OSF DCE SIG **R. Mackey (OSF)**
Request For Comments: 47.0 **R. Salz (OSF)**
December 1993

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 configure which servers should be running in the face of varying system loads.

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

(d) There is no concept of a ''server'', a single program that exports multiple IDL interfaces.

(e) A DCE host has a handful of ''little config files'' ranging from ISO OID names to server key (password) files; there is no remote access to any of these files.

This document describes a new server we plan to 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) what services are 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 expect to merge in the functionality of `sec_clientd`, removing the need for a separate program.

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

2. TERMINOLOGY

This document uses the terms ''daemon'' and ''server'' interchangeably to refer to a single executable program running on a host.

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. For example, 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.

A ''repository'' is a server that manages a large or variable set of objects such that it is not feasible to provide a complete or accurate list of UUIDs to the endpoint mapper.

The `dced` will use the Backing Store Library [RFC 45] and the ACL Library [RFC 46]. Readers might find it helpful to be familiar with the terminology of those documents.

3. 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` 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 also use `dced` (via `dcecp` commands [RFC 42]) to securely manipulate the configuration data currently required by the various DCE programs.

4. GOALS AND NON-GOALS

We want the DCE daemon to provide the following capabilities:

(a) DCE 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 explicitly start registered servers.

(e) The ability to monitor execution state of servers and to restart them if they crash.

(f) A more scalable method for handling servers with large numbers of objects by allowing a server to define itself as a ''repository'' rather than registering each object UUID.

(g) 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.

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

(i) Secure remote management of server encryption keys.

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

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 contraints.

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.

5. STRUCTURE AND REQUIREMENTS

The DCE daemon provides six 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 DCE daemon must provide 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`.

5.1. Endpoint Mapper (ep)

The endpoint mapper is used by the RPC runtime to resolve binding handles 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. The endpoint database currently has no ACL's. This means that anyone in a DCE cell can cause a great deal of damage, e.g., 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 upward-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 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 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.

5.2. 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 service must be able to start and run all DCE servers. It must be able to start servers using a partial ordering (to catch dependencies such as the security server's dependency on the directory service) at boot time. 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 only use the old-style `ep` 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 names, local execution identity, and DCE privileges.

5.3. Server Execution (srvrexec)

This service monitors running server processes. It replaces the existing endpoint management interface and API, `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.

The `srvrexec` service will use the data maintained by the `srvrconf` service to determine if a server should be restarted when it crashes and to determine if it is possible to start a server in order to satisfy an incoming RPC request that cannot otherwise be handled.

The Server Execution 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.

The Server Execution service will 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.

5.4. Host Data (hostdata)

This service manages the configuration data for the local host. 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 host's name and the name(s) of its cell. It must be extensible so that additional vendor-specific data can be retrieved and modified. There will be a separate ACL for each file.

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 program 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, as well as modify the cell alias list.

5.5. Key Management (keymgmt)

This service provides remote administration of server keys and key tables. A key table is backing 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 any current tickets in use with 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. As a compromise, we will call the undocumented 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. 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.

5.6. Host Security Validation (secval)

This service performs two functions. It maintains a login context for the ''self'' identity for the machine. Other programs (on the 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`.)

In addition, if a local privileged process (e.g., a Unix root process) needs DCE credentials and has none of its own, then the security library will inherit the context maintained by this service and use the machine's identity as its own.

This service is currently provided by `sec_clientd`, a separate process which can be independently stopped and started. It must therefore still be possible to enable and disable this service, independently of the other services provided by `dced`.

Recent DCE Change Request's 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.

6. DEFINITION AND SPECIFICATION

In this section we will expand on the formal definition of a DCE service and server that were defined in Section 2, above. We will also define the remote interface for the services to be provided by `dced`.

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, and an optional indication of where it is registered with CDS. Finally, it may have a set of names used for authentication.

The following datatypes introduce some of the basic types used by `dced`, building up to the definition of a service. As defined above, a service is primarily a set of endpoints and objects:

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

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 rpc_binding_vector_list_s_t {
                unsigned32              count;
    [size_is (count)]
                rpc_binding_vector_t    *list;
} rpc_binding_vector_list_t;

typedef struct service_s_t {
                rpc_if_handle_t         ifspec;
                dced_string_t           *ifname;
                dced_string_t           *annotation;
                rpc_binding_vector_list_t *bindings;
                unsigned32              flags;
                uuid_list_t             objects;
                dced_string_t           *entryname;
    }
} 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 list:

```
service_c_repository
service_c_disabled
```

The `repository` bit indicates that the server is a repository, and that it supports the common server operation (see below) to determine whether or not it is responsible for a particular object. 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 implemementations. 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 provide the following datatypes:

```
typedef enum dced_attr_type_e_t {
    dced_attr_type_string,
    dced_attr_type_binary,
    dced_attr_type_stringlist
} dced_attr_type_t;

typedef struct dced_attr_binary_s_t {
                unsigned32              count;
    [size_is (count)]
                byte                    *data;
} dced_attr_binary_t;

typedef struct dced_attr_s_t {
                uuid_t                  id;
                dced_string_t           name;
    union switch (dced_attr_type_t type) {
    dced_attr_type_string:
                dced_string_t           stringdata;
    dced_attr_type_binary:
                dced_attr_binary_t      bindata;
    dced_attr_type_stringlist:
                dced_string_list_t      stringlistdata;
    }
} dced_attr_t;

typedef struct dced_attr_list_s_t {
                unsigned32              count;
    [size_is (count)]
                dced_attr_t             *list;
} dced_attr_list_t;
```

An attribute has a name and the data. By design only a small set of data types are supported. It might be ‘‘better’’ to use the types and interfaces defined by the Extended Registry Attributes [RFC 6] of the DCE 1.1 security service. Doing this would meet the important goal of DCE consistency, however, there are two reasons to prefer the scheme shown above: we cannot afford to introduce a scheduling dependency on the ERA work, and even if it were available now the ERA system is probably overkill for this application.

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

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

                /* Posix-oriented execution attributes.
                 * Might become variable 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_startup
server_c_startup_on_failure
server_c_startup_on_request
server_c_startup_on_rpc
server_c_startup_on_schedule
```

The `at_startup` bit means that `dced` should start the server when `dced` itself is first started. The `on_failure` bit means that the server should be restarted when `dced` detects that it exited with non-succesful exit code. The `on_request` bit means that the server can be started dynamically if `dced` determines that it needs to; this bit is ignored for repositories. The `on_rpc` bit means that the server can be started if `dced` receives an explicit command to start the server. The `on_schedule` bit is a hook so that an administrator can specify things like ''start this server at 6am Sunday morning''. This capability will not be provided in the initial release. Flag bits may be combined; several bits will be reserved for licensee use.

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_fixattr_t        fixed;
                dced_attr_list_t        attributes;

                /* If server is running there will be data
                 * about the executing process. */
                union switch (boolean32 running) {
                false:
                    ;
                true:
                    srvrexec_data_t     data;
                }
} server_t;
```

The `srvrexec_data_t` datatype is defined in section 6.3, below.

6.1. Endpoint Mapper

The `ep` service is used to connect clients to servers. Most servers do not use well-known endpoints and 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 map on that host 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 the internal forwarding mechanism to both forward the message packet and 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. If time permits we might rewrite this API to use the new `dced` services.

6.2. 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]        server_fixattr_t        fixed,
    [out]       error_status_t          *st
);
```

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

```
typedef struct hostdata_list_s_t srvrconf_list_t;

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

The `hostdata_list_t` datatype is defined in Section 6.4, below. It is basically a list of <UUID,name> pairs.

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.

6.3. 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 {
                /* <<To be determined.>> */
} srvrexec_data_t;
```

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

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

This operation does not spawn a program.

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

```
void
srvrexec_create(
    [in]        handle_t                h,
    [in]        server_t                *server,
    [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 `method` parameter indicates how `dced` should stop the service and should be chosen from the following set:

```
srvrexec_c_stop_rpc
srvrexec_c_stop_local_soft
srvrexec_c_stop_local_hard
srvrexec_c_stop_local_error
```

The `rpc` value indicates that an `rpc_mgmt_server_stop_listening` call should be done. The `local` values indicate host-specific ways of doing ''process stop'' requests at different levels of urgency; `local_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
);
```

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]        rpc_if_handle_t         ifspec,
    [out]       error_status_t          *st
);

void
srvrexec_disable_service(
    [in]        handle_t                h,
    [in]        rpc_if_handle_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.

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.

6.4. Host Data

The `hostdata` service provides remote access to view and modify various DCE configuration data. This data is currently stored in a collection of separate files that must be maintained on each DCE hosts. 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,
    [out]       dced_attr_t             *value,
    [out]       error_status_t          *st
);

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

A configuration item will generally be the contents of a file, represented as an array of strings.

To provide extensibility, the `hostdata` service will maintain a list of configuration items using the following datatypes:

```
typedef struct hostdata_s_t {
                uuid_t                  id;
                dced_string_t           name;
                dced_string_t           description;
                dced_string_t           storage_tag;
} hostdata_t;

typedef struct hostdata_list_s_t {
                unsigned32              count;
    [size_is (count)]
                hostdata_t              *list;
} hostdata_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 will be the name of the file where the data is stored. 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,
    [in, out]   hostdata_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]        hostdata_t              id,
    [in]        boolean32               just_import,
    [in]        dced_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 backing storage for the item (e.g., the `dce_cf.db` file) should be removed, or if the host data service should just ''forget'' about this configuration item.

We will provide an initial set of well-known `hostdata` items, with published UUIDs. The list will include the following:

```
host_name
cell_name
cell_aliases
host_type
```

6.5. Key Management

The `keymgmt` service provides remote access to view and modify key tables. In the DCE reference implementation key tables are stored as files. The list of key tables administered by this service is maintained using the same data types used by the `hostdata` service:

```
typedef struct hostdata_s_t keytable_t;

typedef struct hostdata_list_s_t keytable_list_t;
```

There are also related operations to administer the list of key tables. The `keytable_get_list` and `keytable_delete` operations have the same calling sequence and semantics as their `hostdata` counterparts. The calling sequence to create a key table is slightly different but the semantics are the same as `hostdata_create`:

```
void
keytable_create(
    [in]        handle_t                h,
    [in]        keytable_t              id,
    [in]        boolean32               just_import,
    [in]        key_list_t              *value,
    [out]       error_status_t          *st
);
```

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

```
typedef struct key_s_t {
                dced_string_t           principal;
                unsigned32              version;
                sec_key_mgmt_authn_service authn_service;
                rs_acct_key_transmit_t  transmit_rep;
} key_t;

typedef struct key_list_s_t {
                unsigned32              count;
    [size_is (count)]
                key_t                   *list;
} 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 dced-client session key, as described above.

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

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

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

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

void
keymgmt_delete_key(
    [in]        handle_t                h,
    [in]        uuid_t                  id,
    [in]        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
keymgmt_change_key(
    [in]        handle_t                h,
    [in]        uuid_t                  id,
    [in]        key_t                   key,
    [out]       error_status_t          *st
);
```

This operation should use delegation [RFC 3] to change the key stored in the security server.

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.

6.6. Host Security Validation

The secval service exists to validate the DCE security service for programs on the local host. 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 runtime; the implementation consists of a single line that returns a 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
);
```

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.

7. ACCESS CONTROL

Every object in `dced` (except the pseudo-objects provided by the `ep` service) has an ACL. To bind to an ACL (e.g., when using `acl_edit`), objects may be referenced from the existing `rpcd` server entry, `/.:/<hostname>/self`. To edit the ACL for an object, users may specify the object's UUID. All `dced` objects have names that may be used with the `sec_acl_bind` routine to provide a junction-like naming scheme for ACL's. The ACL for any object may be specified as:

```
/.:/<hostname>/self/acl/<service>/<name>
```

where `<service>` is one of `srvrconf`, `srvrexec`, `hostdata`, `keymgmt`, or `secval` and `<name` is the value in the `name` field of the appropriate datatype. For example, the name of the ACL for the DCE security server is:

```
/.:/hosts/sulphur/self/acl/srvrconf/secd
```

In the text below, all relative pathnames are based from the `self` entry. Note that these are the names of ACL objects and are not true CDS names. When federating naming [XFN] becomes available, we expect to enhance `dced` so that it can be part of the federated namespace. The names will probably follow the same system just described except that the `acl/` ''sub-directory'' will not be used. 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 five ACL manager types, one for each new service. In addition, each service will have a container pseudo-object. The container has an ACL whose primary purpose is to specify who can create new objects. These ACLs have the following permission bits:

```
c       control         Modify the ACL
i       insert          Create new 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`).

7.1. 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.

7.2. Server Configuration

The container is `acl/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 are:

```
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
```

This ACL is shared with the `srvrexec` service, so some of the permissions are described in the next section.

7.3. 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 `acl/srvrexec` pseudo-object.

The additional permission bits are:

```
s       stop            Stop server
```

7.4. Host Data

The container is `acl/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 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
```

7.5. Key Management

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

The permission bits are:

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

7.6. Security Validation

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

The permission bits are:

```
c       control         Modify the ACL
s       stop            Stop server
u       update          Make ticket requests
x       execute         Start service
```

8. COMMON SERVER OPERATIONS

In order for `dced` to support repositories, a common ‘‘do you have this object?’’ operation must be defined. Programmers may also use the facilities provided by `dced` to develop more advanced applications. For example, a server can start up, synchronize with its peers, and then make itself available to process client requests.

In order to meet current needs and provide a platform for future commonality, we define an interface that will hold operations common across ‘‘all’’ servers. This is the `cso` interface, for ‘‘common server operations’’. The `cso` interface is not part of the RPC management interface because of a number of reasons that, taken together, indicate a poor fit. These operations are for application development, not management; they are designed to be re-implemented; the interface will expand as we add more commonality to all servers; and finally we do not want to have the binding and registration special-casing that is part of the management interface.

We start by defining an object reference:

```
typedef enum dce_objref_type_e_t {
    dce_objref_type_simple,
    dce_objref_type_with_ifspec
} dce_objref_type_t;

typedef struct dce_objref_with_ifspec_s_t {
                uuid_t                  id,
                rpc_if_t                ifspec
} dce_objref_with_ifspec_t;

typedef union switch (dce_objref_type_t type) {
dce_objref_type_simple:
                uuid_t                  simple;
dce_objref_type_with_ifspec:
                dce_objref_with_ifspec_t with_ifspec;
} dce_objref_t;

typedef struct dce_objref_list_s_t {
                unsigned32              count;
    [size_is (count)]
                dce_objref_t            *list;
} dce_objref_list_t;
```

The intent of the `with_ifspec` reference is to support servers split across multiple executables.

The object lookup operation is:

```
void
cso_locate(
    [in]        handle_t                h,
    [in]        dce_objref_t            objref,
    [out]       unsigned32              *reply,
    [out]       error_status_t          *st
);
```

The `reply` may contain the following bits:

```
cso_c_locate_haveit
cso_c_locate_donthaveit
cso_c_locate_nosuchobject
cso_c_locate_dontknow
```

The first three bits indicate a definitive answer as to whether or not the object is maintained by this server, or does not exist at all (e.g., a file known to have been removed). The `dontknow` bit can be returned when the server's state is very volatile; clients should probably treat it as a `donthaveit` response.

The following operation may return the complete list of objects maintained by a server:

```
void
cso_list_objrefs(
    [in]        handle_t                h,
    [in, out]   dce_objref_list_t       *objref,
    [out]       unsigned32              *reply,
    [out]       error_status_t          *st
);
```

The `reply` parameter must be chosen from the following list:

```
cso_c_list_objrefs_complete
cso_c_list_objrefs_incomplete
cso_c_list_objrefs_impossible
```

Applications that want to find out the status of a server may invoke the following operation:

```
void
cso_status(
    [in]        handle_t                h,
    [out]       unsigned32              *reply,
    [out]       error_status_t          *st
);
```

The `reply` field must be chosen from the following list:

```
cso_c_status_ready
cso_c_status_initializing
cso_c_status_busy
cso_c_status_shutdown
```

Several values will also be reserved for licensee use.

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.

These routines will be specified in a companion document at a later time. We expect them to follow the ''bind-act-free binding'' model common in the DCE API. For example, the bind routine may look like this:

```
void
dce_host_bind(
    [in]        dced_string_t           hostname,
    [in]        dced_binding_type_t     type,
    [in]        void                    *residual,
    [out]       handle_t                *h,
    [out]       error_status_t          *st
);
```

The `type` and `residual` parameters are used to specify which entity managed by `dced` on the specified host is desired. For example, in the following the first call could be used to get a handle to a running printer server (perhaps in order to shut it down), while the second could be used prior to changing the name of the cell:

```
dce_host_bind("sulphur", dced_servrexec, "printer", &h, &st);

dce_host_bind(hostname, dced_hostdata, &cellnameuuid, &h, &st);
```

The user interface, built on top of the programmer's API, will be defined in the same companion document. The commands will be part of `dcecp`. We expect that each `dced` service will define one or two objects, and that each object will have a couple of verbs. For example, the key management commands will probably be able to add, delete, and change keys for a specific server. This could be done using the following syntax (shown on two lines for presentation):

```
keytab verb [server_name | server_uuid] [-host hostname]
    [argument...]
```

For example:

```
keytab add "dts" -host $h "admin" $newkey
```

A server must 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.

9.1. 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.

Assuming the routines described in this section exist, then a session manager could make calls like the following to start up services:

```
dce_host_bind(hostname, dced_server, "calendar", &h, &st);
dce_server_start(h, command_line, environ, &st);
dce_host_bind_free(&h, &st);
```

The `argv` and `environ` parameters are the conventional C NULL-terminated arrays of command line parameters and environment variables. 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 $*
```

10. RESTRICTIONS AND LIMITATIONS

None known.

11. OTHER COMPONENT DEPENDENCIES

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

The `hostdata` service depends on the cell alias design being done as part of the Hierarchical Cells work [RFC 7].

The client interface will be provided as code to be integrated into `dcecp` and is therefore dependent on that component.

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

(a) If pre-authentication [RFC 26] is going to be implemented, then the `secval` service might have to be changed.

(b) 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.

(c) Will client RPC runtime need changes to support autostart? The time required to spawn a server and have it register with `dced` may cause problems with runtime timeouts.

(d) We have dropped the concept of ''one shot'' servers, in favor of keeping the simpler one-to-one mapping of `srvrconf` objects to `srvrexec` objects and to avoid requiring changes to the client RPC runtime which does not rebind a handle for each operation. A ''one shot'' is a server that is instantiated to handle a single RPC request.

(e) There is currently no way to garbage-collect key tables. We have received a request that `dced` periodically do this for all tables that it knows about.

(f) 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.

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

(h) We assume that if `dced` is managing a local file such as a key table, that 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.

REFERENCES

[RFC 3] J. Pato, DCE-RFC 3.0, ''Extending the DCE Authorization Model to Support Practical Delegation (Extended Summary)'', June, 1992.

[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. Romagana, R. Mackey, DCE-RFC 41.1, ''RPC Runtime Support for I18N Characters Functional Specification'', September, 1993.

[RFC 42] H. Melman, DCE-RFC 42.0 ''DCE Shell Functional Specification'', June, 1993.

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

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

AUTHOR'S ADDRESS

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