N. Mishkin (HP)
September 1992

THE REMOTE NETWORK DIRECTORY (RND) NAMING MODEL

1. INTRODUCTION AND FUNCTIONAL OVERVIEW

The Remote Network Directory (RND) naming model meets a fundamental need in distributed systems: a way for users to identify, in human meaningful terms, all the entities in the computing system. It provides a set of interfaces for a directory/naming service in a worldwide distributed computing environment. The RPC interfaces to the name service, expressed in DCE RPC IDL, make it well integrated with the rest of the DCE offering.

The features of the RND directory/name service include the following.

(a) *Single name space.*

All users of an object use the same name for an object, regardless of the location of the object or the user of the name. This eases the sharing of information. It also provides a solid basis from which shorter, context-sensitive names can be implemented.

(b) *Global name space.*

The architecture supports a naming service that must scale to cover worldwide systems. There are no central bottlenecks, and the design allows for massive caching.

(c) *Reliable name space.*

The naming service must be at least as reliable as the objects it names, because objects usually need to be named to be accessed. Reliability is provided via replicated directory servers.

(d) *Secure name space.*

We provide access control and authenticated operations, using the same access control interface as the DCE distributed file system and user account administration system.

(e) *Universal name space.*

We believe that the name space should be able to encompass all objects, including files, services, users, groups, processes, etc. This allows all objects to be named in a uniform way, so that generic tools can be built to operate on objects. Just one example of the power of this concept would be a browser that, instead of just being able to browse the directory service, could browse the directory service, file system, users, etc.

Unfortunately, the implementation characteristics of most core name services (weakly consistent replication, the assumption of a slowly changing name space) do not make them suitable for naming all objects in the DCE. By providing for composition and extension of the name service we permit implementation of a universal name space without needing a single implementation for all requirements.

(f) *Composite naming.*

By this we mean that the core name service is designed to permit various other naming services to be composed with it to get the flexibility to name all DCE entities.

(g) *Extensible naming.*

The separation of name service functionality into orthogonal network interfaces at various levels of generality permits easy extension to new implementations of directory servers. We believe that there will be at least three kinds of name service implementations. Some will use weakly consistent replication, such as DCE CDS. Others will require strongly consistent replication to maintain the integrity of their data. Yet others will need to be frequently updated with changing status information, but not need replication and/or crash survival (OS process names, for example).

(h) *X.500.*

The RND naming architecture provides a high degree of X.500 functional equivalence, plus valuable extensions not provided by X.500. It is designed to allow a DCE cell to act as a directory management domain (DMD), which can: export X.500 DAP and DSP via a DSA; import X.500 DAP via a DUA. Thus, to X.500 clients outside the cell, a cell's name space appears to be an integral part of the X.500 name space, and clients inside the cell see the X.500 name space as an extension of the cell name space. This is possible because the RND architecture supports all X.500 abstract services, including search. This support will also allow access to the name space via the X/OPEN Directory Services interface, which is currently being defined. Valuable extensions to X.500 that we provide with RND naming include: replication, authorization integrated with the rest of DCE, and the tools to provide universal naming.

As can be seen from the above list, RND encompasses most of the features currently being deliberated in the technical community regarding directory/naming service, and going under various titles such as *''junction''* or *''federated''* naming architectures.

2. TECHNICAL INFORMATION

This document describes the Remote Network Directory (RND) model for name, attribute and location services. The concrete RND interfaces mentioned here are described in detail in the Appendices.

2.1. RND Directories

RND directories are simply symbol tables that map names to attribute sets. Each (name, attribute set) pair is called a *directory entry*. For example, a `memos` directory might contain the following entries:[1]

Symbol	*Value*
`naming.doc`	`object-UUID=333b2e690000.0d.00.00.87.84.00.00.00`
`locating.doc`	`object-UUID=44658ae6a000.0d.00.01.d6.d5.00.00.00`
`drafts`	`object-UUID=44658b12c000.0d.00.01.d6.d5.00.00.00`

RND directories require the presence of the *object-UUID* canonical attribute in each directory entry. If the object-UUID associated with the name `drafts` refers to a directory (directories, like all DCE objects, are identified by UUID), then `drafts` is said to be a *subdirectory* of `memos`. Otherwise, the name is said to be a *leaf*.

Directories are permitted to contain entries that refer to other directories. *Pathnames* made of a sequence of *pathname components* separated by a delimiter describe a path through a set of directories, where each non-terminal node in the tree is a directory that points to some other directory. Resolving a pathname consists simply of starting with the UUID of the *root* of the pathnaming tree (this UUID must be known a priori) and iteratively looking up pathname components in succeeding directories, identified by the object-UUID associated with the component in the directory. When the resolution process runs out of component names, the resolution is complete and the last directory entry found is the result of the resolution.

More needs to be said about what we mean by directory. We have already said that directories are simply symbol tables. More properly, we should say directories are objects whose type[2] exports a Directory interface. Any type of object that exports a Directory interface is a directory.[3]

The RND Directory interface contains operations to manipulate directory entries. It also contains an operation called *lookup* that takes a component name and returns the associated object-UUID. The operations in the Directory interface are operations on objects whose type exports the Directory interface.

---

1. It will be noted that the RPC syntaxes appearing in this paper (e.g., the syntax for UUIDs, and in the IDL files) have not yet been updated to the conform to OSF DCE RPC. That will be remedied in a subsequent version of this document.

2. By *type* we mean *implementation* (or what C++ users might call *implementation class*).

3. In general, a system that has a set of well-crafted interfaces has the property that if *I* is the name of some interface (or at least some interface that is expected to have more than one implementation), the statement ''... is an *I* ...'' makes sense.

[Note that when we say ‘‘interface’’, we mean a DCE RPC IDL interface that one or more DCE RPC servers make available to DCE RPC clients. When we say ‘‘invoke an operation on an object’’, we’re referring to making a DCE RPC remote procedure call where the call’s binding (parameter) contains the UUID of the object. We’re leaving vague for the moment the question of how the location of the object (another part of the RPC binding) is found. One scheme for doing so involves a service for mapping object-UUIDs into network addressing information. Another scheme involves making all object references actually be pairs consisting of both the object’s object-UUID and its location; in this scheme, object-UUIDs would never travel alone — they’d always be paired with location information.]

What should be clear from this is that there might be many types of objects that can be called directories. Each type might have a completely different implementation. The way directory entries are represented can vary widely across types. Some types might replicate the directory, some types might not. Some types might promise that changes to the directory are handled in such a way that even if lightning strikes the home machine of the object, recent changes to the state of the object (i.e., the addition and deletion of directory entries) will not be lost. A user registration type might choose to export the Directory interface arranging that a `lookup` with a valid user name as the key (pathname component) yields the UUID of the user’s home directory in a distributed file system. Clients of this interface are in general unaware of the type of the directory they are making a request to.

Note that all these types are ‘‘directories’’ in the pathname resolution process. Unlike traditional file system directories, there is not just one kind of directory. In fact, the number of kinds is not bounded. The directory concept in our scheme is thus *extensible*. Anyone can create a new thing and have it treated by existing software just like every other directory. By adopting UUIDs as the ‘‘coin of the realm’’ (the standard way of identifying everything in the environment), we have enabled extensibility in naming. If the system had allowed multiple primitive forms of naming objects (i.e., UUIDs and other things), then it would not be possible to have a single directory interface since the type of the ‘‘value’’ fields of the directory entry could not be determined statically.

2.2. RND Interfaces: Directory, Directory Attribute, Name List, Link

RND defines four distinct interfaces related to the directory abstraction: Directory, Directory Attribute, Name List, and Link.

A completely general RND directory is a map from component names to one of two kinds of directory entries: attribute set or linktext. An attribute set is a collection of attributes, consisting of an attribute type (a UUID) and an associated attribute value. In a linktext entry, the value of the entry is an arbitrary string of text. Linktext entries support a UNIX-like ‘‘symbolic link’’ feature.

The RND Directory interface provides operations for manipulating and retrieving two particular attributes: object-UUID and type-UUID. Object-UUIDs are often references to directories indicated by the components of pathnames. Among other things, the type-UUID is convenient for name resolution.

Generic attribute sets are accessible through the Directory Attribute interface. As opposed to the Directory interface, which can access only the object- and type-UUIDs, the Directory Attribute interface can access all the attributes of a directory entry.

Name lists are objects (e.g., mailing lists) that simply consist of lists of names. More precisely, the Nlist interface deals only with the name component of an enumeration of entries keyed by name. The Nlist interface provides operations for adding, dropping and changing names in lists and enumerating a list.

The Link interface provides operations to add, change, and lookup linktext directory entries.

The reason the four interfaces described above are not all described as one interface is that it allows for the possibility of types of objects that handle only a subset of the RND directory abstraction. Programs should use the most generic interface possible to do their job. A program that manipulates just the names in a directory should use the Nlist interface; a program that manipulates the object- and type-UUIDs should use the Directory interface; a program that needs access to arbitrary attributes should use the Directory Attribute interface. By adopting this approach, programs maximize the number of different types of objects they can successfully deal with.

2.3. RND Interfaces: Path, Child

The Path interface includes a `resolve` operation to resolve a pathname to an object-UUID and location (i.e., network address) and a `construct` operation to constuct a pathname from an object-UUID. While pathname resolution can be performed in the stepwise fashion described earlier, use of the `resolve` operation makes the process more efficient (in terms of the number of remote calls that need to be performed), since `resolve` takes a pathname (rather than a component of a pathname) as input and resolves as much of the pathname as it can.

The Child interface is used for objects (e.g., directories and cataloged files) that need to (or simply can) maintain a pointer to a parent object. By repeated application of the `get_parent` operation in the Child interface, the Path `construct` operation reconstructs the full pathname of an object.

2.4. RND Interfaces: Attribute

Objects that export the Attribute interface are *attribute databases* (in the same sense that objects that export the RND Directory interface *are* directories). The attribute database can be thought of as a set of tuples: (object-UUID, attribute set), where an ‘‘attribute set’’ is a list of tuples of the form (attribute UUID, attribute value).

The Attribute interface can inquire about and modify the attribute set associated with a particular object-UUID. The interface provides an operation `attrib_lookup_matching_attrs` which returns an array of object-UUIDs that match a set of attributes. This provides the ability to determine a canonical identifier for an object based on that object's attributes.

The Attribute interface provides a way to identify objects based on attributes instead of canonical identifiers such as UUIDs or textual names. Attribute-based naming can help programs identify objects when they lack knowledge of the canonical identifiers. Also, by using the Attribute interface, a program can efficiently narrow a set of choices based on requirements expressed as attributes. Instead of querying each individual object to determine whether it has suitable properties for a specific application, a client can make one query to an attribute service.

2.5. Composite Pathnames

The RND naming model supports a hierarchical name space where the non-leaf nodes of the name space are objects that handle the RND Directory interface; we call such objects *RND directories* and the space of such interconnected directories an *RND namespace*. Many traditional filesystems (e.g., the UNIX filesystem) also support a hierarchy of directories; we call components of this hierarchy *filesystem directories* and the space of such interconnected directories a *filesystem namespace*.

There is yet another class of directories: those that catalog organizations agreeing to cooperate with a ‘‘global naming authority’’. For example, the ARPA Internet community's Domain Name System (DNS) defines a tree of ‘‘domains’’ (which are essentially directories) for naming hosts within the Internet. ISO's X.500 defines similar services (among other things). An X.500-based system will eventually be available to answer queries about the ‘‘upper levels’’ of the name space (i.e., the part that defines countries and other entities that span organizations). We call the objects managed by systems like DNS and X.500 *global directories* and the space of such interconnected directories the *global namespace*.

2.5.1. Connecting global and RND namespaces

The three namespaces can be connected so that users can deal in names which are the composition of names from each space. The result of the composition is a *composite pathname* for objects. First consider a pathname for a non-filesystem object. Suppose the name:

```
/.../c=uk/o=icl/printers/r&d/compiler
```

names the compiler group's printer within the R&D organization of the ICL corporation, which is located in the United Kingdom. (Ignore the `/...` for the moment.) The constituents of the pathname are:

Global: `c=uk/o=icl`
RND: `printers/r&d/compiler`

In general, the global part of the pathname should be thought of as naming an RND directory that is the root of an RND namespace. The first step in resolving a composite pathname is to resolve the global part, using the protocols defined by the administrators of the global namespace, into a UUID and one or more locations (network addresses). The UUID identifies a root RND directory (the root of an RND namespace). The locations are locations of servers that can operate on the root RND directory. (We can live with a global namespace that can provide only locations and not UUIDs as long as the only directory handled by the located server is the cell root.) The second step in name resolution is simply the standard RND resolution mechanism: repeatedly apply lookup operations (dir_lookup or path_resolve) in the RND Directory interface to walk down the rest of the pathname.

It is undesirable that every client wanting to resolve a name be required to have the machinery to resolve the global part of a pathname. Not only is the machinery burdensome in terms of space, but it may have to change over time (e.g., as X.500 becomes enough of a reality to replace the current de facto global namespace (DNS)). The way to handle this issue is to ''pretend'' that the root of the global namespace is an RND directory (one with some canned UUID) and to deploy servers within a cell that offer to handle operations on this directory. Clearly a single-component-name RND directory lookup (dir_lookup) on this directory cannot work since the root of the global namespace, and presumably even directories several levels down from the root, don't hold UUIDs. However, clients will not (should not) in general be doing component-at-a-time lookups. Rather they will be using the multi-component lookup (path_resolve). Thus, when presented with a pathname that contains the whole global part (and optionally more) of a composite pathname, the RND server for the global root can, transparently to the client, use the global namespace protocols to resolve the global pathname.

2.5.2. Connecting a filesystem namespace

Connecting a filesystem namespace into the composite pathnaming model means that individual files can be named in a form like:

```
/.../c=us/o=hp/ou=apollo/fs/rpc_project/memos/naming.txt
```

The constituents of this pathname are:

Global:	c=us/o=hp/ou=apollo
RND:	fs
Filesystem:	rpc_project/memos/naming.txt

In this case, the fs entry in the RND directory named c=us/o=hp/ou=apollo identifies a filesystem which contains the filesystem directories rpc_project and rpc_project/memos and the file named .../naming.txt. Resolution of the global and RND parts of the name proceed as above.

Understanding the resolution of the filesystem part of the pathname requires understanding how filesystems typically deal with names. In traditional local filesystems some component of the system, typically in the OS, is passed a text string pathname and resolves it to some internal identifier of the named file. In UNIX, this internal identifier is an inode number or a vnode reference, or the like. Actual internal operations on the file (read, write, etc.) take this internal identifier as a parameter. Distributed filesystems (e.g., AFS, Apollo DOMAIN/OS, NFS) support distribution of files over multiple machines in a network. As a result, these systems have developed a distributed namespace, but one that names only files, not objects in general (e.g., printers, users).[4]

The use of composite pathnames to name files must affect the filesystem resolution mechanism. The filesystem name resolver must be able to cope with composite pathnames to the extent that it must be able to resolve the non-filesystem part of the pathname into the internal identifier needed to do its real work. Thus, the integration of these namespaces is a function of the OS and filesystem being modified and not in general an architectural issue.

2.5.3. An implementation model for UNIX filesystems vnodes

Many UNIX implementations support the ''virtual filesystem'' (VFS) model. In this model, the UNIX kernel contains support for multiple virtual filesystem types (e.g., local BSD disk filesystem, NFS, AFS). A filesystem is accessed using a generic VFS-defined interface through which filesystem objects are identified by *vnodes*. The interface has multiple implementations, one for each filesystem type. Pathname resolution is implemented as successive calls to the vn_lookup operation in this interface; each call is supplied with one component of the pathname and returns a vnode.

The natural way to support composite pathnames is to add a new filesystem type; call this filesystem type ''GNS''. The implementation of the vn_lookup operation for GNS-type filesystems works roughly as follows. Consider

4. The fact that the filesystem uses the network to resolve filesystem names doesn't bear on this discussion. The distributed filesystem can simply be treated like a local filesystem that happens to go to the network instead of a local disk to do its work.

the /.../c=us/o=hp/ou=apollo/fs/rpc_project/memos/naming.txt pathname above. /... is the ‘‘mount point’’ of a GNS-type filesystem. vn_lookup is passed c=us which it in turn passes (via RPC) to the path_resolve operation at the server that handles the pseudo-RND directory for the root of the global namespace. path_resolve returns an error saying, essentially, ‘‘This isn’t enough of a pathname.’’ The vn_lookup operation reacts to this error by constructing a vnode whose private data consists of the string /o=us.

Processing continues as above on the successive components of the global pathname (the string in the vnode private data having more components appended to it). After having processed c=us/o=hp/ou=apollo the returned vnode contains (in the private part) the UUID and locations of a cell root RND directory. Processing of successive components yields additional, similar vnodes. At each step of the resolution when vnodes with UUIDs are being constructed, the vn_lookup implementation must check to see if the object identified by the UUID is actually a filesystem of the type supported by the kernel. This can be determined by using the type-UUID attribute associated with the directory entry. If the object is discovered to be a filesystem directory (i.e., if the type-UUID is one of the set of type-UUIDs known to the kernel) the vn_lookup implementation returns a vnode that looks like a mount point of a filesystem of the appropriate type.

The GNS filesystem type supports a subset of the VFS interface. In particular, while it might not be able to support operations that modify the filesystem (e.g., creating and renaming files and directories), it can support vn_readdir, the operation that enumerates the contents of a directory. And since a process’s current directory is simply a vnode referenced from the process state information, the current directory can be any point in the global or RND namespace (as well the filesystem namespace).

Note that the above scenario is in some sense the ‘‘worst case’’ scenario for integrating the filesystem namespace. A filesystem that directly supports the RND Directory interface on filesystem directories has a less complicated resolver.

3. ACKNOWLEDGMENTS

Much of the initial work on the RND architecture was done by HP/Apollo (Nathaniel Mishkin, Paul Leach, Elizabeth Martin, Stephanie Bodoff, Joe Pato). Subsequently, representatives of DEC (Danny Cobb, Mark Fox), OSF (Walt Tuvell), and IBM (Scott Snyder) have contributed to RND.

APPENDIX A. THE ORIGINAL RND INTERFACES

There are two versions of the RND interfaces. Although the second version is derived from ideas in the first version, the two versions have not yet been reconciled. The first version was written by HP and most closely represents the ideas described in this document; these interfaces were used in a prototype name server built by HP. The second version incorporates some of the ideas that were learned during the early days of the DCE project; DEC produced this version and was influenced by their experiences with DECdns (the DCE CDS).

A subsequent version of this document will include a single set of interfaces.

A.1. nameb.idl

```
/* Basic datatypes for the name interfaces */

interface nameb {

import 'nbase.idl';

/* Name constants */

const int name_long_complen_max = 255;      /* Max lnth of a single
                                               long component */
const int name_long_complen_size = 256;     /* Max lnth of a single
                                               long component with
                                               null terminator */
const int name_long_pnamlen_max = 1023;     /* Max lnth of whole
                                               long pathname */
const int name_long_pnamlen_size = 1024;    /* Max lnth of whole
                                               long pathname with
                                               null terminator */

typedef string0[name_complen_size]  dir_name_t;     /* component
                                                       name */
typedef string0[name_pnamlen_size]  path_name_t;    /* path name */

/* Common Data Types */

typedef void    *object_handle_t;

typedef struct {
    unsigned32      slen;
    socket_$addr_t  saddr;
}   sock_addr_t;

typedef struct {
    unsigned32                      num_addr;
    [length_is(num_addr)] sock_addr_t saddrs[];
} sock_addr_list_t;

/* Cursor for read/lookup iteration

To read/lookup starting at the beginning of a list set cursor_t.state
to cursor_start.  When all entries have been returned cursor_t.state
will equal cursor_end.

nlist_cursor_start in nlist.idl sets the cursor to point to the
beginning of a list.  nlist_cursor_end in nlist.idl checks whether
```

```
the end of the list has been reached.
*/

typedef enum { cursor_start, cursor_continue, cursor_end }
    cursor_state_t;

const unsigned32 cursor_size = 32;

typedef struct {
    cursor_state_t state;
    byte           x[cursor_size];   /* implementation specific,
                                        opaque cursor */
} cursor_t;

}
```

A.2. dirb.idl

```
/* Basic datatypes for the directory interfaces */

interface dirb {

import 'nameb.idl';

/* directory entry */
typedef struct {
    [string0] char              *name;
    uuid_t                      object;
    uuid_t                      type;
}   dir_entry_t;

}
```

A.3. nlistr.idl

```
[
uuid(468afc968000.0d.00.01.83.dc.00.00.00),
version(1)
]
interface nlist_r {

import 'nameb.idl';

/* Nlist Interface (remote)
The Nlist interface provides operations for adding, dropping and
changing names in lists and enumerating a list.
*/

/*
 * N L I S T _ R _ M U T A B L E
 *
 * return true if the manager which is exporting this interface
 * supports the update operations in this interface.
 * return false otherwise
 */

boolean nlist_r_mutable(
    [in] handle_t       h,
    [out] status_t      *status
);

/*
 * N L I S T _ R _ A D D
 *
 * add a name to a list
 */

void nlist_r_add(
    [in] handle_t       h,
    [in, string0] char  *name,   /* name to add to list */
    [out] status_t      *status
);

/*
 * N L I S T _ R _ C H A N G E _ N A M E
 *
 * change the name in a list from old_name to new_name
 */

void nlist_r_change_name(
    [in] handle_t       h,
    [in, string0] char  *old_name,
    [in, string0] char  *new_name,
    [out] status_t      *status
);

/*
 * N L I S T _ R _ D R O P
```

```
 *
 * drop a name from a list
 */

void nlist_r_drop(
    [in] handle_t       h,
    [in, string0] char  *name,   /* name to drop from list */
    [out] status_t      *status
);

/*
 * N L I S T _ R _ L O O K U P
 *
 * return true if a name is in a list,
 * return false if a name is not in a list
 */

[idempotent] boolean nlist_r_lookup(
    [in] handle_t       h,
    [in, string0] char  *name,   /* name to lookup */
    [out] status_t      *status
);

/*
 * N L I S T _ R _ R E A D
 *
 * return a list of names
 */

[ idempotent ] void nlist_r_read(
    [in] handle_t       h,
    [in, out] cursor_t  *list_cursor,   /* cursor describing
                                           where to begin or
                                           continue reading
                                           the nlist */
    [in] unsigned32     max_num_names,  /* maximum number
                                           of names
                                           to return */
    [out] unsigned32    *num_names,     /* number of names
                                           returned */
    [out, length_is(num_names), size_is(max_num_names), string0] char
                        *names[],       /* list of names */
    [out] status_t      *status
);

}
```

A.4. dirr.idl

```
[
uuid(4649967e1000.0d.00.01.99.ba.00.00.00),
version(1)
]
interface dir_r {

import 'dirb.idl';

/* Directory Interface (remote)

The Directory interface specifies operations to add and replace
directory entries.  The directory is identified by a handle (h).
Entries are identified by a name and consist of a name, object-UUID
and type-UUID.  This interface also supports lookup by name or by
object-UUID and list the contents of a directory.
*/

/*
 * D I R _ R _ M U T A B L E
 *
 * return true if the manager which is exporting this interface
 * supports the update operations in this interface.
 * return false otherwise
 */

boolean dir_r_mutable(
    [in] handle_t       h,
    [out] status_t      *status
);

/*
 * D I R _ R _ A D D
 *
 * add a name and its associated object-UUID and type-UUID to a
 * directory
 */

void dir_r_add(
    [in] handle_t       h,
    [in, string0] char  *name,    /* name to add to directory */
    [in] uuid_t         *object,  /* object-UUID */
    [in] uuid_t         *type,    /* type-UUID of object */
    [out] status_t      *status
);

/*
 * D I R _ R _ R E P L A C E
 *
 * replace the object-UUID and type-UUID associated with
 * a name in a directory
 */

void dir_r_replace(
```

```
    [in] handle_t       h,
    [in, string0] char  *name,    /* name to replace */
    [in] uuid_t         *object,  /* object-UUID */
    [in] uuid_t         *type,    /* type-UUID of object */
    [out] status_t      *status
);

/*
 * D I R _ R _ L O O K U P
 *
 * search a directory for a name and return its associated
 * object-UUID and type-UUID
 */

[ idempotent ] void dir_r_lookup(
    [in] handle_t       h,
    [in, string0] char  *name,    /* name to search for
                                     in directory */
    [out] uuid_t        *object,  /* object-UUID */
    [out] uuid_t        *type,    /* type-UUID of object */
    [out] unsigned32    *ttl,     /* time-to-live */
    [out] status_t      *status
);

/*
 * D I R _ R _ L O O K U P _ U U I D
 *
 * search a directory for an object-UUID and return all entries
 * with the UUID.
 * It is possible to have more than one entry associated with the
 * same object-UUID.
 */

[ idempotent ] void dir_r_lookup_uuid(
    [in] handle_t       h,
    [in] uuid_t         *object,       /* UUID to search for
                                          in directory */
    [in, out] cursor_t  *list_cursor,  /* cursor into entry list */
    [in] unsigned32     max_ents,      /* max number of entries */
    [out] unsigned32    *nents,        /* number of entries */
    [out, size_is(max_ents), length_is(nents)] dir_entry_t
                        entries[],     /* directory entries */
    [out] unsigned32    *ttl,          /* time-to-live */
    [out] status_t      *status
);

/*
 * D I R _ R _ R E A D
 *
 * read all entries in a directory
 */

[idempotent] void dir_r_read(
    [in] handle_t       h,
    [in, out] cursor_t  *list_cursor,  /* cursor into entry list */
```

```
    [in] unsigned32      max_ents,       /* max number of entries */
    [out] unsigned32     *nents,         /* number of entries */
    [out, size_is(max_ents), length_is(nents)] dir_entry_t
                         entries[],      /* directory entries */
    [out] unsigned32     *ttl,           /* time-to-live */
    [out] status_t       *status
);

}
```

A.5. linkr.idl

```
[
uuid(468b09a79000.0d.00.01.83.dc.00.00.00),
version(1)
]
interface link_r {

import 'nameb.idl';

/* Directory Link Interface (remote)
The Directory Link Interface specifies operations to associate a name
with link text in a directory.  A directory is identified by a handle
(h) and a directory entry is identified by a name.
*/

/*
 * L I N K _ R _ A D D
 *
 * add a link entry to a directory
 */

void link_r_add(
    [in] handle_t       h,
    [in, string0] char  *name,      /* name to add */
    [in, string0] char  *linktext,  /* text to associate with name */
    [out] status_t      *status
);

/*
 * L I N K _ R _ R E P L A C E
 *
 * replace a name-link association with new link text
 */

void link_r_replace(
    [in] handle_t       h,
    [in, string0] char  *name,      /* name to replace */
    [in, string0] char  *linktext,  /* text to associate with name */
    [out] status_t      *status
);

/*
 * L I N K _ R _ L O O K U P
 *
 * lookup a link name in a directory and return its associated text
 */

void link_r_lookup(
    [in] handle_t       h,
    [in, string0] char  *name,      /* name to lookup */
    [in] unsigned32     max_len,
    [out, size_is(max_len), string0] char
                        *linktext,  /* text associated with name */
    [out] unsigned32    *ttl,       /* time-to-live */
```

```
    [out] status_t        *status
);

}
```

A.6. attribb.idl

```
/* Basic attribute datatypes
*/
interface attribb {

import 'nameb.idl';

/* Common Data Types for Attribute Interfaces */

/* attribute type enum */
typedef enum {
    attrib_bool_val,
    attrib_int_val,
    attrib_float_val,
    attrib_str_val,
    attrib_byte_val,
    attrib_addr_val,
    attrib_uuid_val
} attrib_type_t;

typedef struct {
    unsigned32            len;
    [length_is(len)] byte byte_array[];
} attrib_byte_t;

/* attribute value type */
typedef union switch ( attrib_type_t type_of_value ) val {
    case attrib_bool_val:  boolean                 bool;
    case attrib_int_val:   long                    integer;
    case attrib_float_val: float                   fp;
    case attrib_str_val:   [string0] char          *string;
    case attrib_byte_val:  attrib_byte_t           *bytes;
    case attrib_addr_val:  sock_addr_t             address;
    case attrib_uuid_val:  uuid_t                  uuid;
} attrib_value_t;

/* attribute type */
typedef struct {
    uuid_t          attrib_id;  /* attribute identifier */
    attrib_value_t  value;      /* attribute value */
} attrib_t;

typedef struct {
    unsigned32                          num_attrib;
    [length_is(num_attrib)] attrib_t    attribs[];
} attrib_list_t;

typedef struct {
    [string0] char  *name;
    attrib_list_t   *attribs;
}   dir_attrib_entry_t;

typedef struct {
    uuid_t          obj_uuid;
```

```
    attrib_list_t   *attribs;
} attrib_entry_t;

/* The following typedefs are for the
   attrib_r.idl/attrib_r_lookup_matching_attrs operation.  The
   attrib client agent takes a text string that specifies the
   attribute values to lookup and constructs an expression tree that
   the remote attrib_r_lookup_matching_attrs operation expects.
   Therefore, client programs should not need to use these types.
*/

/* attribute expression operators */
typedef enum {
    op_or, op_xor, op_and, op_equal, op_not_equal, op_greater,
    op_greater_equal, op_less, op_less_equal, op_min, op_max, op_any,
    op_not, op_str_cmp, op_undef
} attrib_operator_t;

typedef struct {
    uuid_t  attrib_uuid;
    uuid_t  attrib_value_type;
} attrib_id_node_t;

/* attribute tree node types */
typedef enum {
    op_node, attrib_id_node, attrib_val_node, first_node, bad_tree
} attrib_node_type_t;

/* attribute tree node values */
typedef union switch ( attrib_node_type_t type_of_node ) nv {
    case op_node:          attrib_operator_t    operator;
    case attrib_id_node:   attrib_id_node_t     attrib_id;
    case attrib_val_node:  attrib_value_t       attrib_val;
    case first_node:       long                 num_nodes;
} attrib_node_t;

/* attribute tree types */
typedef struct {
    attrib_node_t         node;
    unsigned32            left_index;
    unsigned32            right_index;
} attrib_exp_array_ent_t;

typedef struct {
    unsigned32                                      num_entries;
    [length_is(num_entries)] attrib_exp_array_ent_t ent[];
} attrib_exp_array_t;

typedef struct {
    attrib_node_t       node;
    attrib_exp_tree_t   *left_p;
    attrib_exp_tree_t   *right_p;
} attrib_exp_tree_t;

}
```

A.7. dirattr.idl

```
[
uuid(474f4fc88000.0d.00.01.99.ba.00.00.00),
version(1)
]
interface dir_attrib_r {

import 'nameb.idl';
import 'attribb.idl';

/* Directory Attribute Interface (remote)
The Directory Attribute interface specifies operations to add, drop,
replace, change and return the attributes associated with a directory
entry.  The directory is identified by a handle (h) and the directory
entry is identified by a name.
*/

/*
 * D I R _ A T T R I B _ R _ M U T A B L E
 *
 * return true if the manager which is exporting this interface
 * supports the update operations in this interface.
 * return false otherwise
 */

boolean dir_attrib_r_mutable(
    [in] handle_t       h,
    [out] status_t      *status
);

/*
 * D I R _ A T T R I B _ R _ A D D
 *
 * add a name and associated attribute set to a directory.
 */

void dir_attrib_r_add(
    [in] handle_t       h,
    [in, string0] char  *name,          /* name to add to dir */
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] attrib_t
                        attrib[],       /* attribute/value pairs */
    [out] status_t      *status
);

/*
 * D I R _ A T T R I B _ R _ R E P L A C E _ A T T R I B U T E S
 *
 * replace the attribute set associated with a name
 */

void dir_attrib_r_replace_attributes(
    [in] handle_t       h,
    [in, string0] char  *name,          /* name to add to dir */
```

```
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] attrib_t
                        attrib[],       /* attribute/value pairs */
    [out] status_t      *status
);

/*
 * D I R _ A T T R I B _ R _ D R O P _ A T T R I B U T E S
 *
 * drop some of the attributes associated with a name
 */

void dir_attrib_r_drop_attributes(
    [in] handle_t       h,
    [in, string0] char  *name,
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] uuid_t
                        attrib_id[],    /* ids of attrs to drop */
    [out] status_t      *status
);

/*
 * D I R _ A T T R I B _ R _ C H A N G E _ A T T R I B U T E S
 *
 * change the attribute set associated with a name.  Attributes
 * listed in attrib are changed or added to the entry's attr set.
 * If a  multi-valued attribute is changed, the values specified in
 * this call replace all the values associated with this attribute.
 */

void dir_attrib_r_change_attributes(
    [in] handle_t       h,
    [in, string0] char  *name,
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] attrib_t
                        attrib[],       /* attribute/value pairs */
    [out] status_t      *status
);

/*
 * D I R _ A T T R I B _ R _ L O O K U P
 *
 * search a directory for a name and return the attribute set
 * associated with the name.
 */

[ idempotent ] void dir_attrib_r_lookup(
    [in] handle_t       h,
    [in, string0] char  *name,
    [in, out] cursor_t  *list_cursor,   /* cursor into attrib list */
    [in] unsigned32     max_num_attrib, /* max number of entries */
    [out] unsigned32    *num_attrib,    /* nbr of entries returned */
    [out, size_is(max_num_attrib), length_is(num_attrib)] attrib_t
                        attrib[],       /* attribute/value pairs */
    [out] unsigned32    *ttl,           /* time-to-live */
```

```
    [out] status_t      *status
);

/*
 * D I R _ A T T R I B _ R _ R E A D
 *
 * read all entries (i.e., names and their associated attribute sets)
 * in a directory
 */

[ idempotent ] void dir_attrib_r_read(
    [in] handle_t       h,
    [in, out] cursor_t  *list_cursor,   /* cursor into entry list */
    [in] unsigned32     max_ents,       /* max number of entries */
    [out] unsigned32    *nents,         /* nbr of entries returned */
    [out, size_is(max_ents), length_is(nents)] dir_attrib_entry_t
                        entries[],      /* directory entries */
    [out] unsigned32    *ttl,           /* time-to-live */
    [out] status_t      *status
);

}
```

A.8. pathr.idl

```
[
uuid(468b0c496000.0d.00.01.83.dc.00.00.00),
version(1)
]
interface path_r {

import 'dirb.idl';

/* The Path Interface
The Path Interface specifies operations to resolve a pathname to an
object-UUID and its locations and to return the pathname of a
directory object.
*/

/*
 * P A T H _ R _ C O N S T R U C T
 *
 * determine the full pathname of the object
 * identified by the handle "h".
 */

[ idempotent ] void path_r_construct(
    [in] handle_t       h,
    [out, string0] char *pname,   /* pathname constructed so far */
    [out] uuid_t        *object,  /* UUID of next component whose
                                     name must be retrieved */
    [out] uuid_t        *parent,  /* parent of next component */
    [out] status_t      *status
);

/*
 * P A T H _ R _ R E S O L V E
 *
 * starting at the directory identified by the handle "h" return
 * the entry associated with the pathname's right-most component.
 * If stop_short is true, do not expand the last component
 * if it is a link.
 */

[ idempotent ] void path_r_resolve(
    [in] handle_t       h,
    [in, string0] char  *pname,     /* pathname to resolve */
    [in] boolean        stop_short, /* stop if link? */
    [out] dir_entry_t   *entry,     /* name, object-UUID, type of
                                       last component processed */
    [out] sock_addr_list_t *entry_saddrs,  /* entry's location(s) */
    [out] unsigned32    *ttl,       /* entry's time-to-live */
    [out] uuid_t        *parent,    /* parent of entry */
    [out, string0] char *residual,  /* unresolved part of pathname */
    [out, string0] char *pname_resolved, /* resolved part of pathname
                                            incl. link expansions */
    [out] status_t      *status
);
```

```
}
```

A.9. childr.idl

```
[
uuid(46b71b221000.0d.00.01.99.ba.00.00.00),
version(1)
]
interface child_r {

import 'nameb.idl';

/*  Child Interface (remote)

The child interface provides operations to set and get a reference to
a parent object.  The parent object is the directory in which the
object (i.e., the child) is catalogued.
*/

/*
 * C H I L D _ R _ G E T _ P A R E N T
 *
 * get the UUID of the directory in which the object is cataloged.
 */

[ idempotent ] void child_r_get_parent(
    [in] handle_t       h,
    [out] uuid_t        *parent,  /* parent directory UUID */
    [out] status_t      *status
);

/*
 * C H I L D _ R _ S E T _ P A R E N T
 *
 * set the UUID of the directory in which the object is cataloged.
 */

void child_r_set_parent(
    [in] handle_t       h,
    [in] uuid_t         *parent,  /* parent directory UUID */
    [out] status_t      *status
);

}
```

A.10. attribr.idl

```
[
uuid(47729a417000.0d.00.01.99.ba.00.00.00),
version(1)
]
interface attrib_r {

import 'nameb.idl';
import 'attribb.idl';

/*
Attribute Interface

The Attribute interface can inquire about and modify the attribute
set associated with a particular object-UUID.  The interface also
provides an operation "attrib_lookup_matching_attributes" which
returns an array of object-UUIDs that match a set of attributes.

All operations accept a handle (h) argument as input which denotes
the object that contains the object-UUID/attribute set entries.
*/

/*
 * A T T R I B _ R _ M U T A B L E
 *
 * return true if the manager which is exporting this interface
 * supports the update operations in this interface.
 * return false otherwise
 */

boolean attrib_r_mutable(
    [in] handle_t       h,
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ A D D
 *
 * add an object-UUID and its associated attribute set
 */

void attrib_r_add(
    [in] handle_t       h,
    [in] uuid_t         *object,   /* object to add */
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] attrib_t
                        attrib[],  /* attribute/value pairs */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ D R O P
 *
 * drop an object-UUID
```

```
 */

void attrib_r_drop(
    [in] handle_t       h,
    [in] uuid_t         *object,  /* object to drop */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ R E P L A C E _ A T T R I B U T E S
 *
 * replace the attribute set associated with an object-UUID.
 */

void attrib_r_replace_attributes(
    [in] handle_t       h,
    [in] uuid_t         *object,   /* object to replace */
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] attrib_t
                        attrib[],  /* attribute/value pairs */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ D R O P _ A T T R I B U T E S
 *
 * drop some of the attributes associated with an object-UUID.
 */

void attrib_r_drop_attributes(
    [in] handle_t       h,
    [in] uuid_t         *object,
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] uuid_t
                        attrib_id[],  /* ids of attributes to drop */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ C H A N G E _ A T T R I B U T E S
 *
 * change the attribute set associated with a object-UUID.
 * Attributes listed in attrib are changed or added to the entry's
 * attribute set.  If a multi-valued attribute is changed, the values
 * specified in this call replace all the values associated with this
 * attribute.
 */

void attrib_r_change_attributes(
    [in] handle_t       h,
    [in] uuid_t         *object,   /* object to change */
    [in] unsigned32     num_attrib,
    [in, length_is(num_attrib)] attrib_t
                        attrib[],  /* attribute/value pairs */
    [out] status_t      *status
```

```
);

/*
 * A T T R I B _ R _ L O O K U P
 *
 * return the attribute set associated with an object-UUID.
 */

[ idempotent ] void attrib_r_lookup(
    [in] handle_t       h,
    [in] uuid_t         *object,          /* object to lookup */
    [in, out] cursor_t  *list_cursor,     /* cursor into attrib set */
    [in] unsigned32     max_num_attrib,  /* max number of entries */
    [out] unsigned32    *num_attrib,      /* number of entries */
    [out, length_is(num_attrib), size_is(max_num_attrib)] attrib_t
                        attrib[],         /* attribute/value pairs */
    [out] unsigned32    *ttl,             /* time-to-live */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ L O O K U P _ A T T R I B U T E
 *
 * return all object-UUIDs that have the attribute whose UUID is
 * attrib_id.
 */

[ idempotent ] void attrib_r_lookup_attribute(
    [in] handle_t       h,
    [in] uuid_t         *attrib_id,
    [in, out] cursor_t  *list_cursor,  /* cursor into object list */
    [in] unsigned32     max_num_obj,   /* max number of entries */
    [out] unsigned32    *num_obj,      /* number of entries */
    [out, length_is(num_obj), size_is(max_num_obj)] uuid_t
                        object[],      /* object-UUIDs */
    [out] unsigned32    *ttl,          /* time-to-live */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ R E A D _ O B J E C T S
 *
 * return the UUIDs of the objects whose attributes are stored in the
 * object denoted by h.
 */

[ idempotent ] void attrib_r_read_objects(
    [in] handle_t       h,
    [in, out] cursor_t  *list_cursor,  /* cursor into object list */
    [in] unsigned32     max_num_obj,   /* max number of entries */
    [out] unsigned32    *num_obj,      /* number of entries */
    [out, length_is(num_obj), size_is(max_num_obj)] uuid_t
                        object[],      /* object-UUIDs */
    [out] status_t      *status
);
```

```
/*
 * A T T R I B _ R _ R E A D _ A T T R I B U T E S
 *
 * return the UUIDs of the attributes stored in the object denoted
 * by h.
 */

[ idempotent ] void attrib_r_read_attributes(
    [in] handle_t       h,
    [in, out] cursor_t  *list_cursor,    /* cursor into attr list */
    [in] unsigned32     max_num_attrib,  /* max number of entries */
    [out] unsigned32    *num_attrib,     /* number of entries */
    [out, length_is(num_attrib), size_is(max_num_attrib)] uuid_t
                        attrib_id[],     /* attribute UUIDs */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ R E A D
 *
 * read all the entries stored in the object denoted by h.
 */

[ idempotent ] void attrib_r_read(
    [in] handle_t       h,
    [in, out] cursor_t  *list_cursor,  /* cursor into entry list */
    [in] unsigned32     max_ents,      /* max number of entries */
    [out] unsigned32    *nents,        /* number of entries */
    [out, size_is(max_ents), length_is(nents)] attrib_entry_t
                        entries[],     /* entries */
    [out] unsigned32    *ttl,          /* time-to-live */
    [out] status_t      *status
);

/*
 * A T T R I B _ R _ L O O K U P _ M A T C H I N G _ A T T R S
 *
 * return the UUIDs of the objects whose attributes match the
 * attribute expression whose parse tree representation is in
 * "expression".
 *
 * Note that expression.node.type_of_node must be set to FIRST_NODE,
 * expression.node.nv.num_nodes must be set to the number of nodes in
 * the expression parse tree, and expression.left_p must point to the
 * root of the parse tree.  Expression.right_p is unused.
 * Max_num_obj is the size of the obj_uuid array.  Num_obj is the
 * actual number of objects returned in obj_uuid.
 */

[ idempotent ] void attrib_r_lookup_matching_attrs(
    [in] handle_t       h,
    [in] exp_tree_t     expression,
    [in, out] cursor_t  *list_cursor,  /* cursor into object list */
    [in] unsigned32     max_num_obj,   /* max number of entries */
    [out] unsigned32    *num_obj,      /* number of entries */
```

```
    [out, length_is(num_obj), size_is(max_num_obj)] uuid_t
                        object[],       /* object ids */
    [out] unsigned32    *ttl,           /* time-to-live */
    [out] status_t      *status
);

}
```

A.11. objectr.idl

```
[
uuid(476248fd7000.0d.00.01.99.ba.00.00.00),
version(1)
]
interface object_r {

import 'nameb.idl';

/*
Object interface (remote)

The object interface supports operations on an object identified by
h.  These operations include delete an object, get an object's type,
set/get an object's size, and set/get an object's time created,
modified, or accessed.
*/

/*
 * O B J E C T _ R _ D E L E T E
 *
 * Delete an object
 */

void object_r_delete(
    [in] handle_t       h,
    [out] status_t      *status
);

/*
 * O B J E C T _ R _ G E T _ T Y P E
 */

void object_r_get_type(
    [in] handle_t       h,
    [out] uuid_t        *type,
    [out] status_t      *status
);

/*
 * O B J E C T _ R _ G E T _ S I Z E
 */

void object_r_get_size(
    [in] handle_t       h,
    [out] unsigned32    *size,
    [out] status_t      *status
);

/*
 * O B J E C T _ R _ S E T _ S I Z E
 *
 */
```

```
void object_r_set_size(
    [in] handle_t       h,
    [in] unsigned32     size,
    [out] status_t      *status
);

typedef enum {
    object_time_create,
    object_time_modify,
    object_time_access,
    object_time_attribute_modify
} object_time_type_t;

typedef struct  {
    unsigned32 sec;
    unsigned32 usec;
} time_t;

/*
 * O B J E C T _ R _ G E T _ T I M E
 */

void object_r_get_time(
    [in] handle_t             h,
    [in] object_time_type_t   time_type,
    [out] time_t              *time,
    [out] status_t            *status
);

/*
 * O B J E C T _ R _ S E T _ T I M E
 *
 */

void object_r_set_time(
    [in] handle_t             h,
    [in] object_time_type_t   time_type,
    [in] time_t               *time,
    [out] status_t            *status
);

}
```

A.12. objlocb.idl

```
/* Basic datatypes for the object location */

interface objlocb {

import 'nameb.idl';

typedef struct {
    uuid_t              object;
    sock_addr_list_t    *object_saddrs;
} obj_loc_entry_t;

}
```

A.13. objlocr.idl

```
[
uuid(4772984dd000.0d.00.01.99.ba.00.00.00),
version(1)
]
interface obj_loc_r {

import 'nameb.idl';
import 'objlocb.idl';

/* Object Location Interface (remote)
The Object Location interface supports operations to add, replace,
drop, and lookup object location entries.
The object location database is identified by a handle (h).  Entries
are identified by an object-UUID and contain an object-UUID and its
locations (i.e., a set of socket addresses).
*/

/*
 * O B J _ L O C _ R _ M U T A B L E
 *
 * return true if the manager which is exporting this interface
 * supports the update operations in this interface.
 * return false otherwise
 *
 */

boolean obj_loc_r_mutable(
    [in] handle_t       h,
    [out] status_t      *status
);

/*
 * O B J _ L O C _ R _ A D D
 *
 * add an object and its locations to an object location database
 */

void obj_loc_r_add(
    [in] handle_t       h,
    [in] uuid_t         *object,
    [in] unsigned32     num_addr,
    [in, length_is(num_addr)] sock_addr_t
                        sock_addrs[],
    [out] status_t      *status
);

/*
 * O B J _ L O C _ R _ R E P L A C E
 *
 * replace an object and its locations in an object location database
 */

void obj_loc_r_replace(
```

```
    [in] handle_t       h,
    [in] uuid_t         *object,
    [in] unsigned32     num_addr,
    [in, length_is(num_addr)] sock_addr_t
                        sock_addrs[],
    [out] status_t      *status

);

/*
 * O B J _ L O C _ R _ D R O P
 *
 * drop an object and its locations from an object location database
 */

void obj_loc_r_drop(
    [in] handle_t       h,
    [in] uuid_t         *object,
    [out] status_t      *status
);

/*
 * O B J _ L O C _ R _ L O O K U P
 *
 * lookup an object in an object location database and
 * return its locations
 */

[idempotent] void obj_loc_r_lookup(
    [in] handle_t       h,
    [in] uuid_t         *object,
    [in, out] cursor_t  *list_cursor,  /* cursor into list of locs */
    [in] unsigned32     max_num_addr,  /* max number of entries */
    [out] unsigned32    *num_addr,     /* number of entries */
    [out, size_is(max_num_addr), length_is(num_addr)] sock_addr_t
                        sock_addrs[],  /* locations */
    [out] unsigned32    *ttl,          /* time-to-live */
    [out] status_t      *status
);

/*
 * O B J _ L O C _ R _ R E A D
 *
 * return all objects and their locations
 */

[idempotent] void obj_loc_r_read(
    [in] handle_t       h,
    [in, out] cursor_t  *list_cursor,  /* cursor into entry list */
    [in] unsigned32     max_ents,      /* max number of entries */
    [out] unsigned32    *nents,        /* number of entries */
    [out, size_is(max_ents), length_is(nents)] obj_loc_entry_t
                        entries[],     /* entries */
    [out] unsigned32    *ttl,          /* time-to-live */
    [out] status_t      *status
```

```
);
}
```

A.14. typer.idl

```
[
uuid(4772984dd000.0d.00.01.99.ba.00.00.00),
version(1)
]
interface type_r {

import 'nameb.idl';
import 'acl_base.idl';

/*
 * T Y P E _ R _ C R E A T E
 *
 * Create an object of a particular type, set its protection from the
 * access control list specified in acl, and set its parent.
 * Create the object in the container identified by h.
 */
void type_r_create(
    [in] handle_t       h,
    [in] uuid_t         *parent,
    [in] acl_t          *acl,
    [out] uuid_t        *object,
    [out] status_t      *status
);

}
```

APPENDIX B. THE REVISED RND INTERFACES

B.1. Datatype Definitions

```
/*      BASIC DATA STRUCTURES                                   */

/*
 *      ts_t            A time and space unique identifier used to
 *                      unambiguously identify updates.
 */

typedef struct {
        node byte[6];   /* 48 bit host ID */
        time byte[8];   /* UTC in 100 nanosecond units, smithsonian
                           zero */
} ts_t;

/*      INTERNAL NAME FORMATS                                   */

/*
 *++
 *
 * 1. There are two representations of names.  The first is called a
 * "Path Name".  The path name is used in two contexts:
 *    b) In the RND interface, and hence "on the wire"
 *    c) As an attribute value when an attribute stores a path name
 *       (e.g., symlink target, group member)
 * The second is called a printable path name.  It is the ND’ API
 * name form and is intended for human consumption.
 *
 * 2. Both path name and printable path name consist of a sequence of
 * name components called RDNs (adopting X.500 terminology).  The
 * only difference between the printable and non-printable form of an
 * RDN is that the latter has some extra state associated with it to
 * aid caching.
 *
 * 3. All X.500 RDN forms are permitted, including multi-AVA RDNs
 * (AVA = Attribute Value assertion).  However, the AVA values *must*
 * be represented as strings.  For those RDN types which do not have
 * a natural string representation, the value is encoded as pairs of
 * hex digits.  If the RDN type is a legal X.500 RDN, the hex digits
 * are the exact bit string of the ASN.1 BER encoding of the RDN
 * value.
 *
 * A "binary name value" is indicated by the characters "%x" or "%X"
 * in front of the hex digit pairs.
 *
 * 4. RDN types are denoted by ASN.1 Object identifiers.  They are
 * encoded in an RDN as a sequence of integers, separated by dots
 * ("."), and ending with an equals sign ("=").  A "typeless" RDN is
 * a "naked value", that is, it contains no "=".
 *
 * 5. Mutli-AVA RDNs are represented as a sequence of type=value
 * pairs, separated by commas, e.g., "1.3.43=David,1.3.44=Oran".  In
 * the interest of efficient marshalling, RDNs are restricted to a
 * maximum of 5 AVAs.
```

```
 *
 * 6. Each AVA of an RDN has two flags associated with it.  The first
 * is called the "type tag".  This tag indicates whether the
 * namespace entry of which the RDN is a subordinate guarantees that
 * the value domain is disjoint across all types.  In other words, if
 * the nameservice directory prohibits two subordinates foo=bar and
 * zot=bar from existing simultaneously, then type tag would be TRUE.
 * DECdns directories will always have type tag TRUE.  X.500 DSAs
 * will always set type tag FALSE (except if the X.500 implementation
 * can guarantee through the schema enforcement that a given object
 * class can only have one subordinate object type, in which case it
 * can set type tag to TRUE.
 *
 * The second tag is the "case fold tag".  It is used to indicate
 * whether the type:
 *    a) uses case-sensitive ("Case exact" in X.500ese) matching
 *    b) case-insensitive matching
 *
 * One additional rule is applied.  If an AVA value in an RDN is a
 * binary value, and the case-fold tag is set to TRUE (case
 * sensitive) this is taken to mean "apply an ObjId-specific mathing
 * rule on the AVA".  Pragmatically, since most if not all clients
 * will lack knowledge of how to do type-specific name matching, this
 * will cause clerk caching to be suppressed for this name and all
 * decendents.
 *
 * 7. The following special characters are reserved in DN strings.
 * Each has a defined escape character so that they may be present in
 * an AVA value:
 *      Latin-1 Character   Escape    Reason
 *      /                   \/        Canonical RDN separator
 *      \                   \\        Escape delimiter
 *      =                   \=        type=value
 *      *                   \*        wildcard
 *      ?                   \?        wildcard
 *      ,                   \,        Multi-AVA RDN separator
 *      <space>             \<space>  Name terminator (good idea???)
 */

typedef string0 rdn_t;
typedef string0 printable_path_name_t;

typedef struct {
        boolean disjoint_values_tag[5]  /* type tags for 5 AVAs */
        boolean case_fold_tag[5]        /* case tags for 5 AVAs */
        rdn_t   rdn;
} component_name_t;

typedef struct {
        unsigned32       num_names;
        component_name_t comp_name[num_names];
} path_name_t;

/*      CURSOR STRUCTURE                                               */
```

```
/*
 * The cursor is used as a context handle to continue or restart
 * enumeration operations.
 */

typedef enum {
        cursor_start,     /* Start at the beginning */
        cursor_continue,  /* Start where cursor points */
        cursor_end        /* Reached end of the enumerable data */
} cursor_state_t;

typedef struct {
        cursor_state_t state;
        uuid_t         server_id;
        int            cursor_size;
        byte           x[cursor_size];  /* opaque cursor */
} cursor_t;

/*      NETWORK ADDRESS STRUCTURES                                */

/*
 *      tower_t         A list of protocol selectors which indicate
 *                      one combination of protocols (application,
 *                      presentation, session, transport, network,
 *                      datalink etc) and addresses supported by a
 *                      particular network object.  An object may be
 *                      contacted via one of several protocol towers.
 *                      This is up to Steve Miller and Nat Mishkin.
 */

typedef struct {
    unsigned32 slen;
    byte[slen] tower;
} tower_t;

/*
 *      location_t      A set of protocol towers and additional
 *                      information used to contact a single
 *                      directory replica.
 *
 *              location_uuid   This uniquely identifies the
 *                              clearinghouse which stores this
 *                              replica.
 *
 *              private_info    Additional location discriminator.
 *                              For directories, this maps into
 *                              replica_type_t.
 *
 *              towers          A set of protocol towers which, at
 *                              some point in time, were known to
 *                              reflect the address of this
 *                              clearinghouse which stored this
 *                              replica.  This address information is
 *                              verified by checking the
 *                              location_uuid in the replica pointer
```

```
 *                              against that of the clearinghouse
 *                              itself when
 *                              contacted.
 */
typedef enum {master,    /* Accepts all updates/creates */
              secondary, /* Master minus create child */
              readonly   /* Lookups/reads/tests */
} replica_type_t;

typedef struct {
        uuid_t     location_uuid;  /* UUID of clearinghouse */
        unsigned32 private_info;   /* map to replica type */
        unsigned32 num_towers;
        tower_t    towers[num_towers];
} location_t;

/*
 *      location_set_t  Information which describes a set of service
 *                      replicas.
 *
 *              locations_uuid  A unique identifier for the set as a
 *                              whole, which can be used for later
 *                              update, or caching and possibly for
 *                              sharing.
 *
 *              authoritative   Indicates whether the locations in
 *                              the set are considered to be the
 *                              "truth" by the server returning them,
 *                              or are "hints".  The colloquial
 *                              meaning of this being TRUE is "no
 *                              point in asking anyone else because I
 *                              have as good information as anybody".
 *
 * If authoritative can be set FALSE by a server when returning a
 * location_set_t, the server MUST support a procedure of the form:
 *
 * void location_to_path(
 *      uuid_t [in]       location_uuid;
 *      path_name_t [out] pname;
 *      )
 *
 */
typedef struct {
        uuid_t     locations_uuid;
        boolean    authoritative;
        unsigned32 num_locations;
        location_t locations[num_locations];
} location_set_t;

/*      TTL - time cached values considered valid                  */

/*
 * Each entry may have 2 TTLs.  One is the hint TTL, the other is the
 * authoritative TTL.  Hint would apply to something that can be
 * verified when used and would have a long timeout since the
```

```
 * application can force a cache by-pass when it detects the value is
 * invalid.  Each attribute has a bit which indicates which TTL
 * applies.  If the TTL values are not specified they default to hint
 * of infinity and authoritative of don't cache.
 */

typedef struct {
        byte expiration[8];
        byte extension[8];
} ttl_t;

/* Attribute structures                                                */

typedef struct {
        ttl_t       ttl;     /* Time to cache link */
        path_name_t target;  /* Resultant name */
} link_entry_t;

typedef enum {link, child, other} entrytype_t;
typedef union switch {entrytype_t type_of_value} val {
        case link:  link_entry_t link;
        case child: locations_set_t;
        case other: locations_set_t;
} entry_t;

/*
 * The dir_entry_t is composed of the following known attributes:
 * entrytype, replicas, object- and type-UUID, softlink target and
 * softlink timeout.  These attributes will have pre-assigned
 * object-id's.  (TBD)
 */
typedef struct {
        component_name_t name;       /* entry name */
        uuid_t           object;     /* object-UUID */
        uuid_t           type;       /* type-UUID */
        entry_t          entrydata;
} dir_entry_t;

typedef enum {'types supported'} valuetype_t;
typedef union switch { valuetype_t type_of_value} val {
                /* ONE PER ENUM VALUE */
                case:
                case:
} value_t;

typedef struct {
        ts_t    value_ts; /* timestamp of value */
        value_t value;
attrib_value_t;

typedef struct {
        string0        name;               /* object_id (12.3.4) */
        boolean        auth_or_hint;       /* which ttl to use */
        unsigned32     num_values;
        attrib_value_t value[num_values]; /* attribute value(s) */
```

```
} attrib_t;

typedef enum {
        dropattribute, /* Delete attribute */
        addattribute,  /* Add new attribute and values */
        addvalues,     /* Add values to attribute */
        removevalues   /* Delete values from an attribute */
} attrib_op_t;

typedef {
        attrib_op_t op_code;
        attrib_t    attribute;  /* On a drop num_values = 0 */
} modify_t;
```

B.2. path Interface

```
[
uuid(XXXX),
version(1)
]
interface path_r {

import '??.idl'

/*
 * Path Interface (remote)
 */

/*
 * P A T H _ R _ R E S O L V E
 *
 * Starting at the directory denoted by h return the entry associated
 * with the pathname's right-most component.  Flags include
 * stop_short, which if true, means do not expand the last component
 * if it is a link.  Other values in flags include:
 *
 *      up:             Couldn't resolve any names, include list of
 *                      other servers to try
 *      hit_link:       A link was resolved by the server just
 *                      contacted
 *      ignore_state:   Resolve directory name regardless of its
 *                      state directories and their replicas can be
 *                      in the process of being created/deleted.
 *      stop_short:     Do not expand last component if it is a link.
 *
 * The locations of the parent is the value of the replicas
 * attribute.  This is used for caching.
 *
 * Status: AccessDenied, UnknownEntry
 */

void
path_r_resolve (
        handle_t [in]         h,
        path_name_t [in, out] residual, /* unprocessed part of name */
        path_name_t [in, out] resolved, /* resolved part of name */
        boolean [in]          stop_short,
        boolean [in]          ignore_state,
        boolean [in, out]     up,
        boolean [out]         hitlink,
        dir_entry_t [out]     *entry,
        uuid_t [out]          *parent, /* parent of entry */
        location_set_t [out]  *parent_locations;
        status_t [out, comm_status]
                              *status
)

}
```

B.3. nlist Interface

```
[
uuid(XXXX),
version(1)
]
interface nlist_r {

import '??.idl'

/*
 * Nlist Interface (remote)
 * The Nlist interface provides operations for dropping and changing
 * names in lists and enumerating a list.
 */

/*
 * N L I S T _ R _ A D D
 *
 * Add a name to a list.  If this call is used to add a name to a
 * directory, the entrytype will be other.
 *
 * Status: AccessDenied, EntryExists
 */

void
nlist_r_add (
        handle_t [in]                h,
        component_name_t [in]        name,
        status_t [out, comm_status] *status
);

/*
 * N L I S T _ R _ C H A N G E _ N A M E
 *
 * Change the name in a list from old_name to new_name
 *
 * Status: AccessDenied, UnknownEntry
 */

void
nlist_r_change_name (
        handle_t [in]                h,
        component_name_t [in]        old_name,
        component_name_t [in]        new_name,
        status_t [out, comm_status] *status
);

/*
 * N L I S T _ R _ D R O P
 *
 * Drop a name from a list.
 *
 * Status: AccessDenied, UnknownEntry
 */
```

```
void
nlist_r_drop (
        handle_t [in]                h,
        component_name_t [in]        name,
        status_t [out, comm_status] *status
);

/*
 * N L I S T _ R _ L O O K U P
 *
 * Return true if a name is in a list, return false if a name is not
 * in a list.
 */

boolean
nlist_r_lookup (
        handle_t [in]                h,
        component_name_t [in]        name,
        status_t [out, comm_status] *status
);

/*
 * N L I S T _ R _ R E A D
 *
 * Return a list of names in a directory.  There is no architected
 * sort order.
 *
 * Status: AccessDenied
 */

void
nlist_r_read (
        handle_t [in]         h,
        component_name_t [in] name_filter,   /* optional wildcard */
        cursor_t [in, out]    *list_cursor,  /* cursor for where to
                                                begin or continue
                                                reading the nlist */
        unsigned32 [in]       max_num_names, /* maximum number of
                                                names to return */
        unsigned32 [out]      *num_names,    /* number of names
                                                returned */
        component_name_t [last_is(num_names, max_is(max_num_names),
        out]
                              names[],       /* list of names */
        status_t [out, comm_status]
                              *status
};

}
```

B.4. dir Interface

```
[
uuid(XXXX),
version(1)
]
interface dir_r {

import '??.idl'

/*
 * Directory Interface (remote)
 *
 * The Directory interface specifies operations to add, lookup, and
 * replace entries in a directory.  All entries are identified by a
 * name.
 */

/*
 * D I R _ R _ A D D
 *
 * Add a name and its associated directory entry to a directory.
 *
 * Status: AccessDenied, EntryExists
 */

void
dir_r_add (
        handle_t [in]         h,
        dir_entry_t [in]      entry, /* directory entry */
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ R _ R E P L A C E
 *
 * Replace the dir_entry_t of a directory entry.  Any other
 * attributes will be uneffected.
 *
 * Status: AccessDenied, UnknownEntry, InvalidUpdate
 */

void
dir_r_replace (
        handle_t [in]         h,
        dir_entry_t [in]      entry, /* new entry */
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ R _ L O O K U P
 *
 * Search a directory for a name and return its associated directory
```

```
 * entry If component_name_t is the NULL name, operate on the
 * directory object itself.
 *
 * Status: AccessDenied, UnknownEntry
 */

void
dir_r_lookup (
        handle_t [in]         h,
        component_name_t [in] name,   /* entry name to lookup */
        dir_entry_t [out]     *entry, /* returned directory entry */
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ R _ R E A D
 *
 * Read all entries in a directory.
 *
 * Status: AccessDenied
 */

void
dir_r_read (
        handle_t [in]         h,
        cursor_t [in, out]    *list_cursor, /* cursor into dir */
        unsigned32 [in]       max_ents,     /* max. no. of entries */
        unsigned32 [out]      *nents,       /* no. of rtd. entries */
        dir_entry_t [max_is(max_ents), last_is(nents), out]
                              entries[],    /* directory entries */
        status_t [out, comm_status]
                              *status
);

}
```

B.5. dir_attrib Interface

```
[
uuid(XXXX),
version(1)
]
interface dir_attrib_r {

import '??.idl'

/*
 * Directory Attribute Interface (remote)
 *
 * The Directory Attribute interface specifies operations to operate
 * on arbitrary attributes associated with a directory entry or the
 * directory itself.  The directory itself is specified with a Null
 * name.
 */

/*
 * D I R _ A T T R I B _ R _ A D D
 *
 * Add a name and its associated attribute set to a directory.
 * Cannot be used to create a new directory.
 *
 * Status: AccessDenied, EntryExists
 */

void
dir_attrib_r_add (
        handle_t [in]         h,
        component_name_t [in] name,       /* entry name to add */
        unsigned32 [in]       num_attrib, /* no. of attrs. */
        attrib_t [last_is(num_attrib), in]
                              attrib[],   /* attr/value pairs */
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ A T T R I B _ R _ R E P L A C E
 *
 * Replace a directory entry.  Removes all attributes associated with
 * an entry and adds new ones.
 *
 * Status: AccessDenied, UnknownEntry, InvalidUpdate
 */

void
dir_attrib_r_replace (
        handle_t [in]         h,
        component_name_t[in]  name,       /* entry name to */
        unsigned32 [in]       num_attrib, /* no. of attrs. */
        attrib_t [last_is(num_attrib), in]
                              attrib[],   /* attr/value pairs */
```

```
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ A T T R I B _ R _ M O D I F Y _ A T T R I B U T E S
 *
 * Modify attributes associated with an entry.  The updates either
 * all succeed or all fail at one replica.
 *
 * Status: AccessDenied, UnknownEntry, InvalidUpdate
 */

void
dir_attrib_r_modify_attributes (
        handle_t [in]         h,
        component_name_t [in] name,     /* of entry */
        unsigned32 [in]       num_attr, /* no. of attrs */
        modify_t [last_is(num_attr), in]
                              attrib[],
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ A T T R I B _ R _ L O O K U P_ N A M E S
 *
 * Search a directory for a name and return either all or some of the
 * attributes names associated with an entry.
 *
 * Status: AccessDenied, UnknownEntry
 */

void
dir_attrib_r_lookup_names (
        handle_t [in]         h,
        component_name_t [in] name,           /* of entry */
        unsigned32 [in]       attrib_count,   /* 0 or number in
                                                 attrib_types */
        obj_id_t [in]         attrib_types[attrib_count],
                                              /* attrs to read */
        cursor_t [in, out]    *list_cursor,   /* cursor into attr
                                                 list */
        unsigned32 [in]       max_num_attrib, /* max. no. to
                                                 return */
        unsigned32 [out]      *num_attrib,    /* no. returned */
        obj_id_t [max_is(max_num_attrib), last_is(num_attrib), out]
                              attrib[],
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ A T T R I B _ R _ L O O K U P _ V A L U E S
 *
```

```
 * Search a directory for a name and return either all or some of the
 * attribute values associated the entry.
 *
 * Status: AccessDenied, UnknownEntry
 */

void
dir_attrib_r_lookup_values (
        handle_t [in]         h,
        component_name_t [in] name,           /* of entry */
        unsigned32 [in]       attrib_count,   /* 0 or number in
                                                 attrib_types */
        obj_id_t [in]         attrib_types[attrib_count],
                                              /* attrs to read */
        cursor_t [in, out]    *list_cursor,   /* cursor into attr
                                                 list */
        unsigned32 [in]       max_num_attrib, /* max. no. to
                                                 return */
        unsigned32 [out]      *num_attrib,    /* no. returned */
        attrib_t [max_is(max_num_attrib), last_is(num_attrib), out]
                              attrib[],
        ttl_t [out]           hint_ttl,  /* used to cache attrs */
        ttl_t [out]           auth_ttl,  /* used to cache attrs */
        status_t [out, comm_status]
                              *status
);

/*
 * D I R _ A T T R I B _ R _ C O M P A R E
 *
 * Search a directory for a name and return true if the entry
 * contains an attribute with the purported value.
 *
 * Status: AccessDenied, UnknownEntry
 */

boolean
dir_attrib_r_compare (
        handle_t [in]         h,
        component_name_t [in] name,       /* of entry */
        obj_id_t [in]         attrname,   /* attribute */
        attrib_value_t [in]   attrvalue,  /* value to test */
        status_t [out, comm_status]
                              *status
);

}
```

B.6. dir_admin Interface

```
[
uuid(XXXX),
version(1)
]
interface dir_admin_r {

import '??.idl'

/*
 * Directory Administrative Interface (remote)
 *
 * The Directory Attribute interface specifies operations to create
 * and delete directories.
 */

/*
 * D I R _ A D M I N _ R _ C R E A T E
 *
 * Create a new directory as a child of the directory entry implied
 * by the pathname of the new directory.
 *
 * Status: AccessDenied, UnknownEntry, EntryExists
 */

void
dir_admin_r_create (
        handle_t [in]     h,           /* handle of clearinghouse
                                          holding master replica */
        path_name_t [in]  directory,  /* new directory */
        dir_entry_t [out] dirobject,  /* of directory object */
        status_t [out, comm_status]
                          *status
);

/*
 * D I R _ A D M I N _ R _ D E L E T E
 *
 * Delete a directory
 * All replicas will be deleted upon a successful return; otherwise
 * an error will be returned and the directory entry will not be
 * deleted.
 *
 * Status: AccessDenied, UnknownEntry, NotEmpty
 */

void
dir_admin_r_delete (
        handle_t [in]               h,
        status_t [out, comm_status] *status
);

}
```

B.7. Open Issues and Changes

The following are open issues noted by DEC with the last version of the revised interface:

(a) Specify the semantics of `private_info`

(b) Obtain final format of towers from Nat and Steve

(c) Agree on representation of TTLs

(d) Can an entry type be changed through an update or replace??

(e) Format of wildcard in `nlist_read`

(f) Need an error for UP pointers not supported and replica cannot perform operation.

DEC also noted the following changes from its previous version:

(a) Removed linked flag from PathResolve

(b) Cursor size is variable

(c) Added new `path_name_t` definition

(d) Remove `obj_id_t` from `dir_entry`

(e) Add `location_set_t` to `path_resolve` output to represent parent

(f) Fixed `dir_r_attrib` to read `dir_attrib_r`

(g) Moved create and delete directory to `dir_admin` interface

(h) Removed `dir_attrib_r_enumerate` and added wildcard to `nlist_read`

AUTHOR'S ADDRESS

Nathaniel Mishkin
Distributed Object Computing Program
Hewlett-Packard Co.
250 Apollo Drive
Chelmsford, MA 01824
USA

Internet email: mishkin@apollo.hp.com
Telephone: +1-508-436-4353