part of the federated namespace, using the scheme just described. This will mean that `dced` can become the single entity to name all host-specific items in a cell. This will be a powerful leverage point.

The DCE Daemon has ten ACL manager types, a container for each new service and an ACL for each service that has objects. There is also a config pseudo-object, analogous to the security policy object.

Container ACLs have the following permission bits:

```
c       control         Modify the ACL
i       insert          Create new objects
r       read            Read the list of objects
```

Each container will also have a default object ACL, which specifies the ACL for newly-created objects.

The following sections explain the ACL semantics and naming model for each service.

If a client has any permissions to an object, it will be able to read the ACL for that object. (This is the same model used by `secd`).

8.1. Endpoint Mapper

All `ep` objects are psuedo-objects that refer to pieces of the `srvrexec` objects. As such, access will be filtered through the ACL on the real object. This will only effect the ability to remove or replace endpoints since this operation has no access control.

8.2. Server Configuration

The container is `config/srvrconf`; its ACL specifies who can create new `srvrconf` objects. These objects will get their ACL from the `sec_acl_type_default_object` ACL on the container.

The permission bits for the objects are:

---

3. Note that the service is named `rkeytab` but it appears in the namespace as `keytab`.

4. Note that the hostname is `hosts/sulphur`, another example of a junction-like namespace.

```
c       control         Modify the ACL
d       delete          Delete the object
f       flag            Start server with custom flags
r       read            Read object
w       write           Modify the object's attributes
x       execute         Start server
```

8.3. Server Execution

If a server is started from the `srvrconf` service, then it will get a copy of the ACL from that object. If an ‘‘unconfigured’’ server is started, then the `srvrexec` object will get its ACL from the `sec_acl_type_default_object` ACL attached to the `config/srvrexec` pseudo-object.

The permission bits for the objects are:

```
c       control         Modify the ACL
r       read            Read object
w       write           Modify the object's attributes
s       stop            Stop server
```

8.4. Host Data

The container is `config/hostdata`; its ACL specifies who can create new `hostdata` objects. These objects will get their ACL from the `sec_acl_type_default_object` ACL on the container.

The permission bits for the objects are:

```
c       control         Modify the ACL
d       delete          Delete the item
p       purge           Delete the backing storage for an item
r       read            Read an item
w       write           Modify an item
```

8.5. Key Management

The container is `config/keytab`; its ACL specifies who can create new `rkeytab` objects. These objects will get their ACL from the `sec_acl_type_default_object` ACL on the container.

The permission bits for the objects are:

```
a       append          Add new keys to the table
c       control         Modify the ACL
d       delete          Delete the key table
e       expunge         Remove keys from the table
p       purge           Delete backing store for the key table
r       read            Read the keys in the table
```

8.6. Security Validation

The `config/secval` psuedo-object contains the ACL for this service. This is a simple ACL; no container is involved.

The permission bits for the objects are:

```
c       control         Modify the ACL
r       read            Read the object (i.e., ACL)
s       stop            Stop server
u       update          Make ticket requests
x       execute         Start service
```

8.7. Attribute Schema

The container is `config/attr_schema`; its ACL specifies who can create new attribute definitions. These objects will get their ACL from the `sec_acl_type_default_object` ACL on the container.

The permission bits for the objects are:

```
c       control         Modify the ACL
r       read            Read the attribute definition
w       write           Modify the attribute definition
d       delete          Delete the attribute definition
```

9. USER INTERFACES

The sections above define the remote interface, or ‘‘wire protocol’’, for `dced`. Just as programmers use the `sec_acl_`*`xxx`* API rather then calling the `rdacl_`*`xxx`* operations directly, we will provide a programmer library built on top of what is defined here.

The usage model is the same as that used elsewhere in DCE:

(a) A binding is created that is presented to the programmer as an opaque handle. There may be more than one way to create the binding.

(b) The binding is used as the first parameter in a set of domain-specific routines. These routines are called to do the ‘‘real work’’ required by the application.

(c) A routine is called to release the binding and its associated resources.

All routines described here also follow the same three-part calling sequence: input parameters, output parameters, and an output status parameter.

Wherever possible, the DCED API provides generic routines rather than service-specific routines. For example, a single set of three routines is used to iterate over any set of `dced` objects, whether they are key tables or running servers. We believe this makes things simpler, at the cost of some type-safety.

A server will be able to determine its `srvrexec` instance UUID. This will be provided in a system-specific manner (most likely the reference implementation will use an environment variable), with an API to retrieve the value. A running server will be able to retrieve the list of attributes associated with it.

The user interface, built on top of the programmer’s API, will be part of `dcecp`. It is specified in [RFC 42].

9.1. API Introduction

All constants, datatypes, and routines are declared in the following header file:

```
#include <dce/dced.h>
```

For clarity, we will use IDL syntax in the declarations below; they are not actually defined in IDL files.

9.2. Server-Daemon Routines

The following routines are available for a DCE server to communicate with the DCE daemon. The first returns the UUIDs of the `srvrconf` and `srvrexec` objects for the server. (In the reference implementation the DCE daemon will enter these as known environment variables for the spawned servers.) The second routine can be used to retrieve the value of an attribute that is known to the server. (In the reference implementation the DCE daemon will pickle the attribute list and write it into a file specified by a known environment variable.)

```
void
dce_server_inq_uuids(
        [out]   uuid_t                  *conf_id,
        [out]   uuid_t                  *exec_id,
        [out]   error_status_t          *st
);
```

```
void
dce_server_inq_attr(
        [in]    uuid_t                  *exec_id,
        [in]    uuid_t                  *attr_id,
        [out]   sec_attr_t              *value,
        [out]   error_status_t          *st
);
```

The following routines can be used to set up everything required for a server to do fully-authenticated RPC's as both a client and server. Strictly speaking, they are not part of `dced` but they will be provided to demonstrate the convenience that is possible when a server is fully registered with the DCE daemon:

```
void
dce_server_sec_begin(
        [in]    unsigned32               flags,
        [out]   sec_login_handle_t      *context,
        [out]   pthread_t               *km_thread,
        [out]   error_status_t          *st
);

void
dce_server_sec_done(
        [in]    sec_login_handle_t      *context,
        [in]    pthread_t               *km_thread,
        [out]   error_status_t          *st
);
```

The `begin` routine will find the keytable(s) specified in the `srvrconf` entry for the server and set up a DCE identity based on that information, returning it in `context`. The `flags` parameter may include the following bits:

```
dce_server_c_manage_key
```

If this bit is set, then a thread to manage the server's key (i.e., change it on a regular basis) will be spawned and the handle to that thread returned in `km_thread`. The `done` routine undoes all actions performed by the first routine. These routines will typically be bracketed around the server's call to `rpc_server_listen`.

A server can also call the following routine to register its endpoints:

```
void
dce_server_register(
        [in]    uuid_t                  *id,
        [in]    unsigned32              flags,
        [in]    service_list_t          *services,
        [in,out]
                uuid_t                  *inst,
        [out]   error_status_t          *st
);
```

The `flags` field is for later expansion. In the common case, the `services` will be NULL indicating that all services are enabled. If the `towers` field of any service is empty then all protocols will be supported for that service.

A server may disable and enable services by calling the following routines:

```
void
dce_server_enable_if(
        [in]    rpc_if_id_t            *interface,
        [out]   error_status_t         *st
);

void
dce_server_disable_if(
        [in]    rpc_if_id_t            *interface,
        [out]   error_status_t         *st
);
```

9.3. Binding and Identification Routines

The following routines are available to create and release `dced` bindings:

```
/* [opaque] */
typedef struct dced_binding_handle_s_t *dced_binding_handle_t;

void
dced_binding_create(
        [in]    dced_string_t           name,
        [in]    unsigned32              flags,
        [out]   dced_binding_handle_t   *h,
        [out]   error_status_t          *st
);

void
dced_binding_from_rpc_binding(
        [in]    dced_string_t           name,
        [in]    rpc_binding_handle_t    rpc_handle,
        [out]   dced_binding_handle_t   *h,
        [out]   error_status_t          *st
);

void
dced_binding_free(
        [in]    dced_binding_handle_t   h,
        [out]   error_status_t          *st
);
```

The `name` parameter specifies both the desired host, and the service to be used for the generic routines. It can take any of the following forms:

```
/.../cellname/host/config/service
/.:/host/config/service
service@host
service
```

The first two forms identify a host in the global namespace and the local cell, respectively. The third form is a shorthand version of the second form. The last form is used to refer to either the local cell (for `dced_binding_create`) or to identify the service if using an existing RPC binding handle (`dced_binding_from_rpc_binding`). The `service` names the `dced` service and should be one of those services — `hostdata`, `srvrexec`, and so on.

To following routine is analogous to `rpc_binding_set_auth_info` except that it operates on a `dced` binding handle:

```
void
dced_binding_set_auth_info(
        [in]    dced_binding_handle_t   h,
        [in]    unsigned32              protect_level,
        [in]    unsigned32              authn_svc,
        [in]    rpc_auth_identity_handle_t      authn_identity,
        [in]    unsigned32              authz_svc,
        [out]   error_status_t          *st
);
```

Every object maintained by `dced` is identified by a UUID and a name (the server enforces unique names). The following routines can be used to map from one to the other:

```
void
dced_inq_id(
        [in]    dced_binding_handle_t  h,
        [in]    dced_string_t          name,
        [out]   uuid_t                 *id,
        [out]   error_status_t         *st
);

void
dced_inq_name(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [out]   dced_string_t          *name,
        [out]   error_status_t         *st
);
```

9.4. Object List Routines

Each `dced` service maintains a list of its objects, this list is called the `entry list`. Conceptually, the list is the ‘‘names’’ of the objects, not the objects themselves. For the `srvrconf` and `srvrexec` services, the objects are the in-core `server_t` data, while for the `hostdata` and `rkeytab` services the objects are the data stored in the location specified by the `storage_tag` fields.

There are two ways to examine an entry list: the entire list may be retrieved at once, or a cursor may be used to step through an entry at a time. The following routines provide these capabilities:

```
void
dced_list_get(
        [in]    dced_binding_handle_t  h,
        [out]   dced_entry_list_t      *list
        [out]   error_status_t         *st
);

void
dced_list_release(
        [in]    dced_binding_handle_t  h,
        [out]   dced_entry_list_t      *list
        [out]   error_status_t         *st
);

/* [opaque] */
typedef struct dced_cursor_s_t *dced_cursor_t;
```

```
void
dced_entry_initialize_cursor(
        [in]    dced_binding_handle_t  h,
        [out]   dced_cursor_t          *cursor,
        [out]   error_status_t         *st
);

void
dced_entry_get_next(
        [in]    dced_cursor_t          cursor,
        [out]   dced_entry_t           **entry,
        [out]   error_status_t         *st
);

void
dced_entry_cursor_release(
        [in]    dced_cursor_t          cursor,
        [out]   error_status_t         *st
);
```

The ‘‘cursor’’ routines are provided so that applications can be written that do not require the entire entry list to be transmitted in a single RPC. While direct RPC support for this will not be provided in the initial release, the API is provided so that clients can be written using this style.

For the `hostdata` and `rkeytab` services it is possible to directly manipulate the entry list:

```
void
dced_entry_add(
        [in]    dced_binding_handle_t  h,
        [in]    dced_entry_t           *entry,
        [out]   error_status_t         *st
);

void
dced_entry_remove(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [out]   error_status_t         *st
);
```

The entry lists provided by the other services cannot be directly manipulated by these routines.

9.5. Retrieving Data

The following two routines may be used to read the object data:

```
void
dced_object_read_all(
        [in]    dced_binding_handle_t  h,
        [out]   unsigned32             *count,
        [out]   void                   **list
        [out]   error_status_t         *st
);
```

```
void
dced_object_read(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [out]   void                   **data,
        [out]   error_status_t         *st
);
```

Like the entry list routines, the first returns all objects, while the second returns the objects one at a time.

The following single routine frees either an object or a list of all object data:

```
void
dced_objects_release(
        [in]    dced_binding_handle_t  h,
        [in]    unsigned32             *count,
        [in]    void                   *dataptr
        [out]   error_status_t         *st
);
```

The following examples show how the routine defined above could be used:

```
/* Variable declarations used by the samples below. */
dced_binding_handle     bh;
error_status_t          st;
dced_cursor_t           curs;
dced_entry_t            *e;
dced_key_list_t         *ktab;
server_t                *serv_array;

/* Bind to local dced, set up cursor. */
dced_binding_create("keytab", &bh, &st);
dced_entry_initialize_cursor(bh, &curs, &st);

/* Loop over all objects. */
for ( ; ; ) {
    dced_entry_get_next(curs, &e, &st);
    if (st == dced_st_no_more)
        break;

    /* Read an object and then release it. */
    dced_object_read(bh, &e->uuid, (void *)&ktab, &st);
    dced_objects_release(bh, 1, (void *)ktab, &st);
}

/* Free resources allocated above. */
dced_entry_cursor_release(curs, &st);
dced_binding_free(bh, &st);

/* Bind to a specific host's dced. */
dced_binding_create("/.:/hosts/sulphur/config/srvrexec",
    &bh, &st);
```

```
/* Get info on all running servers and display it. */
dced_object_read_all(bh, &count, (void **)&serv_array, &st);
for (i = 0; i < count; i++)
    display(&serv_array[i]);

/* Free resources allocated above. */
dced_objects_release(bh, count, (void *)serv_array, &st);
dced_binding_free(bh, &st);
```

9.6. Server Routines

The following two routines create and delete `dced srvrconf` objects:

```
void
dced_server_create(
        [in]    dced_binding_handle_t  h,
        [in]    server_t               *object,
        [out]   error_status_t         *st
);

void
dced_server_delete(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [out]   error_status_t         *st
);
```

The following two routines can be used to start a server that has been configured with `dced`:

```
void
dced_server_start(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    dced_attr_list_t       *attributes,
        [out]   error_status_t         *st
);

void
dced_server_stop(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    srvrexec_stop_method   method,
        [out]   error_status_t         *st
);
```

If the `id` parameter given to `dced_server_stop` is a NULL pointer, then all servers on that host (including `dced` itself) are shut down.

The following two routines may be used to control what services are active within a server:

```
void
dced_server_enable_if(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    rpc_if_id_t            *interface,
        [out]   error_status_t         *st
);
```

```
void
dced_server_disable_if(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    rpc_if_id_t            *interface,
        [out]   error_status_t         *st
);
```

The following routine modifies the data of a server, either configured or running:

```
void
dced_server_modify_attributes(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    dced_attr_list_t       attributes,
        [out]   error_status_t         *st
);
```

9.7. Host Configuration Data Routines

The following two routines create and delete host data configuration items. They also cause the `hostdata` entry list to be updated:

```
void
dced_hostdata_create(
        [in]    dced_binding_handle_t  h,
        [in]    dced_entry_t           *entry,
        [in]    dced_attr_list_t       *data,
        [out]   error_status_t         *st
);

void
dced_hostdata_delete(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [out]   error_status_t         *st
);
```

Each `hostdata` item has attributes that contain the ‘‘real’’ data of the item. The following two routines read and write attributes.

```
void
dced_hostdata_read(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    uuid_t                 *attr_id,
        [out]   sec_attr_t              *data,
        [out]   error_status_t         *st
);

void
dced_hostdata_write(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    dced_attr_list_t       *data,
        [out]   error_status_t         *st
);
```

Writing data replaces any existing value. The most common and useful attribute is the `filecontents` attribute which represents the file contents as an array of strings. This will have a well-known UUID.

9.8. Key Management Routines

The following two routines create and delete key tables. They also cause the `keytable` entry list to be updated:

```
void
dced_keytab_create(
        [in]    dced_binding_handle_t  h,
        [in]    dced_entry_t           *entry,
        [in]    dced_key_list_t        *keys,
        [out]   error_status_t         *st
);

void
dced_keytab_delete(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [out]   error_status_t         *st
);
```

The ‘‘object read’’ routines described above will retrieve the contents of a key table. The following datatype and routines can be used to step through individual entries of a key table:

```
/* [opaque] */
typedef struct dced_keytab_cursor_s_t *dced_keytab_cursor_t;

void dced_keytab_initialize_cursor(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [out]   dced_keytab_cursor_t   *cursor,
        [out]   error_status_t         *st
);

void dced_keytab_get_next_key(
        [in]    dced_keytab_cursor_t  cursor,
        [out]   dced_key_t            **key,
        [out]   error_status_t        *st
);

void dced_keytab_release_cursor(
        [in]    dced_keytab_cursor_t   cursor,
        [out]   error_status_t         *st
);
```

The following routines add and delete keys from a key table:

```
void
dced_keytab_add_key(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    dced_key_t             *key,
        [out]   error_status_t         *st
);
```

```
void
dced_keytab_remove_key(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    dced_key_t             *key,
        [out]   error_status_t         *st
);
```

When deleting a key, if the key ‘‘version’’ is zero, then all keys for the specified principal are deleted.

It is also possible to change a key, updating both the registry and the key table, by calling the following routine:

```
void
dced_keytab_change_key(
        [in]    dced_binding_handle_t  h,
        [in]    uuid_t                 *id,
        [in]    dced_key_t             *key,
        [out]   error_status_t         *st
);
```

9.9. Security Validation Routines

The following routines can be used to start or stop the security validation service within `dced`:

```
void
dced_secval_start(
        [in]    dced_binding_handle_t  h,
        [out]   error_status_t         *st
);

void
dced_secval_stop(
        [in]    dced_binding_handle_t  h,
        [out]   error_status_t         *st
);
```

The following routine may be used to validate the credentials returned by a DCE security server. It is most useful when `h` is bound to the local host and the program needs to complete the trust chain:

```
void
dced_secval_validate(
        [in]    dced_binding_handle_t  h,
        [out]   error_status_t         *st
);
```

9.10. Attribute Schema Routines

The `sec_attr_`*xxx* API provided with the ERA work will be able to bind to the DCE daemon and manipulate its attribute schema.

9.11. Session Managers Using DCED

The DCE daemon can be used to start programs that are not DCE servers. One popular example of this is using `dced` as the remote invocation back-end to a window session manager. The DCE daemon listens on a well-known Internet port so that the DCE name service is not required. The daemon can also run in an unauthenticated mode (bypassing most ACL checks) so that DCE security is not needed. Taken together, these mean that `dced` and the necessary client routines can be made to work stand-alone without the full DCE infrastructure.

We will define three well-known attributes: user identity, additional environment values (a list that will typically contain at least a ‘‘DISPLAY’’ variable), and additional command-line text. We will make `dced` aware of these attributes.

The DCE daemon can only start servers that have been configured through the `srvrconf` service. For non-DCE servers, the `server_t` datatype becomes fairly light-weight and it should not be difficult to write a specialized routine to create window-oriented servers that are DCE-ignorant.

In the new X Session Management protocol [Sess], the command line is locale-dependent and cannot be parsed into a conventional parameter list. In this case, `argv` will have two elements, the command line value and a NULL. The server to be started will probably have a shell wrapper script that ends with something like the following four lines:

```
server=$1
shift
shift
exec $server $*
```

Note that the spawned server is `exec`'d. If the script starts the server in the background and then exits, `dced` could treat this as a failure and attempt to restart it.

10. RESTRICTIONS AND LIMITATIONS

It is important to understand the semantics and implications of autostarted servers. The DCE Daemon provides transparent start, not transparent re-start. We expect that auto-started servers, like other DCE servers, are long-lived as opposed to transient processes. If a server crashed and it is restarted, clients will notice. Note that transparent re-start is a CORBA requirement.

Because of the way the datagram RPC endpoint resolution works, it is quite possible that attempts to use auto-started servers will result in client timeouts. We are trying to minimize the cirucmstances under which this can happen, and are considering overloading some current RPC DG PDU's to reduce this even further.

Note that disabling a service only means that new clients will not get bound to that server. Existing clients will still be able to invoke operations on that interface over their existing binding handle. This is analogous to the current practice of removing an endpoint.

Any `hostdata` or `rkeytab` objects created by the DCE daemon will be owned by the local privileged user (e.g., Unix root).

11. OTHER COMPONENT DEPENDENCIES

The DCE server requires the Backing Store and ACL Libraries. These are OSF deliverables for DCE 1.1.

The user interface will be provided as code to be integrated into `dcecp` and is therefore dependent on that component. This is an OSF deliverable for DCE 1.1.

The `secval` service must be able to determine the current names of the cell. This depends on the hierarchical cells work being done in the security server.

12. COMPATIBILITY

Old clients will not be able to see the new server data.

Old management clients will not be able to delete endpoints unless they have permission. This is a new failure status that they may not expect to receive.

13. STANDARDS

The DCE daemon will follow the family of standards specified in the DCE Coding Style document [RFC 42].

Note that to ensure communicability, the ''name'' fields in the structures defined here use the IDL Portable Character Set, not RPC I18N Characters [RFC 41],

14. OPEN ISSUES

There is a small number of issues that should be resolved before the release of DCE 1.1. The first set is related to `dced` itself:

(a) We assume that if `dced` is managing a local file such as a key table, it is the only entity that will write that file. Since `dced` provides no ‘‘open for update’’ operation, local file locking will be of little help here.

(b) It must be possible to ‘‘rate-limit’’ how often a (failing) server is restarted, such as by using a ‘‘maximum number of restarts in `n` seconds’’ parameter.

Items in the second set of issues are dependent upon other components:

(a) If `dced` can start `secd` in ‘‘locksmith’’ mode then the DCE daemon must be considered part of the Trusted Computing Base. It is probably easiest to modify `secd` so that it must be invoked under a different name before entering that mode.

(b) It might be desirable to make the security run-time aware of the `secval_request_ticket` operation, and have credential routines be able to invoke this operation (or a similar one) automatically. This is beyond the scope of the `dced` effort.

REFERENCES

[RFC 6] J. Pato, DCE-RFC 6.0, ‘‘A Generic Interface for Extended Registry Attributes’’, June, 1992.

[RFC 7] J. Pato, DCE-RFC 7.0, ‘‘Hierarchical Trust Relationships for Inter-Cell Authentication’’, July, 1992.

[RFC 26] J. Pato, DCE-RFC 26.0, ‘‘Using Pre-Authentication to Avoid Password-Guessing Attacks’’, June, 1993.

[RFC 34] H. Melman, DCE-RFC 34.2, ‘‘DCE 1.1 Coding Style Guide’’, June, 1993.

[RFC 41] M. Romagna, R. Mackey, DCE-RFC 41.1, ‘‘RPC Runtime Support for I18N Characters Functional Specification’’, September, 1993.

[RFC 42] H. Melman, DCE-RFC 42.2 ‘‘DCE Shell Functional Specification’’, March, 1994.

[RFC 45] R. Mackey, R. Salz, ‘‘DCE Backing Store Library Functional Specification’’, July 1993.

[RFC 46] R. Mackey, R. Salz, ‘‘DCE ACL Library Functional Specification’’, October 1993.

[ERA] Anne Hopkins, Mary Bennett, ‘‘DCE 1.1 Functional Spec: Extended Registry Attributes’’, January, 1994.

[Sess] X Consortium, ‘‘Session Management Protocol Specification’’, 1993.

[XFN] Federated Naming Working Group, X/Open Preliminary Specification, ‘‘Federating Naming Systems’’, in preparation.

16. ACKNOWLEDGEMENTS

John Shirley made many helpful suggestions on routine names and organization.

AUTHOR’S ADDRESS

John Bowe
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: bowe@osf.org
Telephone: +1-617-621-7269

Richard Mackey
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA
Internet email: dmackey@osf.org
Telephone: +1-617-621-8924

Rich Salz
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA
Internet email: rsalz@osf.org
Telephone: +1-617-621-7253

Ping Wang
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA
Internet email: pwang@osf.org
Telephone: +1-617-621-8988