RPC Protocol and Language Specification

B

This appendix specifies a message protocol used in implementing the RPC package. The message protocol is specified with the XDR language. The companion appendix to this one is Appendix C, “XDR Protocol Specification.".

Protocol Overview	*page 207*
Program and Procedure Numbers	*page 210*
Authentication Protocols	*page 218*
The RPC Language Specification	*page 230*

Protocol Overview

The RPC protocol provides for the following:

- Unique specification of a procedure to be called.
- Provisions for matching response messages to request messages.
- Provisions for authenticating the caller to service and vice-versa. In addition, the RPC package provides features that detect the following:
  - RPC protocol mismatches
  - Remote program protocol version mismatches
  - Protocol errors (such as incorrect specification of a procedure’s parameters)
  - Reasons why remote authentication failed

Consider a network file service composed of two programs. One program may deal with high-level applications such as file system access control and locking. The other may deal with low-level file I/O and have procedures like "read" and "write." A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine. In the client-server model a remote procedure call is used to call the service.

The RPC Model

The RPC model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location. The caller then transfers control to the procedure, and eventually regains control. At that point, the results of the procedure are extracted from a well-specified location, and the caller continues execution.

The RPC is similar, in that one thread of control logically winds through two processes. One is the caller's process, the other is a server's process. Conceptually, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

Note that in this description only one of the two processes is active at any given time. However, this need not be the case. The RPC protocol makes no restrictions on the concurrency model implemented. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC disregards how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

RPC does not attempt to ensure transport reliability. Therefore, you must supply the application with information about the type of transport protocol used under RPC. If you tell the RPC service it is running on top of a reliable transport such as TCP, most of the work is already done for it. On the other hand, if RPC is running on top of an unreliable transport such as UDP, the service must devise its own retransmission and time-out policy. RPC does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, it can infer that the procedure was executed at least once.

A server may choose to remember previously granted requests from a client and not regrant them to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction ID is by the RPC client for matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, checking this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

Binding and Rendezvous Independence

The act of binding a client to a service is *not* part of the remote procedure call specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself; see "rpcbind Protocol" on page 240.)

Implementers should think of the RPC protocol as the jump-subroutine instruction (JSR) of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, enabling RPC to accomplish this task.

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice-versa. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header specifies the protocol being used. More information on authentication protocols can be found in the section "Record-Marking Standard" on page 217.

Program and Procedure Numbers

The RPC call message has three unsigned fields that uniquely identify the procedure to be called:

- remote program number
- remote program version number
- remote procedure number

Program numbers are administered by a central authority, as described in "Program Number Registration" on page 213.

The first implementation of a program will most likely have version number 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies the version of the protocol the caller is using. Version numbers make "speaking" old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the individual program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is "read" and procedure number 12 is "write."

Just as remote program protocols may change over several versions, the RPC message protocol itself may change. Therefore, the call message also has in it the RPC version number, which is always equal to 2 for the version of RPC described here.

The reply message to a request message has enough information to distinguish the following error conditions:

- The remote implementation of RPC does not “speak” protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. (This is usually a caller-side protocol or programming error.)
- The parameters to the remote procedure appear to be garbage from the server’s point of view. (Again, this is usually caused by a disagreement about the protocol between client and service.)

Provisions for authentication of caller to service and vice versa are provided as a part of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
    AUTH_NONE = 0,
    AUTH_SYS = 1,
    AUTH_SHORT = 2,
    AUTH_DES = 3,
    AUTH_KERB = 4
    /* and more to be defined */
 };
 struct opaque_auth {
    enum       auth_flavor;        /* style of credentials */
    caddr_t    oa_base;          /* address of more auth stuff */
    u_int      oa_length;        /* not to exceed MAX_AUTH_BYTES */
 };
```

An `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes that are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields are specified by individual, independent authentication protocol specifications. (See "Record-Marking Standard" on page 217 for definitions of the various authentication protocols.)

If authentication parameters are rejected, the response message contains information stating why they are rejected.

Program Number Assignment

Program numbers are distributed in groups of `0x20000000`, as shown in Table B-1.

Table B-1 RPC Program Number Assignment

Program Numbers	Description
00000000 - 1fffffff	Defined by Sun
20000000 - 3fffffff	Defined by user
40000000 - 5fffffff	Transient (Reserved for customer-written applications)
60000000 - 7fffffff	Reserved
80000000 - 9fffffff	Reserved
a0000000 - bfffffff	Reserved
c0000000 - dfffffff	Reserved
e0000000 - ffffffff	Reserved

Sun Microsystems administers the first group of numbers, which should be identical for all customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range.

The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs.

The third group is reserved for applications that generate program numbers dynamically.

The final groups are reserved for future use, and should not be used.

Program Number Registration

To register a protocol specification, send a request by email to `rpc@sun.com`, or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043

Please include a compilable `rpcgen ".x"` file describing your protocol. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard RPC services can be found in the include files in `/usr/include/rpcsvc`. These services, however, constitute only a small subset of those that have been registered.

Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. Some of the non-RPC protocols supported by the RPC package are batching and broadcasting.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching typically uses reliable byte stream protocols (like TCP) for its transport. In batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually finished by a non-batch RPC call to flush the pipeline (with positive acknowledgment). For more information, see “Batching” on page 92.

Broadcast RPC

In broadcast RPC, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses connectionless, packet-based protocols (like UDP) as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the `rpcbind` service to achieve its semantics. See "Broadcast RPC" on page 90 and "rpcbind Protocol" on page 240 for further information.

The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style, as shown in Code Example B-1.

Code Example B-1 RPC Message Protocol

```
enum msg_type {
    CALL = 0,
    REPLY = 1
 };

/*
 * A reply to a call message can take on two forms: The message was
 * either accepted or rejected.
 */
 enum reply_stat {
    MSG_ACCEPTED = 0,
    MSG_DENIED = 1
 };


/*
 * Given that a call message was accepted, the following is the
 * status of an attempt to call a remote procedure.
 */
enum accept_stat {
    SUCCESS = 0,        /* RPC executed successfully */
    PROG_UNAVAIL = 1,   /* remote service hasn't exported prog */
    PROG_MISMATCH = 2, /* remote service can't support versn # */
    PROC_UNAVAIL = 3,   /* program can't support proc */
    GARBAGE_ARGS = 4    /* procedure can't decode params */
};
```

```
/*
 * Reasons a call message was rejected:
 */
enum reject_stat {
    RPC_MISMATCH = 0,  /* RPC version number != 2 */
    AUTH_ERROR = 1     /* remote can't authenticate caller */
};
/*
 * Why authentication failed:
 */
enum auth_stat {
    AUTH_BADCRED = 1,        /* bad credentials */
    AUTH_REJECTEDCRED = 2,   /* clnt must do new session */
    AUTH_BADVERF = 3,        /* bad verifier */
    AUTH_REJECTEDVERF = 4,   /* verif expired or replayed */
    AUTH_TOOWEAK = 5         /* rejected for security */
};

/*
 * The RPC message:
 * All messages start with a transaction identifier, xid, followed
 * by a two-armed discriminated union. The union's discriminant is
 * a msg_type which switches to one of the two types of the message.
 * The xid of a REPLY message always matches that of the initiating
 * CALL message. NB: The xid field is only used for clients matching
 * reply messages with call messages or for servers detecting
 * retransmissions; the service side cannot treat this id as any type
 * of sequence number.
 */
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification, rpcvers must be
 * equal to 2. The fields prog, vers, and proc specify the remote
 * program, its version number, and the procedure within the remote
 * program to be called. After these fields are two authentication
 * parameters: cred (authentication credentials) and verf
```

```
 * (authentication verifier). The two authentication parameters are
 * followed by the parameters to the remote procedure, which are
 * specified by the specific program protocol.
 */
struct call_body {
    unsigned int rpcvers; /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* procedure specific parameters start here */
 };

/*
 * Body of a reply to an RPC request:
 * The call message was either accepted or rejected.
 */
union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;

/*
 * Reply to an RPC request that was accepted by the server: there
 * could be an error even though the request was accepted. The first
 * field is an authentication verifier that the server generates in
 * order to validate itself to the caller. It is followed by a union
 * whose discriminant is an enum accept_stat. The SUCCESS arm of the
 * union is protocol specific. The PROG_UNAVAIL, PROC_UNAVAIL, and
 * GARBAGE_ARGP arms of the union are void. The PROG_MISMATCH arm
 * specifies the lowest and highest version numbers of the remote
 * program supported by the server.
 */
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
        case SUCCESS:
            opaque results[0];
            /* procedure-specific results start here */
        case PROG_MISMATCH:
            struct {
                unsigned int low;
                unsigned int high;
```

```
            } mismatch_info;
        default:
            /*
             * Void. Cases include PROG_UNAVAIL, PROC_UNAVAIL, and
             * GARBAGE_ARGS.
             */
            void;
    } reply_data;
};

/*
 * Reply to an RPC request that was rejected by the server:
 * The request can be rejected for two reasons: either the server is
 * not running a compatible version of the RPC protocol
 * (RPC_MISMATCH), or the server refuses to authenticate the caller
 * (AUTH_ERROR). In case of an RPC version mismatch, the server
 * returns the lowest and highest supported RPC version numbers. In
 * case of refused authentication, failure status is returned.
 */
union rejected_reply switch (reject_stat stat) {
    case RPC_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};
```

Record-Marking Standard

When RPC messages are passed on top of a byte stream transport (like TCP), it is necessary, or at least desirable, to delimit one message from another to detect and possibly recover from user protocol errors. This is called record marking (RM). One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by `0` to `(2**31) - 1` bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is the network byte order.

The header encodes two values:

- A Boolean that specifies whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment).
- A 31-bit unsigned binary value that is the length in bytes of the fragment's data. The Boolean value is the highest-order bit of the header; the length is the 31 low-order bits. (This record specification is *not* in XDR standard form).

Authentication Protocols

Authentication parameters are opaque but open-ended to the rest of the RPC protocol. This section defines some flavors of authentication that have already been implemented. Other sites are free to invent new authentication types, with the same rules of flavor number assignment for program number assignment. SunSoft maintains and administers a range of authentication flavors. To have authentication numbers (like RPC program numbers) allocated (or registered to them), contact the Sun RPC Administrator, as described in "Program Number Registration" on page 213.

AUTH_NONE

Calls are often made where the caller does not authenticate itself and the server disregards who the caller is. In these cases, the `flavor` value (the "discriminant" of the `opaque_auth` "union") of the RPC message's credentials, verifier, and response verifier is `AUTH_NONE`. The body length is zero when `AUTH_NONE` authentication flavor is used.

AUTH_SYS

This is the same as the authentication flavor previously known as `AUTH_UNIX`. The caller of a remote procedure may wish to identify itself using traditional UNIX process permissions authentication. The `flavor` of the `opaque_auth` of such an RPC call message is `AUTH_SYS`. The bytes of the `body` encode the following structure:

```
struct auth_sysparms {
    unsigned int stamp;
    string machinename<255>;
    uid_t uid;
```

```
    gid_t gid;
    gid_t gids<10>;
};
```

- *stamp* is an arbitrary ID that the caller machine may generate.
- *machinename* is the name of the caller's machine.
- *uid* is the caller's effective user ID.
- *gid* is the caller's effective group ID.
- *gids* is a counted array of groups in which the caller is a member.

The `flavor` of the verifier accompanying the credentials should be `AUTH_NONE`.

The AUTH_SHORT Verifier

When using `AUTH_SYS` authentication, the `flavor` of the response verifier received in the reply message from the server may be `AUTH_NONE` or `AUTH_SHORT`.

If `AUTH_SHORT`, the bytes of the response verifier's string encode a `short_hand_verf` structure. This opaque structure may now be passed to the server instead of the original `AUTH_SYS` credentials.

The server keeps a cache that maps shorthand opaque structures (passed back by way of an `AUTH_SHORT` style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials.

The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message will be rejected owing to an authentication error. The reason for the failure will be `AUTH_REJECTEDCRED`. At this point, the caller may wish to try the original `AUTH_SYS` style of credentials. See Figure B-1.

B

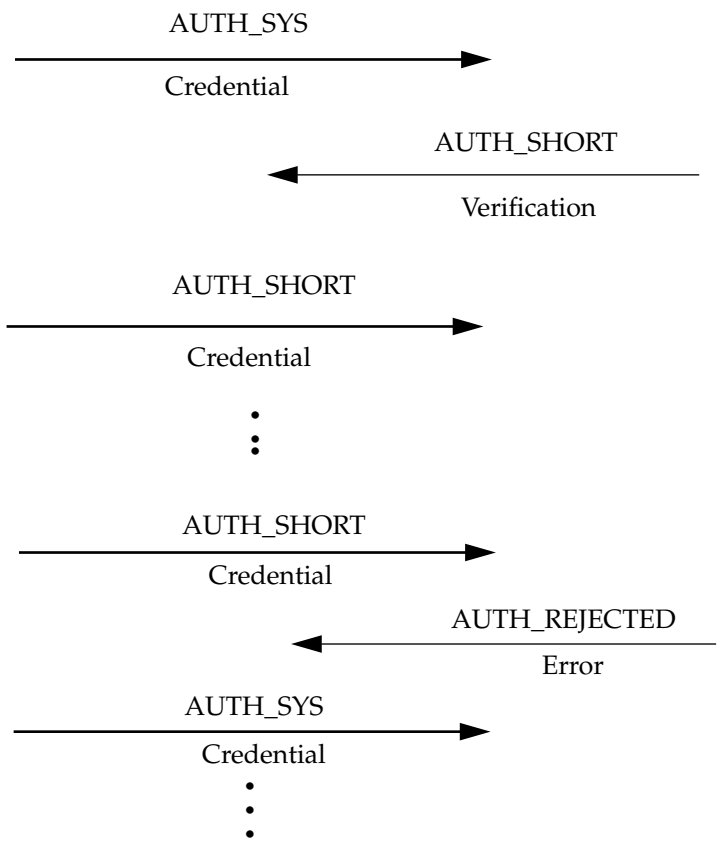

Figure B-1 Authentication Process Map

AUTH_DES Authentication

`AUTH_SYS` authentication has the following problems:

1. Caller identification cannot be guaranteed to be unique if machines with differing operating systems are on the same network.
2. There is no verifier, so credentials can easily be faked. `AUTH_DES` authentication attempts to fix these two problems.

The first problem is handled by addressing the caller by a simple string of characters instead of by an operating system specific integer. This string of characters is known as the netname or network name of the caller. The server should not interpret the caller's name in any way other than as the identity of the caller. Thus, netnames should be unique for every caller in the naming domain.

It is up to each operating system's implementation of `AUTH_DES` authentication to generate netnames for its users that ensure this uniqueness when they call remote servers. Operating systems already distinguish users local to their systems. It is usually a simple matter to extend this mechanism to the network. For example, a user with a user ID of 515 might be assigned the following netname: "`UNIX.515@sun.com`". This netname contains three items that serve to ensure it is unique. Going backward, there is only one naming domain called `sun.com` in the Internet. Within this domain, there is only one `UNIX` user with user ID `515`. However, there may be another user on another operating system, for example `VMS`, within the same naming domain who, by coincidence, happens to have the same user ID. To ensure that these two users can be distinguished you add the operating system name. So one user is "`UNIX.515@sun.com`" and the other is "`VMS.515@sun.com`".

The first field is actually a naming method rather than an operating system name. It just happens that there is almost a one-to-one correspondence between naming methods and operating systems. If the world could agree on a naming standard, the first field could be a name from that standard, instead of an operating system name.

AUTH_DES Authentication Verifiers

Unlike `AUTH_SYS` authentication, `AUTH_DES` authentication does have a verifier so the server can validate the client's credential (and vice versa). The contents of this verifier are primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to its current real time, then the client must have encrypted it correctly. The only way the client could encrypt it correctly is to know the conversation key of the RPC session. If the client knows the conversation key, it must be the real client.

The conversation key is a DES [5] key that the client generates and notifies the server of in its first RPC call. The conversation key is encrypted using a public key scheme in this first transaction. The particular public key scheme used in `AUTH_DES` authentication is Diffie-Hellman [3] with 192-bit keys. The details of this encryption method are described later.

The client and the server need the same notion of the current time for this to work. If network time synchronization cannot be guaranteed, then client can synchronize with the server before beginning the conversation. `rpcbind` provides a procedure, `RPCBPROC_GETTIME`, which may be used to obtain the current time.

A server can determine if a client timestamp is valid. For any transaction after the first, the server checks for two things:

- The timestamp is greater than the one previously seen from the same client.
- The timestamp has not expired. A timestamp is expired if the server's time is later than the sum of the client's timestamp plus what is known as the client's *window*. The window is a number the client passes (encrypted) to the server in its first transaction. The window can be thought of as a lifetime for the credential.

For the first transaction, the server checks that the timestamp has not expired. As an added check, the client sends an encrypted item in the first transaction known as the *window verifier* which must be equal to the window minus 1, or the server will reject the credential.

The client must check the verifier returned from the server to be sure it is legitimate. The server sends back to the client the encrypted timestamp it received from the client, minus one second. If the client gets anything other than this, it will reject it.

Nicknames and Clock Synchronization

After the first transaction, the server's `AUTH_DES` authentication subsystem returns in its verifier to the client an integer *nickname* that the client may use in its further transactions instead of passing its netname, encrypted DES key and window every time. The nickname is most likely an index into a table on the server that stores for each client its netname, decrypted DES key and window. It should however be treated an opaque data by the client.

Though originally synchronized, client and server clocks can get out of sync. If this happens, the client RPC subsystem most likely will receive an RPC_AUTHERROR at which point it should resynchronize.

A client may still get the RPC_AUTHERROR error even though it is synchronized with the server. The reason is that the server's nickname table is a limited size, and it may flush entries whenever it wants. The client should resend its original credential and the server will give it a new nickname. If a server crashes, the entire nickname table will be flushed, and all clients will have to resend their original credentials.

DES Authentication Protocol (in XDR language)

Code Example B-2 AUTH_DES Authentication Protocol

```
/*
 * There are two kinds of credentials: one in which the client uses
 * its full network name, and one in which it uses its "nickname"
 * (just an unsigned integer) given to it by the server. The client
 * must use its full name in its first transaction with the server,
 * in which the server will return to the client its nickname. The
 * client may use its nickname in all further transactions with the
 * server. There is no requirement to use the nickname, but it is
 * wise to use it for performance reasons.
 */
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
 };

/*
 * A 64-bit block of encrypted DES data
 */
 typedef opaque des_block[8];

/*
 * Maximum length of a network user's name
 */
const MAXNETNAMELEN = 255;

/*
 * A fullname contains the network name of the client, an encrypted
 * conversation key and the window. The window is actually a lifetime
 * for the credential. If the time indicated in the verifier
 * timestamp plus the window has passed, then the server should
```

```
 * expire the request and not grant it. To insure that requests are
 * not replayed, the server should insist that timestamps be greater
 * than the previous one seen, unless it is the first transaction.
 * In the first transaction, the server checks instead that the
 * window verifier is one less than the window.
 */
struct authdes_fullname {
    string name<MAXNETNAMELEN>; /* name of client */
    des_block key;              /* PK encrypted conversation key */
    unsigned int window;        /* encrypted window */
};                              /* NOTE: PK means "public key" */

/*
 * A credential is either a fullname or a nickname
 */
unionauthdes_credswitch(authdes_namekindadc_namekind){
    case ADN_FULLNAME:
        authdes_fullname adc_fullname;
    case ADN_NICKNAME:
        unsigned int adc_nickname;
};

/*
 * A timestamp encodes the time since midnight, January 1, 1970.
 */
struct timestamp {
    unsigned int seconds;       /* seconds */
    unsigned int useconds;      /* and microseconds */
};

/*
 * Verifier: client variety
 */
struct authdes_verf_clnt {
    timestamp adv_timestamp;    /* encrypted timestamp */
    unsigned int adv_winverf;   /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client gave
 * it minus one second. It also tells the client its nickname to be
 * used in future transactions (unencrypted).
 */
struct authdes_verf_svr {
```

```
    timestamp adv_timeverf;    /* encrypted verifier */
    unsigned int adv_nickname; /* new nickname for clnt */
};
```

Diffie-Hellman Encryption

In this scheme, there are two constants, `PROOT` and `HEXMODULUS`. The particular values chosen for these for the DES authentication protocol are:

```
const PROOT = 3;
const HEXMODULUS =/* hex */
 "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b";
```

The way this scheme works is best explained by an example. Suppose there are two people “A” and “B” who want to send encrypted messages to each other. A and B each generate a random secret key that they do not disclose to anyone. Let these keys be represented as SK(A) and SK(B). They also publish in a public directory their public keys. These keys are computed as follows:

```
PK(A) = (PROOT ** SK(A)) mod HEXMODULUS
PK(B) = (PROOT ** SK(B)) mod HEXMODULUS
```

The `**` notation is used here to represent exponentiation.

Now, both A and B can arrive at the common key between them, represented here as `CK(A,B)`, without disclosing their secret keys.

A computes:

```
CK(A, B) = (PK(B) ** SK(A)) mod HEXMODULUS
```

while B computes:

```
CK(A, B) = (PK(A) ** SK(B)) mod HEXMODULUS
```

These two can be shown to be equivalent: `(PK(B)**SK(A)) mod HEXMODULUS = (PK(A)**SK(B)) mod HEXMODULUS` Drop the `mod HEXMODULUS` parts and assume modulo arithmetic to simplify the process:

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

Then replace `PK(B)` by what B computed earlier and likewise for `PK(A)`.

```
((PROOT ** SK(B)) ** SK(A) = (PROOT ** SK(A)) ** SK(B)
```

which leads to:

```
PROOT ** (SK(A) * SK(B)) = PROOT ** (SK(A) * SK(B))
```

This common key `CK(A,B)` is not used to encrypt the timestamps used in the protocol. It is used only to encrypt a conversation key that is then used to encrypt the timestamps. The reason for doing this is to use the common key as little as possible, for fear that it could be broken. Breaking the conversation key is a far less serious offense, because conversations are comparatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8 bytes are selected from the common key, and then parity is added to the lower order bit of each byte, producing a 56-bit key with 8 bits of parity.

AUTH_KERB Authentication

To avoid compiling Kerberos code into the operating system kernel, the kernel used in the SunOS 5.x implementation of `AUTH_KERB` uses a proxy RPC daemon called `kerbd`. The daemon exports three procedures. Refer to the `kerbd(1M)` manpage for more details.

1. `KGETKCRED` is used by the server-side RPC to check the authenticator presented by the client.
2. `KSETKCRED` returns the encrypted ticket and DES session key, given a primary name, instance, and realm.
3. `KGETUCRED` is UNIX-specific. It returns the user's ID, the group ID, and groups list, assuming that the primary name is mapped to a user name known to the server.

The best way to describe how Kerberos works is to use an example based on a service currently implementing Kerberos: the network file system (NFS). The NFS service on server *s* is assumed to have the well-known principal name nfs.*s* A privileged user on client *c* is assumed to have the primary name `root` and an instance *c*. Note that (unlike `AUTH_DES`) when the user's ticket-granting ticket has expired, `kinit()` must be reinvoked. NFS service for Kerberos mounts will fail until a new ticket-granting ticket is obtained.

NFS Mount Example

This section follows an NFS mount request from start to finish using `AUTH_KERB`. Since mount requests are executed as root in SunOS, the user's identity is `root`.*c*.

Client *c* makes a `MOUNTPROC_MOUNT` request to the server *s* to obtain the file handle for the directory to be mounted. The client mount program makes an NFS mount system call, handing the client kernel the file handle, mount flavor, time synchronization address, and the server's well-known name, `nfs`.*s*. Next the client kernel contacts the server at the time synchronization host to obtain the client-server time bias.

The client kernel makes the following RPC calls: (1) `KSETKCRED` to the local `kerbd` to obtain the ticket and session key, (2) `NFSPROC_GETATTR` to the server's NFS service, using the full name credential and verifier. The server receives the calls and makes the `KGETKCRED` call to its local `kerbd` to check the client's ticket.

The server's `kerbd` and the Kerberos library decrypt the ticket and return, among other data, the principal name and DES session key. The server checks that the ticket is still valid, uses the session key to decrypt the DES-encrypted portions of the credential and verifier, and checks that the verifier is valid.

The possible Kerberos authentication errors returned at this time are:

- `AUTH_BADCRED` is returned if the verifier is invalid (the decrypted `win` in the credential and `win +1` in the verifier do not match), or the timestamp is not within the window range
- `AUTH_REJECTEDCRED` is returned if a replay is detected
- `AUTH_BADVERF` is returned if the verifier is garbled

If no errors are received, the server caches the client's identity and allocates a nickname (small integer) to be returned in the NFS reply. The server then checks if the client is in the same realm as the server. If it is, the server calls `KGETUCRED` to its local `kerbd` to translate the principal's primary name into UNIX credentials. If it is not translatable, the user is marked anonymous. The server checks these credentials against the file system's export information. There are three cases to consider:

1. If the `KGETUCRED` call fails and anonymous requests are allowed, the UNIX credentials of the anonymous user are assigned.

2. If the KGETUCRED call fails and anonymous requests are not allowed, the NFS call fails with the AUTH_TOOWEAK.

3. If the KGETUCRED call succeeds, the credentials are assigned, and normal protection checking follows, including checking for root permission.

Next the server sends an NFS reply, including the nickname and server's verifier. The client receives the reply, decrypts and validates the verifier, and stores the nickname for future calls. The client makes a second NFS call to the server, and the calls to the server described earlier are repeated. The client kernel makes an NFSPROC_STATVFS call to the server's NFS service, using the nickname credential and verifier described previously. The server receives the call and validates the nickname. If it is out of range, the error AUTH_BADCRED is returned. The server uses the session key just obtained to decrypt the DES-encrypted portions of the verifier and validates the verifier.

The possible Kerberos authentication errors returned at this time are:

- AUTH_REJECTEDVERF is returned if the timestamp is invalid, a replay is detected, or if the timestamp is not within the window range
- AUTH_TIMEEXPIRE is returned if the service ticket is expired

If no errors are received, the server uses the nickname to retrieve the caller's UNIX credentials. Then it checks these credentials against the file system's export information, and sends an NFS reply that includes the nickname and the server's verifier. The client receives the reply, decrypts and validates the verifier, and stores the nickname for future calls. Last, the client's NFS mount system call returns, and the request is finished.

KERB Authentication Protocol (in XDR Language)

Code Example B-3 (AUTH_KERB) has many similarities to the one for AUTH_DES, shown in Code Example B-2 on page 223. Note the differences.

Code Example B-3 AUTH_KERB Authentication Protocol

```
#define AUTH_KERB 4

/*
 * There are two kinds of credentials: one in which the client sends
 * the (previously encrypted) Kerberos ticket, and one in which it
 * uses its "nickname" (just an unsigned integer) given to it by the
 * server. The client must use its full name in its first transaction
 * with the server, in which the server will return to the client
 * its nickname. The client may use its nickname in all further
```

```
 * transactions with the server (until the ticket expires). There is
 * no requirement to use the nickname, but it is wise to use it for
 * performance reasons.
 */
enum authkerb_namekind {
    AKN_FULLNAME = 0,
    AKN_NICKNAME = 1
 };

/*
 * A fullname contains the encrypted service ticket and the
 * window. The window is actually a lifetime
 * for the credential. If the time indicated in the verifier
 * timestamp plus the window has passed, then the server should
 * expire the request and not grant it. To insure that requests are
 * not replayed, the server should insist that timestamps be greater
 * than the previous one seen, unless it is the first transaction.
 * In the first transaction, the server checks instead that the
 * window verifier is one less than the window.
 */
struct authkerb_fullname {
    KTEXT_ST ticket;              /* Kerberos service ticket */
    unsigned long window;        /* encrypted window */
};
/*
 * A credential is either a fullname or a nickname
 */
union authkerb_credswitch(authkerb_namekind akc_namekind){
    case AKN_FULLNAME:
        authkerb_fullname akc_fullname;
    case AKN_NICKNAME:
        unsigned long akc_nickname;
};

/*
 * A timestamp encodes the time since midnight, January 1, 1970.
 */
struct timestamp {
    unsigned long seconds;      /* seconds */
    unsigned long useconds;     /* and microseconds */
};

/*
 * Verifier: client variety
 */
struct authkerb_verf_clnt {
```

```
    timestamp akv_timestamp;   /* encrypted timestamp */
    unsigned long akv_winverf;  /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client gave
 * it minus one second. It also tells the client its nickname to be
 * used in future transactions (unencrypted).
 */
struct authkerb_verf_svr {
    timestamp akv_timeverf;    /* encrypted verifier */
    unsigned long akv_nickname; /* new nickname for clnt */
};
```

The RPC Language Specification

Just as there was a need to describe the XDR data types in a formal language, there is also need to describe the procedures that operate on these XDR data types in a formal language as well. The RPC Language, an extension to the XDR language, serves this purpose. The following example is used to describe the essence of the language.

An Example Service Described in the RPC Language

Code Example B-4 shows the specification of a simple `ping` program.

Code Example B-4 `ping` Service Using RPC Language

```
/*
 * Simple ping program
 */
program PING_PROG {
    version PING_VERS_PINGBACK {
        void
        PINGPROC_NULL(void) = 0;
        /*
         * ping the caller, return the round-trip time
         * in milliseconds. Return a minus one (-1) if
         * operation times-out
         */
        int
        PINGPROC_PINGBACK(void) = 1;
        /* void - above is an argument to the call */
```

```
    } = 2;
/*
 * Original version
 */
    version PING_VERS_ORIG {
        void
        PINGPROC_NULL(void) = 0;
    } = 1;
} = 200000;
const PING_VERS = 2; /* latest version */
```

The first version described is `PING_VERS_PINGBACK` with two procedures, `PINGPROC_NULL` and `PINGPROC_PINGBACK`.

`PINGPROC_NULL` takes no arguments and returns no results, but it is useful for such things as computing round-trip times from the client to the server and back again. By convention, procedure `0` of any RPC program should have the same semantics, and never require authentication.

The second procedure returns the amount of time (in microseconds) that the operation used.

The next version, `PING_VERS_ORIG`, is the original version of the protocol and it does not contain `PINGPROC_PINGBACK` procedure. It is useful for compatibility with old client programs, and as this program matures it may be dropped from the protocol entirely.

RPCL Syntax

The RPC language (RPCL) is similar to C. This section describes the syntax of the RPC language, showing a few examples along the way. It also shows how RPC and XDR type definitions get compiled into C type definitions in the output header file.

An RPC language file consists of a series of definitions.

```
definition-list:
    definition;
    definition; definition-list
```

It recognizes six types of definitions.

```
definition:
```

```
enum-definition
const-definition
typedef-definition
struct-definition
union-definition
program-definition
```

Definitions are not the same as declarations. No space is allocated by a definition – only the type definition of a single or series of data elements. This means that variables still must be declared.

The RPC language is identical to the XDR language, except for the added definitions described in Table B-2.

Table B-2 RPC Language Definitions

Term	Definition
program-definition	`program program-ident {version-list} = value`
version-list	`version;` version; version-list
version	version version-ident {procedure-list} = value
procedure-list	procedure; procedure; procedure-list
procedure	type-ident procedure-ident (type-ident) = value

- The following keywords are added and cannot be used as identifiers: program version.
- Neither version name nor a version number can occur more than once within the scope of a program definition.
- Neither a procedure name nor a procedure number can occur more than once within the scope of a version definition.
- Program identifiers are in the same name space as constant and type identifiers.
- Only unsigned constants can be assigned to programs, versions, and procedures.

Enumerations

RPC/XDR enumerations have the same syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"
enum-value-list:
    enum-value
    enum-value "," enum-value-list
enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is an example of an XDR `enum` and the C `enum` to which it gets compiled.

```
enum colortype {                enum colortype {
    RED = 0,                        RED = 0,
    GREEN = 1,        -->           GREEN = 1,
    BLUE = 2                        BLUE = 2,
};                                  };
                                    typedef enum colortype colortype;
```

Constants

XDR symbolic constants may be used wherever an integer constant is used. For example, in array size specifications:

```
const-definition:
    const const-ident = integer
```

The following example defines a constant, `DOZEN` as equal to `12`:

```
const DOZEN = 12; --> #define DOZEN 12
```

Type Definitions

XDR `typedefs` have the same syntax as C `typedefs`.

```
typedef-definition:
        typedef declaration
```

This example defines an `fname_type` used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

B

Declarations

In XDR, there are four kinds of declarations. These declarations must be a part of a `struct` or a `typedef`; they cannot stand alone:

```
declaration:
        simple-declaration
        fixed-array-declaration
        variable-array-declaration
        pointer-declaration
```

Simple Declarations

Simple declarations are just like simple C declarations:

```
simple-declaration:
        type-ident variable-ident
```

Example:

```
    colortype color; --> colortype color;
```

Fixed-Length Array Declarations

Fixed-length array declarations are just like C array declarations:

```
fixed-array-declaration:
        type-ident variable-ident [value]
```

Example:

```
colortype palette[8]; --> colortype palette[8];
```

Many programmers confuse variable declarations with type declarations. It is important to note that `rpcgen` does not support variable declarations. This example is a program that will not compile:

```
int data[10];
program P {
    version V {
        int PROC(data) = 1;
    } = 1;
} = 0x200000;
```

The example above will not compile because of the variable declaration:

```
int data[10]
```

Instead, use:

```
typedef int data[10];
```

or

```
struct data {int dummy [10]};
```

Variable-Length Array Declarations

Variable-length array declarations have no explicit syntax in C. The XDR language does have a syntax, using angle brackets:

```
variable-array-declaration:
        type-ident variable-ident <value>
        type-ident variable-ident < >
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size:

```
int heights<12>; /* at most 12 items */
int widths<>; /* any number of items */
```

Because variable-length arrays have no explicit syntax in C, these declarations are compiled into `struct` declarations. For example, the `heights` declaration compiled into the following `struct`:

```
struct {
    u_int heights_len;              /* # of items in array */
    int *heights_val;               /* pointer to array */
} heights;
```

The number of items in the array is stored in the `_len` component and the pointer to the array is stored in the `_val` component. The first part of each component name is the same as the name of the declared XDR variable (`heights`).

Pointer Declarations

Pointer declarations are made in XDR exactly as they are in C. Address pointers are not really sent over the network; instead, XDR pointers are useful for sending recursive data types such as lists and trees. The type is called “optional-data,” not “pointer,” in XDR language:

```
pointer-declaration:
    type-ident *variable-ident
```

Example:

```
listitem *next; --> listitem *next;
```

Structures

An RPC/XDR `struct` is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    struct struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

The following XDR structure is an example of a two-dimensional coordinate and the C structure that it compiles into:

```
struct coord {                  struct coord {
    int x;            -->           int x;
    int y;                          int y;
};                                  };
                                    typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This enables one to use `coord` instead of `struct coord` when declaring items.

Unions

XDR unions are discriminated unions, and do not look like C unions – they are more similar to Pascal variant records:

```
union-definition:
    "union" union-ident "switch" "("simple declaration")" "{"
        case-list
    "}"

case-list:
    "case" value ":" declaration ";"
    "case" value ":" declaration ";" case-list
    "default" ":" declaration ";"
```

The following is an example of a type returned as the result of a “read data” operation: If there is no error, return a block of data; otherwise, don’t return anything.

```
union read_result switch (int errno) {
    case 0:
        opaque data[1024];
    default:
        void;
    };
```

It compiles into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the same name as the type name, except for the trailing _u.

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value;

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value;

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value;
```

When the -N option is specified, rpcgen also recognizes the following syntax:

```
procedure:
    type-ident procedure-ident "(" type-ident-list ")" "=" value;
type-ident-list:
    type-ident
    type-ident "," type-ident-list
```

For example:

```
/*
 * time.x: Get or set the time. Time is represented as seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 0x20000044;
```

Note that the void argument type means that no argument is passed.

This file compiles into these #define statements in the output header file:

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Special Cases

There are several exceptions to the RPC language rules.

C-style Mode

In the new features section we talked about the features of the C-style mode of rpcgen. These features have implications with regard to the passing of void arguments. No arguments need be passed if their value is void.

Booleans

C has no built-in boolean type. However, the RPC library uses a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Parameters declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married; --> bool_t married;
```

Strings

The C language has no built-in string type, but instead uses the null-terminated `char *` convention. In C, strings are usually treated as null-terminated single-dimensional arrays.

In XDR language, strings are declared using the `string` keyword, and compiled into type `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the `NULL` character). The maximum size may be omitted, indicating a string of arbitrary length.

Examples:

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

Note – `NULL` strings cannot be passed; however, a zero-length string (that is, just the terminator or `NULL` byte) can be passed.

Opaque Data

Opaque data is used in XDR to describe untyped data, that is, sequences of arbitrary bytes. It may be declared either as a fixed length or variable length array. Examples:

```
opaque diskblock[512]; --> char diskblock[512];

opaque filedata<1024>; --> struct {
                               u_int filedata_len;
                               char *filedata_val;
                           } filedata;
```

Voids

In a void declaration, the variable is not named. The declaration is just `void` and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure, for example no arguments are passed.)

`rpcbind` *Protocol*

`rpcbind` maps RPC program and version numbers to universal addresses, thus making dynamic binding of remote programs possible.

`rpcbind` is bound to a well-known address of each supported transport, and other programs register their dynamically allocated transport addresses with it. `rpcbind` then makes those addresses publicly available. Universal addresses are string representations of the transport-dependent address. They are defined by the addressing authority of the given transport.

`rpcbind` also aids in broadcast RPC. RPC programs will have different addresses on different machines, so there is no way to broadcast directly to all these programs. `rpcbind`, however, has a well-known address. So, to broadcast to a given program, the client actually sends its message to the `rpcbind` process on the machine it chooses to reach. `rpcbind` picks up the broadcast and calls the local service specified by the client. When `rpcbind` gets a reply from the local service, it passes the reply on to the client.

Code Example B-5 `rpcbind` Protocol Specification (in RPC Language)

```
/*
 * rpcb_prot.x
 * RPCBIND protocol in rpc language
 */

/*
 * A mapping of (program, version, network ID) to universal address
 */
struct rpcb {
    unsigned long r_prog;           /* program number */
    unsigned long r_vers;           /* version number */
    string r_netid<>;               /* network id */
    string r_addr<>;                /* universal address */
    string r_owner<>;               /* owner of this service */ };
```

```
/* A list of mappings */
struct rpcblist {
    rpcb rpcb_map;
    struct rpcblist *rpcb_next;
};

/* Arguments of remote calls */
struct rpcb_rmtcallargs {
    unsigned long prog;             /* program number */
    unsigned long vers;             /* version number */
    unsigned long proc;             /* procedure number */
    opaque args<>;                  /* argument */
};

/* Results of the remote call */
struct rpcb_rmtcallres {
    string addr<>;                  /* remote universal address */
    opaque results<>;               /* result */
};

/*
 * rpcb_entry contains a merged address of a service on a particular
 * transport, plus associated netconfig information. A list of
 * rpcb_entrys is returned by RPCBPROC_GETADDRLIST. See netconfig.h
 * for values used in r_nc_* fields.
 */
struct rpcb_entry {
    string          r_maddr<>;      /* merged address of service */
    string          r_nc_netid<>;   /* netid field */
    unsigned long   r_nc_semantics; /* semantics of transport */
    string          r_nc_protofmly<>; /* protocol family */
    string          r_nc_proto<>;   /* protocol name */
};

/* A list of addresses supported by a service. */
struct rpcb_entry_list {
    rpcb_entry rpcb_entry_map;
    struct rpcb_entry_list *rpcb_entry_next;
};

typedef rpcb_entry_list *rpcb_entry_list_ptr;

/* rpcbind statistics */
const rpcb_highproc_2 = RPCBPROC_CALLIT;
const rpcb_highproc_3 = RPCBPROC_TADDR2UADDR;
const rpcb_highproc_4 = RPCBPROC_GETSTAT;
```

```
const RPCBSTAT_HIGHPROC = 13;  /* # of procs in rpcbind V4 plus one */
const RPCBVERS_STAT = 3;  /* provide only for rpcbind V2, V3 and V4 */
const RPCBVERS_4_STAT = 2;
const RPCBVERS_3_STAT = 1;
const RPCBVERS_2_STAT = 0;

/* Link list of all the stats about getport and getaddr */
struct rpcbs_addrlist {
    unsigned long prog;
    unsigned long vers;
    int success;
    int failure;
    string netid<>;
    struct rpcbs_addrlist *next;
};

/* Link list of all the stats about rmtcall */
struct rpcbs_rmtcalllist {
    unsigned long prog;
    unsigned long vers;
    unsigned long proc;
    int success;
    int failure;
    int indirect;   /* whether callit or indirect */
    string netid<>;
    struct rpcbs_rmtcalllist *next;
};

typedef int rpcbs_proc[RPCBSTAT_HIGHPROC];
typedef rpcbs_addrlist *rpcbs_addrlist_ptr;
typedef rpcbs_rmtcalllist *rpcbs_rmtcalllist_ptr;

struct rpcb_stat {
    rpcbs_proc              info;
    int                     setinfo;
    int                     unsetinfo;
    rpcbs_addrlist_ptr      addrinfo;
    rpcbs_rmtcalllist_ptr   rmtinfo;
};

/*
 * One rpcb_stat structure is returned for each version of rpcbind
 * being monitored.
 */
typedef rpcb_stat rpcb_stat_byvers[RPCBVERS_STAT];
```

```
/* rpcbind procedures */
program RPCBPROG {
    version RPCBVERS {
        void
        RPCBPROC_NULL(void) = 0;
        /*
         * Registers the tuple [r_prog, r_vers, r_addr, r_owner,
         * r_netid]. The rpcbind server accepts requests for this
         * procedure on only the loopback transport for security
         * reasons. Returns TRUE if successful, FALSE on failure.
         */
        bool
        RPCBPROC_SET(rpcb) = 1;

        /*
         * Unregisters the tuple [r_prog, r_vers, r_owner, r_netid].
         * If vers is zero, all versions are unregistered. The
rpcbind
         * server accepts requests for this procedure on only the
         * loopback transport for security reasons.  Returns TRUE if
         * successful, FALSE on failure.
         */
        bool
        RPCBPROC_UNSET(rpcb) = 2;

        /*
         * Returns the universal address where the triple [r_prog,
         * r_vers, r_netid] is registered.  If r_addr specified,
         * return a universal address merged on r_addr. Ignores
         * r_owner. Returns FALSE on failure.
         */
        string
        RPCBPROC_GETADDR(rpcb) = 3;

        /* Returns a list of all mappings. */
        rpcblist
        RPCBPROC_DUMP(void) = 4;

        /*
         * Calls the procedure on the remote machine.  If it is not
         * registered, this procedure IS quiet; that is, it DOES NOT
         * return error information.
         */
        rpcb_rmtcallres
        RPCBPROC_CALLIT(rpcb_rmtcallargs) = 5;
```

```
/*
 * Returns the time on the rpcbind server's system.
 */
unsigned int
RPCBPROC_GETTIME(void) = 6;

struct netbuf
RPCBPROC_UADDR2TADDR(string) = 7;

string
RPCBPROC_TADDR2UADDR(struct netbuf) = 8;
} = 3;
version RPCBVERS4 {
bool
RPCBPROC_SET(rpcb) = 1;

bool
RPCBPROC_UNSET(rpcb) = 2;

string
RPCBPROC_GETADDR(rpcb) = 3;

rpcblist_ptr
RPCBPROC_DUMP(void) = 4;

/*
 * NOTE: RPCBPROC_BCAST has the same functionality as CALLIT;
 * the new name is intended to indicate that this procedure
 * should be used for broadcast RPC, and RPCBPROC_INDIRECT
 * should be used for indirect calls.
 */
rpcb_rmtcallres
RPCBPROC_BCAST(rpcb_rmtcallargs) = RPCBPROC_CALLIT;

unsigned int
RPCBPROC_GETTIME(void) = 6;

struct netbuf
RPCBPROC_UADDR2TADDR(string) = 7;

string
RPCBPROC_TADDR2UADDR(struct netbuf) = 8;

/*
 * Same as RPCBPROC_GETADDR except that if the given version
 * number is not available, the address is not returned.
```

```
            */
            string
            RPCBPROC_GETVERSADDR(rpcb) = 9;

            /*
             * Calls the procedure on the remote machine.  If it is not
             * registered, this procedure IS NOT quiet; that is, it DOES
             * return error information.
             */
            rpcb_rmtcallres
            RPCBPROC_INDIRECT(rpcb_rmtcallargs) = 10;

            /*
             * Same as RPCBPROC_GETADDR except that it returns a list of
             * addresses registered for the combination (prog, vers).
             */
            rpcb_entry_list_ptr
            RPCBPROC_GETADDRLIST(rpcb) = 11;

            /*
             * Returns statistics about the rpcbind server's activity.
             */
            rpcb_stat_byvers
            RPCBPROC_GETSTAT(void) = 12;
        } = 4;
} = 100000;
```

rpcbind Operation

`rpcbind` is contacted by way of an assigned address specific to the transport being used. For TCP/IP and UDP/IP, for example, it is port number 111. Each transport has such an assigned well known address. The following is a description of each of the procedures supported by `rpcbind`.

`RPCBPROC_NULL`

This procedure does no work. By convention, procedure zero of any program takes no parameters and returns no results.

RPCBPROC_SET

When a program first becomes available on a machine, it registers itself with the `rpcbind` program running on the same machine. The program passes its program number *prog*; version number *vers*; network identifier *netid*; and the universal address *uaddr*; on which it awaits service requests.

The procedure returns a Boolean response with the value `TRUE` if the procedure successfully established the mapping and `FALSE` otherwise. The procedure refuses to establish a mapping if one already exists for the ordered set *(prog, vers, netid)*.

Note that neither *netid* nor *uaddr* can be `NULL`, and that *netid* should be a valid network identifier on the machine making the call.

RPCBPROC_UNSET

When a program becomes unavailable, it should unregister itself with the `rpcbind` program on the same machine.

The parameters and results have meanings identical to those of `RPCBPROC_SET`. The mapping of the *(prog, vers, netid)* tuple with *uaddr* is deleted.

If *netid* is `NULL`, all mappings specified by the ordered set *(prog, vers, *)* and the corresponding universal addresses are deleted. Only the owner of the service or the super-user is allowed to unset a service.

RPCBPROC_GETADDR

Given a program number *prog*, version number *vers*, and network identifier *netid*, this procedure returns the universal address on which the program is awaiting call requests.

The *netid* field of the argument is ignored and the *netid* is inferred from the *netid* of the transport on which the request came in.

RPCBPROC_DUMP

This procedure lists all entries in `rpcbind`'s database.

The procedure takes no parameters and returns a list of program, version, netid, and universal addresses. Call this procedure using a stream rather than a datagram transport to avoid the return of a large amount of data.

RPCBPROC_CALLIT

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's universal address. It is intended for supporting broadcasts to arbitrary remote programs via rpcbind's universal address.

The parameters *prog*, *vers*, *proc*, and the *args_ptr* are the program number, version number, procedure number, and parameters of the remote procedure.

Note – This procedure sends a response only if the procedure was successfully executed, and is silent (no response) otherwise.

The procedure returns the remote program's universal address, and the results of the remote procedure.

RPCBPROC_GETTIME

This procedure returns the local time on its own machine in seconds since the midnight on January 1, 1970.

RPCBPROC_UADDR2TADDR

This procedure converts universal addresses to transport (netbuf) addresses. RPCBPROC_UADDR2TADDR is equivalent to uaddr2taddr(). See the netdir(3N) manpage. Only processes that cannot link to the name-to-address library modules should use RPCBPROC_UADDR2TADDR.

RPCBPROC_TADDR2UADDR

This procedure converts transport (netbuf) addresses to universal addresses. RPCBPROC_TADDR2UADDR is equivalent to taddr2uaddr(). See the netdir(3N) manpage. Only processes that can not link to the name-to-address library modules should use RPCBPROC_TADDR2UADDR.

Version 4 rpcbind

Version 4 of the rpcbind protocol includes all of the above procedures, and adds several others.

RPCBPROC_BCAST

This procedure is identical to the version 3 RPCBPROC_CALLIT procedure. The new name indicates that the procedure should be used for broadcast RPCs only. RPCBPROC_INDIRECT, defined in the following text, should be used for indirect RPC calls.

RPCBPROC_GETVERSADDR

This procedure is similar to RPCBPROC_GETADDR. The difference is the r_vers field of the rpcb structure can be used to specify the version of interest. If that version is not registered, no address is returned.

RPCBPROC_INDIRECT

This procedure is similar to RPCBPROC_CALLIT. Instead of being silent about errors (such as the program not being registered on the system), this procedure returns an indication of the error. This procedure should not be used for broadcast RPC. It is intended to be used with indirect RPC calls only.

RPCBPROC_GETADDRLIST

This procedure returns a list of addresses for the given rpcb entry. The client may be able use the results to determine alternate transports that it can use to communicate with the server.

RPCBPROC_GETSTAT

This procedure returns statistics on the activity of the rpcbind server. The information lists the number and kind of requests the server has received.

Note – All procedures except RPCBPROC_SET and RPCBPROC_UNSET can be called by clients running on a machine other than a machine on which rpcbind is running. rpcbind accepts only RPCPROC_SET and RPCPROC_UNSET requests on the loopback transport.

Bibliography

For further information on the technologies and architectures discussed in this appendix, reference the following resources.

1. Birrel, Andrew D. & Nelson, Bruce Jay; "Implementing Remote Procedure Calls," XEROX CSL-83-7, October 1983.
2. Cheriton, D.; "VMTP: Versatile Message Transaction Protocol," Preliminary Version 0.3; Stanford University, January 1987.
3. Diffie and Hellman; "New Directions in Cryptography," *IEEE Transactions on Information Theory* IT-22, November 1976.
4. Harrenstien, K.; "Time Server," *RFC* 738; Information Sciences Institute, October 1977.
5. National Bureau of Standards; "Data Encryption Standard," *Federal Information Processing Standards Publication* 46, January 1977.
6. Postel, J.; "Transmission Control Protocol - DARPA Internet Program Protocol Specification," *RFC* 793; Information Sciences Institute, September 1981.
7. Postel, J.; "User Datagram Protocol," *RFC* 768; Information Sciences Institute, August 1980.

B