MSRPC2 User Manual

Timothy Roscoe, Simon Crosby
and Richard Hayton

March 14, 1994

Chapter 1

Introduction

MSRPC2 is a Remote Procedure Call system designed to run over the Multi-Service Network Architecture. At the low level it is very lightweight: bindings are implicitly single-threaded and are in one-to-one correspondence with lightweight virtual circuits. Interface references are also single-protocol affairs. At the higher levels it is based on some aspects of the ANSA Computational Model and borrows a number of ideas from the DEC SRC Modula-3 Network Objects system. It also incorporates transparent optimisations for calls within a single address space (for use in shared libraries) which map invocations onto single procedure calls.

MSRPC2 was written by Simon Crosby and Timothy Roscoe as an attempt to correct many of the grosser shortcomings of MSRPC, which itself was based heavily on SUN RPC. While the wire representation of MSRPC2 is very similar (the only obvious difference being the use of procedure numbers to represent exceptions), the tale grew in the telling. There is a totally new stub compiler based on MIDDL (a minor extension of ANSA IDL), the marshalling code has been rewritten from the ground up, and the run-time system has been completely re-implemented over the POSIX pthreads interface. Other implementations to run over raw Wanda threads, both inside and outside the kernel, are anticipated.

To understand this manual, it helps to have had some experience with ANSAware and/or Modula-3 Network Objects. It is essential to have a knowledge of the interface definition language, though a working knowledge of ANSA IDL might do. You *have* to know what an RPC system looks like.

The structure of the system splits quite nicely into layers and much of this manual is organised into chapters about the various layers and the interfaces between them. Broadly speaking, from the top down:

1.1 The user level

This is the code that the user writes, whether as a client or a provider of a particular service. The key concepts are those of *interfaces* and *bindings*.

A service provider creates an interface by instantiating a particular interface type. This operation returns a handle on the interface itself (used for shutting it down), and an *interface reference* to be used for naming the interface to remote clients.

A client in possession of a reference for an interface can attempt to *bind* to it. This operation returns a binding which appears to be an abstract data type: operations on a binding are restricted to the methods it provides. These are the operations defined in the interface specification, plus ones for setting protocol-specific aspects of the binding connection, and ultimately shutting it down.

A request from a client to bind to an interface results in the creation of a similar binding at the server end. Server code also initialises state it can hold on a per-binding (per-client) basis (or refuse the bind request if it so desires). Servers can also alter protocol-specific aspects of the binding and destroy the binding themselves.

Unlike many RPC systems, there can be many interfaces active in a single server process. These interfaces can be of arbitrary types. Interfaces of the same type in a single server process can have different implementations.

1.2 The stub code

The stub code is generated from the interface specification by the stub compiler. It is the portion of the system that is specific to a given interface type, and the only portion of the system that user code needs to interact with. It consists of type definitions for use by client and server user code and library routines to create and destroy interfaces and bindings, along with the stubs which make remote invocations and exceptions look like their local equivalents.

1.3 The marshalling code

The marshalling code is also partially generated by the stub compiler; for each data type defined in an interface it provides C functions or macros for marshalling an instance of the type into a buffer, unmarshalling it into user memory, and freeing up memory which may have been allocated to hold entities whose size the user cannot know in advance. MSRPC2 allows user code considerable control over where data are unmarshalled and how the marshalling code allocates memory for them.

1.4 The runtime system

The runtime system is a library which handles all network transmission of packets, timeouts and the like. See a later section.

1.5 The wire representation

The wire representation is the format of data in the packets sent between machines in the course of an invocation. MSRPC2 uses a data representation called MSDR, which is based in on XDR used by Sun RPC.

Chapter 2

MIDDL to C language mapping

2.1 Primitive Types

Primitive MIDDL types are output by the MSRPC2 stub compiler as their IDL name in mixed case, prefixed by the string `MSRPC`. They are then mapped to system C datatypes by a file called `MSRPCMachineTypes.h`. The current definitions are nothing mysterious; the file reads:

```
typedef u_int32      MSRPCBoolean;
typedef u_int16      MSRPCShortCardinal;
typedef u_int32      MSRPCCardinal;
typedef u_int64      MSRPCLongCardinal;
typedef int16        MSRPCShortInteger;
typedef int32        MSRPCInteger;
typedef int64        MSRPCLongInteger;
typedef float32      MSRPCReal;
typedef float64      MSRPCLongReal;
typedef char        *MSRPCString;
typedef u_char       MSRPCOctet;
typedef char         MSRPCChar;
```

–where `u_int32`, etc. are defined elsewhere in a machine-dependent file.

2.2 Interface References

The basic interface reference type is defined in the base types interface (see below). An interface reference is defined in MIDDL as:

```
MSRPCInterfaceRef : TYPE = RECORD [
        id      : MSRPCInterfaceID,
        attrs   : MSRPCAttributes,
        hint    : MSRPCAddressHint
];
```

–which maps to a C definition of:

```
typedef struct MSRPCInterfaceRef MSRPCInterfaceRef ;
struct MSRPCInterfaceRef {
    MSRPCInterfaceID id ;
    MSRPCAttributes attrs ;
    MSRPCAddressHint hint ;
} ;
```

1. The `id` is supposed to be an identifier for the interface that is unique in time in space. It is intended to be checked at bind time to allow a client to be sure that it is talking to the interface it thinks it is. It can be regarded as "the Truth".
2. The attrs field contains information about which endian the server uses for data, and whether to perform an end-to-end check on the interface id at bind time.
3. The address hint is a discriminated union with a member for every protocol supported by MSRPC, including the local (same protection domain) optimisation. Its members are protocol specific addresses, with possible additional information such as Quality of Service hints.

Any interface reference should be regarded as nothing more than a hint as to how to bind to a service. If a bind fails it may be that the interface is still extant and even accessible, but other means may be required in order to find it, such as an additional Trader operation.

It is useful to examine the `MSRPCBaseTypes` interface definition and the files it includes, since they contain a fair number of useful types for representing protocol-specific addresses and may also change as new protocols are added.

2.3 Constructed Types

The 7 MIDDL type constructors map in a fairly natural way onto (slightly less safe) C types as follows. They are described by examples, many taken from the base types interface.

2.3.1 Aliases

```
MSNLQosSpec : TYPE = STRING;
```

–maps to:

```
typedef MSRPCString MSNLQosSpec ;
```

2.3.2 Arrays

```
MSRPC2InterfaceID : TYPE = ARRAY 8 OF OCTET;
```

–maps to:

```
typedef MSRPC2Octet MSRPC2InterfaceID[8];
```

2.3.3 Enumerations

```
MSRPCEndian     : TYPE = { Big, Little };
```

–maps to:

```
typedef enum {
    Big ,
    Little ,
} MSRPCEndian ;
```

Note that all enumeration types are guaranteed to start at 0. This is important for mapping of the power set type, described below.

2.3.4 Sequences

```
Crawl : TYPE = SEQUENCE OF Bar ;
```

–maps to:

```
typedef struct Crawl Crawl;
struct Crawl {
        MSRPCCardinal len;
        Bar          *data;
};
```

Sequence and strings are the only variable-length data types in MSRPC2. This has implications for unmarshalling, since the user cannot easily allocate space for them in advance as she can for other data types. MSRPC2 provides a facility for the user to control memory allocation for storage of unmarshalled sequences and strings, described later.

2.3.5 Records

```
MSRPCAttributes : TYPE = RECORD [
        endian  : MSRPCEndian,
        check   : BOOLEAN
];
```

–maps to:

```
typedef struct MSRPCAttributes MSRPCAttributes ;
struct MSRPCAttributes {
    MSRPCEndian endian ;
    MSRPCBoolean check ;
} ;
```

2.3.6 Choices

```
MSRPCProtocol : TYPE = { MSRPC_PROTO_MSNL,
                         MSRPC_PROTO_TCP,
                         MSRPC_PROTO_SDOM };

MSRPCAddressHint : TYPE = CHOICE MSRPCProtocol OF {
        MSRPC_PROTO_MSNL        =>      MSNLAddressHint,
        MSRPC_PROTO_TCP         =>      TCPAddressHint,
        MSRPC_PROTO_SDOM        =>      SDomAddressHint };
```

–maps to:

```
typedef enum {
    MSRPC_PROTO_MSNL ,
    MSRPC_PROTO_TCP ,
    MSRPC_PROTO_SDOM ,
} MSRPCProtocol ;

typedef struct MSRPCAddressHint MSRPCAddressHint ;
struct MSRPCAddressHint {
    MSRPCProtocol d ;
    union {
        MSNLAddressHint MSRPC_PROTO_MSNL ;
        TCPAddressHint  MSRPC_PROTO_TCP ;
        SDomAddressHint MSRPC_PROTO_SDOM ;
    } u ;
} ;
```

2.3.7 Power Sets

```
Bar : TYPE = SET OF Beers ;
```

–maps to:

```
typedef set_t Bar ;
```

–which is a generic type to represent sets. Since a `set_t` is actually an unsigned integer, to get information about a set type you can use the enumeration value and a shift operator to mask bits. This works because enumeration values start at 0 and is the intended mechanism for manipulating set types (indeed, set types were introduced into MIDDL as an abstraction of bitmasks).

There is currently a restriction on the maximum cardinality of a set type of 32 elements (for obvious reasons). This may be increased to 64 in the near future (for equally obvious reasons).

2.3.8 InterfaceRefs

Typed interface references are currently treated as type aliases.

2.4 Subtyping of Interfaces

There is *no* inheritance in MSRPC2.

If a MIDDL interface imports datatypes by means of the `NEEDS` or `IS COMPATIBLE WITH` directives, there is no duplication of type definitions in the C files. It remains sufficient, however, to only include the type and marshalling header files from the interface one is interested in, as this will include all the type files it needs. All header files output by the stub compiler have C preprocessor directives to ensure that they are only parsed once.

Operations and exceptions, on the other hand, are always defined from scratch for a given interface. It would in theory be possible to re-use stub routines from supertype interfaces (as the Modula-3 Network Objects implementation over ANSA does), but doing this in C would blow away what little type safety the bindings system gives you, and the stubs are very small anyway. Of course, the user level method functions can be re-used trivially.

Every interface in MSRPC2 is implicitly compatible with (in the MIDDL sense) an interface called `MSRPCBaseTypes`. This interface defines the engineering failure exception and the `MSRPCInterfaceRef` type, along with all the protocol-specific information it contains in the address hint.

2.5 Interrogations and Announcements

Operations in MIDDL are mapped onto function pointers which are members of binding structures (see bindings below). A new C type is defined for each MIDDL operation, which is a pointer to a C function returning a boolean value. Under ANSI C, these function pointers are prototyped and should take as parameters a pointer to the binding being used, followed by the parameters of the operation, followed by the results the operation returned passed by reference.

Any parameters of type RECORD, ARRAY or CHOICE (or types which are aliases to these types) are passed by reference for efficiency. Note, when used as arguments, SEQUENCES are passed by value, as they map to C structures which are more efficiently passed in this way.

Suppose a MIDDL interface called `Beer` defined an operation like:

```
BuyRound : INTERROGATION OPERATION [ num : INTEGER; order : Bar ]
              RETURNS [ pints : CARDINAL ]
              RAISES WrongOrder, TimeCalled ;
```

Then the stub compiler would output a type definition:

```
typedef (bool_t)(*Beer_BuyRound) _((
        Beer_Binding  *self,
        MSRPCInteger   num,
        Bar            order,
        MSRPCCardinal *pints ));
```

–and a `Beer_Binding` structure would contain a member called `ops.BuyRound` of this type.

This happens on both client and server sides, so a client would call an instance of this function type and the server would implement one. Indeed, in the case of a local (same domain) binding, the client would directly call the server routine.

2.6 Exceptions

Exceptions in MIDDL are mapped onto C functions in the server, and to an exception structure in the client. The server calls the exception routine which results in the arguments to the exception being unmarshalled into an exception structure in the client. These can then be accessed by an in-line exception handler, as described in a later section.

Thus if the `Beer` interface defined an exception:

```
WrongOrder : EXCEPTION [ who : CARDINAL ;
                         msg : STRING ] ;
```

–the stub compiler would output a type definition for the server of:

```
typedef (bool_t)(*Beer_WrongOrder) _((
        Beer_Binding   *self,
        MSRPCCardinal   who,
        MSRPCString     msg ));
```

–and a `Beer_Binding` structure would contain a member called `excs.BuyRound` of this type. The binding exception structure structure would also contain a union of all possible exception results, for this example this would include

```
struct Beer_Binding
{ ...
   struct { ...
           union {
                   struct {
                            MSRPCCardinal who;
                            MSRPCString   msg;
                          } WrongOrder;
                   ...
                 } u;
          } excs;
};
```

The binding structure also contains two fields used to signal which of the exceptions occurred (if any). In the present implementation this consists of a pointer to a table of exceptions names, and a pointer to the entry signifying the exception that occured. For gcc this allows exception name matching to be done by pointer comparison, for ansi-c compilers, string comparison is used between exception names.

The interface called `MSRPCBaseTypes` defines an exception:

```
InternalError : EXCEPTION [ status : CARDINAL ;
                            err_no : INTEGER ];
```

This exception is used for signalling what ANSA call "engineering failures": internal errors in the MSRPC2 infrastructure and the like. It is implicitly the first operation of any kind defined in an interface.

Chapter 3

Action of the stub compiler

The MSRPC2 stub compiler is constructed around the `middlc1` front end for MIDDL, and a set of Python modules to implement the back end. It takes as input a MIDDL file, and outputs a grand total of 8 files. An input file called `Aubergine.idl` would result in the following being generated, described below:

- `Aubergine.sif`, the MIDDL signature file.
- `t_Aubergine.h`, the MSRPC2 type definitions.
- `m_Aubergine.h`, the MSDR marshalling code header file
- `m_Aubergine.c`, the MSDR marshalling code.
- `s_Aubergine.h`, the MSRPC2 server function prototypes.
- `s_Aubergine.c`, the MSRPC2 server stub code.
- `c_Aubergine.h`, the MSRPC2 client function prototypes.
- `c_Aubergine.c`, the MSRPC2 client stub code.

All C header files generated by the stub compiler are protected from multiple inclusion by C preprocessor macros.

3.1 The Signature File

The `.sif` file is generated by the front end of the MIDDL compiler, and is used for keeping track of interface subtyping and type imports. It does not form part of any MSRPC2 code.

3.2 The Type Definition Header File

The type definition file contains the following:

- Include directives for the type header files for all interface types that this one is compatible with or `NEEDS`.

- A definition of the type of references to instances of this interface type. All interface references are currently defined as `MSRPCInterfaceRef`s. Given the state of C's type system, this probably doesn't matter.
- Definitions of C types for all the constructed types defined in the interface type. Note that imported types are catered for by the include directives.
- Definitions of C types for all the operations and exceptions exported by the interface type. These are defined as pointers to prototyped functions.
- The definition of the binding structure for the interface type.
- The definition of the interface structure for this interface type.
- Definitions of the user state constructor and destructor functions for interfaces of this type.

3.3 The Marshalling Code Header File

The marshalling code header file contains the following:

- An include directive for the corresponding type definition file.
- Include directives for the marshalling header files for all interface types that this one is compatible with or `NEEDS`.
- For each MIDDL constructed type (including the type-specific interface reference), either a set of macros or a set of function prototypes to:
  - Marshal an instance of the data type
  - Unmarshal an instance of the data type
  - Free any memory allocated during the unmarshalling of an instance of the data type.

  For details of when macros are used instead of real function, see the chapter on MSDR in this manual. Basically they are used wherever the obvious optimisation can be made.

3.4 The Marshalling Code Source File

The marshalling code source file contains the following:

- An include directive for the MSRPC2 system header file.
- An include directive for the MSDR code header file corresponding to this source file.
- For each MIDDL constructed type (including the type-specific interface reference) which is not marshalling by means of a set of macros in the header file, C source for the MSDR functions to:
  - Marshal an instance of the data type

- Unmarshal an instance of the data type
- Free any memory allocated during the unmarshalling of an instance of the data type.

3.5 The Server Function Prototype File

The server header file contains the following:

- The declaration of the function to create an interface of this type.
- The declaration of the function to destroy an interface of this type.

3.6 The Server Stub Source File

The server stub file contains the following:

- An include directive for the MSRPC2 system header file.
- An include directive for the MSDR code header file corresponding to this interface. Note that this includes the type definitions.
- An include directive for the corresponding server header file.
- The function to create an interface of this type.
- The function to destroy an interface of this type.
- The internal binding control method function.
- The internal routine to create a binding to an interface of this type.
- The internal binding destroy method function.
- Server stub functions for all exceptions raised by interfaces of this type.
- Server stub functions for all interrogations and/or announcements exported by interfaces of this type.
- The internal server dispatcher function

3.7 The Client Function Prototype File

The server header file contains the following:

- The declaration of the function to bind to an interface of this type.

3.8 The Client Stub Source File

The client stub file contains the following:

- An include directive for the MSRPC2 system header file.
- An include directive for the MSDR code header file corresponding to this interface. Note that this includes the type definitions.
- An include directive for the corresponding client header file.
- Client stub functions for all exceptions raised by interfaces of this type, for each exception there are two functions, a single domain version and a generic version.
- The internal client exception dispatcher function
- An array of exception names. This is to aid the ansi-c version of the TRY–ENDTRY block. There is a gcc specific optimisation which does not require this.
- The function to bind to an interface of this type.
- The function to destroy a binding to an interface of this type.
- The internal binding control method function.
- The internal routine to create a binding to an interface of this type.
- The internal binding destroy method function.
- Server stub functions for all interrogations and/or announcements exported by interfaces of this type.

Chapter 4

Bindings and Interfaces

4.1 Interfaces

In MSRPC2 terms, an *interface* represents an instantiation of an interface type. The operations supported by an interface and the types of parameters they take are given in the MIDDL specification for the interface type. Once created, an interface can support many active clients, each one having made a *binding* to the interface.

In implementation terms, MSRPC2 provides mechanisms for creating an interface for a particular type, managing it, and eventually destroying it. An interface called `Aubergine` would be represented at the stub level by the following C structure:

```
typedef struct Aubergine_Interface {
    struct {
          ...
    } ops;
    bool_t (*bind) (  Aubergine_Interface  *self,
                      Aubergine_Binding   **b,
                      caddr_t               sess );
    svcInterface         *svcIntf;
    Aubergine_Constructor userBind;
    Aubergine_Destructor  userDestroy;
    caddr_t               userState;
}
```

The `ops` structure contains pointers to the C functions which implement the operations of the interface. It has a reference to a corresponding structure inside the runtime system, and two methods which are called to create a new binding to the interface and destroy an existing one.

The implementer of the interface is provided with a pointer to interface-specific state, which he or she can use in an arbitrary way, and two methods to create and destroy binding (client) specific state when bindings are created and destroyed.

4.2 Bindings

In MSRPC2 terms, a *binding* represents a logical connection between a client and a server. To the client, the binding represents an object which performs the server's operations.

In MSRPC2 bindings are inherently single threaded. If a client wishes to make multiple concurrent requests to a server, then should first obtain multiple bindings.

A binding structure contains information on all operations and and exceptions used by it. Some of this state is client specific, and some server specific, but most is shared. In the same domain case, the binding structure is shared between the client and sever which optimises RPC calls and exception returns to a single indirect procedure call.

An binding called `Aubergine` would be represented at the stub level by the following C structure:

```
struct Aubergine_Binding {
        void   (*destroy) _(( Aubergine_Binding *self ));
        bool_t (*control) _(( Aubergine_Binding *self,
                              u_int32 op, caddr_t param ));
        void (*release) _((Aubergine_Binding *self ));
        Aubergine_Interface *intf;
        msdr_mm mm;
        caddr_t session;
        caddr_t userState;
        struct {
                Aubergine_enames_t *elist;
                char **name;
                int retries;
                union {
                        struct {
                                MSRPC2Cardinal  status;
                                MSRPC2Integer   err_no;
                        } InternalError;
                       ...
                } u;
                Triv_InternalError      InternalError;
                ...
        } excs;
        struct {
          ...
        } ops;
};
```

`control`, `destroy` and `release` are server specific functions and are described in the next section. `ops` contains pointers to the procedures implementing the operations of the binding. The `excs` structure contains pointers to functions implementing each of the exception dispatchers (for the server) and a union of structures of exceptions arguments (for the client). It also contains three fields used by the TRY block.

Chapter 5

User Code Interface, Server Side

5.1 Creation of interfaces

An interface type in MSRPC2 is instantiated via a function of the form:

```
bool_t  Aubergine_Interface_Create(
        Aubergine_Interface    **intf,
        MSRPCInterfaceRef       *ref,
        Aubergine_Binding_Constructor  bind,
        Aubergine_Binding_Destructor   destroy,
        PFC            allocFunc,
        PFV            freeFunc,
        caddr_t        state,
        Aubergine_Op1  Op1,
        ...
        Aubergine_Opn  Opn )
```

The function returns true if it has succeeded. The parameters are:

- `intf` is the address of a pointer to the interface structure itself. The structure is `malloc`ed (using the real `malloc`, not `allocFunc`. If the function succeeds, `*intf` will point to the interface.

- `ref` is a pointer to an interface reference. The reference should already partially exist; in particular the type of address hint should be specified since the run time library needs to know this to offer the service. If the function succeeds, the reference address hint will be completed and the interface identifier will be filled in. The interface ref can then be marshalled and passed to other entities.

- `bind` and `destroy` are pointers to functions the functions to construct and destroy user-defined per-binding state for the interface. Either or both of them can be null, in which case it will not be called at the appropriate moment during binding construction or destruction (see the various "creation of bindings" sections below).

- `allocFunc` and `freeFunc` are pointers to functions with the same signatures as `malloc` and `free`. They are only used to allocate and free memory when unmarshalling strings and sequence types on the server side of the interface. They default to the real `malloc` and `free` if null. They are present to allow users to optimise memory management in the marshalling code, for example by allocating fixed-size buffers extremely fast or ensuring that variable-length data is unmarshalled into a special area of memory.

- `state` is a user-defined pointer to per-interface state. This is available in the interface structure. Typically it should be created before the interface itself is created and disposed of before the interface is destroyed.
- The final parameters are pointers to the methods of the interface, i.e. the functions which are called when a client invokes an operation.

5.2 Destruction of interfaces

For each interface type in MSRPC2 there is a destructor function:

```
void Aubergine_Interface_Destroy(
        Aubergine_Interface *intf )
```

The sole parameter is a pointer to the interface to destroy. There is no way to destroy user-defined per-interface state (given as the `state` parameter when the interface was created, so this should be disposed of before this function is called.

When this function is called, all bindings are destroyed and then the interface is dismantled. Any invocations on the interface which are already in progress are allowed to complete, but no new bindings can be created. The function only returns when all the bindings have been disposed of cleanly.

5.3 Creation of bindings

When a client attempts to bind to the interface, the stub code will attempt to create a new binding. The user code will not be notified of the new binding being created unless it has registered a state constructor (a method called `userBind`) in the interface structure. If this exists, the thread handling the binding will call the constructor giving the interface, the new binding, and an address hint which will typically contain information about the client. The `userBind` routine should initialise any state it needs, set the `userState` field of the binding and return the boolean `TRUE`. If `FALSE` is returned, the binding is assumed to have been rejected and the client side bind call will return a failure code.

The constructor may also set the `release` method of the binding, which will be called after the result of an RPC has been dispatched, but before the server thread exits. This will typically be used to free any buffers the server routines has allocated for the RPC.

5.4 Destruction of bindings

A Binding can be destroyed by calling its `destroy` method. This must be done by the thread handing the binding, i.e. it can only be called from within an invocation.

When a binding is destroyed, the user code implementing the interface is called via the `userDestroy` method, if present. The user code should clean up all per-client state to prevent memory leaks and return. The thread calling this method is guaranteed to be the thread handling the binding that is to be destroyed. After the call returns the binding will not exist.

5.5 Invocations

When a client invokes an operation on a binding, the thread handling the binding calls the C function for the operation that was specified when the interface was created (i.e. relevant member of the `ops` structure in both the interface and server binding structures.

This function takes as parameters a pointer to the binding and the parameters and results listed in the interface specification. It should perform the service.

The function returns a `bool_t`. If this is `TRUE`, it implies that the operation was in some sense successful that the results can be dispatched. If `FALSE` is returned, it implies that the operation failed in some way and that an exception has been raised.

To raise an exception within the user routine implementing an operation, simply call the relevant member of the `excs` structure in the binding, and then return whatever boolean value that function returned.

5.6 Control parameters

MSRPC2 provides a call which allows bindings to tweak protocol-specific parameters such as timeouts. Each binding structure has a method which looks like:

```
(bool_t)(*control)( Aubergine_Binding *b,
                    u_int32   op,
                    caddr_t   param )
```

As bindings are inherently single-threaded, this call must only be invoked by the thread handling the binding. `op` is an operation code whose meaning depends on the type of protocol the interface is using and is determined by the runtime system. `param` is a generic parameter whose meaning is determined by the value of `op`.

To set the protocol parameters for a binding to values other than the runtime library defaults, register a user binding constructor with the interface which makes the appropriate calls to `control` (see `userBind` above). This is guaranteed to be invoked when the protocol session structure and binding have been created, and runs in the binding thread. It is also the only time that `control` can be invoked outside an actual operation.

5.7 Local-case differences

A key aim of the local case version of MSRPC2 is that looks pretty much the same as the remote one. You create and destroy interfaces in the same way, and the same events occur when bindings are created and destroyed.

However, in the local case there are no threads waiting to service requests: invocations optimise right down to single function calls. This means that withdrawing a service is quite dangerous: no checks are made on existing bindings, indeed there is no information on which bindings exist. Withdrawing services should be done with caution. The same goes for destroying a binding from the server side. As the binding structure is shared between the client and server, problems will arise if both try and use the userState field for their own purposes.

There is currently a real problem involving local bindings and subtyping. Suppose that interface type `Aubergine` is a subtype of the `Vegetable` interface (Aubergines are technically fruit, but in practice they conform to vegetables). If a client does a

```
Vegetable_Binding_Create
```

to an Aubergine interface, the client gets a binding of type `Vegetable` and the server gets one of type `Aubergine`. In the local case, the client gets an `Aubergine_Binding` cast to a `Vegetable_Binding`. This would be fine if the methods matched up. They don't. Beware. This should be fixed in a revision of MSRPC2. It's not a showstopper. Control functions are ignored by local bindings. Attempts to invoke the control function for a local interface will return `FALSE`.

Chapter 6

User Code Interface, Client Side

6.1 Creation of bindings

A client of an MSRPC2 interface binds to the interface by means of a call like:

```
bool_t Aubergine_Client_Bind (
        Aubergine_Binding  **b,
        MSRPCInterfaceRef  *ref,
        PFC          allocFunc,
        PFV          freeFunc,
        excpt_t     *exc)
```

The function returns `TRUE` if it succeeds. The parameters are:

- `b` is the address of a pointer to the binding structure. The structure is `malloc`ed (using the real `malloc`, not `allocFunc`. If the function succeeds, `*b` will point to the new binding.
- `ref` is a pointer to the interface reference for the interface to be bound to.
- `allocFunc` and `freeFunc` are pointers to functions with the same signatures as `malloc` and `free`. They are only used to allocate and free memory when unmarshalling strings and sequence types on the client side of the binding. They default to the real `malloc` and `free` if null. They are present to allow users to optimise memory management in the marshalling code, for example by allocating fixed-size buffers extremely fast or ensuring that variable-length data is unmarshalled into special area of memory.

6.2 Destruction of bindings

As on the server side, a binding can be destroyed by calling its `destroy` method. Again, as bindings are single-threaded, there should not be an invocation in progress over the binding. The binding pointer will be invalid after the call.

6.3 Invocations

To invoke an operation on a binding, simply call the relevant member of the `ops` structure of the binding. The parameters of this method are:

- A pointer to the binding itself,
- Each parameter of the operation, and
- A pointer to each result of the operation.

This will return a boolean; `TRUE` indicates that in some way the operation went OK and it's fine to carry on. `FALSE` means that something went wrong, and exceptions results have been returned. (see below).

Note that MIDDL `MUTABLE` parameters are not supported by MSRPC2 (at present).

6.4 Exceptions

A TRY–ENDTRY set of macros has been implemented to aid exception handing. For efficiency, these are not implemented using setjmp and longjmp, but instead use gotos. The disadvantage is a slight clouding of syntax. The exception mechanism does not pretend to be general purpose, and in particular no notion of stack unwinding is provided.

The syntax of the TRY block is as follows:

```
TRY(<label>,<binding>,<operation>)
{
  CATCH(<exceptionname>)
      <handling code>
  ...
} ENDTRY;
```

`label` is a user provided label name that is used to allow retries. All exceptions *must* be caught, but maybe ignored by using the macro `SUCCEED(<label)` from within the handling code. It is a fatal run time error to fail to catch an exception. The statement `CATCHALL` may be used as the last catch statement, and matches all exceptions not mentioned in a catch block. There is no restriction on the style of C programming using within the handling code, but control must not drop through to the next CATCH statement. (In particular 'break', 'continue' and 'return' can all be used and will apply to the enclosing structures). The following macros are provided for use within the handling code.

`FAIL(<label>)` Raise the generic pseudo-exception Failure, which must be handled within the same TRY block. This is intended as a generic fail point for use by library handler routines.

`RETRY(<label>)` Try the RPC again. This is equivalent to a jump to the line containing the TRY statement, so arguments to the RPC will be re-evaluated.

`SUCCEED(<label>)` Jump to beyond the ENDTRY.

`HANDLER(<label>,procedure call)` This calls the named procedure, and allows it to perform RETRY, FAIL or SUCCEED calls (or just return). Due to the limitations of C macros, it must use the commands hRETRY, hFAIL, hSUCCEED and hRETURN respectively (with no arguments). The procedure implementing the handler should be declared to return and integer.

In addition to these macros, the macro `RETRIES(<binding>)` may be used within a TRY block, and returns the number of retries (so far) on the operation that raised the exception. Within the handling code, the arguments to the exception are available via the excs structure of the binding. If variable length, it is the handlers responsibility to free any memory allocated for them. See the section on memory below.

As pthreads also provides an exception mechanism (which is unsuitable for use by MSRPC2), the macros TRY, CATCH etc are already defined for use by pthreads. Client code can either precede each of the macros by `msrpc2_`, #define the macro `USE_MSRPC2_EXCEPTIONS` before including `msrpc2.h`. In the latter case, pthreads own exception mechanism cannot be used.

An example TRY block is as follows;

```
TRY(mylab,b,Read(b,fid,offset,len,&data))
{
  CATCH(InternalError)
    printf("Internal Error status=%d\n",b->excs.u.InternalError.status);
    FAIL(mylab);
  CATCH(ReadBeyondEnd)
    len=b->excs.u.ReadBeyondEnd.actual_len;
    SUCCEED(mylab);
  CATCH(Access)
    HANDLER(mylab,handle_accessexpt(b->excs.u.Access.reason));
  CATCH(Failure)
     exit(1);
} ENDTRY;
```

6.5 Control parameters

As on the server side, the `control` method allows protocol-specific parameters of a binding such as timeouts, etc. to be set. This method must not be invoked concurrently with any invocation over the binding. `op` is an operation code whose meaning depends on the type of protocol the interface is using and is determined by the runtime system. `param` is a generic parameter whose meaning is determined by `op`.

6.6 Local-case optimisations

In the remote case, the `ops` methods of the binding object point to stub routines which marshal the arguments, send the data, receive the reply, unmarshal the results, and optionally raise exceptions in response to the reply received or local "engineering failures". In the local case, the methods point to the real server routines. Only one indirection is required to call a server routine.

In fact, in the local case there is only one binding object, shared by client and server. The session field is null (the run-time library is never used for local bindings).

Client binding is done by directly calling the `bind` method of the *interface* structure (see below). The address of this structure in memory is obtained from the address hint in the InterfaceRef. `bind` is called with a null session argument to indicate a local binding. The server binding constructor builds the binding and the client bind routine then fills in the exception routines as normal. Thus raising exceptions also optimises down to a single function call.

Control methods on a local binding always return `FALSE`.

6.7 Memory

If an RPC call returns variable length results (or exceptions) then the returned memory 'belongs' to the caller, and it is his responsibility to free it once it is no longer required. Help is provided in the form of msdr macros to free types declared in the idl file. These required a pointer to a `mm` structure which contains details of the relevant memory freeing and allocating functions. A copy of the mm structure is stored in the binding for the clients use. This is also available for server code.

For example

```
f_msdr_MSRPC2String(& b->mm,barman);
```

Chapter 7

Stub Internals, Server Side

7.1 Creation of interfaces

<*Interface*>`_Interface_Create` first `malloc`s the structure and fills in the operation methods, along with `userBind` and `userDestroy`. It also sets the `bind` method to the internal `S_Binding_Create` routine.

In the case of a local interface (divined by grubbing around in the partial interface reference), this is all there is to do. The stub sticks the address of the interface structure into the interface ref address hint and returns.

For remote interfaces the stub calls `svc_offer` to offer the interface and fill in the interface reference.

7.2 Destruction of interfaces

Destruction of interfaces in very simple. <*Interface*>`_Interface_Destroy` first checks to see if the interface's `svcIntf` field is non-null, indicating a remote interface. If this is so, it calls `svc_withdraw` to remove the service. Once this returns, it simply `free`s the structure.

7.3 Creation of bindings

There is an internal stub routine for each interface which is the sole constructor for any binding to the interface. Its prototype is:

```
static bool_t S_Binding_Create ( Aubergine_Interface *self,
                                 Aubergine_Binding  **b,
                                 svcSession          *s )
```

The address of this routine is held in two place: it is the `bind` method of the binding object, and it is also passed to the runtime library during interface creation and is associated with the `svcInterface` This means that it can be called both directly by user-level server code or by the runtime library in response to a remote bind request.

When an incoming bind request arrives in the runtime library, it creates a server session object (an `svcSession`) and a thread to handle requests. This thread then calls the stub level constructor with a pointer to the session. The stub builds a stub-level binding structure, calling user-level initialisation code if necessary, and then returns `TRUE` with the binding pointer filled in.

If the stub returns `FALSE`, something has gone wrong and there is no state at the stub or user levels. The runtime library thus deletes the session and the thread terminates.

7.4 Destruction of bindings

There is an internal stub routine for each interface which is the sole destructor for any binding to the interface. Its prototype is:

```
static void S_Binding_Destroy ( Aubergine_Binding *b )
```

The address of this routine is held in two place: it is the `destroy` method of the binding object, and it is also passed to the runtime library during interface creation and appears as the `destr` method of the run-time server session. This means that it can be called both by user-level server code or by the runtime library. *It should only ever be called by its binding's thread.*

What it does is first to call the server session's `destroy` method, if the session exists. This completely shuts down and frees the session from the runtime library point of view. Note that for the runtime system to destroy a binding, it must call its `destr` method, and never invoke `destroy` itself, so as to allow the stubs and user code to free up their resources.

Next it calls the `userDestroy` method, if present, then frees up the binding structure itself and returns. The binding is now totally gone.

7.5 Operation dispatching

There is one operation dispatcher in the stubs for an interface. This is called by the runtime system whenever a request comes in on a binding, and looks like:

```
static bool_t S_Dispatch( Aubergine_Binding *b,
                          excpt_t           *exc,
                          MSDR              *msdrs,
                          u_int32            procno )
```

Note that neither the dispatcher, nor any operation stub routines, are involved in an invocation across a local binding.

- `b` is the binding on which the request has come in on. The dispatcher is always executed by the thread handling the binding.
- `exc` is the exception structure used to pass exceptions back down to the runtime system so that they can be returned to the client.
- `msdrs` is set up such that arguments associated with the operation can be immediately unmarshalled.

- `procno` is the procedure number of the operation.

The dispatcher is nothing more than a case statement which calls the operation stubs. Procedure numbers that are out of range or that refer to exceptions as opposed to operations raise an `InternalError` exception which is returned to the client by filling in the `exc` structure and returning `FALSE` to the runtime.

7.6 Operation stub routines

An operation stub routine prototype looks like:

```
static bool_t S_BuyRound( Beer_Binding *self,
                          excpt_t      *exc,
                          MSDR         *msdrs )
```

The meanings of the parameters are exactly as for the operation dispatcher.

The stub routine simply unmarshals the arguments, raising an `InternalError` exception if it goes wrong and calls the relevant `ops` field of the binding structure.

In the case of an interrogation, if the user routine (the one supplied in the binding structure) returns `FALSE`, it is assumed that the user has called an exception-raising routine (see below) to signal an exception. In this case the buffer specified by `msdrs` has been loaded with the parameters of the exception and is ready to go, so the stub simply returns `TRUE` to the calling runtime routine to indicate that the buffer can be sent.

If the user returns `TRUE`, then the server implementation is assumed to have terminated normally. The stub invokes the session method `reply_init` with an exception number of zero to signal successful termination and the results are marshalled. The routine returns `TRUE` to send the packet off.

If anything else happens, then `exc` will contain the details of the failure, filled in either by the stub or the runtime library. `FALSE` is returned to the runtime which will marshal it into an `InternalError` exception and send it off.

In the case of an announcement, the user routine is assumed to be of void return type. The stub simply returns `TRUE`, and the runtime takes no further action.

7.7 Operation terminations

An operation termination is either successful completion, or the raising of a parametrised exception. User routines signal successful completion by returning `TRUE`, as described above. User-defined exceptions are signalled by the user returning the results of the relevant exception method of the binding.

Note that it is possible for a user-routine to signal a fake engineering failure, since there is a `InternalError` method available. This is discouraged, because it's pretty confusing apart from anything else. However, it could conceivably be used for RPC-level protocol converters.

The exception stubs being methods of the binding have signatures determined by the parameters of the exceptions themselves as given in the interface specification. In fact, they look a lot like client side announcement stub routines. They simply call the `reply_init` method of the session, marshal their arguments, and return `TRUE`. If any engineering failures occur the routine returns false; enough state is left in the session structure for the runtime to marshal and transmit the correct `InternalError` exception.

It is important to realise that the stubs' semantics are the same as for the local case exception handlers. Though the meanings of `TRUE` and `FALSE` as returned by a user operation routine may at first sight seem confusing, they ensure consistency across remote and local bindings.

7.8 Control operations

The `control` method of any server-side binding is bound to a routine that looks like:

```
static bool_t S_Binding_Control( Aubergine_Binding *b,
                                 u_int32           op,
                                 caddr_t           param )
```

–which simply acts as a filter for the `control` method of the session structure: if the session structure is there, it calls it. Otherwise, it returns false.

7.9 Local-case optimisations

Interface creation in the local case does not involve a call to `svc_offer`, instead the address of the newly created interface is stuffed into the interface reference. Destroying an interface just frees it. Care must be taken in this case with outstanding client bindings, as detailed previously.

Construction of bindings is pretty much as in the remote case, except that exception stubs are not filled in. A local bind is indicated by a null session pointer in the `bind` call.

Destruction of bindings on the server side is the same as the remote case, except that the binding destructor will not attempt to destroy a null session.

Chapter 8

Stub Internals, Client Side

8.1 Creation of bindings

The <*Interface*>`_Client_Bind` function does pretty much the obvious thing. The first thing is to check inside the interface ref to see if the interface is local.

Local bindings are created by calling the `bind` method on the *interface* itself. The address hint in the interface ref provides the address of the interface structure. If this works, the exception handlers are written in and the operation is complete.

For the remote case the binding structure is `malloc`ed. The exceptions are filled it, as are the operation stub routines, which are called `C_`<*Operation Name*>. The special methods `C_Binding_Destroy` and `C_Binding_Control` are also filled in.

Finally `clnt_bind` is called to create the binding.

8.2 Destruction of bindings

The stub routine:

```
static void C_Binding_Destroy( Beer_Binding * self )
```

is bound to the `destroy` method of a binding at bind time. If the binding is a true remote one (indicated by a non-null `session` field, it calls the `destroy` method of the client session block. It then `free`s the binding structure.

8.3 Operation invocation

Operation methods for local clients are simply the server routines themselves. For remote sessions they are bound to internal routines, for example:

```
static bool_t C_LastOrders( Beer_Binding *self, MSRPCBoolean time )
```

These stubs work by calling methods on the `clntSession` structure, passing the session itself along with an `MSDR *` for holding marshalling state and an `excpt_t` for signalling failures. If any session method returns false, the stub simply calls the exception dispatcher to signal an engineering failure.

The first method called is `call_init` to set up a packet for marshalling into. The arguments to for the call are then marshalled in.

Announcements and interrogations are handled differently since in the announcement case the run time library should not be expecting any return. Thus when marshalling has been successfully completed the stub either invokes the `announce` or `call` method to send the packet.

In the case of an interrogation the arguments are simply unmarshalled before returning.

8.4 Exception dispatching

Each client interface stub has a single exception handler:

```
static bool_t C_Exception_Dispatch( Beer_Binding * self, excpt_t * exc )
```

–which is called whenever the rest of the stub detects something may have gone wrong. This includes user-defined exception conditions from the server. `exc` will contain the details of the exception (its procedure number, which is always 1 for engineering failures), plus a status code and error number which only have meaning for engineering failures. A simply case statement calls an internal exception stub for each exception. These have the same naming convention as operation stubs, and the same signature as the dispatcher itself. It is their responsibility to unmarshal exception arguments and set the exception name pointer in the binding.

Bad exception procedure numbers raise an `InternalError` exception in the dispatcher.

Note that local exceptions are not dispatched; special same domain stubs are invoked directly through the binding structure. these fill in the binding excs structure exactly like the normal stubs (except that the arguments do not need to be unmarshalled).

8.5 Local-case optimisations

See the same section for the server side, above.

Chapter 9

Runtime library interface

This chapter describes the structure of the MSRPC2 runtime system which supports the MSRPC2 system for all protocols which involve Inter-Process Communication (IPC). The runtime system implements the RPC mechanism which forms the basis of MSRPC2, including service, session and protocol specific communication management. This chapter defines the interface to the MSRPC2 runtime library, and describes its functionality. The goal of this is to facilitate the addition of new functionality to to the runtime library by those unfamiliar with its structure.

9.1 Introduction

Before describing the MSRPC2 runtime system, a few definitions and some motivation are required. These are given below. Thereafter the runtime system interface is described, first by addressing the IPC protocol independent layer, and thereafter a specific implementation of a protocol dependent layer – for MSNL.

Like MSNL, MSRPC2 aims to provide improved performance by reducing multiplexing and by relying to some extent on the properties of the underlying communication facilities. Since MSNL offers a transport which is fairly reliable, MSRPC2 exploits this by neglecting to implement a reliable transport protocol on top of MSNL. Errors arising at the network level are left to the user to sort out, and the semantics of the RPC mechanism reflect this.

MSRPC2 is a simple and efficient RPC mechanism for distributed systems programming. It originates from MSRPC, a lightweight RPC system based on SUN RPC, implemented by the authors at the Computer Laboratory. Whilst the RPC mechanism offered by MSRPC was efficient, its library interface, the stub generation utility and the interface description language it used were unfriendly and lacked the requisite functionality for a tool which could be used for distributed systems programming. A complete redesign of the system, combined with a complete re-implementation of the runtime system have resulted in the new system, MSRPC2.

The MSRPC2 library has been written to run on both Wanda and Unix. The only difference in the code between the two versions is the platform specific protocol interface. The bulk of the code is platform independent and makes use of a portable threads package, named "pthreads", and standardised as the POSIX 1003.4a standard.

The MSRPC2 runtime system was written to support a lightweight protocol stack. The initial implementation supported only MSNL as the transport protocol, but recently a TCP transport has been added. MSRPC2 relies on the use of virtual circuit based transport protocols (MSNL and TCP) to reduce the per RPC overhead. State associated with the binding between client and server is maintained at the endpoints, and the connection identifier is the only identifier needed to uniquely specify a communicating (client, server) tuple. The semantics of the RPC system are dependent on the protocol being used. The use of MSNL has the following consequences:

- MSNL is an unreliable protocol, and hence cells and packets may be lost by the network. Because ATM networks aim to deliver very high reliability, however, such losses are expected to occur infrequently. MSRPC2 thus implements an unreliable protocol, which does not guarantee that an RPC will be executed once only in the absence of failures. On the contrary, the only semantics supported for MSNL are At Most Once (AMO) semantics. Options exist to request the runtime system to re-try calls which fail due to loss or time out.
- MSNL is an ATM protocol and fragments large packets into cells. By keeping the per RPC overhead to a minimum, MSRPC2 permits a typical RPC call and its arguments to fit into a single ATM cell. The overhead per MSRPC2 call is only eight octets of data for both the call and the returned results.
- MSNL permits connections to request different quality of service parameters (QOS) at connection setup time. Currently the QOS used for all MSRPC2 connections is “best-effort”, however it is possible to support RPC connections which offer enhanced QOS. The MSRPC2 interface reference permits a service to specify its QOS requirements, so that data transferred across its interface will be properly delivered by the network.

For TCP the reliable communication protocol means that an Exactly Once semantics can be supported.

The following sections describe the components of the MSRPC2 runtime library and utilities in their current state.

9.2 Library Interface

The MSRPC2 library presents an interface to clients and servers which permits the stubs to invoke RPC calls on remote (i.e. not same domain) servers, and servers to export their interfaces.

9.3 Client

The client stubs are responsible for binding to the server, and management of the communication with the server for a given session. The user (i.e. the stubs) are presented with (dare I say it) an object-oriented binding structure which represents all state and methods associated with the binding. To create a binding, the prospective client calls the function:

```
bool_t
   clnt_bind(cl, ifref, m_alloc, m_free, exc_disp, exc)
```

```
clntSession             **cl;
MSRPC2InterfaceRef      *ifref;
PFC                     m_alloc;
PFV                     m_free;
caddr_t                 exc_disp;
excpt_t                 *exc;
```

Here the supplied pointer to the interface reference should be a pointer to the interface reference which refers to the service to which the client wishes to bind. The memory management functions `m_alloc`, `m_free` are used by the MSRPC2 library on behalf of the client, to allocate any memory required for RPC arguments or results. The standard C library functions may be supplied for this purpose. The function `exc_disp` is a pointer to the stub level exception handler which will manage the dispatching of all RPC library exceptions to the client. The exception pointer is supplied to permit the user to establish why the binding operation failed, if the return code is FALSE. A return code of TRUE indicates success, and the variable `cl` will point to a client session structure which contains all state for the binding, as well as stub level methods for accessing the underlying RPC protocol. The session structure has the following fields:

```
typedef struct _clntSession {
   LIBS bool_t  (*call_init)    _((struct _clntSession *self, MSDR **m,
                                   int32 procno, excpt_t *exc));
   LIBS bool_t  (*call)         _((struct _clntSession *self, excpt_t *exc));
   LIBS bool_t  (*announce)     _((struct _clntSession *self, excpt_t *exc,
                                  bool_t isretry));
   LIBS bool_t  (*control)      _((struct _clntSession *self, int32 op,
                                   caddr_t param));
   LIBS void    (*destroy)      _((struct _clntSession *self));
   LIBS caddr_t                 private; /* private proto  dependent state */
   LIBS caddr_t                 exc_disp; /* opaque exception dispatch state */
} clntSession;
```

The specification `LIBS` ensures that the fields are write only by the libraries, and compile time errors will occur if any session fields are modified outside the library. The session structure is the client's only access to the runtime library. Each method is described in more detail below, using the MSNL transport service as an example. Note that the session structure is transport protocol independent, and represents the session-layer functions of the RPC runtime system. Protocol specific data is stored on the structure as a private opaque field `private` which can be accessed and interpreted by the methods associated with the session.

The methods, whilst presenting a protocol independent interface, are protocol specific, and are initialised to the appropriate methods the protocol required for the binding, when the binding is created by `clnt_bind`. At bind time the protocol specific binding function is called to establish a connection of the appropriate type to the remote service instance. The required information for the protocol is stored in the interface reference passed to the library. For MSNL the binding function is:

```
bool_t
   clntmsnl_bind(cl, ref, m_alloc, m_free, exc_disp, exc)
```

```
clntSession           *cl;
MSRPC2InterfaceRef    *ref;
PFC                   m_alloc;
PFV                   m_free;
caddr_t               exc_disp;
excpt_t               *exc;
```

This function returns TRUE if it succeeds, and FALSE if the binding failed. In this case the exception block has been set to indicate the cause. This function initiates communication with the remote service instance. If the interface reference requires the client to check the identity of the service, then when the connection has been established, the client waits for a message from the server which contains the interface identifier. This is checked against the identifier in the interface reference. If the two do not match then the binding is destroyed. If no check is requested then this sequence is not performed.

Once the client has established binding with the remote service instance, it can send RPC requests and obtain results. Note that each session is single threaded. It is not possible to multiplex RPC calls onto a single session. Before making an RPC, the client stubs must request the library to initialise state required for a call. It uses the method `call_init` on the binding structure for this purpose. For MSNL this function is:

```
bool_t
   clntmsnl_call_init(cl, msdrs, procno, exc)
clntSession     *cl;
MSDR            **msdrs;
int32           procno;
excpt_t         *exc;
```

This function initialises the marshalling state required for the session, in the variable `msdrs`. To do this the library requests an I/O buffer from the I/O buffer system on the platform. For wanda the I/O buffer system which forms part of the IPC system in wanda is used. For unix, a library of I/O buffer management functions is supplied as a part of the RPC library. This permits I/O buffers to be used universally throughout the MSRPC2 library, giving a platform independent interface. The integer `procno` is the number representing the operation in the MSRPC2 interface which is to be invoked. This is defined by the interface compiler. This function returns TRUE on success, and FALSE, with exception, on failure.

Once it has successfully initialised the runtime system, the client stubs can make an RPC call. To do this the stubs call either the method `announce` or the method `call` to make the RPC. `announce` is used for announcement operations, whilst `call` is used for operations which return results. When either of these methods is invoked, the library assumes that the arguments to the call have already been marshalled. Marshalling is done by the stubs, using the initialised `msdrs` supplied for this purpose by the library as a result of the `call_init` operation.

```
bool_t
   clntmsnl_announce(cl, exc, isretry)
clntSession     *cl;
excpt_t         *exc;
bool_t           isretry;
```

The MSNL versions of `call` and `announce` make an RPC call on half of the client. The underlying MSNL service layer is used to transmit the I/O buffer containing the marshalled arguments to the server. The client library then blocks in a receive until either a timeout expires or results are received from the server. In the case of an announcement, no results are expected, and the library returns immediately after making the call. The timeout can be set using the control operation described below. Received results from a `call` are partially unmarshalled within the library, to determine whether the call executed successfully or not. In the latter case the client's exception handler is called with the exception, in the former, the call simply returns TRUE and the stubs can unmarshal the results.

The results or exceptions are passed to the stub code in the `msdrs` which is the marshalling object associated with the call. This will permit easy unmarshalling of the results. If the call fails then the exception structure contains a reason for the failure, and a pointer to the `msdrs` which can be used to unmarshal the arguments to the exception, returned by the server.

```
bool_t
   clntmsnl_call(cl, exc)
clntSession     *cl;
excpt_t         *exc;
```

Several properties of the underlying protocol may be controlled via the library interface. Each binding object has a method `control` which can be used to set these parameters. These parameters may differ according to the transport protocol being used. For MSNL the function used is:

```
bool_t
   clntmsnl_control(cl, request, info)
clntSession     *cl;
int32           request;
caddr_t         info;
```

The parameters which may be set are:

- `MSRPC2_CLSET_TIMEOUT` – sets the timeout before the exception is raised which declares that the RPC has failed.
- `MSRPC2_CLGET_TIMEOUT` – retrieves the current value of the timeout.
- `MSRPC2_CLSET_BUFSIZE` – sets the size of I/O buffer to use for the session. This should be fixed. It is a hangover from msrpc, which requires that the maximum buffer size be specified before the session is set up. Currently the limit is set at 1500 bytes, which is the size of the largest I/O buffer on a default wanda kernel.
- `MSRPC2_CLGET_BUFSIZE` – retrieves the current value of the buffer size been used by the MSDR library.
- `MSRPC2_CLSET_RETRIES` – this is a feature which is supported by MSNL to permit the library to make several retries of a failed RPC request before declaring that the call has failed. The value of this variable will be used to determine how many retries are made.
- `MSRPC2_CLGET_RETRIES` – retrieves the current value of the count.

A client should terminate its session when it no longer requires it. This should be done by calling the session's destroy method. For MSNL this is:

```
void
   clntmsnl_destroy(cl)
clntSession     *cl;
```

9.4 Server

The server side of the library is far more complicated. A service offer, corresponding to an instance of a middl interface, is made by calling the function

```
bool_t
   svc_offer(svc_int, ifref, intf, m_alloc, m_free, cons, dest, exc)
svcInterface            **svc_int;
MSRPC2InterfaceRef      *ifref;
caddr_t                 intf;
PFC                     m_alloc;
PFV                     m_free;
svcBindingConstructor   cons;
svcBindingDestructor    dest;
excpt_t                 *exc;
```

This is a protocol independent function, and takes as argument a pointer to an interface reference in which the protocol field must be set. This allows the library to determine which protocol to support for this service instance. Also passed as arguments are memory allocation and deallocation routines, which will be used by the library to allocate memory for all RPC arguments and results which require it. The argument `svc_int` is modified by the library and on return points to the interface instance which will be listening for client binding requests. The argument `intf` is a pointer used the library to refer to the stub level state representing the interface instance. It is not interpreted by the library. The methods `cons,dest` are user-level methods which are supplied to the library to permit validation of new binding requests and notification of binding termination. When a new client binding request is made, the user level constructor function will be called, to allow the user to vet the client. If this function returns FALSE, the new client binding request will be rejected. If it returns TRUE then the client will be accepted. The destructor will be called when the binding is broken, for any reason. This allows the server to free any per-client binding state which it has attached to the binding object.

The MSNL version of the offer function is used to initiate the service. It creates an offer, by forking off a thread which listens for new client binding requests. This thread has as argument a pointer to the section object created for the interface. This is known as the rendezvous thread, and it is responsible for creating all new bindings. The interface object created contains list of all bindings which belong to this interface instance. When the interface is withdrawn, this list is used to ensure that all bindings for the interface are terminated appropriately.

```
bool_t
```

```
   svcmsnl_offer(svc_int, ifref, intf, m_alloc, m_free, constr, destr, exc)
svcInterface            **svc_int;
MSRPC2InterfaceRef      *ifref;
caddr_t                 intf;
PFC                     m_alloc;
PFV                     m_free;
svcBindingConstructor   constr;
svcBindingDestructor    destr;
excpt_t                 *exc;
```

The offer routine returns a pointer to an `svcSession` . This is the server-side representation of the interface/binding. In the case of an interface, it represents the interface instance. For each new client binding, a new `svcSession` is created. This maintains all per session state for the binding. It also has a private field which is for use by the protocol specific code which manages the connection with the client. The notation `LIBS` indicates that the fields may only be modified by the library. The notation `BOTH` indicates that the stub level code and the library may modify the fields.

```
struct _svcSession {
   LIBS void            (*destroy)      _((svcSession *self));
   LIBS bool_t          (*control)      _((struct _svcSession *self,
                                           u_int32 op, caddr_t param));
   LIBS bool_t          (*recv)         _((svcSession *self,
                                           u_int32 *procno, MSDR **m,
                                           excpt_t *exc));
   LIBS bool_t          (*reply_init)   _((svcSession *self, MSDR **m,
                                           u_int32 raises));
   LIBS bool_t          (*reply)        _((svcSession *self));
   LIBS void            (*excpt_intnl)  _((svcSession *self,
                                           excpt_t *exc));
   BOTH svcBindingDispatcher dispatch;
   LIBS bool_t          alive;          /* used by svc_run for all protos */
   LIBS caddr_t         private;        /* protocol dependent state */
   LIBS svcSession      *next;          /* chain for linked list of sessions */
   LIBS svcType         type;           /* rendezvous or connected */
   LIBS svcList         *svc_list;      /* list of all sessions for service */
   LIBS pthread_t       thread;         /* the thread managing the session */
   LIBS caddr_t         binding;        /* stub per session state block */
   BOTH MSRPC2InterfaceRef      *ifref; /* stores state on endian proto etc */
   LIBS svcBindingDestructor    destr;  /* stub state destructor */
};
```

Note that each binding is single threaded. For each binding a thread is forked to manage and perform all RPC requests for the associated client. No other thread should act on behalf of the client, otherwise the library will be unable to withdraw the interface.

When a new binding is created, the rendezvous thread calls the internal function

```
bool_t
```

```
   svcmsnl_intnl_create(session, sock, ifref, m_alloc, m_free, exc)
svcSession              *session;
int32                   sock;
MSRPC2InterfaceRef      *ifref;
PFC                     m_alloc;
PFV                     m_free;
excpt_t                 *exc;
```

to create a new binding object for the binding. It also calls up into the user level code by invoking the binding constructor for the interface. The session object is initialised with all appropriate state for the interface to which it refers, and the methods are initialised to the protocol specific methods for managing the RPC session specified in the interface reference.

Each binding is managed by a single thread. Each thread, on starting, calls the function (protocol independent) `svc_run` which contains an infinite loop. On each iteration, the binding's `recv` method is called. For a rendezvous thread, the `recv` method is initialised to the function which is responsible for forking off new sessions when new binding requests are received. For MSNL this is the function

```
bool_t /* ALWAYS FALSE */
   svcmsnl_rendezvous(session, procp, msdrs, exc)
svcSession      *session;
u_int32         *procp;         /* DUMMY argument, never used */
MSDR            **msdrs;        /* DUMMY argument, never used */
excpt_t         *exc;
```

This function waits for connection requests on the MSNL sap associated with the interface reference. When a request is received, if the binding constructor returns TRUE, a new binding is created. This involves forking a new thread, with the `recv` method to the real receive method for the protocol. In the case of MSNL this is `svcmsnl_recv`. The other methods will also be initialised to the appropriate values.

```
bool_t
   svcmsnl_recv(session, procp, msdrs, exc)
svcSession      *session;
u_int32         *procp;
MSDR            **msdrs;
excpt_t         *exc;
```

This function waits for RPC calls from its associated client. The function receives an I/O buffer from the MSNL layer managing the communication with the client. This function unmarshals the arguments required by the stub layer code, and then calls the dispatch function associated with the binding. This function then unmarshals the arguments and calls the user level code. If an error occurs within this function, an exception is raised. Exceptions are marshalled and returned to the calling user. This includes both user level exceptions and library exceptions. If this function returns value of TRUE to the caller (`svc_run`) then the dispatch function will be invoked. If not then the associated exception, if there is one will be returned. If the user-level dispatcher returns TRUE, then the call succeeded, otherwise it failed.

If the invocation is expected to return results then the stub level code must call the function `reply_init` on the binding to initialised appropriate state in the library for the reply. The MSDR structure will be initialised with an I/O buffer in preparation for the reply. For MSNL this function is

```
bool_t
   svcmsnl_reply_init(session, msdrs, raises)
svcSession      *session;
MSDR            **msdrs;
u_int32         raises;
```

Once the reply state has been initialised, the stub level code can use the MSDR object to marshal either the results or the exception arguments. Note that it is the responsibility of the stub code to do all marshalling. The stub code may then call the reply function for the session. This will transmit the marshalled results or exception and arguments to the client.

```
bool_t
   svcmsnl_reply(session)
svcSession      *session;
```

Similar to the client side, the server is provided with an interface which permits the properties of the session to be modified on a per binding basis. This is the control function, which for MSNL has the following form. All arguments `info` for the time functions are a `struct timeval *`, otherwise they are unsigned integers.

```
bool_t
   svcmsnl_control(session, request, info)
svcSession      *session;
u_int32         request;
caddr_t         info;
```

The parameters which may be modified/retrieved include

- `MSRPC2_SVCSET_TIMEOUT` – set the timeout which the server uses before declaring the session to have died.
- `MSRPC2_SVCGET_TIMEOUT` – get the timeout above.
- `MSRPC2_SVCSET_BUFSIZE` – set the buffer size used by the MSDR library when initialised an MSDR object. The default value is 1500 bytes.
- `MSRPC2_SVCGET_BUFSIZE` – get the buffer size above.

Chapter 10

Marshalling code library

MSRPC2 uses the MSDR library to marshal and unmarshal data to/from the underlying communication format. MSDR defines a set of encoding and decoding rules for each possible data type. Each rule is specified in a functional form. Three types of operations can be invoked on an instance of a particular type, namely encoding, decoding and freeing. The encoding operation is used to encode an instance of a type into a form suitable for transmission using MSRPC2 protocols to the destination. The decoding rule is used by a receiver to convert the underlying protocol data into an instance of the appropriate type. The freeing rule is used to discard an instance of a particular type, once it is no longer required.

For each of the basic MSRPC2 types there is a corresponding MSDR type, listed below. For each MSDR type 3 functions are provided in the MSDR library to encode, decode and free data of that type. The encoding function is prefixed with `m_`, the decoding function is prefixed with `u_`, and the freeing function is prefixed with `f_`. The basic MSDR types are:

`msdr_void`	`msdr_int32`	`msdr_u_int32`
`msdr_int64`	`msdr_u_int64`	`msdr_int16`
`msdr_u_int16`	`msdr_bool`	`msdr_enum`
`msdr_char`	`msdr_u_char`	`msdr_float`
`msdr_double`	`msdr_set`	

Constructed types, declared in MSRPC2 MIDDL interfaces, are also marshalled and unmarshalled using MSDR primitives. For each type in the interface, the interface compiler generates the 3 MSDR functions for encoding, decoding and freeing data of that type. These functions will make use of the basic MSDR primitives declared in the MSDR library. It is therefore possible to marshal data of any type, simply by making use of a set of basic marshalling primitives.

Associated with each MSRPC2 session binding at the client and server is an MSDR "handle" which refers to an object which maintains the state on the (de)serialisation of data corresponding to arguments and results of RPC calls. Both the client and server link with the same stub library and use the same MSDR routines to perform the (un)marshalling of the data. The MSDR object is initialised with a memory management structure containing pointers to 2 functions, for allocating and freeing memory. These are passed into the library by the client or server when creating the binding.

10.1 Wire representation

MSDR is endian dependent. Each interface reference contains information which describes the endian of the interface instance. A client is expected to marshal data in the endian format of the server. Similarly the client must convert data from the endian of the server into its own endian when results are received. This conversion is carried out automatically by the MSDR library. Each MSDR object has an associated endian, which it uses to marshal the data in the correct way. The endian of the data on the wire is thus the endian of the interface instance.

The wire representation of MSRPC2 RPC calls can differ according to the protocol being used for communication. For MSNL, the protocol defines two words of header per message. These are as follows:

- For an RPC call/announcement: word 0 contains a sequence number indicating the sequence number of the call since the beginning of the session; word 1 contains an integer representing the remote operation which is to be invoked.
- For returned results: word 0 contains the sequence number used in the invocation of the operation, to allow the client to associate the results with the appropriate invocation; word 1 contains a status code which indicates whether or not the RPC succeeded. If the status is not `MSRPC2STAT_SUCCESS` , then it indicates that an exception has been raised at the server, and the appropriate exception handler should be invoked.

For TCP, the protocol defines only one word of header per message. The protocol is as follows:

- For an RPC call/announcement: word 0 contains an integer representing the remote operation which is to be invoked.
- For returned results: word 0 contains a status code which indicates whether or not the RPC succeeded. If the status is not `MSRPC2STAT_SUCCESS` , then it indicates that an exception has been raised at the server, and the appropriate exception handler should be invoked. The internal exceptions which can be raised are listed below.

- `MSRPC2STAT_CLNT_NO_BUFS` – the client cannot obtain an I/O buffer for communication
- `MSRPC2STAT_CLNT_CANT_SEND` – failure in sending call/reply
- `MSRPC2STAT_CLNT_CANT_RECV` – failure in receiving call/reply
- `MSRPC2STAT_CLNT_OUT_OF_MEM` – out of memory
- `MSRPC2STAT_CLNT_IPC_ERROR` – failed in setting up IPC
- `MSRPC2STAT_CLNT_CANT_ENCODE_ARGS` – the client cannot encode the arguments to the call, possibly due to a lack of buffer space
- `MSRPC2STAT_CLNT_CANT_DECODE_RES` – client cannot decode results received from the server
- `MSRPC2STAT_CLNT_TIMED_OUT` – call timed out
- `MSRPC2STAT_CLNT_BAD_EXCEPTION` – bad exception raised at client

- MSRPC2STAT_CLNT_NO_EXCPT_HANDLER – exception raised has no handler
- MSRPC2STAT_UNKNOWN_PROTO – unknown protocol requested in interface reference
- MSRPC2STAT_BIND_FAILED – probably server has moved on
- MSRPC2STAT_OFFER_FAILED – server cannot initialise a new offer
- MSRPC2STAT_SVC_IPC_CLOSED – client closed connection
- MSRPC2STAT_SVC_NO_BUFS – the server has run out of I/O buffers
- MSRPC2STAT_SVC_CANT_SEND – failure in sending call/reply - usually due to a protocol error at a lower level
- MSRPC2STAT_SVC_CANT_RECV – failure in receiving call/reply - usually due to a protocol error at a lower level
- MSRPC2STAT_SVC_OUT_OF_MEM – out of memory
- MSRPC2STAT_SVC_IPC_ERROR – failed in setting up server side of communication path
- MSRPC2STAT_SVC_CANT_DECODE_ARGS – server couldn't decode arguments
- MSRPC2STAT_SVC_CANT_ENCODE_RES – server cannot encode results
- MSRPC2STAT_SVC_TIMED_OUT – listen timed out
- MSRPC2STAT_SVC_BAD_PROCNO – illegal procedure (operation) requested
- MSRPC2STAT_THREAD_ERROR – thread package problem
- MSRPC2STAT_USER_ERROR – a user level exception has been raised