Remote Procedure Calls: Protocol Specification

1. Status of this Memo

Note: This chapter specifies a protocol that Sun Microsystems, Inc., and others are using. It has been designated RFC1050 by the ARPA Network Information Center.

2. Introduction

This chapter specifies a message protocol used in implementing Sun's Remote Procedure Call (RPC) package. (The message protocol is specified with the External Data Representation (XDR) language. See the *External Data Representation Standard: Protocol Specification* for the details. Here, we assume that the reader is familiar with XDR and do not attempt to justify it or its uses). The paper by Birrell and Nelson [1] is recommended as an excellent background to and justification of RPC.

2.1. Terminology

This chapter discusses servers, services, programs, procedures, clients, and versions. A server is a piece of software where network services are implemented. A network service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters, and results are documented in the specific program's protocol specification (see the *Port Mapper Program Protocol* below, for an example). Network clients are pieces of software that initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

For example, a network file service may be 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 IO 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.

2.2. The RPC Model

The remote procedure call 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 (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The remote procedure call 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. That is, 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 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 model, only one of the two processes is active at any given time. However, this model is only given as an example. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. 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.

2.3. Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP[6], then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP[7], it must implement is own retransmission and time-out policy as the RPC layer 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 such as UDP/IP. 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, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them in order 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 is by the client RPC layer in matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. 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/IP, 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.

There are other possibilities for transports besides datagram- or connection-oriented protocols. For example, a request-reply protocol such as VMTP[2] is perhaps the most natural transport for RPC.

NOTE: At Sun, RPC is currently implemented on top of both TCP/IP and UDP/IP transports.

2.4. 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 the *Port Mapper Program Protocol* below).

Implementors 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, using RPC to accomplish this task.

2.5. Authentication

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 indicates which protocol is being used. More information on specific authentication protocols can be found in the *Authentication Protocols* below.

3. RPC Protocol Requirements

The RPC protocol must provide for the following:

1. Unique specification of a procedure to be called.
2. Provisions for matching response messages to request messages.
3. Provisions for authenticating the caller to service and vice-versa.

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

1. RPC protocol mismatches.
2. Remote program protocol version mismatches.
3. Protocol errors (such as misspecification of a procedure's parameters).
4. Reasons why remote authentication failed.
5. Any other reasons why the desired procedure was not called.

3.1. Programs and Procedures

The RPC call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority (like Sun). Once an implementor has a program number, he can implement his remote program; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies which 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 specific 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 actual RPC message protocol could also change. Therefore, the call message also has in it the RPC version number, which is always equal to two for the version of RPC described here.

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

1. The remote implementation of RPC does speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
2. The remote program is not available on the remote system.
3. The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
4. The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)
5. 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.)

3.2. Authentication

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_NULL       = 0,
    AUTH_UNIX       = 1,
    AUTH_SHORT      = 2,
    AUTH_DES        = 3
    /* and more to be defined */
};

struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

In simple English, any *opaque_auth* structure is an *auth_flavor* enumeration followed by bytes which are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. (See *Authentication Protocols* below, for definitions of the various authentication protocols.)

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

3.3. Program Number Assignment

Program numbers are given out in groups of *0x20000000* (decimal 536870912) according to the following chart:

Program Numbers	*Description*
0 - 1fffffff	*Defined by Sun*
20000000 - 3fffffff	*Defined by user*
40000000 - 5fffffff	*Transient*
60000000 - 7fffffff	*Reserved*
80000000 - 9fffffff	*Reserved*
a0000000 - bfffffff	*Reserved*
c0000000 - dfffffff	*Reserved*
e0000000 - ffffffff	*Reserved*

The first group is a range of numbers administered by Sun Microsystems and should be identical for all sites. The second range is for applications peculiar to a particular site. This range is intended primarily for debugging new programs. When a site develops an application that might be of general interest, that application should be given an assigned number in the first range. The third group is for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

3.4. 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. Sun currently uses, or perhaps abuses, the RPC message protocol for the following two (non-RPC) protocols: batching (or pipelining) and broadcast RPC. These two protocols are discussed but not defined below.

3.4.1. 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/IP) for its transport. In the case of 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 terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgement).

3.4.2. Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (like UDP/IP) 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 Port Mapper RPC service to achieve its semantics. See the *Port Mapper Program Protocol* below, for more information.

4. 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.

```
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 hasn't exported program  */
    PROG_MISMATCH = 2, /* remote can't support version #  */
    PROC_UNAVAIL  = 3, /* program can't support procedure */
    GARBAGE_ARGS  = 4  /* procedure can't decode params   */
};

/*
* Reasons why 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,  /* client must begin new session */
    AUTH_BADVERF      = 3,  /* bad verifier */
    AUTH_REJECTEDVERF = 4,  /* verifier expired or replayed  */
    AUTH_TOOWEAK      = 5   /* rejected for security reasons */
};
```

```
/*
* 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;
};
```

5. Authentication Protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication implemented at (and supported by) Sun. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

5.1. Null Authentication

Often calls must be made where the caller does not know who he is or the server does not care who the caller is. In this case, the flavor value (the discriminant of the *opaque_auth*'s union) of the RPC message's credentials, verifier, and response verifier is *AUTH_NULL.* The bytes of the opaque_auth's body are undefined. It is recommended that the opaque length be zero.

5.2. UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX system. The value of the credential's discriminant of an RPC call message is *AUTH_UNIX*. The bytes of the credential's opaque body encode the following structure:

```
struct auth_unix {
     unsigned int stamp;
     string machinename<255>;
     unsigned int uid;
     unsigned int gid;
     unsigned int gids<10>;
};
```

The *stamp* is an arbitrary ID which the caller machine may generate. The *machinename* is the name of the caller's machine (like "krypton"). The *uid* is the caller's effective user ID. The *gid* is the caller's effective group ID. The *gids* is a counted array of groups which contain the caller as a member. The verifier accompanying the credentials should be of *AUTH_NULL* (defined above).

The value of the discriminant of the response verifier received in the reply message from the server may be *AUTH_NULL* or *AUTH_SHORT*. In the case of *AUTH_SHORT*, the bytes of the response verifier's string encode an opaque structure. This new opaque structure may now be passed to the server instead of the original *AUTH_UNIX* flavor credentials. The server keeps a cache which 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 due 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_UNIX* style of credentials.

5.3. DES Authentication

UNIX authentication suffers from two major problems:

1. The naming is too UNIX-system oriented.
2. There is no verifier, so credentials can easily be faked.

DES authentication attempts to fix these two problems.

5.3.1. Naming

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 is not allowed to interpret the contents of the caller's name in any other way except to identify the caller. Thus, netnames should be unique for every caller in the internet.

It is up to each operating system's implementation of DES authentication to generate netnames for its users that insure this uniqueness when they call upon remote servers. Operating systems already know how to distinguish users local to their systems. It is usually a simple matter to extend this mechanism to the network. For example, a UNIX user at Sun with a user ID of 515 might be assigned the following netname: "unix.515@sun.com". This netname contains three items that serve to insure it is unique. Going backwards, 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 that, by coincidence, happens to have the same user ID. To insure that these two users can be distinguished we 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 today 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 the name of that standard, instead of an operating system name.

5.3.2. DES Authentication Verifiers

Unlike UNIX authentication, DES authentication does have a verifier so the server can validate the client's credential (and vice-versa). The contents of this verifier is primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to what the real time is, 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. And if the client knows the conversation key, then it must be the real client.

The conversation key is a DES [5] key which 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 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 in order for all of this to work. If network time synchronization cannot be guaranteed, then client can synchronize with the server before beginning the conversation, perhaps by consulting the Internet Time Server (TIME[4]).

The way a server determines if a client timestamp is valid is somewhat complicated. For any other transaction but the first, the server just checks for two things:

1. the timestamp is greater than the one previously seen from the same client.
2. 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. You can think of it as a lifetime for the credential.

This explains everything but the first transaction. In the first transaction, the server checks only that the timestamp has not expired. If this was all that was done though, then it would be quite easy for the client to send random data in place of the timestamp with a fairly good chance of succeeding. 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 too 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 different than this, it will reject it.

5.3.3. Nicknames and Clock Synchronization

After the first transaction, the server's DES authentication subsystem returns in its verifier to the client an integer "nickname" which 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 which stores for each client its netname, decrypted DES key and window.

Though they originally were synchronized, the client's and server's clocks can get out of sync again. When this happens the client RPC subsystem most likely will get back *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. A client should resend its original credential in this case and the server will give it a new nickname. If a server crashes, the entire nickname table gets flushed, and all clients will have to resend their original credentials.

5.3.4. DES Authentication Protocol (in XDR language)

```
/*
* 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 fullname 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 past, then the server
* should expire the request and not grant it. To insure that
* requests are not replayed, the server should insist that
* timestamps are 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 */
};

/*
* A credential is either a fullname or a nickname
*/
union authdes_cred switch (authdes_namekind adc_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
* The window verifier is only used in the first transaction.  In
* conjunction with a fullname credential, these items are packed
* into the following structure before being encrypted:
*
* struct {
*   adv_timestamp;              -- one DES block
*   adc_fullname.window;        -- one half DES block
*   adv_winverf;                -- one half DES block
* }
* This structure is encrypted using CBC mode encryption with an
* input vector of zero.  All other encryptions of timestamps use
* ECB mode encryption.
*/
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 client */
};
```

5.3.5. Diffie-Hellman Encryption

In this scheme, there are two constants, *BASE* and *MODULUS*. The particular values Sun has chosen for these for the DES authentication protocol are:

```
const BASE = 3;
const MODULUS =
        "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b"; /* hex */
```

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. So, A and B both generate "secret" keys at random which they do not reveal 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) = ( BASE ** SK(A) ) mod MODULUS
PK(B) = ( BASE ** SK(B) ) mod MODULUS
```

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 revealing their secret keys.

A computes:

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

while B computes:

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

These two can be shown to be equivalent:

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

We drop the "mod MODULUS" parts and assume modulo arithmetic to simplify things:

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

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

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

which leads to:

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

This common key CK(A, B) is not used to encrypt the timestamps used in the protocol. Rather, it is used only to encrypt a conversation key which 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, since conversations are relatively 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.

6. Record Marking Standard

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). Sun uses this RM/TCP/IP transport for passing RPC messages on TCP streams. 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 from highest to lowest. The number encodes two values—a boolean which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which 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. (Note that this record specification is NOT in XDR standard form!)

7. The RPC Language

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. We use the RPC Language for this purpose. It is an extension to the XDR language. The following example is used to describe the essence of the language.

7.1. An Example Service Described in the RPC Language

Here is an example of the specification of a simple ping program.

```
/*
* Simple ping program
*/
program PING_PROG {
     /* Latest and greatest version */
     version PING_VERS_PINGBACK {
     void
     PINGPROC_NULL(void) = 0;

     /*
     * Ping the caller, return the round-trip time
     * (in microseconds). Returns -1 if the operation
     * timed out.
     */
     int
     PINGPROC_PINGBACK(void) = 1;
} = 2;

/*
* Original version
*/
version PING_VERS_ORIG {
     void
     PINGPROC_NULL(void) = 0;
     } = 1;
} = 1;

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 computing round-trip times from the client to the server and back again. By convention, procedure 0 of any RPC protocol should have the same semantics, and never require any kind of authentication. The second procedure is used for the client to have the server do a reverse ping operation back to the client, and it 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.

7.2. The RPC Language Specification

The RPC language is identical to the XDR language, except for the added definition of a *program-def* described below.

```
program-def:
    "program" identifier "{"
        version-def
        version-def *
    "}" "=" constant ";"

version-def:
    "version" identifier "{"
        procedure-def
        procedure-def *
    "}" "=" constant ";"

procedure-def:
    type-specifier identifier "(" type-specifier ")"
    "=" constant ";"
```

7.3. Syntax Notes

1. The following keywords are added and cannot be used as identifiers: "program" and "version";
2. A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.
3. A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of version definition.
4. Program identifiers are in the same name space as constant and type identifiers.
5. Only unsigned constants can be assigned to programs, versions and procedures.

8. Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

This is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

The port mapper also aids in broadcast RPC. A given RPC program will usually have different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that picks up the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply on back to the client.

8.1. Port Mapper Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;        /* portmapper port number */

/*
* A mapping of (program, version, protocol) to port number
*/
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
};

/*
* Supported values for the "prot" field
*/
const IPPROTO_TCP = 6;        /* protocol number for TCP/IP */
const IPPROTO_UDP = 17;       /* protocol number for UDP/IP */

/*
* A list of mappings
*/
struct *pmaplist {
    mapping map;
    pmaplist next;
};

/*
* Arguments to callit
*/
struct call_args {
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque args<>;
};

/*
* Results of callit
*/
struct call_result {
    unsigned int port;
    opaque res<>;
};
```

```
/*
* Port mapper procedures
*/
program PMAP_PROG {
     version PMAP_VERS {
          void
          PMAPPROC_NULL(void)          = 0;

          bool
          PMAPPROC_SET(mapping)        = 1;

          bool
          PMAPPROC_UNSET(mapping)      = 2;

          unsigned int
          PMAPPROC_GETPORT(mapping)    = 3;

          pmaplist
          PMAPPROC_DUMP(void)          = 4;

          call_result
          PMAPPROC_CALLIT(call_args)   = 5;
     } = 2;
} = 100000;
```

8.2. Port Mapper Operation

The portmapper program currently supports two protocols (UDP/IP and TCP/IP). The portmapper is contacted by talking to it on assigned port number 111 (SUNRPC [8]) on either of these protocols. The following is a description of each of the portmapper procedures:

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

PMAPPROC_SET:
When a program first becomes available on a machine, it registers itself with the port mapper program on the same machine. The program passes its program number "prog", version number "vers", transport protocol number "prot", and the port "port" on which it awaits service request. The procedure returns a boolean response whose value is *TRUE* if the procedure successfully established the mapping and *FALSE* otherwise. The procedure refuses to establish a mapping if one already exists for the tuple "(prog, vers, prot)".

PMAPPROC_UNSET:
When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine. The parameters and results have meanings identical to those of *PMAPPROC_SET*. The protocol and port number fields of the argument are ignored.

PMAPPROC_GETPORT:
Given a program number "prog", version number "vers", and transport protocol number "prot", this procedure returns the port number on which the program is awaiting call requests. A port value of zeros means the program has not been registered. The "port" field of the argument is ignored.

PMAPPROC_DUMP:
This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

PMAPPROC_CALLIT:
This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It is intended for supporting broadcasts to arbitrary remote programs via the well-known port mapper's port. The parameters "prog", "vers", "proc", and the

bytes of "args" are the program number, version number, procedure number, and parameters of the remote procedure.

Note:

1. This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.
2. The port mapper communicates with the remote program using UDP/IP only.

The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

9. References

[1] Birrell, 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 & 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.

[8] Reynolds, J. & Postel, J.; "Assigned Numbers", RFC 923; Information Sciences Institute, October 1984.