Sun Network Data Representation and Remote Procedure Call

Bob Lyon

Company Confidential

ABSTRACT

This document proposes two standards for implementing Sun's network services - the Network Data Representation standard (NDR) and the Remote Procedure Call standard (RPC). NDR accomplishes two things: (1) a language/OS/machine independent representation of elementary and constructed data types, and (2) a way of documenting these data types in protocol specifications.

RPC is a message passing protocol specified using NDR. RPC lies between a transport protocol and a specific service protocol. RPC also specifies a way of documenting argument(result) parameters to(from) remote service "procedures".

1. Principles

NDR is not a protocol

The specification of Sun's Network Data Representation is NOT a protocol. Rather, it is a method for describing Sun's yet-to-be defined protocols. NDR standardizes the representation of data types and how these types are serialized by a producer and deserialized by a consumer. Given this standard, NDR can then be used to send call and reply messages between two processes.

RPC is a protocol

Sun's Remote Procedure Call is a Protocol; it will be completely specified using NDR. All of Suns network services will be based on the Sun RPC protocol. (See the next principle concerning what RPC is based on.) RPC addresses the following issues which are common to all protocols: (1) The structure of messages passed between two cooperating processes on an internet, (2) The semantics of these messages. (3) Authentication / Verification / Security of the messages passed between cooperating processes over unsecure media such as phone lines and Ethernets. Authentication certifies one network entity to another (for example, "I am Bob Lyon and I am talking to you, Krypton, a network file server"). Verification assures that (a) Authentication parameters are genuine, (b) Authentication parameters cannot be "ripped off" by a third party who wishes to masquerade as a client, and (c) complete messages cannot be replayed at some later time by a third party (the third party may not know what the messages meant, but he could still try to rape a system by replaying previously valid messages. A message that I would like to replay many times is "add two weeks salary to Bob Lyon's account".)

(With respect to authentication and verification, you cannot have one without the other. In the example of Bob Lyon talking to Krypton, one process is certain that he represents Bob Lyon, but has very serious doubts that he is talking to Krypton. The other process is certain that it represents Krypton, but has doubts about the authenticity of Bob Lyon.)

Finally, security addresses two passive and one active threats concerning the protocol specific parameters that are passed in the messages between the cooperating processes. All threats are avoided by allowing the protocol specific parameters themselves to be encrypted. The first passive threat is that of violation of privacy; a third party should not be able to see how much money Bob Lyon makes in two weeks time by watching the message passed over the internet. The other passive threat is that of clear-text traffic analysis; the third party should not be able to see exactly which procedures to which programs I am calling over the net (however, just by watching the size, direction, and frequency of messages, considerable amount of information may be inferred). The active threat is that of parameter modification. Encryption of parameters also protects against this threat.

RPC is transport independent

The specification of RPC is not concerned with how data is transported from one process to another (with one exception concerning arbitrary stream protocols). Given that data was moved from one process to another, RPC is concerned about the structure of that data. Originally Sun's protocols will most likely be transported by TCP/IP or IP. However, there is no reason to disallow other transportation methods such as SNA or Xerox's NS SPP or Packet-Exchange protocols. (For example, a "native" immediate addressing 10MBit Ethernet protocol would be most useful for booting.)

Transport independence means that in many cases RPC messages will have redundant information. For example, in the TCP/IP world, each service has its well known socket number that calls are made to. However, the program number corresponding to that service should still be passed in a message so that someday the same message can be transported by a protocol like XNS which does not associate one-to-one mappings of services and sockets. (In actuality, I can find counter examples of the above observations in both the IP and XNS worlds.)

NDR and RPC are machine and OS independent

Given that NDR concerns the interpretation of data, it is machine and operating system independent. The protocols built on top of RPC should strive to maintain this independence (how do we get rid of pids and gids for access-control purposes?). NDR (de)serializers and RPC dispatchers are very machine and OS dependent.

NDR is not the design for server or client data structures

The structure and sequencing of data as it is transmitted over media has nothing to do with the data structures that a service instance uses to implement a function for some client. The converse is also true - a clever set of C data structures does not define a protocol (RPC based or otherwise). People who have implemented protocols know that the two are usually related and that in some trivial cases, the two are the same.

RPC (almost) behaves like calling a procedure remotely

With the exception that a client can time-out and give-up or retry an operation, RPC behaves like calling a procedure. That is, the client blocks and waits for a return (a reply) before continuing execution. Furthermore, the results of a call (as sent by one server(replier)) is one reply message. That is, one call does not generate more than one reply.

Parallel execution is achieved at a higher level by use of parallel processes, but not by one process sending multiple calls and then attempting to match replies as they trickle back.

(The reader may conclude that I consider Sun's ND protocol ill-designed because it does not adhere to either of these implications. I believe that ND lets a client have two outstanding calls and a call generates four replies!)

(Client) Violators will Not be shot

The previous statement always applies to the servers of a protocol. Obviously, clients of the protocol may violate this model. For example, a client may broadcast a call and wait for n answers where $0 \leq n \leq infinity$, depending on exactly what the client is trying to accomplish. Other examples of (reasonable) client-side RPC model violations are available on request.)

NDR-RPC is a library package

So far I have only said how NDR and RPC will (more or less) restrict the protocol implementors' freedom. But since RPC is (will be) a standard, common utility routines will be supplied that will assist in (de)serializing C data structures (from)to the NDR standard bits. Other routines can help in sending, receiving, filtering, and dispatching-on RPC messages.

2. Important issues not addressed in this document

The following is a list of issues that must be resolved before reasonable network services and their protocols can be implemented.

Object names

How are objects named in an internetwork-wide manner? Objects represent clients (like Bob Lyon), service instances (like Krypton), and most likely service-specific resources. Note that message certification is based upon a public/private key scheme where these keys are associated with the named objects.

Access Control

People often confuse authentication with access control. RPC authenticates clients to services, but each service must implement its specific access control mechanisms.

Transactions, Locking, and Sessions

Some of these three are desirable in some of the protocols. All are difficult. We do not want to complicate RPC such that every client pays overheads for transactions, locking, or sessions. I claim that all three of these concepts can be implemented on top of RPC by the specific protocols that need them.

3. Basic Data Types

Definitions for all the well known, basic data types are described in this section. Constructed types for structures, arrays, variable lengthed arrays, and unions are described in the next section. Note that there will be no direct way to specify C's "fields" (packed data).

Using the definitions for basic and constructed types, the structure of messages will be defined.

(I chose the notation of the following types to be different from the C language types; I need to know if reviewers like this or if I should use C types. The confusion factor is probably high in either case. I have also added a Xerox Courier-like notation to each notation.)

Basic block size

Every NDR data type comprises $32n$ bits of arbitrary data. The bits of data are numbered from 0 to some $m-1$, where $(mMOD32)=0$. Although it has already been stated that NDR types are machine and OS independent, aligning values on "32-bit word" boundaries tends to make (de)serialization of arbitrary data structures easier on many machine architectures. In the cases where some number ($p<32$) of the bits are not used to carry information about the value of a type, the "pad" bits must be zero. Justification of this is given in a later section.

Long Integer

The data object of type "long integer" (Courier LONG INTEGER) represents a signed integer in the closed interval [-2147483648, 2147483647] (2147483648 is 2 raised to the 31st power). The NDR standard representation of this type is a 32-bit field that encodes its value as a two's complement binary number whose most significant bit and least significant bit are bits 0 and 31, respectively.

Long Unsigned

The data object of type "long unsigned" (Courier LONG CARDINAL) represents an (unsigned) integer in the closed interval [0, 4294967295] (4294967295 is one less than 2 raised to the 32nd power). The NDR standard representation of this type is a 32-bit field that encodes its value as a unsigned binary number whose most significant bit and least significant bit are bits 0 and 31, respectively.

Note that long integers and long unsigneds in the closed interval [0, 2147483647] have identical bit encodings.

Hyper Integer

The data object of type "hyper integer" (Courier HYPER INTEGER) represents a signed integer in the closed interval [-9223372036854775808, 9223372036854775807] (9223372036854775808 is 2 raised to the 63rd power). The NDR standard representation of this type is a 64-bit field that encodes its value as a two's complement binary number whose most significant bit and least significant bit are bits 0 and 63, respectively.

Hyper Unsigned

The data object of type "hyper unsigned" (Courier HYPER CARDINAL) represents an (unsigned) integer in the closed interval [0, 18446744073709551615] (18446744073709551615 is one less than 2 raised to the 64th power). The NDR standard representation of this type is a 64-bit field that encodes its value as a unsigned binary number whose most significant bit and least significant bit are bits 0 and 63, respectively.

Boolean

The data object of type "boolean" (Courier BOOLEAN) represents a logical datum whose value can either be TRUE or FALSE. The NDR standard representation of this type is a 32-bit field whose first 31 bits (bits 0 through 30, inclusive) are zero and whose bit 31 encodes either TRUE or FALSE. TRUE is defined as bit value 1 while FALSE is defined as bit value 0.

C programmers should be aware of the subtle difference between NDR's boolean and C's int that is sometimes treated as a logical datum. Ignoring word sizes, C's interpretation of FALSE matches NDR's. However, C's interpretation of TRUE is any int that is not zero. This implies that NDR's booleans map well to C, but that C's "boolean" do not map well to NDR (i.e., care must taken when serializing an int that represents a logical datum).

Also note that many booleans cannot be packed into a single 32-bit object. This tends to frustrate super-efficient programmers. (Too bad!) Protocol writers can do tricks with integers to accomplish boolean packing if they believe it is absolutely necessary. For example, if we wish to describe which of the mouse buttons are currently "down", we could use a short integer to document that 0 => no buttons down, 1 => right button down, 2 => middle button down, 4 => left button down, ... , 7 => all buttons down.

(Forward reference) Both the previous example and Boolean could be implemented as a type "enumerated".

Counted String

The ability to pass sequences of characters that represent strings (like server names or file path names) is very important. Counted strings accomplish this. The NDR standard representation of a counted string is a 32-bit field (a NDR long unsigned) that encodes n, the number of 8-bit bytes that comprise the string. n is immediately followed by $32*((n+3)/4)$ bits. (n is followed by enough 32-bit objects to hold n 8-bit bytes.) For $0 \leq b < n$, the most significant bit and least significant bit of byte b are bits $8*b$ and $(8*b)+7$, respectively. If $nMOD 4 \neq 0$, then the remaining (eight, sixteen, or twenty four) unused bits must be all zeros.

Note that this document does not say how to interpret these eight bit bytes. Sun's protocols will most likely interpret the bytes as ascii characters. However, the bytes could just as easily represent the run-encoding of a string in "the" international character set. (The encoding scheme uses null-characters (\0) to change character sets - a good reason not to pass C's null terminated character arrays. The international character set supports characters in most languages including Japanese and Chinese.)

How do you pass one character as a parameter? (My answer is that "I don't".) An international character is actually 16 bits. The eight most significant bits represent the character's set, while the eight least significant bits represent the character's value. All ascii characters are conveniently in character set zero. Therefore, a C 8-bit char occupying the low-order 8 bits of an NDR long unsigned could be a "network-character". However, such a representation is radically different than the representation of a counted string of length one whose byte is the same character.

The notation for specifying a counted string, s at protocol documentation time is

```
string s<>;
```

or

```
string s<max-allowed-byte-count>;
```

Either notation specifies that s is a counted string. The second notation lets the protocol designer specify a static upper bound on the byte count of a (valid, protocol specific) counted string parameter. For example, a lousy file server protocol may have the following in its specification:

```
string local_name<255>;
string full_name<10240>;
```

The corresponding Courier declarations to the previous four C-like declarations are as follow:

```
s: STRING;
s: STRING[max-allowed-byte-count];
local_name: STRING[255];
full_name: STRING[10240];
```

Opaque

In many cases, fixed-size uninterpreted data needs to be passed in a protocol. Handles for files, sessions, or clients could be described as opaque data types. The NDR notation for such a field is

```
opaque name[n];
```

or

```
typedef opaque[n] type_name;
```

where n is the number of 8-bit bytes (that is right - BYTES) that comprise the opaque type. Note that n is specified at protocol specification time.

The NDR standard representation of opaque data is $32*((n+3)/4)$ bits. (The opaque data is represented by as many 32-bit objects that are necessary to hold the data.) The bits 0 through

$(8*n)-1$ contain the n bytes of opaque data. If $nMOD4 \neq 0$ then the remaining (eight, sixteen, or twenty four) bits must be zero.

For example a file server could return a handle to a file that the client would use at some later time. Assume the the file handle is represented by 7 bytes. It could be declared as:

```
opaque fhandle[7];
```

or

```
typedef opaque[7] fht;
fht fhandle;
```

The data associated with fhandle would require 64 bits to represent.

Note that varying sized opaque objects can be implemented via the counted string data type.

4. Constructed Data Types

Building upon the basic types described above, we can now specify more complicated data structures which are usually necessary in real world protocols. In some instances, types will effectively be equivalent to the types described above. These types will serve to more clearly document the specific protocol.

The constructed data types may be highly recursive. That is, elements of arrays, unions, or structures may be themselves arrays, unions, or structures.

Fixed and Varying Arrays

NDR's fixed array type (Courier ARRAY) is a protocol documentation trick. The fixed array is used to state at protocol documentation time that some fixed n values of one type will appear one after another. The notation for fixed arrays is similar to C's notation for allocated arrays:

```
type-name variable-name[fixed-element-count];
```

or in Courier

```
variable-name: ARRAY fixed-element-count OF type-name;
```

For example, the days of the week could be represented in a fixed array of (multi-national) strings:

```
counted string days[7];
```

or in Courier ...

```
days: ARRAY 7 OF STRING;
```

Varying arrays (Courier SEQUENCE) specify that the size of the array of homogeneous data elements will be specified at transfer time. Since the size of the array varies, the sequence of array elements is preceded by the array's element count represented by a NDR long unsigned. At protocol specification time, the designer will most likely want to specify an upper bound on the number of elements that may exist in the array; this number only appears in protocol specs and does not appear "on the wire". The syntax for specifying a varying length array of some element type will be either of the following (note the angle brackets rather than the square brackets):

```
type-name variable-name<max-element-count>;
type-name variable-name<>;  /* no specified max element means up to 4294967295 ok */
```

Courier notation looks like:

```
variable-name: SEQUENCE max-element-count OF type-name;
variable-name: SEQUENCE OF type-name;  /* no specified max element means up to 4294967295 ok */
```

For example an "open directory" function on a file server may return handles for all the file

handles in the directory. In this example, assume the protocol designer decides that a directory could not have more than 64 files. The specification may be something like:

```
typedef fht<64> fhts;
fhts file_handles;
```

Courier ...

```
Fhts: TYPE = SEQUENCE 64 OF Fht;
file_handles: Fhts;
```

Structures

NDR structure notation (Courier RECORD) is also only a protocol documentation aid. Structures represent an ordered, heterogeneous collection of data objects whose types are specified at documentation time. Any component of the structure may be any NDR types, including other structures.

The NDR standard representation of a structure is simply the standard representation of its components, one followed by another, in proper order. The specification of a NDR structure is similar to C's structs:

```
structure structure-type-name {
        first-component-type first-field-name,
        . . .
        last-component-type last-field-name
} structure-variable-name;
```

The corresponding Courier notation is (everything that could be different is):

```
structure-type-name: TYPE = RECORD [
        first-field-name: first-component-type,
        . . .
        last-field-name: last-component-type ];
structure-variable-name: structure-type-name;
```

The previous example could be modified so that a function returned both the files' handles and the files' names. The type declaration would look like:

```
structure h_and_n {
        fht handle,
        counted string name
};
typedef h_and_n<64> h_and_ns;
```

Or in Courier ...

```
h_and_n: TYPE = RECORD [
        handle: fht,
        name: STRING];
h_and_ns: TYPE = SEQUENCE 64 OF h_and_n;
```

Enumeration and Unions

Enumeration is just another way to document the meaning of NDR's long signed or long unsigned integers. It lets the protocol writer specify that only some subset of the integers are valid and attach names to those values. The specification of an enumeration will be like C's typedef and define statements. A contrived example of an enumeration would be a procedure that returned a variable that said it worked or not (of course a boolean would work just as well here).

```
typedef enumeration { error=-1, ok=0 } RESULT;
```

Courier's representation has no key word

```
RESULT: TYPE = { error(-1), ok(0) };
```

This means that short integers of type RESULT only have two valid values - zero and negative one. Enumeration types are not very exciting by themselves, but are very useful in specifying (discriminated) unions.

A NDR "union" (Courier CHOICE) allows a variable to hold different types (of potentially different sizes) at different times. That is, NDR type union represents a data object whose NDR type is chosen at run time from a set of candidate types specified at protocol documentation time. The candidate types are designated by values of NDR enumerations. Candidate types may be any NDR type, simple or constructed. A candidate type may itself be NDR type union.

The NDR standard representation of a union is the (32-bit) enumeration value followed by the NDR standard representation of implied candidate type value. The notation for specifying a union in a protocol document looks more like Mesa than like C; it involves declaring the enumeration types followed by a union type with a switch in its declaration. The previous example could be extended so that if an error occurred, a system error number was returned along with an error message (a nasty string); the type of the return value would be declared as follows (notice that there is no "default" or "break"):

```
typedef enumeration { error=-1, ok=0 } RESULT;
typedef union switch ((RESULT)r) {
        case ok: structure {};  /* this takes zero bits to represent */
        case error: structure { /* this takes at least 64 bits */
                long integer sys_err_no;
                counted string nasty_msg;
        };
} result;
result rslt;
```

Couriers notation begins to shine...

```
RESULT: TYPE = { error(-1), ok(0) };
Result: TYPE = CHOICE RESULT OF {
        ok => RECORD[], /* this takes zero bits to represent */
        error => RECORD [ /* this takes at least 64 bits */
                sys_err_no: LONG INTEGER;
                nasty_msg: STRING ];
};
```

NDR's union differs from C's in two ways. First, the current type of the union must be passed as part of the value of the union. Second, the size of the NDR union is dynamic since the NDR types in the union are potentially dynamic. (In the example, the number of bits needed to express the OK variant is 32, where the number of bits needed for the ERROR variant is at least 96 (when the message text has zero bytes).)

The key word "default" (Courier ENDCASE) is allowed in specifying unions and has the obvious semantics. The lack of a default (as in the above example) means that default values are not allowed (i.e., they are a protocol violation).

Defaults in unions will typically allow a protocol writer to design a protocol before she has figured out how to build a server that serves the protocol. (The nice way of saying this is that the default allows an open-ended architecture at the parameter level.) The default case will usually select a varying length array of opaques. For example, file handles that the server returns to the client may be constructed in this way.

Float and Double Float

The ability to describe floating point numbers is desirable in many applications. Describing floating point numbers in machine independent formats almost guarantees that the NDR representation will not match any machine architecture. I propose that we either punt the issue or suggest the following representations:

```
typedef structure {
        boolean is_negative;  /* encodes the sign of the mantissa */
        long integer exponent;  /* the unbiased exponent */
        long unsigned mantissa; /* the absolute value of the mantissa */
} float;

typedef structure {
      boolean is_negative;
      long integer exponent;
      hyper unsigned mantissa;
} double float;
```

Tom Lyon and Bill Shannon propose that (short = long) floats be represented as counted (ascii) strings. If I understand them correctly, there exists an (IEEE?) standard that specifies how to convert between floats and ascii strings. David Goldberg would rather see a straight forward IEEE bit representation that matches that of a Vax or Sun machine.

As a protocol documentation aid, we still may want to distinguish shorts from longs from quads.

5. The RPC Protocol (Call and Reply Messages)

The passing of messages is a protocol. The protocol's layer is between the transport method and the particular "service" protocol. This message layer provides Authentication, Verification, and Security of messages to the higher protocol layers. Corresponding to this protocol layer will be a common software utility package that does basic message processing.

The message protocol will be specified by NDR types.

Arguments, Parameters, and Results

Parameters flow in either direction. That is, argument parameters are passed to procedures. The procedures return result parameters. This document often leaves the word "parameter" off of "argument parameters" and "result parameters".

Introduction to Authentication

The bare-bones, opaque declaration of parameters necessary to pass credentials and verifiers are defined here. How these opaque data are constructed and validated is discussed in detail in a later section.

```
typedef opaque[4] long_opaque;  /* an opaque, 32 bit object */

typedef union switch((long unsigned)flavor) {
        default: long_opaque body<>;
} CREDENTIAL;  /* there are many flavors of credentials, but they all have the same opaque structure */

typedef union switch((long unsigned)flavor) {
      default: long_opaque body<>;
} VERIFIER;  /* there are many flavors of verifiers, but they all have the same opaque structure */
```

Courier ...

```
LongOpaque: TYPE = OPAQUE[4];

CREDENTIAL: TYPE = CHOICE LONG CARDINAL OF {
        ENDCASE => SEQUENCE OF LongOpaque };

VERIFIER: TYPE = CHOICE LONG CARDINAL OF {
      ENDCASE => SEQUENCE OF LongOpaque };
```

Call Message Bodies

In order for client-software to call a remote procedure, it must pass the following information (not necessarily in order): the "program" that contains the desired procedure, the procedure itself, the arguments to the procedure, and the credentials-verifier pair that allows implementation of Authentication, Verification, and Security. Examples of programs are "File Service", "Authentication Service" and "Print Service". The procedures and their arguments are protocol dependent. Credentials and verifiers are defined in this protocol level, but addressed in a later section. For now, just assume that a credential is a long-lived, static handle that identifies a "client" (like, "I am Bob Lyon") to a service (like, "you are Krypton") whereas a verifier is once-used, dynamic parameter that guarantees the validity of the credential.

There are two other desirable data that should be passed with each call message. First, there is the version number of NDR/RPC, itself. Although we never expect NDR/RPC to change, this version number will distinguish one generation from another if this protocol ever does evolve. This document is only concerned with version 1 of NDR/RPC. Second, there is the version number of the program being called. Again, we do not expect well-specified, well-implemented protocols to evolve; however, if they do, then we are covered. (There are exceptions to this. A "worm" server (a server that accepts executable code over the network) would probably change its version number every time the user/kernel interface changed. This could prevent old programs from being executed on new, incompatible kernels.)

```
typedef union switch (long unsigned)rpc_vers) {
        case 1: structure {  /* note ONLY version one is defined */
                CREDENTIAL cred;
                VERIFIER ver;
                long unsigned program;
                long unsigned version;
                long unsigned procedure;
                protocol_specific arguments;
        };
} call_body;  /* what calling parameters look like in rpc version one */
```

The same bits in Courier ...

```
call_body: TYPE = CHOICE LONG CARDINAL OF {
        1 => RECORD [  /* note ONLY version one is defined */
                cred: CREDENTIAL,
                ver: VERIFIER,
                program: LONG CARDINAL,
                version: LONG CARDINAL,
                procedure: LONG CARDINAL,
                arguments: protocol_specific_type;
        ];
};
```

Reply Messages

Eventually, the client expects the server to reply to his call. Either the call successfully completed (protocol specific errors are considered successful completions) or one of the following errors occurred: 1) the targeted machine no longer executes RPC protocol-version 1, but it does know how to generate the rpc-version-mismatch reply, 2) the authentication parameters of the call body were not acceptable, 3) the desired remote program is not executing on the targeted machine, 4) the remote program does not implement the desired version number, 5) the remote program does not implement the desired procedure, and 6) the remote procedure could not make sense out of the serialized arguments. The seven cases can be broken into two major categories - the call message was accepted by the remote rpc implementation or it was not.

```
typedef enumeration { mess_accept=0, mess_denied=-1 } REPLY_TYPE;

typedef enumeration { call_complt=0, prog_not_here=-1, wrong_vers=-2,
                      no_such_proc=-3, garbage_args=-4 } ACCEPT_TYPE;

typedef enumeration { rpc_v1_not_here=-1, auth_error=-2 } REJECT_TYPE;

typedef enumeration {
        bad_credential=-1, /* bogus credential (broken seal) */
        rejected_credentials=-2, /* client should get new credentials */
        bad_verifier=-3, /* bogus verifier (broken seal) */
        rejected_verifier=-4, /* verifier expired in transit or was replayed */
        auth_too_weak=-5 /* the remote program refused the particular
                            parameters on security grounds. */
} AUTH_PROBLEM;

typedef union switch ((REPLY_TYPE)rt) {
        case mess_accept: structure {
                VERIFIER reply_verifier; /* authenticates this reply message */
                union switch (ACCEPT_TYPE)ma) {
                        case call_complt: structure {
                                protocol_specific results;
                        };
                        case prog_not_here: structure {};
                        case wrong_vers: structure {long unsigned low_ver, high_ver};
                        case no_such_proc: structure {};
                        case garbage_args: structure {};
                        default: structure {};
                }; /* end message accepted case */
        };
        case mess_denied: union switch ((REJECT_TYPE)mr) {
                case rpc_v1_not_here: structure {long unsigned low_ver, high_ver};
                case auth_error: AUTH_PROBLEM why;
        };
} reply_body;
```

Courierized, it looks like:

```
REPLY_TYPE: TYPE = { mess_accept(0), mess_denied(-1) };

ACCEPT_TYPE: TYPE = { call_complt(0), prog_not_here(-1), wrong_vers(-2),
                      no_such_proc(-3), garbage_args(-4) };

REJECT_TYPE: TYPE = { rpc_v1_not_here(-1), auth_error(-2) };

AUTH_PROBLEM: TYPE = {
        bad_credential(-1), /* bogus credential (broken seal) */
        rejected_credentials(-2), /* client should get new credentials */
        bad_verifier(-3), /* bogus verifier (broken seal) */
        rejected_verifier(-4), /* verifier expired in transit or was replayed */
        parms_too_weak(-5) }; /* valid, but refused on security grounds */

reply_body: TYPE = CHOICE REPLY_TYPE OF {
        mess_accept=> RECORD [
                reply_verifier: VERIFIER, /* authenticates this reply message */
                CHOICE ACCEPT_TYPE OF {
                        call_complt=> RECORD [results: protocol_specific_type],
                        prog_not_here=> RECORD [],
                        wrong_vers=> RECORD [low_ver, high_ver: LONG UNSIGNED],
                        no_such_proc=> RECORD [],
                        garbage_args=> RECORD [],
                        ENDCASE => RECORD []
                },
        ],
        mess_denied=> CHOICE REJECT_TYPE OF {
                rpc_v1_not_here=> RECORD [low_ver, high_ver: LONG UNSIGNED],
                auth_error=> AUTH_PROBLEM
        },
};
```

Message Specification

The call and reply bodies are encapsulated with RPC headers that distinguish the direction and provide message transaction ids so that duplicate call messages may be filtered on the server side and late replies can be filtered out on the client side.

```
typedef opaque[4] x_id;  /* 32-bit, transaction id for filtering / matching messages */

typedef enumeration { call=0, reply=1 } direction;

typedef structure {
        x_id id;  /* reply ids should equal call ids */
        union switch ((direction)d) {
                case call: call_body;
                case reply: reply_body;
        };
} message;
```

Courier ...

```
x_id: TYPE = OPAQUE[4];  /* 32-bit, transaction id for filtering / matching messages */

direction: TYPE = { call(0), reply(1) };

message: TYPE = RECORD [
        id: x_id,  /* reply ids should equal call ids */
        CHOICE direction OF {
                call => call_body,
                reply => reply_body
        }
];
```

The type message (and its implied sub-fields) completely specify the RPC protocol. However, for each transport that is used to move messages, we must be able to recognize the boundaries between messages. In streaming environments, an "end-of-record" marker should be placed after each message. (Streaming environments include SPP, TCP/IP, disk drives (raw files), and tapes. Unfortunately, I have just discovered that TCP does not support "record demarcation", so a thin record marker protocol must be interfaced to TCP so that RPC may run on top of it.) Exactly one message should fit into a datagram used by the non-streaming protocols (UDP/IP, and IP).

Remote procedure call notation

How about:

```
typedef long unsigned PROGRAM;
typedef long unsigned VERSION;
/* tell what transport methods are used by the protocol */
define program_name (PROGRAM)program_number;
define program_version (VERSION)version_number;
proc_number: proc_name(arguments) -> (results);
        /* now declare all the types of the arguments and results */
```

6. The Marriage of C and UNIX to RPC - Descriptions and Filters

The non-C-like data structures can be very intimidating to a C/UNIX programmer. The desperate one could deduce what needs to emitted or read-in in order to adhere to the protocol. However, we would probably wish to provide a utility package that generated standard call messages and received standard reply messages for the user. The client side may look something like:

```
typedef struct {
        char *parm_ptr; /* where the interesting data comes from / goes to */
        int (*parm_descrpt)(); /* proc that describes the structure of the data */
} parameters;
rpc_callit(fd, prog, ver, proc, args, results);
        int fd;  /* a handle to an i/o stream */
        long int prog;  /* the desired program number */
        int ver;  /* the program's version number */
        int proc; /* the procedure number that we are calling */
        parameters args;
        parameters result;
```

Here, all the arguments to the procedure are gathered up in a structure. The results will also be placed in a structure. The addresses of the arguments and results are passed to rpc_callit along with a procedure that describes what fields inside the structures are meaningful and in what way. The RPC utility package will then build the complete call message using some internal routines and calling parm.parm_descrpt at the appropriate time. The RPC specific (header) data of the reply will be deserialized and inspected by the utility and the results will be placed into

results.parm_ptr-> with the help of results.parm_descrpt.

How parameter descriptors work will be addressed in another memo. However, some properties of parameter descriptors are worth noting now. First, the descriptors tell which fields of the structure are important to the protocol and in what way; other fields of the structure may be completely ignored. Next, the parameter descriptors are recursive in the sense that the structures can contain sub-structures (or pointers to sub-structures) that have their own parameter descriptors. Finally, the parameter descriptors work in either direction - they serialize C structures into NDR bits or deserialize NDR bits into C structures. Therefore, the descriptors that the client uses to serialize data can be used by the server to deserialize data and vise versa.

(This may sound impossible to some, but I have a data point that says it can easily be done ... at least in an object-oriented environment ...)

7. More Epistles

Why Protocol should be general

(De)serialization and Real Databases

The Near Uselessness of Protocol Compilers

8. Credentials and Verifiers in Gory Detail

The section discusses in gory detail the architecture and construction of credentials and verifiers. As stated above, all credentials and verifiers have the same opaque structure:

```
typedef opaque[4] long_opaque;

typedef union switch((long unsigned)flavor) {
        default: long_opaque body<>;
} CREDENTIAL;  /* there are many flavors of credentials, but they all have the same opaque structure */

typedef union switch((long unsigned)flavor) {
        default: long_opaque body<>;
} VERIFIER;  /* there are many flavors of verifiers, but they all have the same
opaque structure */
```

We will first look at the two extreme flavors of credentials and verifiers. Then suggest some other flavors. (I am trying to convince David Goldberg to put some English into this section.) Consider:

```
typedef ??? CLIENT_ID;  /* defines what a "name" is the internet */

typedef long unsigned TIMESTAMP;  /* seconds since Jan 1, 1900, 0:00, gmt */

typedef hyper unsigned OCCURRENCE;  /* 64 bit, microsecond counter */

typedef enumeration { null=0, complete=1 } CRED_FLAVOR;

typedef enumeration { null=0, complete=1, complete_but_change_cred=2 } VER_FLAVOR;
```

```
typedef long_opaque<0> NULL_CRED_FLAVOR;

typedef long_opaque<0> NULL_VER_FLAVOR;

typedef structure {
        long unsigned the_cred_length;  /* the 32-bit object length of the_cred */
        structure {
                CLIENT_ID c_id;  /* un-encrypted id of the client */
                long unsigned clear_checksum;  /* the checksum of the serialization
                                of the following record BEFORE it is encrypted */
                /* the serialization of the clear text is encrypted with the a key that is the
                  product of the the serializer's private key and the deserializer's public key.
                  Note that the size is a multiple of 64-bit in length */
                structure {
                        TIMESTAMP creation_time;
                        hyper unsigned conversation_key;
                        TIMESTAMP expiration_time;
                } secret_crud;  /* only the client and server can understand this stuff */
        } the_cred;
} COMPLETE_CRED_FLAVOR;

typedef structure {
        long unsigned the_ver_length;  /* the 32-bit object length of the_ver (always 5). */
        structure {
                long unsigned clear_checksum;  /* the checksum of the serialization
                                of the following record BEFORE it is encrypted */
                /* the serialization of the clear text is encrypted with the
                  conversation_key that was passed in the credential.
                  Note that the size is a multiple of 64-bit in length */
                structure {
                        OCCURRENCE current_time;
                        boolean parms_are_also_encrypted;
                        long unsigned filler;  /* makes this struct 64*2 bits long */
                } secret_crud;  /* only those with the conversation_key can understand this stuff */
        } the_ver;
} COMPLETE_VER_FLAVOR;

typedef structure {
        long unsigned the_ver_length;  /* the 32-bit object length of the_ver (always odd) */
        structure {
                long unsigned clear_checksum;  /* the checksum of the serialization
                                of the following record BEFORE it is encrypted */
                /* the serialization of the clear text is encrypted with the
                  conversation_key that was passed in the credential.
                  Note that the size is a multiple of 64-bit in length */
                structure {
                        OCCURRENCE current_time;
                        boolean parms_are_also_encrypted;
                        CREDENTIAL new_credential;  /* always treated as "opaque" by the client */
                        long_unsigned filler<1>; /* use to make this struct 64*n bits long */
                } secret_crud;  /* only those with the conversation_key can understand this stuff */
        } the_ver;
} COMPLETE_VER_BUT_CHANGE_CRED_FLAVOR;
```

```
/* now we can exactly define credentials and verifiers */

typedef union switch((CRED-FLAVOR)flavor) {
        case null: NULL_CRED_FLAVOR;
        case complete: COMPLETE_CRED_FLAVOR;
        default: long_opaque body<>;
} CREDENTIAL;

typedef union switch((VER_FLAVOR)flavor) {
        case null: NULL_VER_FLAVOR;
        case complete: COMPLETE_VER_FLAVOR;
        case complete_but_change_cred: COMPLETE_VER_BUT_CHANGE_CRED_FLAVOR;
        default: long_opaque body<>;
} VERIFIER;
```

9. Example: The remote RPC information program/protocol (RRPCI)

The RPC remote information program is very much like the unix ps command. ps tells the user which processes are currently executing on a unix machine whereas a remote procedure in the information program can tell the caller which remote programs are currently being executed on a server machine.

The remote program should also have a procedure that lets a client find the machine (and discover its id) and a null procedure to do performance measurement with. The following is a complete specification of RRPCI:

```
define RRPCI (PROGRAM)1;
define VERS_RRPCI (VERSION)1;
/* RRPCI is currently supported by RPC/UDP/IP on port xx
  and RPC/RP/TCP/IP on port xx. */

/* data type needed for this protocol */
/* Actually, all "addressing" schemes should be standardized and declared in one place */
typedef enumeration { darpa=0, xns=1 } TRANSPORT_FAMILY;

typedef enumeration { ip=0, udp_ip=1, tcp_ip=2, spp_ip=3 } DARPA_TRANSPORT;
        /* spp_ip=3 really is not a darpa protocol */

/* if we knew what we doing, XNS transport protocols would be defined here */
typedef long unsigned XNS_TRANSPORT;

typedef union select ((TRANSPORT_FAMILY)t_family) {
        case darpa: structure {
                long unsigned ip_addr;
                long unsigned port;  /* only the low order 16 bits have meaning */
                DARPA_TRANSPORT supported_trans;
        } darpa_addr;
        case xns: structure {  /* this will remain unimplemented for a while.... */
                long unsigned xns_net;  /* 32 bit net number */
                hyper unsigned xns_host; /* 48 bit host number in the low order 48 bits */
                long unsigned port;     /* only the low order 16 bits have meaning */
                XNS_TRANSPORT supported_trans;
        } xns_addr;
} SUPPORTED_ADDRESS;
```

```
/* now for some remote procedures */

1: please_responder () -> (responders_addrs, responders_id);
        SUPPORTED_ADDRESS responders_addrs<>;
        CLIENT_ID: prog_id;
        /* this procedure takes no arguments and returns to the caller
           all the addresses that the responder can be reached.
           The responder's id is also returned. Guaranteed to respond to
           null credentials and null verifiers. */
        /* EVERY SUN SERVICE SHOULD HAVE A PROCEDURE LIKE THIS IN ITS PROTOCOL*/

2: null_proc () -> ();
        /* the procedure takes no arguments and returns no results.
           It will be useful in doing performance measurements.
           Also guaranteed to respond to  null credentials and null verifiers,
           although other flavors may be used for debugging or performance
           purposes. */

3: enumerate_progs: () -> (current_progs);
        structure {
                PROGRAM prog;
                VERSION low_version, high_version;  /* current protocol range supported */
                CLIENT_ID: prog_id;  /* the remote program's name */
                counted string prog_name;  /* human readable, like "krypton"*/
                counted string prog_dscrpt; /* human readable, like "file server" */
        } current_progs<>;
        /* the procedure returns an array whose elements are the above structure.
           Given a machine's address, you can discover what remote programs are
           currently being executed on that machine. */

/* end of RRPCI program specification */
```

Serialized messages carrying RRPCI calls.

This section will someday have the argument and result parameters serialized into their corresponding call / result RPC messages.

10. Why NDR/RPC is not Xerox Courier

This document specifies a system that looks alot like Xerox's Courier system. This section describes the differences and the reasons for the differences.

Protocol (In)flexibility

Courier is only based upon xns SPP. Since Courier is only based on this connection oriented transport protocol, the versions supported by Courier are negotiated once per stream creation. A good aspect of this technique is that it saves a few bits in the call message header. A bad aspect of this technique is that a single call may require multiple packet overheads for the following: (1) SPP stream creation, (2) Courier version number negotiation, and (3) SPP stream destruction. The sole use of SPP often appears to be using a cannon to kill a mouse; much computing may be involved with "pingingu" the other machine while a stream is idle, but not destroyed. More cycles may be required to determine just when to shut down a connection that cost so much to build.

PRC tries to be transport media/protocol independent. Firstly, many service protocols do not need the reliability and flexibility of a stream protocol. Therefore, they should not pay the price

of set-up and tear-down of these transport methods if they are not needed. However, each message now has a little more overhead than a similar message would have in Courier's world.

Secondly, different protocols have different transport needs. In many cases (like a boot protocol), basing the protocol on something as complicated as SPP or TCP would make implementing a function infeasible.

Finally, there is little advantage in locking our services' protocols into one transport method. By adding a little complexity now, Sun can be sure that its protocols can easily migrate to new markets when and if the markets develop (read this as "I want to put them on xns").

Basic object size

Courier's objects are all represented as n 16-bit words. Sun's NDR represents objects as m 32-bit words. This "biggerism" comes about because the world is gradually moving toward bigger architectures. (Particularly, Sun is already there.) Using more bits to represent similar data should not bother protocols that run on high band-width networks. However, this biggerism could severely affect protocol performance when calls must traverse slow networks. Again, the world is moving toward faster nets - 300 baud is dead and 1200 baud will soon die with respect to gateway links.

Because of this 32-bit word representation scheme, some of Courier's types disappear in NDR representation (like INTEGER and CARDINAL). In anticipation of huge numbers (like DES keys), NDR defines 64-bit integers - Courier does not.

Message authentication

Courier is not in the business of authentication. Individual protocols implement their own authentication via protocol/procedure specific parameters. There are two engineering problems with Courier's lack of authentication.

Firstly, many protocol/service don't bother to do authentication since they view as hard. These protocols must then be evolved when customers complain about the insecure systems. Secondly, authentication is re-implemented n times in n different ways by each protocol/service writer.

By having RPC guarantee the authenticity of each message, each protocol/service implementor is freed to worry only about her specific service problems. By insisting on authentication on a per message basis, RPC actually prevents insecure protocols from being released.

The down side of this attitude is that RPC itself become more complicated and harder to implement.

Mesa signals on a network?

RPC has call messages and reply messages. Courier has call messages and three specific flavors of reply messages - reject, return, and abort. Reject messages implemented what RPC calls a call-message-denied reply. The return message is only used to note a successful procedure call. If the procedure returns with an error condition, the abort message is used.

The return/abort messages emulated what Signals are like in Mesa, though the implementation and protocol aspects of network-signals were severely limited in their functionality (network-signals could not be resumed) and they tended to consume large amounts of cpu time to implement.

What they arn't telling you

Courier was never meant to transport huge arguments or results. (I claim that RPC should not be abuse in this way, either.) Riding on top of Courier is a kludgy, "Bulk Data Transfer" protocol that implements arbitrary sequence of record streaming after the call message is sent but before the result message is sent. The protocol specification of Bulk Data Transfer is a little hairy and the implementation is a gross package that causes its maintainers and the services implementors endless grief. (The worst part of this scheme is that Courier must share its SPP stream with a

client-service at each end the bulk data connection. The client can then rape the stream in one of many ways and then give it back to Courier. Often Courier cannot detect the problem and various network processes hang in various strange ways. I claim a Sun implementation of this mess would be no better than Xerox's.)

The right way to move huge amounts of data is with a parallel (not shared!) stream whose rendezvous specifications are passed in the remote procedures arguments or results.