# MSRPC
# A Lightweight RPC Mechanism for MSNL

Simon Crosby

February 17, 1993

## 1 Introduction

This document describes the current implementation of MSRPC, a simple and lightweight RPC mechanism for use in the world of MSNL. Like MSNL, MSRPC 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, MSRPC 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.

MSRPC was based on an implementation of SUN RPC for Wanda and currently suffers as a result from certain restrictions imposed by its parentage. Proposals have been made to improve the user interface to the MSRPC library and to use a different interface definition language. These are discussed in more detail in the sections which follow. To begin with, however, the basic design of the MSRPC library routines in their current incarnations for Unix and Wanda, are described. Where appropriate, proposals for improvements in the library are discussed. Currently it is proposed to subsume the Wanda and Unix implementations of the MSRPC libraries into a single, portable implementation using the "pthreads" library.

## 2 MSRPC and MSNL

A server, in MSRPC terminology, is an object which is capable of responding to requests from a client according to a pre-defined interface which describes the data transferred between them and a set of procedures (methods) offered by the server. This definition is broad, and might include a SCSI disk driver, or an ATM camera control interface. A server exports an interface which describes the data types and the operations which the server supports. When a server wishes to export its interface, it does so by registering an offer with a binding service. Prospective clients wishing to invoke operations on the server's interface can query the binding service to gain information on service offers matching their requirements. As a result of its query to the binding service, the client is handed one or more interface references. Each interface reference is a unique reference to an instance of the required service. Using the interface reference the client can then create a session binding with the required server, and invoke operations on the exported interface.

MSRPC relies on the use of a virtual circuit based protocol (MSNL) to reduce the per RPC overhead. State associated with the binding between client and server is maintained at the endpoints, and the VCI of the network connection between them is the only identifier needed to uniquely specify a communicating (client, server) tuple. MSRPC relies on the properties of MSNL in several ways:

- 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. MSRPC 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 are At Most Once (AMO) semantics.

- MSNL is an ATM protocol and fragments large packets into cells. By keeping the per RPC overhead to a minimum, MSRPC permits many RPC calls and their arguments to fit into a single ATM cell. The overhead per MSRPC call is only eight octets of data, and only four octets for returned results of an RPC.

- MSNL permits connections to request different quality of service parameters (QOS) at connection setup time. Currently the QOS used for all MSRPC connections is "best-effort" however it is possible to support RPC connections which offer enhanced QOS. The MSRPC interface reference (will soon) permit a service to specify its QOS requirements, so that data transferred across its interface will be properly delivered by the network.

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

## 3 Interface Definitions

Interface definitions in MSRPC are written in a modified form of the SUN RPC description language, called MSDR, which has a syntax similar to that of C. The language provides constructs for the description of the data structures and programs which make up a service interface. The language supports all conventional C programming language types and several other types such as strings, opaque data and tag discriminated unions. Program definitions describe the services offered by an interface. A program may have several versions, each of which defines a set of procedures which a client can invoke. An example of an interface description is given below. The reader is referred to the SUN RPC network programming manual for a full specification of the SUN RPC description language (XDR) which differs from MSDR as described below.

```
const   BLOCK_SIZE = 512;
typedef opaque block_type[BLOCK_SIZE];
typedef struct blk_req {
        int    blk_size;
        string blk_name<20>;
} blk_req;

program MEAS_PR {
        version MEAS_VER {
            void PUTBLOCK (block_type);
            block_type GETBLOCK (void);
            block_type GETNAMEDBLOCK (blk_req);
    } = 1;
} = "meas_program";
```

In this example the only data transferred across the interface are the opaque type named `block_type`, and the structure `blk_req`. The interface is of type `MEAS_PR` and has the name "meas_program", and it has a single version, 1. This version offers three procedures – a `PUTBLOCK` procedure to send a block to the server, a `GETBLOCK` procedure to get a block from the server, and a `GETNAMEDBLOCK` procedure to get a block of a specified length, named by a string of length 20 characters. By convention the procedure names are specified in upper case in the interface description, and implemented in lower case in the client and server stubs.

One change has been made to the SUN RPC language for MSRPC, to support the use of strings for the specification of program names. This is to avoid the problem which occurs in SUN RPC of having to administer program number assignment centrally. The use of a string to define a program name is attractive because it permits the use of meaningful names; for instance a SCSI disk driver could be registered as "`/dev/scsi0`". The use of strings also permits client requests to be matched more intelligently with a list of available services.

### 3.1 A Note for ANSA Fans

MSRPC does not attempt to replace ANSA, nor pretend to offer all of the useful features of ANSA. It might be desirable, however, to implement an "ANSA–lite" in which the MSRPC library replaces the ANSA REX, MPS and kernel functions, and the useful ANSA features such as interface trading and other elements of the ANSA architecture are retained. This is a nice idea but would take a substantial amount of work to achieve (volunteers ?).

The ANSA Interface Description Language (IDL), whilst it offers no advantages over MSDR in terms of the data types which it can describe, has other advantages. IDL interfaces are properly typed, whereas the MSRPC program type is currently used only to ensure uniqueness of the stub names generated for an interface. In IDL it is possible to declare an interface to be a subtype of another interface, whereupon the subtype inherits the supertype's operations (procedures) and data declarations. IDL and the embedded language DPL permit the handling of exceptions raised by the ANSA sublayers. In IDL a procedure can take many arguments, whereas in MSDR each procedure takes only one argument which is a compound data structure composed of the individual arguments to the remote procedure. This forces the programmer to use somewhat unnatural data structures in the program. Currently it is planned to replace MSDR with a different interface description language. This may be an extended form of ANSA IDL or may be derived from other interface description languages published in the literature.

## 4 Stub Generation

The aim of using an interface description language is to provide an implementation independent specification of the data and procedures which a server supports. The interface description can be compiled into client and server stubs which transparently establish and maintain the connection between the client and server when the client starts up, and which manage the transfer of data to and from the server. The client stubs provide a procedural interface to the server which can be invoked as though the server procedures were procedures in the client program. The stubs also ensure that the data which traverse the interface are marshalled into an appropriate form for the server. On the server side the stubs provide a mechanism for unmarshalling the supplied arguments, invocation of the appropriate server procedure, and marshalling of the returned results for return to the client.

From an MSDR description of a data structure the marshalling and unmarshalling routines can be automatically generated. In general for a data type `t` defined in an interface an MSDR function named `msdr_t()` is generated to marshal and unmarshal data of that type. The same function is used for marshalling and unmarshalling, to ensure that exactly the same steps are carried out for each direction of transfer, thereby reducing the possibility of errors. For each of the basic MSDR types, listed below, a function is provided in the MSRPC library to serialise data of that type:

| | | |
|---|---|---|
| `msdr_void` | `msdr_int` | `msdr_u_int` |
| `msdr_long` | `msdr_u_long` | `msdr_short` |
| `msdr_u_short` | `msdr_bool` | `msdr_enum` |
| `msdr_array` | `msdr_opaque` | `msdr_string` |
| `msdr_union` | `msdr_char` | `msdr_u_char` |

```
msdr_vector          msdr_hyper           msdr_float
msdr_double          msdr_reference       msdr_pointer
msdr_string          msdr_msrpc_port_t
```

Associated with each MSRPC session binding at the client and server is an MSDR "handle" which refers to an object which maintains the state on the 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.

When a client makes a `clnt_call` (an RPC invocation) it passes as an argument a pointer to the appropriate MSDR routine (either a library routine or one generated by `msrpcgen`). The MSRPC library at the client side initialises its MSDR object with the buffer into which data is to be marshalled. Data are marshalled into the buffer by the MSDR routine passed as argument to the `clnt_call` and then transmitted by the client to the server. At the server side, the buffer which contains the arguments received from the client is used to initialise its MSDR object, and the arguments are unmarshalled by the same MSDR routine to reproduce the the arguments to the call. Returned results are marshalled in a similar fashion at the server, and unmarshalled at the client. Memory may be allocated by the MSDR routines when unmarshalling data such as linked lists and strings. This can be freed, at the client side, using the function `clnt_freeres`.

MSDR supports different endian orderings for the data transferred across an interface. By convention the server specifies the endian to be used. Supporting different endian architectures can yield considerable performance benefits. By convention, if the endian is not specified then big endian ordering is used. Endian specifications are dynamic, and are not bound to an interface definition. When a server exports an interface reference to the binding service and to clients, the endian of the server is expressed in the interface reference. When the session binding the client and server is established, the MSRPC library ensures that the correct endian ordering for the data transfer is used.

## 4.1 `msrpcgen`

MSRPC interface definitions are compiled by `msrpcgen` to produce client and server stubs for different operating system platforms. `msrpcgen` also generates the MSDR routines for (un)marshalling the data transferred across an interface. The arguments to `msrpcgen` include:

- `-t <transp>` Stubs for the "transport" protocol `transp` are generated. Two protocols are available: `msnl` and `sdom`. The `msnl` transport simply generates stubs which use MSNL for communication between client and server. The `sdom` transport is used when client and server are located within the same protection domain (Wanda only). In this case the RPC calls are optimised to local procedure calls and a different session binding mechanism is used. For `sdom` transports the portmapper is not used. The default on wanda systems is to generate stubs which use both protocols. On Unix the sdom protocol is not currently supported.

- `-u` Stubs are generated for a Unix platform: The `main()` routine generated does not assume that the server is multithreaded, and the `sdom` transport is not permitted. This will change when a pthreads multithreaded Unix library implementation is done in the near future. If the `-u` argument is not given then it is assumed that Wanda stubs are required. In this case no server `main()` routine is generated and the stubs spawn Wanda threads to perform server registration and initialisation.

- `-i` Client side RPC calls are placed inline in the client code. This saves the overhead of a procedure call to the client stub routines and is recommended in all cases.

- `-m, -c, -s, -h` Generate (respectively for each argument) only the MSDR routines for (un)marshalling the data types in the interface definition, the client side stubs, the server

side stubs, or the header file. It would appear (who knows why ?) that when using these arguments the argument `-o <output_file_name>` should also be supplied, otherwise `stdout` is used.

- CPP arugments: `msrpcgen` uses `cpp` to preprocess interface definitions. The normal CPP arguments are available, such as `-I` for an include path, -D to define variables etc. The use of CPP permits the usual preprocessor directives to be used within interface defintion files. This allows a primitive "subtyping" relationship by simply `#include`ing the required interface definitions.

In the Wanda Snap 4 version of `msrpcgen` procedure names in the generated stubs are simply the procedure names from the interface, in lower case, with the interface version and a `c` or `s` appeded to indicate client and server respectively. In the current version of `msrpcgen` (new name pending to avoid confusion) procedure names are in addition prefixed by the interface type in lower case.

In all interfaces which contain a program definition, a NULL procedure is generated automatically *iff* there is no other procedure which has been assigned the procedure number 0. The NULL procedure can be used as a "ping" procedure to check whether a server is still alive and kicking. in the Wanda Snap 4 `msrpcgen` the NULL procedure is called `NULLPROC`. In the current version it is called `<iftype>_NULLPROC`, where `<iftype>` is the interface type. As a final note, `msrpcgen` is an ugly beast and should be shot, however until we change to using a different IDL it will probably remain with us, so beware.

## 5 Semantics of the RPC Mechanism

MSRPC currently supports only AMO (At Most Once) semantics. Under AMO semantics, a client will transmit a service request exactly once, to ensure that retried requests do not result in invocation of a service more than once. Such requests may be lost by the network, or alternatively the reply from the server may be lost. In either case the RPC system will time out the request and will indicate to the client that the RPC failed. It is the responsibility of the client to manage such failures. A simple recovery mechanism is to retry each request some number of times before declaring the server dead. In practice MSNL loses data infrequently and time outs can be set to very small values. Servers should be coded in an idempotent fashion – duplicate requests should not have unwanted side effects. Alternatively EXO (EXactly Once) semantics can be implemented on top of the AMO semantics offered by the library or the EXO libraries currently being developed (see below) should be used. In the current, AMO, implementation the server does not maintain state on the invocation of each operation in the interface and will re-execute a retransmitted RPC request which the client believes was lost. A server which fails to receive a request after (a settable) timeout will declare the session to be dead, and close the connection to the client. Clients should thus be aware that the session may be destroyed by the server at any time, and should therefore be coded in such a way as to re-establish lost sessions after timeouts. The client discards replies to all requests other than its most recent request. If the reply to the most recent request is lost then the client will time out and report the error. The client timeout is also user settable.

In two special cases the AMO semantics are enhanced to EXO semantics. When the client and server are co-located on the same host the MSNL protocol is reliable. Also, when the client and server are located in the same protection domain, invocations across the service interface can be implemented as normal procedure calls. In these two cases MSRPC offers EXO semantics. An implementation of EXO semantics using TCP/IP over MSNL is currently in progress.

## 6 The Portmapper

In a world with eager clients and subservient servers, a mechanism is required to bring the two together. This is the binding service, known in SUN RPC as the portmapper, itself an RPC service. In MSRPC the portmapper is provided as a standalone server, `msportmapper`. One instance of the portmapper is required per host, where an MSNL address is given by a (host, port) pair. The portmapper listens for service requests on a well known port. The portmapper maintains information about all services which are available on its host, and informs clients how to contact a particular service. A service registers with the portmapper by invoking the routine `PMAPPROC_SET` on the portmapper interface, providing the following data to fully specify its interface (in MSDR):

```
struct pmap {
       string pm_prog<>;
       u_long pm_vers;
       u_long pm_prot;
       u_long pm_port;
       u_long pm_semantics;
       u_long pm_endian;
};
```

The fields are, respectively, the program name (eg: "`/dev/wombat`"), the version number of the interface, the protocol which the interface supports (eg: `PROTOCOL_MSNL`), the port at which the service is listening to receive requests, the semantics of the interface, and the endian which clients should use when communicating with the service. Two "protocols" are supported, namely MSNL and SDOM. SDOM is the same domain protocol which is used when the client and server reside in the same protection domain on a particular host. A service can de-register itself by calling the function `PMAPPROC_UNSET`.

As part of the session binding procedure prospective clients request information on the services they require from the portmapper. The procedures used are `PMAPPROC_DUMP`, which lists all registered services, and `PMAPPROC_GETPORT` which returns the particulars of a specified service. The portmapper returns to a client the data which are dynamically associated with a service instance, called the port semantics (`prt_sem`). They include:

```
struct prt_sem {
       u_long port;
       u_long semantics;
       u_long server_endian;
};
```

The fields have the same meaning as described above. In particular the client is informed of the endian which the server wishes to use for communication. By convention the portmapper talks big endian, irrespective of the endian of the platform supporting it. The structures `pmap` and `prt_sem` together form the MSRPC equivalent of an interface reference, and in future releases an interface reference type containing these fields (and others) will be one of the MSDR basic data types. In future versions of MSRPC it is proposed to implement the interface reference as a standard type in the IDL used to define interfaces. The interface ref will contain additional fields to permit the specification of the QOS requirements of the MSNL connection used to bind client and server.

It is proposed to replace the current portmapper with a trading service similar to the ANSA trader. This would relieve us of the requirement that a portmapper runs on every host where services are located, and that clients must be able to locate the portmapper on the host machine before being able to obtain information on services on that particular host. A step in the right direction has been made with the implementation of the MSNL name server (see section 9 below). Extensions to this service will permit more flexible trading of interface references.

In the current implementation portmapper queries are automatically made during creation of the client–server session binding. The portmapping service resides on MSNL port 222, and is assumed to run on every machine on which services reside. A client cannot contact a service directly without first obtaining the appropriate information from the portmapper. A server can, however bind itself to a particular address and fail to inform the portmapper of the binding.

# 7 Implementation

This section gives a broad overview of how MSRPC is implemented. Basically the user interface to the MSRPC library routines is similar to the SUN RPC user interface: A session binding between client and server is represented as an object with private data and several public methods. The objects are typed in the sense that the underlying communication protocol used causes the methods presented to the user to differ. The method signatures, however, are independent of the protocol. The object definitions, the definitions of the standard MSDR data types and marshalling routines and the signatures (C profiles) of the object methods are defined in a set of header files which are part of the MSRPC library.

Implementations of the MSRPC libraries exist for both UNIX and Wanda. On Wanda the libraries are available both within and outside the kernel, and form part of the standard Wanda libraries. There is *no difference* in the code between the Wanda user space and kernel libraries. The user space code makes use of the Wanda IPC system calls, and these are mapped onto their kernel counterparts by a simple header file which permits the use of the IPC system within the kernel. This permits clients and servers to be moved transparently from user space to kernel.

The Wanda MSRPC library sources are located in `master/libs/wanda/msrpc`. The Unix library sources are in `master/tools/msrpc`. The header files are located, for Unix, in the Wanda `master/tools/msrpc` library directory (snap 4). The Wanda MSRPC header files are located in `master/libs/include/msrpc`. The portmapper is located in: Wanda: `master/wapps/msrpcsvc`; Unix: `master/tools/msrpcsvc`. An example MSRPC client and server are also provided in the same directory in each case. The portmapper is an independent server and must be started before any services attempt to register themselves. On Wanda this leads to a problem when the server is located in the kernel. Since the kernel starts first, the portmapper, if run in user space, will not be available when the server starts up. Thus when services are located in the kernel, the portmapper must also be started as a kernel thread. Within the kernel, there is no requirement on the startup order of MSRPC threads. Semaphores in the MSNL code and within the MSRPC library ensure that (a) the host is capable of doing IPC and (b) the portmapper is running before any servers are allowed to start up. In user space the portmapper must be running before a server tries to register. Servers should re-try the registration procedure if the portmapper is not yet running.

Details on the client & server session object operations are given in the appendix.

## 7.1 Client

A client creates a binding to a server (either `msnl` or `sdom`, using the routine `clnt_create()` which returns a client handle representing the session binding. This handle is then used in every `clnt_call()` of an RPC. `clnt_create()` takes as arguments the transport to use, and the name of the host on which the server resides. In the case of an MSNL transport, `clnt_create()` contacts the portmapper on the machine on which the server resides to obtain the `prt_sem` for the connection. The `prt_sem` are used to determine the port to use in establishing the connection, the semantics, and the endian mapping to use. These are associated with the client object. The client sets up an MSDR stream with the appropriate endian to deal with the server. Every `clnt_call()` has associated with it a sequence number, which is generated by the client. The sequence number is opaque to the server. The sequence number and the remote procedure number are marshalled into a call body by the client and sent to the server. The header of every RPC message thus looks like:

```
struct msrpc_call_msg {
       u_long xid;  /* unique message id */
       u_long proc; /* the service number */
};
```

For the SDOM transport (EXO semantics) the setup procedure is as follows: The `clnt_create()` routine checks with an intra-domain registration service whether the service is available within the domain. If so then a session is created which simply maps client calls onto procedure calls at the server. The RPC message header does not require the `xid` field, and this is omitted.

In both cases each `clnt_call()` marshals the client call arguments using the MSDR routines generated from the interface description by `msrpcgen`. The complete message is sent to the server. Under MSNL the client transmits the call to the server and then blocks awaiting a reply. A timer is set to ensure that the client does not block indefinitely. If when the timer expires the server has not yet replied (the call or reply may have been lost), the client call will return a failure condition to the caller with the condition "timed out". Note that each (client, server) tuple has its own MSNL connection.

When a reply is received, the client checks to see that it has the same sequence number as the transmitted request. If it does not then the reply is discarded and the client re-listens on the connection until the right reply arrives or the call times out, whereupon control returns to the client process with appropriate error conditions set.

### 7.2 Server

A server, when it starts up, initialises its data structures, opens a port (may be well known, but shouldn't be) and then makes a conventional client call (as described above) to its local portmapper (MSNL only). It invokes the routines `PMAPPROC_UNSET` and `PMAPPROC_SET` to first de-register and then register its interface. The registration also registers with the server's RPC mechanism the address of a dispatcher function which will handle the dispatching of service requests to the appropriate service routines.

Having registered itself, the server does a blocking listen on the socket which it has created for the service. At this stage the service thread is called a `rendezvouser` and is incapable of servicing RPCs. When a client connects to the server, the server thread changes class, from a `rendezvouser` to a `data handler`, which permits it to execute RPCs. A new `rendezvouser` thread is spawned to continue listening for new requests on the service port, and the old thread begins to execute client requests. Each service thread thus created services *all requests* from its client. A service thread services only one client. Service implementors should be aware that multiple clients can connect to a single instance of the service, with each connection represented by an instance of a service thread. State maintained by a server should thus be protected by semaphores where appropriate. Currently no per-client state is available to a server thread when it is processing a client request. This will be changed in the near future. In the UNIX MSRPC library the concurrency of the server is "faked" by serialising the processing of requests arriving at the set of sockets managed by the server. One connection is maintained per client. Connections which are silent for an extended period are timed out and torn down. Currently this timeout mechanism is clumsy for the Unix implementation. For the Wanda snap 4 (unix) tools it is not implemented at all. A quick extension has been made to time out connections when *all* connections to the server are silent for 10 seconds ! This will be improved in the next release.

Each request received by the server is unmarshalled using the appropriate MSDR routines generated by `msrpcgen`. The service dispatcher function (also generated by `msrpcgen` is then called with the unmarshalled arguments. The reply message is marshalled appropriately by the server. On the wire the reply is preceeded by the RPC call number which the client uses to match replies with requests. Marshalled data returned as the body of the reply follow. Session bindings are terminated by calling `clnt_destroy()` or `svc_destroy()` appropriately.

## 8 Conventions

By convention all servers which are intended to be long running should export a NULL procedure in their interfaces, to permit remote clients to test whether they are operational or not. In addition such servers should be provided together with a client program which is capable of exercising the server interface by "pinging" its NULL procedure. This permits us to continually check whether or not a server is running, and if it is not, to re-start it.

## 9 MSNL Name Server

Sensible network administration at the Computer Laboratory requires a mechanism for the resolution of host- name to MSNL addresses, similar to the DNS. It was decided to implement a simple MSNL name service for use at the Laboratory which could be used to resolve MSNL address queries. All known MSNL addresses are stored in the name server or in the ANSA trader. For details on the MSNL name server, see the chapter in this document [1].

The name server queries the ANSA trader to resolve queries for which it does not have the appropriate information. The MSNL name server exports the `name_service` interface, and is assumed to be located at a well known address. Any Wanda or Unix program which wishes to use MSNL must use the MSNL name service for address resolution if the required MSNL address is not known. The equivalent of the Unix `gethostname, gethostbyname` and `gethostbyaddr` routines are implemented within the MSRPC library on all platforms to support MSNL address resolution. On Unix these routines are named `getmsnlhostname, getmsnlhostbyname` and `getmsnlhostbyaddr`.

If it isn't working: See The Don't Panic chapter in this document.

## 10 Acknowledgements

The original SUN RPC port for Wanda was written by Sai Lai Lo. Simon Crosby and Tristan Richardson (ORL) produced the first iteration of the Wanda MSRPC libraries, using Sai Lai's code. The Unix libraries were written by Simon Crosby. The MSRPC libraries and tools have been infinitely improved by several people, including Tristan Richardson (also the intrepid maintainer of `msrpcgen`), Shaw Chuang and Paul Barham, who seems to be particularly good at finding bugs. The MSNL name server was written by Shaw Chuang.

## References

[1] University of Cambridge Systems Research Group. ATM document collection. Technical note, University of Cambridge Computer Laboratory, Pembroke Street, Cambridge, CB2 3QG, UK, February 1993.

## A Client Session Operations

A client session object offers the following operations:

- `clnt_call`: Used to invoke a remote operation at the server.
- `clnt_abort`: Used to tidy up and kill a session binding if things go wrong (NOT IMPLEMENTED - use `clnt_destroy`).

- `clnt_freeres`: Should be called after every RPC when the results are no longer required. This frees all memory allocated for the results returned from the server. Note that it is important that results are freed after use or memory shortages could arise.
- `clnt_destroy`: Close the MSNL connection and destroy the session binding.
- `clnt_control`: Used to set and query various parameters of the session binding including: The buffer size used to marshal the arguments of calls (`MSRPC_CLSET_BUFSIZE`) and the timeout before declaring an RPC to have failed (`MSRPC_CLSET_TIMEOUT`). `CLGET_SERVER_ADDR` is used to obtain the MSNL address of the server to which the client is bound. Note that the buffer size should not exceed the current Wanda IO Buffer size limit in default kernels of 2 KB. Note also that attempts to marshal data which require buffer space in excess of the buffer size will result in an MSDR failure, reported by the MSRPC library as `MSRPC_CANT_ENCODE_ARGS`.

## B Server Session Operations

The server session object offers the following methods:

- `svc_recv`: Receive an RPC from a client.
- `svc_getargs`: Unmarshal the arguments recieved in a buffer from the underlying communication mechanism.
- `svc_freeargs`: Free memory associated with arguments from a client call.
- `svc_destroy`: Destroy a session binding to a particular client.
- `svc_control`: Get and set parameters associated with a binding. These include: Get/Set the timeout to wait before declaring the session dead. If the client does not send a request in this time then the session will be timed out and destroyed (`MSRPC_SVCSET_TIMEOUT`), get the client address (`MSRPC_SVCGET_CLIENT_ADDR`), and get/set the marshalling buffer size (`MSRPC_SVCSET_BUFSIZE`). The same limitations on buffer sizes apply to the server as to the client.

## C Errors

Anything could go wrong, at any time. The best advice is to always carry a blue towel. The following error messages may be produced, and may help:

- `MSRPCSTAT_CANT_SEND`: failed to send call (client) or reply. (server).
- `MSRPCSTAT_CANT_RECV`: failed to receive call (server) or reply. (client)
- `MSRPCSTAT_CANT_DECODE_ARGS`: server couldn't decode arguments.
- `MSRPCSTAT_OUT_OF_MEM`: out of memory.
- `MSRPCSTAT_IPC_ERROR`: failed in initialisation of the communication subsystem.
- `MSRPCSTAT_CANT_ENCODE_ARGS`: can't encode arguments (client).
- `MSRPCSTAT_CANT_DECODE_RES`: can't decode results (client).
- `MSRPCSTAT_TIMED_OUT`: call timed out (client).

- MSRPCSTAT_PROC_UNAVAIL: procedure unavailable at server (client).
- MSRPCSTAT_UNKNOWN_HOST: unknown host name used in binding call (client).
- MSRPCSTAT_PMAP_FAILURE: failed in call to portmapper (portmapper may not be running).
- MSRPCSTAT_PROG_NOT_REGISTERED: server is not registered with portmapper.
- MSRPCSTAT_UNKNOWN_PROTO: unknown protocol requested for server registered at the portmapper.
- MSRPCSTAT_PROG_ALREADY_REGISTERED: interface already registered on a different port at the portmapper (server).
- MSRPCSTAT_CANT_ENCODE_RES: can't encode results (server).
- MSRPCSTAT_IPC_CLOSED_CLEANLY: client closed connection cleanly (server).
- MSRPCSTAT_FAILED: any of the other random errors could have occurred. Use your blue towel.