ONC XDR for Lisp

Gene Michael Stover

created Wednesday, 2003 August 6
updated Friday, 2007 July 13

Contents

Chapter 1

Introduction

This is an implementation of Open Network Computing (ONC) XDR for Common Lisp. ONC XDR is also known as Sun XDR. XDR stands for eXternal Data Representation. It is the data format used by ONC RPC (which is also known as Sun RPC, which is the RPC behind such favorites as the Network File System (NFS)).

The protocol for ONC RPC is specified in RFC 1831 ([Sri95a,)].

Chapter 2

XDR Library

XDR is defined in RFC 1832 ([Sri95b,)].

xdr.lisp. *Unfinished, but I last worked on it on 2004 November 29.*

XDR allows type definitions. I'd like XDR types to be defined in Lisp dynamically (without using the EVAL function). So at run-time you could define XDR types & an RPC service that uses it, like we do with SOAP & XML-RPC these days.

This means that XDR types must map to types which already exist in the Lisp environment. For example, an XDR `struct` cannot map to a Lisp `struct` that was created just for that XDR `struct`. Instead, the XDR `struct` must map to some Lisp type which already exists.

Table 2.1 is a map that shows XDR types & their corresponding representations in Lisp.

Many XDR numeric types map to Lisp's integer or number. The Lisp programmer sends or receives a particular XDR type of number by calling a function that sends or receives that XDR type of number regardless of the type of the Lisp number.

To a Lisp programmer, this might be odd. After all, if all those numeric types map to Lisp's number, why do all those XDR numeric types exist? If Lisp programmers were designing a data exchange language for Lisp[1], they probably wouldn't specify all those numeric types because Lisp doesn't normally require programmers to think about types of numbers, but XDR was designed for languages which distinguish between types of numberes. So the Lisp programmer using XDR must live with that. Internally, the Lisp programmer just has numbers, but when dealing with other programs through XDR, she must know whether an external datum in question is a `short` or `unsigned` from XDR's point of view.

[1] Lisp expressions *are* such a Lisp-to-Lisp data exchange language.

XDR	**Lisp**	**comment**
integer (4 octets)	integer	fixnum or bignum depends on Lisp implementatio
unsigned integer	integer	fixnum vs bignum depends on Lisp implementatio
enumeration	set of symbols	
Boolan	symbol	decodes as strictly T or NIL
hyper integer (8 octets)	integer	probably bignum
hyper unsigned integer	integer	probably bignum
character	character	
const	symbol	
double	number	
dynamic array	resizable array	
fixed array	fixed array	
float	number	
long	integer	
opaque	array of (UNSIGNED 8)	
short int	integer	
string	string	
struct	association list	could be a hash table
typedef	symbol	
union	association list	
unsigned long	integer	fixnum vs bignum depends on Lisp implementatio
program	symbol	
version	symbol	
procedure	symbol	

Table 2.1: XDR types & their corresponding representations in Lisp

1. a number, in which case the type is really a constant,
2. a symbol, in which case the type is a synonym for another type or for a constant,
3. a list of the form (:FUNCTION reader-fn writer-fn), where reader-fn & writer-fn can be used in a FUNCALL,
4. a list of the form (:STRUCT ($attr_0$. $type_0$) ...),
5. a list of the form (:ENUM ($symbol_0$. $value_0$) ...),
6. a list of the form (:UNION ($attr_0$. $type_0$) ...),
7. a list of the form (:VECTOR length type), in which case the type names a fixed-length array of *length* elements, each of type *type*,
8. a list of the form (:ARRAY length type), in which case the type names a variable-length array of at most *length* elements,
9. a list of the form (:OPAQUE length), in which case the type names a fixed-length array of *length* octets, or
10. a list of the form (:STRING length), in which case the type names a string of at most *length* characters.

Table 2.2: Possible values for the definition column in the XDR type symbol table

2.0.1 Dyanmic Data Definition

Inside the Lisp program, there is a symbol table for XDR types. That table has two columns: the type's name & it's definition.

The name is a symbol.

The definition may be any one of the things in Table 2.2.

This type table allows us to have general-purpose functions for XDR input & output. Here is pseudo-Lisp code for an XDR-READ function:

```
(defun xdr-read (type xdrs &optional (error-p t)
      (error-value xdrs))
  "Read the next datum from the XDR input stream, xdrs.
Assume that the next datum is of type TYPE.  On end-of-input
or other condition, if error-p is NIL, return error-value;
otherwise, raise an error."
  (cond ((and (symbolp type) (xdr-lookup-type type))
         ;; TYPE is a symbol, so it's probably a
         ;; synonym for some other type.  We process
         ;; it recursively.
         (xdr-read (xdr-lookup-type type) xdrs
```

```
                  :error-p error-p
                  :error-value error-value))
       ((listp type)
;; I don't like the brute-force, hard-coded
;; decomposition in this CASE expression, but
;; it seems to work fine.
(case (first type)
  (:array
   ;; A variable-length array.
   ;; SECOND is maximum length.
   ;; THIRD is the element type.
   (xdr-read-array (third type)
   (second type)
   xdrs
   :error-p error-p
   :error-value error-value))
  (:function
   ;; TYPE holds the I/O functions for a built-in
   ;; XDR type.
   (funcall (second type) xdrs
    :error-p error-p
    :error-value error-value))
  (:enum
   (xdr-read-enum (rest type) xdrs
  :error-p error-p
  :error-value error-value))
  (:string
   ;; A string is an array of characters.
   ;; Doesn't XDR allow for both fixed-length
   ;; strings & variable-length strings?  (Do
   ;; fixed-length strings make sense?)
   ;; We treat them all as variable-length
   ;; strings.
   ;; SECOND is length.
   (xdr-read-array 'char (second type) xdrs
   :error-p error-p
   :error-value error-value))
  (:struct
   ;; TYPE holds the description of an XDR
   ;; structure type.
   (xdr-read-struct (rest type) xdrs
    :error-p error-p
    :error-value error-value))
  (:union
   (xdr-read-union (rest type) xdrs
   :error-p error-p
```

```
    :error-value error-value))
   (:vector
    ;; TYPE describes a fixed-length array.
    ;; Its SECOND is the length.  It's THIRD
    ;; is the element type.
    (xdr-read-vector (third type)
     (second type)
     xdrs
     :error-p error-p
     :error-value error-value))))
(t
         ;; It wasn't a symbol or a list, or it was a
 ;; symbol, but it wasn't defined in the type table.
 (when error-p
   (cerror "Undefined XDR type: ~A" type))
 error-value)))
```

I dislike the big CASE , but it seems to work, & I don't know an alternative other than defining a more complex form for the values in the type table. I could replace that CASE with a lookup in another table of what to do, but that would only be a syntactic change; the hard-coded decomposition would still be there.

You might notice that there is no provision for arrays & strings of unlimited length, though XDR allows them. I chose not to implement arrays & strings of unlimited length because:

1. besides I/O functions for arrays & strings with maximum lengths, I'd have to create functions for unlimited arrays & strings,

2. current security practices say that you should always know an input buffer's length & should check it, &

3. memory is a finite resource in reality, so there is no such thing as an unlimited array or string.

My XDR library simulates arrays & strings of unlimited length by using arrays & strings with very large maximum lengths.

So the type table contains all the type & constant definitions, but the generic XDR reader function (XDR-READ) & the generic XDR writer function (XDR-WRITE) must know how to use those definitions.

The type table is primed with :FUNCTION entries for all the built-in XDR types: `bool`, `char`, `int`, `long`, `unsigned`, `unsigned long`, `float`, & `double`.

If a performance-conscious Lisp/XDR programmer hard-codes Lisp I/O functions for her XDR types, she can insert them into the type table with :FUNCTION entries, & they will effectively become built-in XDR types.

It would also be possible to write a function to "compile" the dynamic XDR type definitions by creating Lisp functions that read & write them, then replac-

ing all the entries in the type table with :FUNCTION entries that contained those new functions.

A generic XDR-WRITE function is the inverse of XDR-READ. Note that it handles constants. It recursively evaluates them until it finds a number, then writes that as an integer.

2.1 XDR Streams

The XDRS argument to all the XDR input & output functions is an XDR stream.

Because the XDR input & output functions need read & write only octets, the XDR streams could be Common Lisp binary input streams with an element type of `(:unsigned 8)`. This won't work if your Lisp doesn't provide a Common Lisp-style stream interface to network I/O. What's more, I'd like to allow file-based IPC with the aid of a small C program, for Lisps that don't offer network I/O at all & for debugging. And for a personal quirk.

So we need an actual XDR stream type.

```
(defstruct xdr-stream
  read-octet-fn    ; return next octet or self
  write-octet-fn   ; write an octet or return self
  sync-fn          ; ensure input & output buffers are flush
  close-fn         ; won't need this any more
  datum)
```

Each member of XDR-STREAM is a function or NIL. Each of those functions returns NIL on error or anything else on success. We use wrapper functions so that you needn't see the ugliness of calling functions through members of a structure.[2]

```
(defun xdr-read-octet (xdrs)
  "Return next octet or nil on error"
  (declare (type xdr-stream xdrs))
  (assert (xdr-stream-read-octet-fn xdrs))
  (funcall (xdr-stream-read-octet-fn xdrs) xdrs))

(defun xdr-write-octet (octet xdrs)
  "Write octet & return it or nil on error"
  (declare (type xdr-stream xdrs))
  (assert (xdr-stream-write-octet-fn xdrs))
  (funcall (xdr-stream-write-octet-fn xdrs)
           (mod octet 256) xdrs))
```

The XDR stream's "sync" function ensures that the entire content of the output buffer has been sent, & it clears the content of the input buffer. That is, it does these things where appropriate. For a network socket in a virtual circuit

[2] Yup, I could have used CLOS, Grey streams, or simple-stream's, but I didn't. *shrug*

protocol, it probably ensures that the output buffer is send & the input buffer is cleared. For a datagram protocol, it would send the datagram. It depends on the XDR stream.

The XDR stream's "close" function aborts[3] the buffers & does whatever cleanup is necessary when we will not be using the XDR stream again.

XDR streams are most frequently requested to read & write 32-bit quantities. It says so in the XDR specification. I'll call that 32-bit quantity an "XDR word" & provide a couple of conventience functions for it:

```
(defun xdr-read-word (xdrs &key (type 'vector))
  "Return vector or list of the next 4 octets or nil.
The keyword argument TYPE may be VECTOR or LIST."
  (let (a b c d)
    (and (setq a (xdr-read-octet xdrs))
         (setq b (xdr-read-octet xdrs))
         (setq c (xdr-read-octet xdrs))
         (setq d (xdr-read-octet xdrs))
         (if (eq type 'vector)
             (make-array 4
                         :adjustable nil
                         :element-type 'integer
                         :fill-pointer nil
                         :initial-contents (list a b c d))
           (list a b c d)))))

(defun xdr-write-word (word xdrs)
  "Word is a vector or list of 4 integers.  Write each
(mod 256).  Return xdrs on success, nil on error."
  (declare (type (or list array) word) (type xdr-stream xdrs))
  (assert (xdr-stream-write-octet-fn xdrs))
  (and (xdr-write-octet (elt word 0) xdrs)
       (xdr-write-octet (elt word 1) xdrs)
       (xdr-write-octet (elt word 2) xdrs)
       (xdr-write-octet (elt word 3) xdrs)))
```

You make an XDR stream by, well, making one, then setting all of its members. You may use the DATUM member for private data. For example, if your Lisp offers network sockets as streams, then you might keep that socket in an XDR stream's DATUM member.

I presume there will be functions to create XDR streams with one function call rather than with a MAKE-XDR-STREAM & a bunch of SETF forms. For example, there might be functions like these:

```
(defun make-tcp-xdr-stream (hostname port) ...)
```

[3]Should it abort or sync the buffers?

```
(defun make-udp-xdr-stream (hostname port) ...)

(defun make-x25-xdr-stream (circuit) ...)

(defun make-soap-xdr-stream (url) ...)

(defun make-subspaceradio-xdr-stream (frequency direction) ...)

(defun make-xdr-stream-from-url (url)
   (declare (type string url))
   ...)
```

Chapter 3

Source Files

It's best not to link to these files for now (September 2003) because this document is still under construction. These links will probably change.

- xdr.lisp

Chapter 4

Usage Scenarios

4.1 Load from a service description file

If the RPC service is described in a `*.x` file, the Lisp programmer can create all the data types in that file, plus client calls, with `load-rpc`. If the RPC service is described in the file `myservice.x`, the programmer just needs to evaluate "`(load-rpc "myservice.x")`". That will define Lisp types & client-side functions for all the types & remote procedures in the RPC file.

For example, let's say that `myservice.x` defines two types & one program with three procedures. The types are an enumeration & a composite. The file is like this:

```
/* this is myservice.x */

enum Color {
  red = 1,
  green = 2,
  blue = 3
};

struct Shoe {
  color c;
  int size;
};

program MyService {
  version Latest {
    /* Return the number of shoes in stock
     * that have the color. */
    int Inventory (Color) = 1;

    /* Buy a shoe, decrement the inventory. */
```

```
    void Buy (Shoe) = 2;

    /* Put more shoes in stock. */
    void Receive (Shoe) = 3;
  } = 1;
} = 0x19283;
```

If I evaluate "`(load-rpc "myservice.x")`", it will insert items into the XDR symbol table to describe the two types & one program from `myservice.x`. Any symbols it creates from `myservice.x` are in the current package. After that `load-rpc`, I could read & write Colors & Shoes like this:

```
;; We'll assume that *xdrs* is bound to an XDR stream
;; that is open for reading & writing.
@ (defvar *xdrs* ...)
*XDRS*

@ (xdr-write 'red 'color *xdrs*) ; Write the Color Red
#<XDR-STREAM ...>

;; Assume the program on the other end of *xdrs*
;; writes the Color Green
@ (xdr-read 'color *xdrs*)
GREEN

@ (xdr-write '(:struct (c . red) (size . 7)) 'shoe *xdrs*)
#<XDR-STREAM ...>

;; Assume the othe process writes a blue, size 12 Shoe.
@ (xdr-read 'shoe *xdrs*)
(:STRUCT (SIZE . 12) (C . BLUE))
```

sc load-xdr might also create convenience functions as if they were defined like this in the current package:

```
(defun read-color (xdrs &optional (error-p t) (error-value xdrs))
  (xdr-read 'color error-p error-value))

(defun write-color (color xdrs &optional (error-p t)
                    (error-value xdrs))
  (xdr-write color 'color xdrs error-p error-value))
```

To Do: LOAD-XDR also takes into account the information about the `MyService` RPC program, but I have not yet decided how that will be implemented.

4.2 XDR Translation Cases

4.2.1 Const

XDR constants must be integral. Translate them to constants in Lisp, but evaluate them as we initialize the XDR symbol table. So if, for example, an XDR `*.x` file defines a constant called `magic_number` to be 42, we stuff 42 into the XDR symbol table.

XDR constants are not transmitted. Rather, if you send an XDR constant, you send an integer. So the XDR symbol table doesn't have information to translate an incoming integer into a constant.

Even if I didn't think the XDR decoder should not translate incoming integers into symbolic constants, consider this case which demonstrates why it would be effectively impossible.

Let's say we have an XDR file that contains these items:

```
/* file.x */
const magic_number = 42;
const max_length = 42;
const bias = 42;
```

Let's say the remote process uses XDR-WRITE to write `magic_number` as an integer. We should use XDR-READ to read that integer. What would tell XDR-READ that the incoming integer is actually `magic_number`, `max_length`, `bias`, or just plain 42?

If you want integers to be decoded as symbols, use an XDR `enum`.

4.2.2 Enum

Let's say we have an XDR enum, described like this:

```
/* an XDR enumeration */
enum Color {
    red,
    yellow,
    green = 5,
    blue
};
```

This enumeration declares four symbols & associates them with integral values. "red" is associated with 0; when you write the "red" symbol, XDR writes a zero. When you read a Color enumeration value & that value is zero, XDR translates it to the symbol "red". "yellow" is associated with 1; "green", with 5; "blue", with 6.

Translated to Lisp, only the XDR functions need to know the numeric associations. A Lisp application can concern itself with the symbols & ignore the numbers.

The symbols *red*, *yellow*, *green*, & *blue*, become Lisp symbols declared in the package that is current when the XDR enumeration is parsed & converted to Lisp. If the current package is *myserver*, then parsing the XDR enumeration will produce the four Lisp symbols MYSERVER:RED, MYSERVER:YELLOW, MYSERVER:GREEN, & MYSERVER:BLUE. They will all be converted to the case that *read* would have used if it had read them from a stream of Lisp expressions. Normally, they will be converted to upper-case. This could cause problems sometimes, but it should work most of the time. Maybe there could be a global variable that selected between "let *read* convert case" & "preserve case under any circumstances".

We'll also need a symbol for the enumeration itself. In this case, that will be COLOR. The XDR symbol table associates the symbol COLOR with an association list that connects the enumeration symbols with their numeric values; that would be `((red . 0) (yellow . 1) (green . 5) (blue . 6))`.

4.2.3 Structures

Structures are composite types.

This XDR structure declaration:

```
struct point {
  int x;
  int y;
};
```

must translate to a Lisp structure type, functions for reading & writing the structure type, & maybe some introspection information.

The Lisp structure is easy enough:

```
(defstruct point x y)
```

It will not use all the features available to a Lisp structure. It'll use only basic Lisp structure features. It might not be the way a Lisp programmer would define a similar structure, but the idea is not to synthesize an Lisp-optimum data type from the XDR declaration. The idea is to create a data structure which can be translated between Lisp form & XDR form. If a Lisp programmer wants a more Lispy internal format, she must write those conversions herself.

We'll also need I/O functions for type `point`:

```
(defun read-point (strm)
  (make-point :x (xdr:read-int strm) :y (xdr:read-int strm)))
(defun write-point (val strm)
  (xdr:write-int (point-x val) strm)
  (xdr:write-int (point-y val) strm))
```

Because the Lisp XDR functions will toss an exception on error, there is no need to wrap them in an `and` like C XDR functions do. In the future, maybe

Lisp XDR functions will offer an option to raise an exception or return `nil`, in which case we'll want to wrap them in `and`, but that can be done later, when & if we need it.

Should there be introspection data? How should it be stored? As a member of the `point` structure? As a property of the `point` symbol?

4.2.4 Unions

Here is an XDR union:

```
/* an XDR union */
union result switch (int status) {
  case 0: int i;
  case 1: double j;
  case 2: int k;
  default: void; /* no payload with error */
};
```

In C & other bit-twiddling languages, unions & variant records exist to save space. They use overlapping memory to do that.

In Lisp, the manifest typing removes the need to have separately named members in a union. In other words, that union could translate to Lisp like this:

```
(defstruct result status arm)
```

Because we have manifest typing in Lisp, we don't need the `i` & `j` arms of the union. We can create the correct type of datum & bind it with the `arm`.

I know that would work, but I wonder if it would be difficult to debug. What if the status & the type of datum didn't agree? Would it be difficult to locate that bug? It wouldn't be difficult if the value in the arm was of some particular type (like a structure) & it should be some other particular structure type, but what if it was an incorrect integer & what it really should be was some other integer. (Like in the XDR example above, there are two arms that are integers. Could be confusing.

What might make debugging more useful & make the Lisp code more self-documenting is to create the actual arms that are in the XDR union. We'll bind a value to only one of them, selected by the descriminant. We'll need functions to read & write the union, too. So the XDR union above would translate to this Lisp code:

```
(defstruct result
  status
  i ; int, when status 0
  j ; double, when status 1
  k ; int, when status 2)
(defun read-result (strm)
```

```
  (let ((status (xdr:read-int strm)))
    (apply #'make-result
      (list :status status
            :i (if (eql status 0)
                   (xdr:read-int strm)
                 nil)
            :j (if (eql status 1)
                   (xdr:read-double strm)
                 nil)
            :k (if (eql status 2)
                   (xdr:read-int strm)
                 nil)))))
(defun write-result (val strm)
  (xdr:write-int (result-status val) strm)
  (case (result-status val)
    (0 (xdr:write-int (result-i val) strm))
    (1 (xdr-write-double (result-j val) strm))
    (2 (xdr-write-int (result-k val) strm))
    (t)))           ; don't write anything when
                    ; status < 0 or status > 2
```

I'm pleasantly surprised with the code. I had thought it would be crap, too difficult to understand, but it's pretty straightforward. I'm glad that a program, not I, will parse the XDR file & produce the code, but it's still not bad.

4.2.5 Typedefs

XDR allows `typedef` statements, but it does not process them. They are for C.

The `typedef` statements for C probably don't have much value to a Lisp programmer, so we'll ignore them, as does XDR. My XDR parser can recognize the `typedef` statements in the XDR file. It can convert them to Lisp data, but the macros (or whatever) which process that data & produce Lisp code can ignore the typedefs.

Maybe there will be a use for typedefs in the future.

4.2.6 Programs

Here is an XDR program declaration:

```
/* an XDR program declaration */
program Blackbook {
  version Vers2_0 {
    /* This version's interface differs significantly
     * from those of the previous versions. */
    int Count () = 1;
    Address FetchByName (string) = 2;
```

```
    Address FetchById (string) = 3;
    NameList ListBegin () = 10;
    NameList ListMore (string) = 12;
    void ListEnd () = 17;

    NameList Search (SearchCriteria) = 22;
  } = 3;

  version Vers1_1 {
    /* Like Vers1 but with a Count feature. */
    Address LookupByName (string) = 1;
    NameList GetList () = 2;
    int Count () = 2983;
  } = 2;

  version Vers1 {
    /* The original version */
    Address LookupByName (string) = 1;
    NameList GetList () = 2;
  } = 0;
} = 0x12345;
```

There are three version symbols: `Vers2_0`, `Vers1_1`, & `Vers1`. Their values are applicable only when considered in context with the program itself, Blackbook. Some other program could use the same version symbols but could associate them with different version numbers. For that matter, the same symbols might have entirely different meanings in some other, non-RPC context. So the values associated with the version symbols depend on the RPC program.

For that matter, the same is true for the symbols which identify the procedures.

The RPC-to-C compiler produces version symbols which begin with the program symbol. It produces procedure symbols which begin with the program symbol & the version number. For example, it would translate `Vers2_0` to `Blackbook_Vers2_0`. It would translate the `Count` procedure of Blackbook version 2.0 to a C function called `Count_3`, & it would translate the `Count` procedure of version 1.1 to `Count_2`.

My RPC-to-Lisp compiler could do that, but I'd rather it didn't.

Instead, I'd rather have a Program object that used the version symbols & procedure symbols to identify the client-side remote procedure proxy. Without knowing its implementation, you'd use it like this:

```
;;; Example of how to use a Program object
;;; derived from the example XDR program
;;; above.

;; Use a client object.  I haven't discussed
```

```
;; them yet, but just roll with it.
(with-rpc-client (clnt "other" blackbook vers2_0 "tcp")
  ;; Now CLNT is connected to the BLACKBOOK RPC
  ;; program on the computer called "other" via
  ;; TCP.

  ;; We can call procedures on the remote.
  (format t "~&There are ~D items in the blackbook."
          (rpc clnt 'count))

  ;; Here's an RPC with a string argument.
  ;; Assume that it returns an Address
  ;; structure that has already been defined in
  ;; Lisp.
  (format t "~&Mike's address is ~A."
          (rpc clnt 'fetchbyname "Mike"))

  ;; For convenience (if you can call it
  ;; convenient), we can make a remote procedure
  ;; call with the procedure number instead
  ;; of its symbol.
  (format t "~&Jill's address is ~A."
          (rpc clnt 2 "Jill"))

  ;; When we end the WITH-RPC-CLIENT, it'll
  ;; automagicaly end the client connection
  ;; in a well-behaved way.
  )
```

So a program could be a hash table bound to a symbol. The keys of the hash table could be (*version procedure*) pairs, & the values could be the XDR functions. The values could be lists. The first item in a list would be the XDR procedure to decode the return value. The remaining arguments could be XDR procedures to encode the procedure arguments. That would work for any number of arguments.

The version in the keys could be saved as both symbols & version numbers. Same goes for the procedure identities: both symbols & numbers.

How about type information? I guess the RPC program doesn't need type information at run-time, but it would be nice. It'd allow for introspection. Might be useful when debugging. *Might* allow for some cool functionality, though I can't imagine anything in particular at the moment.

I could put the type information directly in the RPC program object, or I could use RPC procedure objects which contained the type information as well as the XDR procedures. So the values in the program object's hash table would be procedure objects, & those procedure objects would contain run-time type information & the XDR procedures. Yup, not a bad idea. To discover what

procedures an RPC program offered, a Lisp client program could scan the RPC program object. The Lisp client might need to be careful to filter out duplicates, but it could work.

So my Lisp RPC library might contain an RPC procedure type like this:

```
(defstruct rpc-proc
  n                 ; procedure number
  sym               ; procedure symbol
  vers              ; version symbol ???
  prog              ; program symbol ???

  ;; OUT is a pair.  Its CAR is an XDR function
  ;; to decode the return value.  Its CDR is
  ;; the symbol whic defines the XDR type.
  ;; How does this work for arrays & such?
  out

  ;; INS is like OUT except that it's a list of
  ;; pairs, & it describes the procedure arguments.
  ins)
```

So Lisp code to convert the previous RPC program declaration example to Lisp, assuming the XDR types & functions were already defined, would look like this:

fixme: I'm too tired to construct this Lisp code for the moment. It will need to take each version into account, stuff all the procedure objects into the RPC program's hash table. Finally, it must bind the hash table to a symbol. (Or maybe that binding should happen first.)

Chapter 5

API

5.1 Common Functions & Other Entities

5.1.1 *make-network-stream*

defvar `*make-network-stream*` `nil`

`*Make-network-stream*` indirectly indicates how to create a stream to a port on a computer. It is bound to a function that does the work or to a symbol that names a global function that does the work. (Basically, Lizard uses it like this: "`(funcall *make-network-stream* ...)`".)

The application should initialize `*make-network-stream*` with a function that works for its Lisp implementation. Alternatively, the application program could avoid the functions which use `*make-network-stream*` & could create network streams directly & use them to initialize client structures.

The function must allow these arguments, in order:

1. *hostname* to designate a remote computer. It can be a string that gives the hostname or network address of a remote computer. Implementations might allow other types of host indicators, maybe network addresses in a list or some type of structure.

2. *port*, the port number for a TCP, UDP, or similar protocol. For other protocols, it might specify some other datum which is needed for a connection.

3. *protocol* indicates the protocol. It can be a string, such as "tcp", "TCP", or "udp". It is not case-sensitive. It may be a symbol whose name is a string that satisfies those other conditions. Implementations might allow other indicators for the protocol.

If the function can create a bi-directional stream that is connected in the indicated protocol to the indicated port on the indicated computer, it should return a 3-element list whose elements are, in order:

1. *the stream*,

2. *a function to write one octet* to the stream. That function accepts the same arguments that `write-byte` does. Depending on the type of the stream, this value in the list could be `#'write-byte`.

3. *a function to read one octet* from the stream. That function accepts the same arguments that `read-byte` does, & it might actually be `#'read-byte`.

5.1.2 client

structure
`client`

A `client` has these members:

strm A stream connected to the appropriate port on the apropriate remote computer. It must be open for both input & output.

hostname Hostname (or other host identifier) this client is connected to.

prognum Program number that this client is connected to.

versnum Version number this client is connected to.

timeout For this client, the number of seconds to wait for a reply after sending a remote procedure call. If this number of seconds expires without a reply, return an error instead of a true return value.

5.1.3 xdr-read-array

function
`xdr-read-array` *strm maxsize elsize elproc* &optional (*eof-error-p* t)

Reads an array of at most *maxsize* elements from the XDR stream. Creates a new Lisp array to hold them & stuffs the elements into that array.

If *eof-error-p* is true & there is any kind of I/O error on the stream, this function will raise an error. Otherswise (*eof-error-p* is `nil`), this function will return *strm* on error.

The C function `xdr_array` has *sizep* & *arrp* arguments, but this Lisp version doesn't need them. A Lisp program can obtain the length of the array by applying `length` to the array returned by `xdr-read-array`.

5.1.4 xdr-read-bool

function
`xdr-read-bool` *strm* &optional (*eof-error-p* t)

Reads a Boolean value from the stream & returns it as `t` or `nil`.

5.1.5 xdr-read-bytes

function
`xdr-read-bytes` *strm maxsize* &optional (*eof-error-p* t)

Reads at most *maxsize* bytes from the XDR stream. Stores them in a newly allocated Lisp vector & returns the vector.

5.2 Internal Functions & Other Entities

5.2.1 rpc

function
`rpc` *strm return-on-error-p prognum versnum procnum outproc inproc* &rest *args*

I don't like this. Looked like a good idea initially, but it will require too many arguments in practice. It needs arguments to determine whether it will return or raise a condition on a stream error. It needs a timeout argument. It probably needs others. Could I solve these problems if the first argument was a client object, not just a stream?

Makes a remote procedure call. Encodes the arguments (*args*) with *inproc*, sends the request to the specified procedure (in the program & version). Decodes the return value with *outproc* & returns the return value.

Chapter 6

Stream of Consciousness

6.1 5 August 2003

I'm writing this simply because I want to do some programming, but I'm in a coffee shop without a computer or descriptions of other things I could program. I know RPC/XDR well enough to do it off the top of my head.

RPC describes intefaces to remote services. In Lisp, we'd like to load the `*.rpc` file for a service & have the client functions defined automatically. Let's think about the API the `*.rpc` parser would use to define the client functions.

We'll need ot define RPC data types becauase they often are not equivalent to Lisp types (even via `defstruct`).

Let's see. XDR has `int`s, characters, strings, floating point, arrays, pointers, & references. Maybe built-in Lisp types would suffice for those. See Figure 6.1.

6.1.1 Questions about details of types

Most commonly, the Lisp programmer doesn't worry too much about integers versus floating point. XDR definitely distinguishes between the two. The Lizard XDR library needs to know whether it is sending or receiving an integer or

XDR type	**Lisp type**
int	number. How to handle out-of-range?
float	number. How to handle out-of-range?
char	character
string hline array	string array with type restrictions
union	Hmmm...
struct	struct ?

Figure 6.1: Some built-in XDR types & notes about how they might correspond to Lisp types.

floating point, but when sending an integer, should it quietly coerce its numeric argument to an integer, or should it raise an error if its numeric argument is not an integer? When sending a floating point number, should it quietly coerce its numeric argument to floating point, or should it raise an error if the argument is not already floating point?

XDR arrays are fully typed, in their length (which can be variable) & in the types of the elements they contain. Lisp arrays are more maleable. When reading an array, the Lizard XDR functions need to know the type they are reading. When writing an array, do they somehow rely on the declared types of the Lisp array argument, or do they try sending it & raise an error if the type can't be converted to the XDR array type? I guess that arrays are not like the integers. The Lizard XDR functions should marshal the Lisp array into the required XDR type. If that can't be done, maybe because the Lisp array contains elements that are not compatible with the XDR array's type, it's an error.

XDR arrays may have variable length or fixed length. Lisp array are the same, but Lisp programmers don't normally worry too much about whether an array is extensible. Should the Lizard functions, when marshalling a Lisp array into an XDR array of fixed length, require the Lisp array to be of fixed length? I think not. They should check the Lisp array's effective length, & raise an error if it is not the same as the XDR array's fixed length. The Lisp array's effective length is probably the fill pointer, or the array's actua length if it doesn't have a fill pointer. Hell, it's probably best to just use the Lisp `length` function, whether it uses the fill pointer or the actual length, whatever. Just use `length`. The fixed length of the XDR array should probably be an argument of the marshalling function.

When the Lizard functions marhsal a Lisp array into an XDR array of variable length, they should use the Lisp array's `length` or its fill pointer? I don't know.

Except for `struct`, maybe Lisp types will work. Lisp struct will work for XDR structures, too.

We must tell the client functions exactly what types to send & receive, but their Lisp-side APIs may be forgiving.

Given an RPC program number, version number, procedure number, procedure name, return type, & argument types, it'd be nice to do something like this:

`(defclient` *func* `(`*argtype*`)` *rettype prognum versnum funcnum*`)`

The user wouldn't need to know that the `defclient` expanded to

```
(defun FUNC (clnt arg)
  (client-call clnt PROGNUM VERSNUM FUNCNUM
               (xdr-func-for ARGTYPE) arg
               (xdr-func-for RETTYPE)))
```

Oh yeah, that'd be pretty nice for a first try.

Should the function go into a namespace? Should I follow the lead of XDR for C & use the same functions for input & output? I never liked those, butthey could help reduce code. Maybe CLOS could help.

```
(defclass xdr-stream () ...)
(defgeneric xdr-int (strm))
(defgeneric xdr-array (strm))

(defclass xdr-input-stream (xdr-stream) ...)
(defmethod xdr-int ((strm xdr-input-stream))
  ;; Read the bytes from STRM, assemble
  ;; into an integer, return the integer.
  )

(defclass xdr-output-stream ((strm xdr-stream)) ...)
(defclass xdr-int ((strm xdr-output-stream) n)
  ;; Write the bytes for N to the stream.
  )
```

Oh! See that the integer argument isn't needed for input, but it is for output. For C, the XDR frunctions use pointers, so the integer argument (a pointer to an integer) is useful for both input & output.

Not so in Lisp. For Lisp, it is better to follow the lead of other Lisp I/O functions, returning what is read, raising error on error unless specified otherwise by optional arguments. Might still use CLOS.

```
(defun xdr-write-int (strm n &optional (error-p t)
                      (error-val nil))
  ;; Send the bytes of N, return T.
  )
(defun xdr-read-int (strm &otional (error-p t)
                     (error-val nil))
  ;; Read bytes, assemble into an integer,
  ;; return the integer.
  )
```

When we define an XDR struct, we should get a Lisp structure & I/O functions. It should be possible to override the auto-generation of I/O functions, but ignore that for now.

We should be able to do something like this:

```
(defxdrstruct NAME
  (member0 type0)
  (member1 type1)
  ...)
```

That should give us a Lisp structure type *name*, as if it were created this way:

```
(defstruct name
  member0 member1 ...)
```

and two I/O functions, as if defined like this:

```
(defun xdr-write-name (strm val)
  (and (xdr-write-type0 strm (name-member0 val))
       (xdr-write-type1 strm (name-member1 val))
       ...))
(defun xdr-read-name (strm)
  ;; I can't remember actual make-WHATEVER
  ;; syntax for the function automatically
  ;; created by DEFSTRUCT WHATEVER.  So forgive
  ;; errors, please.
  (make-name
    :member0 (xdr-read-type0 strm)
    :member1 (xdr-read-type1 strm)
    ...))
```

6.2 6 August 2003

Experiments & research showed that I can't create a Lisp function at run-time without calling `eval`. Can't do it at all.

```
;; Idea was to type "(lambda (x y) (+ x y))"
;; for the READ.
;; Fails because READ is not a lambda expression:
((read) 1 2)

;; This:
(funcall (read) 1 2)
;; fails if I type "(lambda (x y) (+ x y))"
;; because that is a list, not a function.
;; Fails if I type "(function (lambda (x y)
;; (+ x y)))" because that, too, is a list,
;; not a function.  Also fails for "#'(lambda
;; (x y) (+ x y))" beacause that is a synonym
;; for the previous ("(function ...)") form.
```

This won't do: "`((lambda (x y) (+ x y)) 1 2)`". The lambda expression is hard-coded. I need dynamism.

There are two alternatives:

1. Completely dynamic data structures, interpreted by a virtual machine, or
2. use macros (or whatever) to define functions.

Since a common case will be to parse the `*.rpc` file at load time or to specify manually a service with `defwhatever` statements, I'll use item number 2. when (if) I need to dynamically load a service, I'll use `eval`.

The common high-level, convenient case is to load the service description from a `*.rpc` file, like this:

```
(defrpc-from-file *pmap* "pmap.rpc")
```

That will read the service description from the file, create the appropriate date & functions, & bind the object to symbol `*pmap*`. then you can use it like this:

```
(defvar clnt)
(setq clnt (clnt-create "somehost"
             (prognum *pmap*) (versnum *pmap*)
             "tcp"))
(format t "~&Port for ~A.~A is ~A."
  some-prognum some-versnum
  (rpc clnt 'getmap some-prognum some-versnum))
(clnt-destroy clnt)
```

Function `clnt-create` returns a new client object connected to the indicated program (& version) on the indicated host. When you have finished with the client object, dispose of it with `clnt-destroy`.[1]

Question: A `*.rpc` file may contain more than one RPC program description. Does that mean it describes a *service* & that a service is a collection of programs? Does `defrpc-from-file` create a service object, not just a program object? Or should it create multiple program objects?

Let's look at it from the other side, using the client library.

If I've loaded the `pmap` RPC program (not a service, a program), I guess I'd like to use it this way:

```
(with-rpc (clnt
            ("hostname" 'pmap pmap-versnum "tcp"))
  (format t "~&The RPC programs on that host are")
  (format t " ~A." (pmap-get-list clnt)))
```

Let's look at the API again.

6.2.1 clnt-create

`defun clnt-create (`*`service prog vers host proto`*`)`

Service is a service object containing one or more programs. *Prog* is an RPC program number, name as string, or name as symbol. *Vers* is a version number

[1] It would agree more with other Lisp functions to dispose of a client object with `clnt-close`, but this is a case in which it might be good to follow the naming convention of the RPC library for C.

or the symbol :LATEST. *Host* identifies a computer (hostname, IP address, whatever). *Proto* identifies a protocol; it's probably a string such as "tcp".

Returns a new client object that is connected to the program/version in the service on the host.

6.2.2 clnt-call

`defun clnt-call (clnt proc &rest args)`

Call *proc* on the other end of *clnt*. Marshalls *args*.

Returns two values: The actual return value (`nil` on error) & the result code (`t` for success or some other value indicating what type of error happened).

Always stuffs the result code into the client as well as returning it as the second value.

Maybe it would be better in Lisp if there was a keyword argument that could indicate whether to raise an error on error. By default, it might raise an error & be done with it, but if the programmer preferred, it could ignore the error & return, with the second return value indicating the error.

Function `clnt-call` gets type & marshalling functions from the program/version object to which *clnt* is connected.

6.2.3 callrpc

`defun callrpc (host service prog vers proc &rest in)`

This is a Lisp analog of the C RPC function of the same name. This function doesn't need *inproc* & *outproc* arguments because they are in the service. *Prog* is the program number or name (string or symbol). *Vers* is the version number or :LATEST. *Proc* is the procedure number or name (string or symbol).

Returns two values: actual return value & status. Should there be a keyword to indicate whether to raise an error on error?

```
(defun callrpc (host service prog vers proc &rest in)
  (with-clnt (clnt (service prog vers host "udp"))
    (apply #'clnt-call clnt (cons proc in))))
```

6.2.4 Out of API again

A service is a collection of types & RPC programs. Could we implement it as a list or two hash tables (one for types, one for programs)? Also, let's revisit the idea of dynamic descriptions...

6.3 Friday, 8 August 2003

Make hand-written functions to send & receive XDR primitives. They might look like these:

```
(defun xdr-read-int (strm &optional
                     return-on-error-p error-val)
  ...)
(defun xdr-write-int (strm n &optional
                      return-on-error-p errorval)
  ...)
(defun xdr-read-string (strm &optional
                        return-on-error-p error-val)
  ...)
```

That's just three. There will be many.

We can make a generic XDR encoder like this:

```
(defun xdr-super-write (strm val typeinfo service
                        &optional return-on-error-p
                        error-val)
  (cond ((functionp typeinfo) (funcall typeinfo strm val
                                       return-on-error-p
                                       error-val))
        ((or (stringp typeinfo) (symbolp typeinfo))
         (xdr-super-write strm val
                          (get-typeinfo typeinfo service)
                          service
                          return-on-error-p error-val))
        ((consp typeinfo)
         ;; This is important.  It's for composite types.
         ;; Also, this implementation assumes we raise a
         ;; condition on error.  A production implementation
         ;; must also be able to return on error.
         (mapc #'(lambda (typeinfo)
                   (xdr-super-write strm val typeinfo
                                    service))
               typeinfo))
        (t (error "Don't know how to deal with ~A"
                  typeinfo))))
```

That's the idea. For a value, we have `typeinfo`. It might be a function that is responsible for writing, in which case we call that function. Ultimately, we always call the XDR primitives. `Typeinfo` might be a string or symbol that names a type in the RPC service. In that case, get the typeinfo for the type & process it recursively. Finally, it might be a list. Recursively process each part.

A problem is handling I/O errors. The error-handling code is tedius. It might be nicer to write an entire, assembled block of data. Well, maybe not; maybe it wouldn't be that much simpler. Let's experiment with error-handling code. It's just the composite case that is tricky. What about using `return`? And we need keywords to select between return on error or signal.

Let's try `do` instead of return.

```
(cond ...
      ((consp typeinfo)
       (do ((is-error t ???)
            (x typeinfo (rest x)))
           ((or is-error (endp x))
            (if is-error
                error-val
              last-val)))))
```

No, that's crap. It's going nowhere. Probably just plain dead wrong. Try again.

```
(labels ((moo (typeinfo)
           (xdr-super-write strm val typeinfo
                            service
                            return-on-error-p
                            error-val)))
  (cond ...
        ((consp typeinfo)
         (do ((x typeinfo (rest x))
              (rc (moo (first typeinfo)) (moo (first x))))
             ((or (endpx) (equal rc error-val)) rc)))
        ...))
```

That's much better, & it's not so bad. It assumes that `error-val` has a value that cannot be returned on success. The code makes this assumption even if we raise a condition on error because it would exit early if a success value was equal to the value of `error-val`. I believe this is a safe assumption because the only value that is likely to conflict would be `nil`. You can send & receive lists in XDR, but they are intrusively linked lists, not Lisp lists, so it is unlikely that an XDR operation would send or receive a `nil` value. Wait: What about pointers, which XDR supports? Maybe the error value should always be `strm`. I think that might solve the problem, & anyway, I'll deal with it later.

I don't think this will work for composite types. You need a single function to write all parts of a composite, not functions to write specific parts of a composite.

Given a composite & functions for writing each part (by its type), we need way to access each part & send the part to the writer. Like this:

```
(defun xdr-write-composite (strm comp parts)
  (mapc #'(lambda (part)
            (funcal (part-writer part)
                    strm
                    (funcall (part-getter part) comp)))
        parts))
```

I think this (or something like it) will work. Instead of recursing, `xdr-super-write` calls `xdr-writer-composite` when the `typeinfo` is a list.

For hand-coded composites, the typeinfo can be a function that doesn't have anything to do with `xdr-write-composite`, so that will work.

A dynamic service loader could always use hash tables to implement composites. The part-getter is a parameterized closure that calls `gethash`. The writer must be `xdr-super-write` for nested composites? The arity doesn't work with the `xdr-write-composite` I have here. How about a closure that calls `xdr-write-composite`? Yes, that might work. Not sure, though. Let's try it.

```
;; Instance of a SHOE as it might be
;; implemented by the dynamic XDR service
;; loader.
#S(hashtable eql
   (color . "red")
   (size . 1))

;; The parts (a.k.a. typeinfo) for a SHOE
;; as it might be implemented by the
;; dynamic loader.
(list '(writer #'xdr-write-string)
      (list 'getter
            (let ((fld 'color))
              #'(lambda (shoe)
                  (gethash fld shoe)))))
```

6.4 Saturday, 9 August 2003

I've spent a lot of time thinking of how to make a dynamic solution in which the RPC service need not be known at compile time. I've had two realizations over night:

1. That idea distills to functions to read & write XDR types that are not known at compile-time.
2. Do I need completely dynamic types, anyway?

6.4.1 Dynamic XDR Functions

Because I know how to do the rest of the library at run-time, the idea of dynamic RPC service description comes to dynamicly encoded composite types & to dynamically generated XDR functions. Binding client-side Lisp function calls to server-side Lisp functions is entirely dynamic except for the marshalling functions & the data types.

I already know how to store composite types without knowing about them at compile-time. Just use a hash table (or an association list or other implementation of a dictionary). All that's left are the XDR functions that are known only at run-time. Maybe that realization can help clarify the problem.

I could implement the RPC system with XDR functions that must be known at compile-time. I know how to do that already. Then I can go back & create the XDR-encoding überfunctions so that the only XDR functions needed at compile-time are the primitives.

6.4.2 Do I Need Dynamic XDR Functions?

What is the need for the dynamic XDR functions, anyway? I'll need dynamic RPC services rarely. If I have an RPC service compiler (Lisp macro), then in the rare instances where I don't know the service name at compile-time, I can wrap the service comiler invocation in an `eval`, like this:

```
;; Normal use, I know the service name at compile-time.
(defvar *rwho* (compile-rpc "rwho.x"))

;; Sometimes, I don't know the service name until
;; run-time.  Here, the user enters the pathname
;; for the RPC service's *.x file.
(defvar *service*
        (progn (format t "~&RPC filename? ")
               (compile-rpc (read))))
```

6.4.3 Service Known at Compile-Time

This is a usage scenario. Or a use case. Whatever buzzword you believe is necessary to lend legitimacy to it.

Most commonly, service is known at compile-time. Programmer should be able to tell his Lisp about the service by parsing the `*.rpc` file for the service, like this:

```
;; Create the Lisp description of the port-mapper
;; service & bind it to symbol *PMAP*.  *PMAP*
;; is created as if by DEFVAR.  This should
;; definitely load the client-side description of
;; the service, but should it load the server-side,
;; too?  I think so.
(defvar *pmap* (compile-rpc "pmap.rpc"))
```

Should `compile-rpc` define XDR functions in the global namespace? For example, let's say the RPC file this composite type:

```
/* inside whatever.x */
/* maybe it defines all sorts of types, but
 * look at just this one example type here: */
struct dohicky {
    int n;
    double d;
```

```
    string<100> s;
};
```

When processing the definition of `struct dohicky`, should `compile-rpc` create an `xdr-write-dohicky` function & an `xdr-read-dohicky` function?

If so, then maybe it's not "compile rpc". Maybe it's "load rpc" because it might define types & functions, just like `load`ing a Lisp file might. I would use `load-rpc` like this:

```
;; There ain't much to its use, really.
(load-rpc "myservice.x")
```

That one Lisp form could define XDR functions, client calls, server stubs, & maybe some symbols bound to program numbers, version numbers, & procedure numbers. The potential for name collisions with other parts of a program is astounding, but maybe no worse than that of `load`. I suppose you could always do it in a package, like this:

```
;; Declare a package with the same name as the
;; service.  LOAD-RPC doesn't care about this,
;; but it might be a useful convention for the
;; programmer.
(defpackage myservice
  (:use "COMMON-LISP")
  ;; It's a good idea to export the public
  ;; symbols first, though we must know the
  ;; symbols from the RPC file.  Seems like
  ;; a reasonable assumption for cases in
  ;; which we know the RPC service at
  ;; compile-time.
  (:export "MYSERVICE-PROG"
           "MYSERVICE-VERSION"
           "NULL-PROC"
           "DOSUMIN"
           "DOSUMIN-SVC"
           "XDR-READ-THING"
           "XDR-WRITE-THING"))

(load-rpc "myservice.x")
```

6.4.4 Stream Errors

I've decided that these functions don't need to return on an I/O error. They should raise a condition on any I/O error.

The reason is that the standard Lisp I/O functions raise a condition on any I/O error except end-of-file. For end-of-file, they raise a condition by default, but you can tell them toreturn instead.

If I follow their example, the RPC & XDR functions should raise a condition on any error other than end-of-file. Except for EOF, that solves that.

For EOF, they could duplicate the possibility of returning instead of raising a condition, but for RPC, an end-of-file where you don't expect it is an error, deserving a condition or exception.

So these functions should raise a condition on any error. Since that's what the standard Lisp I/O functions do, these functions simply don't need to worry about I/O errors.

6.5 18 August 2003

(defxdrtype *type synonym*)

Declares *type* as a synonym for some other type, called *synonym*. Synonym is a symbol.

```
(defxdrtype type
  (mem0 type0)
  (mem1 type1)
  (mem2 type2)
  ...)
```

Declares *type* as a composite. Members are *mem0* ... Their type must be XDR types.

With the right arguments, `defxdrtype` could create the XDR procedures, too.

How about enumerations? Could `defxdrtype` figure it out, or do we need `defxdrenum`?

How about unions? Because they are always(?) part of a structure, the notation should be part of `defxdrtype`.

However it works, you can create all the XDR types & their I/O procedures.

6.5.1 Example

If XDR file says:

> enum ColorEnum (Red, Yellow, Green); struct Car Color color; double weight; int price; string serial; ;

then in Lisp, however it gets ther, I have:

- The symbols `red`, `yellow`, & `green`.
- Maybe some functions to test membership in the Color enumeration.
- The `car` structure type, as if declared like this: `(defstruct car color weight price serial)`.

- XDR functions for I/O on these types. The functions are named `xdr-read-color`, `xdr-write-color`, `xdr-read-car`, & `xdr-write-car`.

Then the XDR file creates this service:

```
program Dealer {
  version Latest {
    void ping () = 1;
    Car<1000> inventory () = 3;
    Car<1000> searchbyprice (int max) = 3;
  } = 1;
| = 123;
```

In Lisp, we get

```
(defconstant dealer 123 "program number")
(defconstant latest 1 "One of the potentially many
  versions.  Should probably be inside the program
  object.")
(defun ping (clnt) ...)
(defun inventory (clnt) ..)
(defun seacrchbyprice (max clnt) ...)
```

Problem: Version symbol should be in program context. Proecdure should be in (program version) context.

What if Program is some kind of data structure (such as hash table), containing version. (Program Version) is hash containing procedures. Too cumbersome to use? Efficiency?

Let's try it. Use the same previous example, Lisp conversion would be

```
(defun puthash (value key ht) (setf (gethash key ht) value))

(let ((ping #'(lambda ...))
      (inventory #'(lambda ...))
      (searchbyprice #'(lambda ...))
      (procs (make-hash-table))
      (vers (make-hash-table)))
  (puthash ping pint ht)
  (puthash inventory inventory ht)
  (puthash procs latest ht)
  (puthash procs 1 ht)

  (defvar Dealer vers))
```

The setup for Dealer is kind of involved. I could write some functions to do it, given a structured data description of hte program.

Given the hash table description of an RPC program & a client object, you can call a procedure like this:

```
(rpc dealer 'latest 'ping clnt)
```

That will lookup `latest` in the hash, then `ping`. From the lookups, it has a client function. It'll call the client function with just one arg, `clnt`. If the remote procedure needs args, where to put them in RPC's arg list? Before client? It could work, but it's unnecessary complexity at run-time because `clnt` is a required argument. Let's put `clnt` first. The `rpc` function then has definition like this:

```
(defun rpc (clnt program version proc &rest args)
  (apply (gethash (cons vers proc) program)
         (append args (list clnt))))
```

With that `append`, maybe clnt at the beginning is not more efficient. let's try `rpc` with `clnt` at the end again:

```
(defun rpc (program vers proc &rest args)
  (apply (gethahs (cons vers proc) program) args))
```

We need a Client object so we can bundle socket, timeout, & maybe other info together.

6.5.2 20 August 2003

Could make the RPC program be a CLOS object & let CLOS connect it with the right method. There would be no need for data within the program object.

It'd work like this:

```
/* XDR file */
program TheProg {
  version VersA {
    int Count () = 1;
    int Add (int, int) = 2;
  } = 3;
} = 456;
```

would translate to this Lisp:

```
;; The Program object.  Only identity matters.
(defvar TheProg (make-program ...))

;; Version? Damn.  Potential name collision.
(defvar VersA 3)

;; CLOS methods for RPC procedures
(defmethod count ((clnt Client) (prog eq TheProg)
                  (versnum eq 3))
  ...)
```

```
(defmethod Add ((clnt Client) (prog eq TheProg)
                (versnum eq 3) i j)
  ...)
```

heck, we don't even need the Program object. Can we use just the program number?

This won't work the way RPC normally does. We need to specify the client like program, & the version to call an RPC; that's reluctant. Could specify the stream, not the client, but then must specify the timeout. It could be keyword arg, but how many more?

But I like the fixed number of arguments; might help detect programmer errors.

The client is a connection to a particular `(program version)`. What if the client's type indicated that? Then you could do this:

```
(defmethod Count ((clnt TheProg-VersA)) ...)
(defmethod Add ((clnt TheProg-VersA) i j) ...)
```

I like that! From the programmer's point of view, it's minimal but complete. It's probably reasonably efficient because it lets CLOS do much of the work. I write less code beause I don't screw around with program data structures. Easy to debug because every remote procedure is a method.

Difficulties include:

1. Must declare all the generics. This could interfere with other RPC programs. Resolve it with namespaces?
2. Need a class for each `program version` combination. Three possibilities:
   (a) Combine the program & version symbol to forma classname.
   (b) Use a gensym for classname. Use a table to map `(program version)` to the gensym.
   (c) Some other approach I haven't imagined yet.

So I have two approaches ...

1. CLOS, or
2. Program object that can map version & procedures to the correct client proxy function.

Number 1 (CLOS) might require less code, might be more efficient at runtime, is more direct, might be easier to debug, might provide more type-safety.

Number 2 (home-grown Program object) pollutes the namespace less, is more data-driven.

There are more benefits potentially to number 1, but it puts so many symbols in the namespace, & I don't look forward to declaring the generics. let's go with number 2 (program object). As a solution as a whole, it is more encapsulated.

If a client is connected to a `(program vrsion)`, we can call a remote procedure like this:

```
(rpc clnt 'Count)
(rpc clnt 'Add 1 2)
```

Which isn't as terse as the CLOS solution, but it's not bad.

I could implement the RPC function like this:

```
(defun rpc (clnt proc &rest args)
  ;; Get the data about the procedure.
  (let ((proc (gethash (cons (client-vers clnt) proc)
                       (client-prog clnt))))
    ;; Encode & send the arguments.
    (apply (proc-encode proc)
           (append args (list (client-strm clnt))))
    ;; Read & decode, then return, the reply value.
    (funcall (proc-decode proc) (client-strm client))))
```

6.5.3 21 Augut 2003

What if I don't convert RPC service description to Lisp code? What if I convert it to data & allow some "primitives" (Lisp RPC functions I wrote by hand) to interpret the data?

Primitives include functions for builtin XDR types, a "call RPC" function for client, & a "request-to-Lisp" mapping function for the server (a server container?).

A special function to encode any XDR type, primitive or otherwise, is needed. Alsoa decoder. These are "generic wirte" & "generic read".

Store composites as Lisp hash tables. Store an entire RPC file – all its XDR data & RPC program declarations – in a single data structure. Make it a hash.

Keys can be:

1. data type name (a symbol), maybe more complex expressions, such as `(array 4 int)`,
2. a program's name (symbol). Values could be list of versions (symbols & numbers), oh, & the RPC prognum.
3. a `(version program version)` list identifying a version. Program & version may be symbols &/or numbers. Values may be information about the version: number, list of procedures.
4. a `(proc prog vers proc)` identifying a remote procedure. Values are the procedure number, argument types, & result types.

To make a remote call given a stream, program, version, args, & timeout:

1. Lookup prognum, versnum, procnum in the RPC datum.
2. Lookup types for the procedure.
3. Tell generic-write to encode the args to the stream.
4. Tell generic-read to decode the results & return them.

```
(defun rpc (stream timeout rpcdata
            program version procedure
            &rest args)
  ;; Write the RPC call header.
  (write-call strm rpcdata program version procedure)

  ;; Write the actual arguments.
  (mapc #'(lambda (typeinfo arg)
            (generic-write strm typeinfo arg))
        (argtypes program version procedure rpcdata)
        args)

  ;; Read the RPC result header.  It might indicate failure.
  (if (success-p (read-reply strm))
      ;; Success, so read & return the actual return value.
      (generic-read strm
                    (resulttype program version procedure
                                rpcdata))
    ;; The reply header indicated an error.
    (cerror "RPC error")))
```

Now I need `generic-read` & `generic-write`.

In the RPC datum, primitive XDR types are keys whose values are functions. For example:

1. (reader 'int rpcdata) ⇒ #'xdr:read-int
2. (reader 'double rpcdata) ⇒ #'xdr:read-double
3. (writer 'int rpcdata) ⇒ #'xdr:write-int

All of those reader functions have one argument: `strm`. All the writer functions have two arguments: `strm` & `value`.

The XDR function for arrays is a problem because it has a length argument. (Actually, length & maxlength & the XDR function to encode or decode its elements.) We can use a closure to make it look like the standard reader (1 argument) or standard writer (2 arguments).

```
(defun generic-read (strm type rpcdata)
  (if (functionp type)
      (funcall type strm)
    (progn
      (setq type (typeinfo type rpcdata))
      (cond ((eq (first type) 'builtin)
             (generic-write strm (second type) rpcdata))
            ((eq (first type) 'fixarray)
             (read-fixarray strm (second type)
               #'(lambda (strm)
                   (generic-read strm (third tye) rpcdata))))
            ((eq (first type) 'vararray)
             (read-vararray strm (second type)
               #'(lambda (strm)
                   (generic-read ...))))
            ((eq (first type) 'enum)
             (read-enum strm (second type)))
            ((eq (first type) 'struct)
             (make-hash-from-assoc
               (mapcar
                 #'(lambda (pair)
                     (cons (car pair)
                           (generic-read strm (cdr pair)
                                         rpcdata)))
                 '(something goes here but I cant
                   remember what))))
            ((eq (first type) 'typedef)
             (generic-read strm (second type) rpcdata))
            (t
             (cerror "This shouldn't happen.
                      Should it?"))))))
```

Rewrite that `cond`:

```
(let ((t2 (typeinfo type rpcdata)))
  (case (first t2)
    (typedef (generic-read strm (second t2) rpcdata))
    (builtin (funcall (second t2) strm))
    (fixarray
      (read-fixarray strm (second t2)
                     #'(lambda (strm)
                         (generic-read strm (third t2)
                                       rpcdata))))
    (vararray ...)
    (enum ...)
    (struct ...)))
```

Also must handle string. XDR has a `vector` type, which I've called "fixarray" here.

Here's `read-opaque`, possibly of use as the fundamental function for reading XDR data. It reads `count` octets from `strm`. It stores them in the array `arr` unless `arr` is `nil`, in which case it stores the octets into a newly created array of length `count`. Always returns the array into which it stored the octets (unless there is an error, in which case it probably doesn't return at all).

```
(let ((discard (make-array 3)))
  (defun read-opaque (strm count &optional arr)
    (unless arr
      (setq arr (make-array count)))
    (labels ((read-raw (strm count arr)
               (do ((j 0 (1+ j)))
                   ((>= j count) j)
                   (setf (aref arr i) (read-byte strm)))))
      ;; Read & store the opaque bytes.
      (read-raw strm count arr)

      ;; Read & discard the remaining bytes.  (XDR always
      ;; reads/writes blocks of 4 octets.)
      (read-raw strm (- 4 (mod count 4)) discard)

      ;; Return the array into which we read the octets.
      arr)))
```

I can use that function as the underlying I/O function for all other XDR functions, including the functions for numeric types.

```
(defun read-uint (strm)
  "Reads & returns a 32-bit, unsigned integer from the
XDR-encoded stream."
  (let ((ab4 (read-opaque strm 4)))     ; array of 4 octets
    (+ (* (aref ab4 3) (expt 2  0))
       (* (aref ab4 2) (expt 2  8))
       (* (aref ab4 1) (expt 2 16))
       (* (aref ab4 0) (expt 2 24)))))

(defun rad-char (strm)
  (aref (read-opaque strm 4) 0))
```

At least I think that'll work. Something like it will work.

Here's a first attempt at a functio nto read a string from an XDR-encoded stream. XDR strings are written in blocks of 4 octets. The final block is padded with zeros.

```
(defun read-string (strm max)
```

```
(let ((arr (make-array 1024 :element-type 'char
                            :resisable t))
      (buf (make-array 4 :element-type 'char))
      donep)
  (while (not donep)
    (read-opaque strm 4 buf))
  (do ((i 0))
      ((or (= i 4) (zerop (aref buf i)))
       (when (zerop (aref buf i))
         (setq donep t)))
      (push (aref buf i) arr)
      (decf max)
      (when (<= max 0)
        (error "string too long")))))
```

6.6 1 September 2003

Wrote & tested `xdr:read-opaque` & `xdr:write-opaque`. Turn out that, in clisp at least, you must create a stream for XDR with an `:element-type` of `'(unsigned-byte 8)`. (I sure hope that's portable.)

Wrote & tested `xdr:read-uint` & `xdr:write-uint`.

6.7 Sunday, 2004 November 28

6.7.1 Prior Work

There is a Lisp implementation of ONC RPC already. I found it at CMU Artificial Intelligence Repository[2]. The URL is http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/io/rpc/0.html.

I like the programmer's interface to the CMU Lisp RPC system, but I don't care for the implementation. The API resembles the one I have envisioned, which was cool because it showed that similar goals produce similar solutions.

I didn't care for the implementation because it seemed to rely on more static code than I want. I'd like my RPC implementation to allow for more dynamic service & data definitions.

Also, the CMU Lisp RPC implementation seemed to be cluttered with (platform-dependant) networking code. I'd like to isolate the networking & I/O code to keep the RPC logic simpler.

Also, the CMu Lisp RPC implementation uses lots of functions from an IL name-space. I don't know what that is; I presume it contains platform-dependent objects from a particular Lisp implementation. Seems like it will be easier to re-write than to figure out those platform-dependent functions.

[2] http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/0.html

I have used ONC RPC with the C programming language for years. For decades, in fact. I wrote my own implementation for the fun of it. The image I have of RPC architecture is pretty much relies on an XDR stream class. That may be a limitation in my thinking, but that's how I want to write my Lisp RPC system.

That being said, I'm fully aware that rewriting *always* looks easier than reusing, & that rewriting is *always* harder than you predict. So yes, I'm deluding myself by re-writing, but I'm deluding myself with conscious intent, so it's okay. *grin*

6.7.2 What about (unsigned-byte 32)?

The XDR block size is 32 bits. Wouldn't this be a great application for Common Lisp streams's ability to define an element type of "`(unsigned-byte 32)`"?

The answer is No.

The reason is that the XDR specification says that each block is big-endian, but Common Lisp does not define the order the external octets are encoded into an "`(unsigned-byte 32)`". The only platform-independant way to ensure that we read XDR blocks in a big-endian way is to treat each XDR block as four octets which we read & encode explicitly.

Appendix A

Other File Formats

- This document is available in multi-file HTML format at http://lisp-p.org/lizard/.
- This document is available in Pointless Document Format (PDF) at http://lisp-p.org/lizard/lizard.pdf.

Bibliography

[200] Floating point. Wikipedia. http://en.wikipedia.org/wiki/Floating%5Fpoint.

[20099] The ieee standard for floating point arithmetic. Pittsburgh Supercomputing Center, November 1999. http://www.psc.edu/general/software/packages/ieee/ieee.html.

[aDV] Tom Parker Dick Valent. http://www.scd.ucar.edu/zine/96/fall/articles/4.ieee.formats.html.

[Gol91] David Goldberg. *Numerical Computation Guide*, chapter What Every Computer Scientist Should Know About Floating-Point Arithmetic. 1991. http://docs.sun.com/source/806-3568/ncg%5Fgoldberg.html.

[Sri95a] R. Srinivasan. Rfc 1831: Rpc: Remote procedure call protocol specification version 2. *RFC Editor*, August 1995. ftp://ftp.rfc-editor.org/in-notes/rfc1831.txt.

[Sri95b] R. Srinivasan. Rfc 1832: Xdr: External data representation standard. *RFC Editor*, August 1995. ftp://ftp.rfc-editor.org/in-notes/rfc1832.txt.