Remote Procedure Call
Protocol Specification

Remote Procedure Call
Protocol Specification

1. Introduction

This document 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.

This document assumes that the reader is familiar with both RPC and XDR. It does not attempt to justify RPC or its uses. Also, the casual user of RPC does not need to be familiar with the information in this document.

1.1. Terminology

The document discusses servers, services, programs, procedures, clients and versions. A server is a machine where some number of network services are implemented. A 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 Appendix C 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 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.

1.2. 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, except that one thread of control 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. That is, the RPC protocol does not explicitly support multi-threading of caller or server processes.

1.3. 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 only deals with the specification and interpretation of messages.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Some semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, RPC message passing using UDP/IP is unreliable. Thus, if the caller retransmits call messages after short time-outs, the only thing he can infer from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). On the other hand, RPC message passing using TCP/IP is reliable. No reply message means that the remote procedure was executed at most once, whereas a reply message means that the remote procedure was exactly once. (Note: At Sun, RPC is currently implemented on top of TCP/IP and UDP/IP transports.)

1.4. 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 Appendix C.)

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.

1.5. Message Authentication

The RPC protocol provides the fields necessary for a client to identify himself to a service and vice versa. Security and access control mechanisms can be built on top of the message authentication.

2. 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 (like mis-specification of a procedure's parameters).
4. Reasons why remote authentication failed.
5. Any other reasons why the desired procedure was not called.

2.1. Remote 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 the RPC version number in it; this field must be two (2).

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 caused by a disagreement about the protocol between client and service.)

2.2. Authentication

Provisions for authentication of caller to service and vice versa are provided as a wart on the side 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
        /* and more to be defined */
};

struct opaque_auth {
        union switch (enum auth_flavor) {
                default: string auth_body<400>;
        };
};
```

In simple English, any `opaque_auth` structure is an `auth_flavor` enumeration followed by a counted string, whose bytes 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. Appendix A defines three authentication protocols.

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

2.3. Program Number Assignment

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

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 Sun customers. The second range is for applications peculiar to a particular customer. This range is intended primarily for debugging new programs. When a customer 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 reservered for future use, and should not be used. The exact registration process for Sun defined numbers is yet to be established.

3. Other Uses and Abuses 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 (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.

3.1. Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching uses reliable bytes stream protocols (like TCP/IP) for their 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).

3.2. Broadcast RPC

In broadcast RPC based protocols, the client sends an a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet based protocols (like UDP/IP) as their transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors.

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. Note: This is an XDR specification, not C code.

```
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,       /* remote procedure was successfully executed */
        PROG_UNAVAIL = 1,  /* remote machine exports the program number */
        PROG_MISMATCH = 2, /* remote machine can't support version number */
        PROC_UNAVAIL = 3,  /* remote program doesn't know about procedure */
        GARBAGE_ARGS = 4   /* remote procedure can't figure out parameters */
};

/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
        RPC_MISMATCH = 0,  /* RPC version number was not two (2) */
        AUTH_ERROR = 1     /* caller not authenticated on remote machine */
};

/*
 * Why authentication failed:
 */
enum auth_stat {
        AUTH_BADCRED = 1,       /* bogus credentials (seal broken) */
        AUTH_REJECTEDCRED = 2,  /* client should begin new session */
        AUTH_BADVERF = 3,       /* bogus verifier (seal broken) */
        AUTH_REJECTEDVERF = 4,  /* verifier expired or was replayed */
        AUTH_TOOWEAK = 5,       /* rejected due to 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; the service side cannot treat this id as any type of
 * sequence number.
 */
struct rpc_msg {
        unsigned        xid;
        union switch (enum msg_type) {
                CALL:   struct call_body;
                REPLY:  struct reply_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,
 * and the procedure within the remote program to be called.  These fields are
 * followed by two authentication parameters, cred (authentication credentials)
 * and verf (authentication verifier).  The authentication parameters are
 * followed * by the parameters to the remote procedure; these parameters are
 * specified by the specific program protocol.
 */
struct call_body {
        unsigned rpcvers;       /* must be equal to two (2) */
        unsigned prog;
        unsigned vers;
        unsigned proc;
        struct opaque_auth cred;
        struct opaque_auth verf;
        /* procedure specific parameters start here */
};
/*
 * Body of a reply to an RPC request.
 * The call message was either accepted or rejected.
 */
struct reply_body {
        union switch (enum reply_stat) {
                MSG_ACCEPTED:   struct accepted_reply;
                MSG_DENIED:     struct rejected_reply;
        };
};
```

```
/*
 * Reply to an RPC request that was accepted by the server.
 * Note: there could be an error even though the request was accepted.
 * The first field is an authentication verifier which 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_ARGS arms
 * of the union are void.  The PROG_MISMATCH arm specifies the lowest and
 * highest version numbers of the remote program that are supported by the
 * server.
 */
struct accepted_reply {
        struct opaque_auth      verf;
        union switch (enum accept_stat) {
                SUCCESS: struct {
                        /*
                         * procedure-specific results start here
                         */
                };
                PROG_MISMATCH: struct {
                        unsigned low;
                        unsigned  high;
                };
                default: struct {
                        /*
                         * void. Cases include PROG_UNAVAIL,
                         * PROC_UNAVAIL, and GARBAGE_ARGS.
                         */
                };
        };
};
/*
 * Reply to an RPC request that was rejected by the server.
 * The request can be rejected because of two reasons - either the server is
 * not running a compatible version of the RPC protocol (RPC_MISMATCH), or
 * the server refused to authenticate the caller (AUTH_ERROR).  In the case of
 * an RPC version mismatch, the server returns the lowest and highest supported
 * RPC version numbers. In the case of refused authentication, the failure
 * status is returned.
 */
struct rejected_reply {
        union switch (enum reject_stat) {
                RPC_MISMATCH: struct {
                        unsigned low;
                        unsigned high;
                };
                AUTH_ERROR: enum auth_stat;
        };
};
```

Appendix A: Authentication Parameter Specification

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some ‘‘flavors’’ of authentication which have been implemented at (and supported by) Sun.

A.1. Null Authentication

Often calls must be made where the caller does not know who he is and the server does not care who the caller is. In this case, the auth_flavor value (the discriminant of the opaque_auth’s union) of the RPC message’s credentials, verifier, and response verifier is AUTH_NULL (0). The bytes of the auth_body string are undefined. It is recommended that the string length be zero.

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 (1). The bytes of the *credential’s* string encode the the following (XDR) structure:

```
struct auth_unix {
        unsigned        stamp;
        string          machinename<255>;
        unsigned        uid;
        unsigned        gid;
        unsigned        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 callers 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 discriminate of the *response verifier* received in the reply message from the server may be AUTH_NULL or AUTH_SHORT (2). In the case of AUTH_SHORT, the bytes of the *response verifier’*s string encode an `auth_opaque` structure. This new `auth_opaque` structure may now be passed to the server instead of the original AUTH_UNIX flavor credentials. The server keeps a cache which maps short hand *auth_opaque* structures (passed back via a 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 short hand *auth_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.

Appendix B: 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!)

Appendix C: Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to UDP/IP or TCP/IP 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.

C.1. The Port Mapper RPC Protocol

The protocol is specified by the XDR description language.

```
Port Mapper RPC Program Number: 100000
        Version Number: 1
        Supported Transports:
                UDP/IP on port 111
                RM/TCP/IP on port 111

/*
 * Handy transport protocol numbers
 */
#define IPPROTO_TCP     6       /* protocol number used for rpc/rm/tcp/ip */
#define IPPROTO_UDP     17      /* protocol number used for rpc/udp/ip */

/* Procedures */

/*
 * Convention: procedure zero of any protocol takes no parameters
 * and returns no results.
 */
0. PMAPPROC_NULL () returns ()

/*
 * Procedure 1, setting a mapping:
 * 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 success whose value is TRUE if the procedure
 * successfully established the mapping and FALSE otherwise.  The
 * procedure will refuse to establish a mapping if one already exists
 * for the tuple [prog, vers, prot].
 */
1. PMAPPROC_SET (prog, vers, prot, port) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned port;
        boolean success;
```

```
/*
 * Procedure 2, Unsetting a mapping:
 * 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.
 */
2. PMAPPROC_UNSET (prog, vers, dummy1, dummy2) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned dummy1;  /* this value is always ignored */
        unsigned dummy2;  /* this value is always ignored */
        boolean success;

/*
 * Procedure 3, looking-up a mapping:
 * 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 that the program has not been registered.
 */
3. PMAPPROC_GETPORT (prog, vers, prot, dummy) returns (port)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned dummy;  /* this value is always ignored */
        unsigned port;  /* zero means the program is not registered */

/*
 * Procedure 4, dumping the mappings:
 * This procedure enumerates all entries in the port mapper's database.
 * The procedure takes no parameters and returns a ''list'' of
 * [program, version, prot, port] values.
 */
4. PMAPPROC_DUMP () returns (maplist)
        struct maplist {
                union switch (boolean) {
                        FALSE: struct { /* void, end of list */ };
                        TRUE: struct {
                                unsigned prog;
                                unsigned vers;
                                unsigned prot;
                                unsigned port;
                                struct maplist the_rest;
                        };
                };
        } maplist;
```

```
/*
 * Procedure 5, indirect call routine:
 * The procedures allows a caller to call another remote procedure
 * on the same machine without knowing the remote procedure's port
 * number.  Its intended use is 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 the the remote
 * procedure.
 *
 * NB:
 * 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 via
 * UDP/IP only.
 *
 * The procedure returns the port number of the remote program and
 * the bytes of results are the results of the remote procedure.
 */
5. PMAPPROC_CALLIT (prog, vers, proc, args) returns (port, results)
        unsigned prog;
        unsigned vers;
        unsigned proc;
        string args<>;
        unsigned port;
        string results<>;
```