As an example, lets us consider a filter to represent a person whose name is either "Robbins" *or* ("Steve" *and* "Kille"). The structure would look something like:

```
flt_type = FILTER_OR
        - The filter is an OR filter
flt_un_filter.flt_type = FILTER_ITEM
        - First component or the OR is an item
flt_un_filter.flt_un_item = filter_item (Robbins)
        - The item should match "robbins"
flt_un_filter.flt_next->flt_type = FILTER_AND
        - Second component or the OR is an AND filter
flt_un_filter.flt_next->flt_un_filter.flt_type = FILTER_ITEM
        - First component or the AND is an item
flt_un_filter.flt_next->flt_un_filter.flt_un_item =
                                        filter_item (Steve)
        - The item should match "Steve"
flt_un_filter.flt_next->flt_un_filter.flt_next->flt_type =
                                        FILTER_ITEM
        - Second component or the AND is an item
flt_un_filter.flt_next->flt_un_filter.flt_next->flt_un_item =
                                        filter_item (Kille)
        - The item should match "Kille"
```

17.11 Modification Operations

There are four operations available to modify the directory.

Note: with this version of QUIPU, modify operations are only allowed over DAP, attempts to modify over DSP will return referral errors.

17.11.1 Add

To add an entry to the DIT use

```
ds_addentry (arg, error)
    struct ds_addentry_arg      * arg;
    struct DSError              * error;
```

You will need to include *quipu/add.h* to use this procedure. The argument you must supply is made up as follows:

```
struct ds_addentry_arg {
    CommonArgs    ada_common;
    DN            ada_object;
    Attr_Sequence ada_entry;
};
```

`ada_common`: The common arguments as described in Section 17.2.1.

`ada_object`: The DN of the object you want to add.

`ada_entry`: The attributes you want to add for this entry. This must contain the RDN of the entry, and an "objectclass" attribute. Then, the other attribute *must* conform to the set of "optional" and "mandatory" attribute for that object class.

17.11.2 Remove

This is used to remove an entry from the DIT. Only leaf entries can be removed.

```
ds_removeentry (arg, error)
    struct ds_removeentry_arg   * arg;
    struct DSError              * error;
```

You will need to include *quipu/remove.h* to use this procedure. The argument you must supply is made up as follows:

```
struct ds_removeentry_arg {
    CommonArgs rma_common;
    DN         rma_object;
};
```

`rma_common`: The common arguments as described in 17.2.1.

`rma_object`: The DN of the object you want to remove.

17.11.3 Modify

`ModifyEntry` is used to modify the attributes of the specified entry. There are strong restrictions on this operation as required by X.500. Invalid operations result in errors, and as such none of the requested modifications are made. To modify an attribute you must first `remove` it then `add` a new attribute. You cannot modify the RDN; you must use the `ModifyRDN` operation described in Section 17.11.4 to do this.

```
ds_modifyentry (arg, error)
    struct ds_modifyentry_arg   * arg;
    struct DSError              * error;
```

You will need to include *quipu/modify.h* to use this procedure. The argument you must supply is made up as follows:

```
struct ds_modifyentry_arg {
    CommonArgs        mea_common;
    DN                mea_object;
    struct entrymod * mea_changes;
};
```

`mea_common`: The common arguments as described in Section 17.2.1.

`mea_object`: The DN of the object you want to modify.

`mea_changes`: A tree structure defining the changes you want to make to the entry.

```
struct entrymod {
    int               em_type;
#define EM_ADDATTRIBUTE         1
#define EM_REMOVEATTRIBUTE      2
#define EM_ADDVALUES            3
#define EM_REMOVEVALUES         4
    Attr_Sequence     em_what;
    struct entrymod * em_next;
};
```

`em_type`: The type of modification this is.

`em_what`: The attribute you want to add or remove. If the operation type is remove, then this structure should only contain an attribute type, and not a value.

`em_next`: A modify operation is built up with a series of small modifications. Hence this structure is a linked list. The operation is seen as one atomic operation.

17.11.4 ModifyRDN

To modify the distinguished attribute values of an entry, you must use this operation; `modifyEntry` cannot be used.

```
ds_modifyrdn (arg, error)
    struct ds_modifyrdn_arg     * arg;
    struct DSError              * error;
```

You will need to include *quipu/modifyrdn.h* to use this procedure. The arguments are:

```
struct ds_modifyrdn_arg {
    CommonArgs mra_common;
    DN         mra_object;
    RDN        mra_newrdn;
    char       deleterdn;
};
```

`mra_common`: The common arguments as described in Section 17.2.1.

`mra_object`: The DN of the object you want to modify the name of.

`mra_newrdn`: The new RDN.

`deleterdn`: if `TRUE` then the old RDN will be deleted as an attribute; otherwise it will remain as a nondistinguished attribute.

17.12 Abandon

The abandon operation shown below is slightly different from the other DUA operations. It does not make much sense for a synchronous interface to handle abandon directly. In the next release, transparent use of abandon will be provided.

```
ds_abandon (arg, error)
    struct ds_abandon_arg       * arg;
    struct DSError              * error;
```

The argument structure contains a single parameter supplies the operation invocation id.

```
struct ds_abandon_arg {
    int aba_invokeid;
};
```

In the synchronous interface, an operation may be abandoned by the user entering a "control–c" (unless trapped by user code). In this case, the abandon will be handled by the lower layers of QUIPU, and the outstanding operation will return. It may have results if the abandon operation failed, or may indicate "abandon successful" via the DAP error structure!

17.13 Multiple Associations

The procedural interface described so far has been based on the assumption of a connection to a single DSA. However, it is possible to make connection to multiple DSAs.

17.13.1 Multiple Binds

The bind and unbind routines needed to make multiple association are as follows:

```
dap_bind (ad, arg, error, result, addr)
    int                         * ad;
    struct ds_bind_arg          *arg;
```

```
    struct ds_bind_arg          *result;
    struct ds_bind_error        *error;
    struct PSAPaddr             *addr;

dap_unbind (ad)
    int                         ad;
```

The `arg`, `error` and `result` arguments are as defined in Section 17.5 for the routine `secure_ds_bind`.

The argument `ad` is the association descriptor of the association. It will be different for each association.

The `addr` argument is the address of the DSA you wish to contact.

17.13.2 Other DAP Operations

The other DAP operations (read, list, modify...), use the same format as already described, but with two extra integer parameters; thus

```
ds_read (arg, error, result)
```

becomes

```
dap_read (ad, id, arg, error, result)
```

where `ad` is the association descriptor returned by the bind, and `id` uniquely identifies the operation with respect to other operation on the same association.

Chapter 18

The Async DAP Procedural Interface

Originally, the procedural interface providing a representation of the X.500 DAP operations was only required to provide synchronous behaviour. It is likely that many useful applications will be written which only require synchronous representation of operations; so that having bound to a DSA, operations are constructed in some way and then the routine representing the relevant operation is called and will block until an appropriate response has been received from the DSA.

However, in more sophisticated DUAs it will be desirable to have asynchronous or non-blocking routines representing the various DAP operations, so that the DUA can construct and send operations, perform useful work whilst waiting for the operations to complete, and collect and use the responses to operations as they become available.

For the above reasons, a new set of procedure calls has been written as part of the *libdsap*(3n) library. These calls provide a representation of the X.500 operations (including the bind operation) which can be used to program asynchronous performance of operations within a DUA. This chapter describes those routines and their intended use.

To use the async DAP interface described here it is necessary to include the file *quipu/dap2.h*.

18.1 Procedure Model

18.1.1 Styles of Behaviour

The routines which it is desirable to have to represent X.500 DAP operations depends on the behaviour required within a DUA. A simple DUA may be more easily developed by expecting a synchronous behaviour from routines representing operations; DAP operations are invoked as routines which block until a response to the operation arrives and the DUA can then continue according to that response.

If, for whatever reasons, it is undesirable to block every time a DAP operation is requested, then a different behaviour is required from the routines representing operations; the operation request should be issued and the routine returned. Subsequently a routine should be available to accept a response to the request issued and process that response.

An additional twist is that there are actually three styles of behaviour which it is desirable to make available to DUA programs: a simple synchronous interface, an interruptible synchronous interface, and an asynchronous interface.

The interruptible synchronous interface is the style of behaviour described in the previous section. If an operation is called synchronously and then interrupted (a Control-C to the user interface for instance) during performance of the operation, then an abandon DAP operation should automatically be invoked on the outstanding operation and appropriate responses gathered from the DSA before returning to the user. The interruptible synchronous style of behaviour is only available for the remote operations of the directory access context (read, add entry, ...) and not for the bind and unbind operations.

Mixing of styles of behaviour is deprecated, although there are no known problems. An asynchronous operation request followed by a synchronous operation request may well get the response to either operation, which means checking the responses to all synchronous requests in the same ways as the responses to asynchronous requests must be checked, thus losing most of the advantages that make the synchronous style worthwhile in the first place.

In order to enable the incorporation of all three styles, a generalised interface has been written over which the interruptible synchronous interface described in Section 17.1 has been reimplemented. Thus, the routines in that

previous section can be thought of as a specialised restriction of the interface described here.

The generalised interface also differs, in that it does not provide automatic decoding of results and errors into particular parameters, instead returning an indication parameter which may include decoded structures within it. This means that the implementation of the old style interface using the generalised routines is not as straightforward as the above comments might suggest.

Future revisions should take on board the separation of the provision of routines which perform DAP operations from the provision of routines which provide DUA services in the construction of DAP operations (e.g., getting passwords for a bind request is a high-level DUA task; parameterisation which allows the use of pre-allocated result and error return parameters is a lower-level DUA task). All that the DAP operations should provide is the encoding of arguments, sending of operations and appropriate subsequent behaviour. Hopefully, future documentation will be produced which is structured so as to enable a clearer relationship between the different interfaces and behaviour styles provided for DAP operations.

Where each routine of the synchronous interface described in the previous chapter had a prefix of "`ds_`" (when using an implicit association identifier) or of "`dap_`" (when using an explicit association identifier), all the routines in the asynchronous interface have a prefix of "`Dap`". All operations in the asynchronous interface take an explicit association identifier.

For each of the DAP bind operations, the DAP unbind operations and the DAP remote operations (read, list, search, addentry, removeentry, modifyentry, modifyrdn and abandon) a request routine is provided. The request routine takes a parameter indicating the style of behaviour to be used (synchronous, or asynchronous for the bind and unbind operations requests; synchronous, interruptible or asynchronous for the DAP remote operations).

If the style indicated is synchronous, then the request routine will attempt to construct and send the appropriate operation and wait for a response. If the operation completes successfully (a result or an error is returned) then the routine returns `OK`. Otherwise, on failure, the routine returns `NOTOK`.

If the style indicated is interruptible, then the operation is sent and an interruptible wait initiated for a response from the DSA. If an interruption occurs (see a description of the routine `RoIntrRequest` in the rosap module for details of what constitutes an interruption and how it is indicated), then an abandon operation is sent for the outstanding operation and re-

sponses from the DSA are awaited for both the abandon operation and for the outstanding operation. Except in the case of an interruption the style of behaviour is exactly the same as the synchronous style.

If the style indicated is asynchronous, then the operation is sent and the routine returns. For the DAP bind and DAP unbind operations a retry routine is provided for each operation which when called will either complete the outstanding operation or indicate that it is not yet complete. For the DAP remote operations a single wait routine is available which will complete an outstanding operation, if there is one which can be completed, or indicate that there is no completable operation.

The value used to indicate the style of behaviour is borrowed from the "`rosap`" module:

`ROS_SYNC`: Simple synchronous behaviour: block until DSA responds.

`ROS_INTR`: Interruptible synchronous behaviour: block until DSA responds or an interruption occurs. (Untidily, the definition of the value `ROS_INTR` is currently contained in *quipu/dap2.h*.)

`ROS_ASYNC` : Asynchronous behaviour: send operation, do not block.

18.1.2 Arguments

Arguments taken by async DAP request operations are, in general, a binding (or association) identifier which identifies the association on which to invoke the operation; an operation identifier which it is the responsibility of the DUA to generate, maintain and use as a reference (e.g., for abandoning a previously requested operation); a representation of the argument for the operation; an indication return parameter through which information on the outcome of the operation can be returned to the invoker of the routine; and of course the behaviour style indicator.

18.1.3 Indications

All of the routines representing X.500 operations in this procedural interface take an indication parameter, the type of which is a pointer to a pre-allocated `DAPindication` structure.