OSF DCE SIG
Request For Comments: 45.0

D. Mackey (OSF)
R. Salz (OSF)
July 1993

DCE BACKING STORE LIBRARY — FUNCTIONAL SPECIFICATION

1. INTRODUCTION

Most DCE servers have some data that must persist across program invocations. DCE currently provides no facility to help with this. As a result, each DCE server has its own code to serialize its internal data structures and write them to disk, as well as a similar amount of code to do the inverse set of operations. Application developers must also develop this technology in order to build servers that deal with persistent data.

The DCE daemon (`dced`) [RFC 47] is a replacement for `rpcd` that will be part of DCE 1.1. Since `dced` needs persistent data (both for its own data and for its ACL's), we will have to solve the same problem. Rather than reinvent the wheel yet again, we would like to write a general package that can be used in any server. The IDL Encoding Services ([RFC 2], [IDL-ES]), or ''pickling,'' that is part of the DCE 1.0.3 IDL compiler makes this feasible.

2. TERMINOLOGY

The term ''database'' in this document refers to the backing store library only for convenience. The library does not support SQL or other query capabilities associated with databases, but using the term makes many sentences less awkward. The API uses the abbreviation `db` rather than `bs` for more obvious reasons.

The ''backend'' is that part of the library that handles disk access.

3. TARGET

This document describes a convenience library for programmers writing DCE servers.

4. GOALS AND NON-GOALS

The goals of the backing store library are:

(a) Provide tagged retrieval of typed data. The tag will be either a UUID or a standard C string. The data will be a run-time specified datatype using IDL Encoding Services.

(b) A program must be able to have more than one database open at the same time.

The following are not goals:

(a) General (SQL-like) access to data will not be provided.

(b) It will not be possible to use an arbitrary array of bytes as a retrieval key.

(c) Storing different data types in a single database will not be supported.

(d) Automatic remote retrieval by name will not be supported — this is not RND ([RFC 18]) or any other directory service.

(e) There will be no attempt to synchronize across multiple writers to the same database.

5. REQUIREMENTS

Except for the backend, the entire library should be implemented in under 1,000 lines of code.

The library must be structured so that the backend can be implemented as either a disk-based system, or one that assumes the entire database is loaded into memory while it is used.

The interface to the backend must be written so that it is feasible to replace it with a high-performance commercial database.

We will provide a default backend.

6. FUNCTIONAL DEFINITION

The backing store library provides simple access to persistent storage. There is a unique key that is used to identify data. The key can be either a DCE UUID or a C string (which should be limited to the DCE `idl_char` PCS (Portable Code Set) codeset if it will be used in a remote interface). The key type is specified when the database is created.

There are routines available to create a new database and access an existing one, in either read-only or update mode. Both operations return a handle to be used in subsequent calls. A routine is available that will free the handle, closing any open files and releasing all other resources associated with the store.

Once a database has been opened, data can be retrieved or stored. A function specified at open time will be called to convert between native format and on-disk (serialized) format.

A set of iterator routines allows an application to step through all the keys in the database. The order in which the keys are retrieved is indeterminate, although no key will be returned more than once. No writes must be performed while the iterator is in use.

While the library will be thread-safe, there is limited concurrency support. There are routines available to lock and unlock a database, and to query if it is already locked. Only write operations require the lock. A database can be opened multiple times but the library will not synchronize across multiple writers.

7. DATA STRUCTURES

A ‘‘standard prolog’’ datatype will be available. If the prolog is used then the library will automatically maintain some of the fields. Other standard fields will help encourage standardized server development and use models.

```
typedef struct dce_db_dataheader_s_t {
        uuid_t          creator;
        uuid_t          uuid;
        char            *name;
        uuid_t          acl_uuid;
        union switch (boolean32 is_container) {
        case false:
            ;
        case true:
            uuid_t          iacl_uuid;
            uuid_t          cont_iacl_uuid;
        }
        unsigned32      ref_count;
        /* The following fields are updated by the library */
        utc_t           created;
        utc_t           modified;
        unsigned32      modified_count;
} dce_db_dataheader_t;
```

```
typedef enum dce_db_header_type_e_t {
       dce_db_header_std
} dce_db_header_type_t;

typedef
union switch (dce_db_header_type type) dce_db_header_u_t {
    case dce_db_header_std:
        dce_db_dataheader_t h;
} dce_db_header_t;
```

The `acl` and `iacl` fields are intended to be used as indices into the server's ACL database. No semantics are attached to these fields, but we do intend to promote some standard use of them.

There will be routines available to initialize a header.

8. USER INTERFACES

We want to encourage standardized use of the backing store library. We recommend that the IDL interface for a server be written in the following way:

```
interface XXX
{
        /* other imports */
        import "dce/db"

        /* other datatypes */
        typedef XXX_data_s_t {
            dce_db_header_t *header;
            /* server-specific data */
        } XXX_data_t;

        /* other operation definitions */
        void XXX_data_convert(
            [in] handle_t h,
            [in, out] XXX_data_t *data
        );
}
```

This should be compiled with the following ACF:

```
interface XXX
{
        /* other operation attributes */
        [encode, decode] XXX_data_convert();
}
```

9. API'S

The library routines are divided into three parts:

(a) Open/close.

(b) Fetch/store.

(c) Miscellaneous.

The following sub-sections explain each part in turn.

9.1. Open/Close Routines

The following routine opens an existing database or creates a new one:

```
void dce_db_open(
    const char          *name,
    const char          *backend_type,
    unsigned32          flags,
    dce_db_convert_funct_t convert,
    dce_db_handle_t     *handle,
    error_status_t      *st
);
```

The name parameter is the filename of the database. Depending on the implementation of the backend, the library may actually create multiple files with new extensions.

The backend_type parameter specifies the database backend type. This is for licensees who add multiple backends. For the OSF-supplied implementation this parameter is ignored.

The flags parameter is a combination of the following bits:

```
dce_db_c_std_header
dce_db_c_index_by_name
dce_db_c_index_by_uuid
dce_db_c_readonly
dce_db_c_create
```

The first flag indicates that the data being stored has the standard database header as its first field. The next two flags two are mutually exclusive and specify the index type. It is not possible to change the index type at run time; a new database must be built. Choosing the index type is the responsibility of the application developer. If all a server's objects have entries in the CDS namespace then it is probably best to use a UUID index. If the server provides a junction or other name-based lookup operation then it is probably best to use a name index.

An existing database may be opened in read-only mode; the default is read-write mode. The last flag, if specified, creates an empty database if it does not already exist; it is an error to use this flag on an existing database. It is an error to open an existing database using an index type other then the one with which it was created.

The convert parameter specifies the function that should be called to do the serialization. This will normally be a function generated by the IDL Encoding Services.[1] It must follow a standard format. Using the IDL example above, a typical call to dce_db_open would be like this:

```
dce_db_open("XXX_db", NULL,
        dce_db_c_std_header | dce_db_c_index_by_uuid,
        (dce_db_convert_func_t)XXX_data_convert,
        &handle, &st);
```

To close a database and release all resources associated with it, the following routine must be used:

```
void dce_db_close(
    dce_db_handle_t     *handle,
    error_status_t      *st
);
```

9.2. Fetch/Store Routines

Once a database has been identified, data may be retrieved by calling one of the following routines:

1. Note that automatic codeset conversion is not available when using IDL Encoding Services.

```
void dce_db_fetch(
    dce_db_handle_t     handle,
    void                *key,
    void                **data,
    error_status_t      *st
);

void dce_db_fetch_by_name(
    dce_db_handle_t     handle,
    char                *key,
    void                **data,
    error_status_t      *st
);

void dce_db_fetch_by_uuid(
    dce_db_handle_t     handle,
    uuid_t              *key,
    void                **data,
    error_status_t      *st
);
```

The first routine is a general retrieval routine, and interprets the `key` according to the type of index with which the database was created. The other two are type-safe wrappers that will not work with a database of the wrong index type.

Note that the `data` parameter is a pointer to an arbitrary pointer. In actual use, this will be the address of the database-specific datatype.

The following routines are available to store data in the database:

```
void dce_db_store(
    dce_db_handle_t     handle,
    void                *key,
    void                *data,
    error_status_t      *st
);

void dce_db_store_by_name(
    dce_db_handle_t     handle,
    char                *key,
    void                *data,
    error_status_t      *st
);

void dce_db_store_by_uuid(
    dce_db_handle_t     handle,
    uuid_t              *key,
    void                *data,
    error_status_t      *st
);
```

Again, the first routine is for run-time dispatching while the other two provide more type-safety.

Once an application is done with the data it should be released by using the following routine:

```
void dce_db_free(
    dce_db_handle_t     handle,
    void                *data,
    error_status_t      *st
);
```

9.3. Miscellaneous Routines

Each database handle has a lock associated with it. The ‘‘store’’ routines acquire the lock before doing any updates. This lock may also be acquired by application code:

```
void dce_db_lock(
    dce_db_handle_t     handle,
    error_status_t      *st
);

void dce_db_unlock(
    dce_db_handle_t     handle,
    error_status_t      *st
);
```

The following routines are used to iterate over all keys in a database.

```
void dce_db_iter_start(
    dce_db_handle_t     handle,
    error_status_t      *st
);
```

This routine sets up a database handle to begin retrieving all keys. Right now only one iterator can be in use at any time; if more than one iterator is needed simultaneously, then the database must be opened multiple times.

Once the iterator has been set up, the following routine can be called (for the sake of brevity, the ‘‘_by_name’’ and ‘‘_by_uuid’’ analogs are not shown):

```
void dce_db_iter_next(
    dce_db_handle_t     handle,
    void                **key,
    error_status_t      *st
);
```

This routine returns a pointer to private space associated with the handle. The space will be re-used on the next call to the iterator. We do not return a pointer to allocated space which would have to be free'd. We expect that most uses of the key will be to immediately do a database fetch, and that the keys themselves are rarely of interest.

To free up the state associated with an iterator the following routine must be called:

```
void dce_db_iter_done(
    dce_db_handle_t     handle,
    error_status_t      *st
);
```

Note that most uses of the iterator will probably want to be wrapped inside calls to lock the database so that incoming writes do not upset any internal state of the iterator. For example:

```
dce_db_lock(h, &st);
dce_db_iter_start(h, &st);
for ( ; ; ) {
        dce_db_iter_next(h, &key, &st);
        if (st == dce_db_c_no_more)
                break;
        dce_db_fetch(h, key, &data);
        process(data);
}
dce_db_iter_done(h, &st);
dce_db_unlock(h, &st);
```

10. REMOTE INTERFACES

There are no remote interfaces.

A simple remote interface could be provided that gave ‘‘raw’’ access to the database. This could be useful for building a general replication facility. We will not do this for DCE 1.1.

11. MANAGEMENT INTERFACES

There is no management interface.

12. RESTRICTIONS AND LIMITATIONS

Documented in other sections of this document.

13. OTHER COMPONENT DEPENDENCIES

The backing store library will require the IDL Encoding Services. This dependency has already been met with the release of DCE 1.0.3. The library also requires ‘‘reasonable’’ performance from the Encoding Services. We do not know if this will be a problem, or how it can be addressed if so.

We do not want to write the database backend. We have, instead, identified three possibilities:

(a) The ‘‘dsm’’ facility currently used by `rpcd`. The primary drawback of this package is that it does not provide any tagged retrieval.

(b) The ‘‘balanced_tree’’ facility used by the `secd`. A possible drawback of this package is that it keeps the entire database in memory.

(c) The Berkeley 4.4 ‘‘db’’ facility. The primary drawback of this package is that it is not known to be thread-safe, and that earlier versions have had stability problems. The developers are interested in working with OSF.

14. COMPATIBILITY

This is a new facility; compatibility with previous releases is not applicable.

15. STANDARDS

There are no applicable standards.

16. OPEN ISSUES

The backend must be chosen.

REFERENCES

[IDL-ES] T. Hinxman, ‘‘Design Note RPCLang 016: IDL Encoding Services,’’ June, 1993.

[RFC 2] J. Harrow, DCE-RFC 2.1, ‘‘Proposed Enhancements for DCE 1.1 IDL’’, July, 1992.

[RFC 47] D. Mackey, R. Salz, DCE-RFC 47.0, ‘‘DCE Daemon Functional Specification’’, to appear.

[RFC 18] N. Mishkin, DCE-RFC 18.0, ‘‘The Remote Network Directory (RND) Naming Model’’, September 1992.

AUTHOR'S ADDRESSES

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