Network Working Group
Request for Comments: 1831
Category: Standards Track

R. Srinivasan
Sun Microsystems

August 1995

RPC: Remote Procedure Call Protocol Specification Version 2

Status of this Memo

ABSTRACT

This document describes the ONC Remote Procedure Call (ONC RPC Version 2) protocol as it is currently deployed and accepted. "ONC" stands for "Open Network Computing".

TABLE OF CONTENTS

1. INTRODUCTION

This document specifies version two of the message protocol used in ONC Remote Procedure Call (RPC). The message protocol is specified with the eXternal Data Representation (XDR) language [9]. This document assumes that the reader is familiar with XDR. It does not attempt to justify remote procedure calls systems or describe their use. The paper by Birrell and Nelson [1] is recommended as an excellent background for the remote procedure call concept.

2. TERMINOLOGY

This document discusses clients, calls, servers, replies, services, programs, procedures, and versions. Each remote procedure call has two sides: an active client side that makes the call to a server, which sends back a reply. 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. A server may support more than one version of a remote program in order to be 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 input and output and have procedures like "read" and "write". A client of the network file service would call the procedures associated with the two programs of the service on behalf of the client.

The terms client and server only apply to a particular transaction; a particular hardware entity (host) or software entity (process or program) could operate in both roles at different times. For example, a program that supplies remote execution service could also be a client of a network file service.

3. THE RPC MODEL

The ONC RPC protocol is based on the remote procedure call model, which 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 register window). It then transfers control to the procedure, and eventually regains 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 model is similar. One thread of control logically winds through two processes: the caller's process, and a server's process. The caller process first sends a call message to the server process and waits (blocks) for a reply message. The call message includes the procedure's parameters, and the reply message includes the procedure's results. 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.

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 ONC 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 separate task to process an incoming call, so that the original server can be free to receive other requests.

There are a few important ways in which remote procedure calls differ from local procedure calls:

1. Error handling: failures of the remote server or network must be handled when using remote procedure calls.

2. Global variables and side-effects: since the server does not have access to the client's address space, hidden arguments cannot be passed as global variables or returned as side effects.

3. Performance: remote procedures usually operate one or more orders of magnitude slower than local procedure calls.

4. Authentication: since remote procedure calls can be transported over unsecured networks, authentication may be necessary. Authentication prevents one entity from masquerading as some other entity.

The conclusion is that even though there are tools to automatically generate client and server libraries for a given service, protocols must still be designed carefully.

4. TRANSPORTS AND SEMANTICS

The RPC protocol can be implemented on several different transport protocols. The RPC protocol does not care how a message is passed from one process to another, but only with specification and interpretation of messages. However, the application may wish to obtain information about (and perhaps control over) the transport layer through an interface not specified in this document. For example, the transport protocol may impose a restriction on the maximum size of RPC messages, or it may be stream-oriented like TCP with no size limit. The client and server must agree on their transport protocol choices.

It is important to point out that RPC does not try to implement any kind of reliability and that the application may need to 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 [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 [7], it must implement its own time-out, retransmission, and duplicate detection policies as the RPC protocol does not provide these services.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution requirements. 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. If an application retransmits RPC call messages after time-outs, and does not receive a reply, it cannot infer anything about the number of times the procedure was executed. 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 message. The main use of this transaction ID is by the client RPC entity in matching replies to calls. However, a client application may choose to reuse its previous transaction ID when retransmitting a call. The server may choose to remember this ID after executing a call and not execute calls 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, 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 that 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 a natural transport for RPC. ONC RPC uses both TCP and UDP transport protocols. Section 10 (RECORD MARKING STANDARD) describes the mechanism employed by ONC RPC to utilize a connection-oriented, stream-oriented transport such as TCP.

5. BINDING AND RENDEZVOUS INDEPENDENCE

The act of binding a particular client to a particular service and transport parameters is NOT part of this RPC protocol specification. This important and necessary function is left up to some higher-level software.

Implementors could 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 binding software makes RPC useful, possibly using RPC to accomplish this task.

6. AUTHENTICATION

The RPC protocol provides the fields necessary for a client to identify itself to a service, and vice-versa, in each call and reply message. Security and access control mechanisms can be built on top of this 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 is in section 9: "Authentication Protocols".

7. 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.

7.1 RPC Programs and Procedures

The RPC call message has three unsigned integer fields -- remote program number, remote program version number, and remote procedure number -- which uniquely identify the procedure to be called. Program numbers are administered by a central authority (rpc@sun.com). Once implementors have a program number, they can implement their remote program; the first implementation would most likely have the version number 1. Because most new protocols evolve, a version field of the call message identifies which version of the protocol the caller is using. Version numbers enable support of both old and new protocols through the same server process.

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 not support 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 client 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.)

7.2 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 credential 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 (in the eXternal Data Representation (XDR) language [9]):

```
enum auth_flavor {
   AUTH_NONE       = 0,
   AUTH_SYS        = 1,
   AUTH_SHORT      = 2
   /* and more to be defined */
};

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

In other words, any "opaque_auth" structure is an "auth_flavor" enumeration followed by up to 400 bytes which are opaque to (uninterpreted by) the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. (Section 9 defines the various authentication protocols.)

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

7.3 Program Number Assignment

Program numbers are given out in groups of hexadecimal 20000000 (decimal 536870912) according to the following chart:

```
          0 - 1fffffff   defined by rpc@sun.com
   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 rpc@sun.com 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. Application developers may apply for blocks of RPC program numbers in the first range by sending electronic mail to "rpc@sun.com". The third group is for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

7.4 Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. Normally, each call message is matched with a reply message. However, the protocol itself is a message-passing protocol with which other (non-procedure call) protocols can be implemented.

7.4.1 Batching

Batching is useful when a client wishes 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 the case of batching, the client never waits for a reply from the server, and the server does not send replies to batch calls. A sequence of batch calls is usually terminated by a legitimate remote procedure call operation in order to flush the pipeline and get positive acknowledgement.

7.4.2 Broadcast Remote Procedure Calls

In broadcast protocols, the client sends a broadcast call to the network and waits for numerous replies. This requires the use of packet-based protocols (like UDP) as its transport protocol. Servers

that support broadcast protocols usually respond only when the call is successfully processed and are silent in the face of errors, but this varies with the application.

The principles of broadcast RPC also apply to multicasting - an RPC request can be sent to a multicast address.

8. THE RPC MESSAGE PROTOCOL

This section defines the RPC message protocol in the XDR data description language [9].

```
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         */
   SYSTEM_ERR    = 5  /* errors like memory allocation failure */
};
```

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_OK           = 0,  /* success                        */
```

```
        /*
         * failed at remote end
         */
        AUTH_BADCRED      = 1,  /* bad credential (seal broken)     */
        AUTH_REJECTEDCRED = 2,  /* client must begin new session    */
        AUTH_BADVERF      = 3,  /* bad verifier (seal broken)       */
        AUTH_REJECTEDVERF = 4,  /* verifier expired or replayed     */
        AUTH_TOOWEAK      = 5,  /* rejected for security reasons    */
        /*
         * failed locally
         */
        AUTH_INVALIDRESP  = 6,  /* bogus response verifier          */
        AUTH_FAILED       = 7   /* reason unknown                   */
    };
```

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 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 credential) 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.

The purpose of the authentication verifier is to validate the authentication credential. Note that these two items are

historically separate, but are always used together as one logical entity.

```
   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 call:

```
   union reply_body switch (reply_stat stat) {
   case MSG_ACCEPTED:
      accepted_reply areply;
   case MSG_DENIED:
      rejected_reply rreply;
   } reply;
```

Reply to an RPC call that was accepted by the server:

There could be an error even though the call was accepted. The first field is an authentication verifier that the server generates in order to validate itself to the client. 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, GARBAGE_ARGS, and SYSTEM_ERR 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,
             * GARBAGE_ARGS, and SYSTEM_ERR.
             */
            void;
        } reply_data;
    };
```

Reply to an RPC call that was rejected by the server:

The call can be rejected for two reasons: either the server is not running a compatible version of the RPC protocol (RPC_MISMATCH), or the server rejects the identity of 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 invalid 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;
};
```

9. AUTHENTICATION PROTOCOLS

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines two standard "flavors" of authentication. Implementors are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment. The "flavor" of a credential or verifier refers to the value of the "flavor" field in the opaque_auth structure. Flavor numbers, like RPC program numbers, are also administered centrally, and developers may assign new flavor numbers by applying through electronic mail to "rpc@sun.com". Credentials and verifiers are represented as variable length opaque data (the "body" field in the opaque_auth structure).

In this document, two flavors of authentication are described. Of these, Null authentication (described in the next subsection) is mandatory - it must be available in all implementations. System authentication is described in Appendix A. It is strongly recommended that implementors include System authentication in their implementations. Many applications use this style of authentication, and availability of this flavor in an implementation will enhance interoperability.

9.1 Null Authentication

Often calls must be made where the client does not care about its identity or the server does not care who the client is. In this case, the flavor of the RPC message's credential, verifier, and reply verifier is "AUTH_NONE". Opaque data associated with "AUTH_NONE" is undefined. It is recommended that the length of the opaque data be zero.

10. RECORD MARKING STANDARD

When RPC messages are passed on top of a byte stream transport protocol (like TCP), it is necessary to delimit one message from another in order to detect and possibly recover from 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 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!)

11. 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. The RPC Language is an extension to the XDR language, with the addition of "program", "procedure", and "version" declarations. The following example is used to describe the essence of the language.

11.1 An Example Service Described in the RPC Language

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

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

```
            /*
             * Ping the client, 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.

11.2 The RPC Language Specification

The RPC language is identical to the XDR language defined in RFC 1014, 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
        "," type-specifier )* ")" "=" constant ";"
```

11.3 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.

APPENDIX A: SYSTEM AUTHENTICATION

The client may wish to identify itself, for example, as it is identified on a UNIX(tm) system. The flavor of the client credential is "AUTH_SYS". The opaque data constituting the credential encodes the following structure:

```
struct authsys_parms {
   unsigned int stamp;
   string machinename<255>;
   unsigned int uid;
   unsigned int gid;
   unsigned int gids<16>;
};
```

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 credential should have "AUTH_NONE" flavor value (defined above). Note this credential is only unique within a particular domain of machine names, uids, and gids.

The flavor value of the verifier received in the reply message from the server may be "AUTH_NONE" or "AUTH_SHORT". In the case of "AUTH_SHORT", the bytes of the reply verifier's string encode an opaque structure. This new opaque structure may now be passed to the server instead of the original "AUTH_SYS" flavor credential. The server may keep a cache which maps shorthand opaque structures (passed back by way of an "AUTH_SHORT" style reply verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the shorthand credential.

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 client may wish to try the original "AUTH_SYS" style of credential.

It should be noted that use of this flavor of authentication does not guarantee any security for the users or providers of a service, in itself. The authentication provided by this scheme can be considered legitimate only when applications using this scheme and the network can be secured externally, and privileged transport addresses are used for the communicating end-points (an example of this is the use of privileged TCP/UDP ports in Unix systems - note that not all systems enforce privileged transport address mechanisms).

REFERENCES

[1] Birrell, A. D. & Nelson, B. J., "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] Mills, D., "Network Time Protocol", RFC 1305, UDEL, March 1992.

[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", STD 7, RFC 793, USC/Information Sciences Institute, September 1981.

[7] Postel, J., "User Datagram Protocol", STD 6, RFC 768, USC/Information Sciences Institute, August 1980.

[8] Reynolds, J., and Postel, J., "Assigned Numbers", STD 2, RFC 1700, USC/Information Sciences Institute, October 1994.

[9] Srinivasan, R., "XDR: External Data Representation Standard", RFC 1832, Sun Microsystems, Inc., August 1995.

[10] Miller, S., Neuman, C., Schiller, J., and J. Saltzer, "Section E.2.1: Kerberos Authentication and Authorization System", M.I.T. Project Athena, Cambridge, Massachusetts, December 21, 1987.

[11] Steiner, J., Neuman, C., and J. Schiller, "Kerberos: An Authentication Service for Open Network Systems", pp. 191-202 in Usenix Conference Proceedings, Dallas, Texas, February 1988.

[12] Kohl, J. and C. Neuman, "The Kerberos Network Authentication Service (V5)", RFC 1510, Digital Equipment Corporation, USC/Information Sciences Institute, September 1993.

Security Considerations

Security issues are not discussed in this memo.

Author's Address

Raj Srinivasan
Sun Microsystems, Inc.
ONC Technologies
2550 Garcia Avenue
M/S MTV-5-40
Mountain View, CA 94043
USA

Phone: 415-336-2478
Fax: 415-336-6015
EMail: raj@eng.sun.com