

An ONC RPC Framework for Guile

for GNU Guile-RPC 0.3

Ludovic Courtès

Edition 0.3
26 August 2008

Copyright © 2007, 2008 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the author.

Table of Contents

An ONC RPC Framework for Guile	1
1 Introduction	3
2 Obtaining and Installing GNU Guile-RPC ...	5
3 Quick Start	7
3.0.1 Defining Data Types	7
3.0.2 Creating the Client	8
3.0.3 Creating the Server	9
4 API Reference	11
4.1 Usage Notes	11
4.2 Implementation of XDR	11
4.2.1 XDR Type Representations.....	11
4.2.2 XDR Standard Data Types.....	13
4.2.3 XDR Encoding and Decoding.....	15
4.3 Implementation of ONC RPC	15
4.3.1 Building an RPC Client	15
4.3.2 Building an RPC Server.....	16
4.3.3 ONC RPC Message Types.....	19
4.3.4 Record Marking Standard	20
4.4 Standard RPC Programs	21
4.4.1 The Portmapper Program	21
4.5 XDR/RPC Language Compiler.....	23
4.5.1 Parser	23
4.5.2 Code Generation Compiler Back-End.....	24
4.5.3 Run-Time Compiler Back-End.....	25
5 Stand-Alone Tools	27
5.1 Invoking <code>grpc-compile</code>	27
5.2 Invoking <code>grpc-rpcinfo</code>	28
5.3 Invoking <code>grpc-nfs-export</code>	28
6 References	31
Appendix A Portability Notes	33
Appendix B GNU Free Documentation License	35

Concept Index 43

Function Index 45

Variable Index 47

An ONC RPC Framework for Guile

This document describes GNU Guile-RPC version 0.3. It was last updated in August 2008.

1 Introduction

GNU Guile-RPC is a framework for distributed programming under GNU Guile. It is a pure Scheme implementation of the ONC RPC standard, i.e., the “Open Network Computing” Remote Procedure Call standard. ONC RPC is standardized by the Internet Engineering Task Force (IETF) as RFC 1831. It is based on the External Data Representation standard (XDR), known as RFC 4506 (see [Chapter 6 \[References\]](#), page 31).

Remote procedure calls allow programs on different, potentially remote machines to interact together. A *remote procedure call* is the invocation of the procedure of a program located on a remote host (the *RPC server*), as the name implies. Doing so requires the procedure arguments on the client-side to be encoded, or *marshalled*, i.e., converted to a representation suitable for transfer over the network. On the server-side, upon reception of the RPC, those arguments must be decoded or *unmarshalled*, i.e., converted back to a form that is directly understandable by the server program—for instance, data using Scheme data types, should the server program be written in Scheme. The value returned by the RPC server must be encoded and decoded similarly.

When using the ONC RPC protocol, the way data items are encoded is dictated by the XDR standard. This encoding has the advantage of being particularly compact, allowing for relatively low bandwidth requirements and fast implementations, especially compared to more verbose RPC protocols such as XML-RPC and SOAP.

The XDR encoding is not *self-describing*, i.e., it is impossible given an arbitrary XDR encoded sequence to determine the XDR type of the encoded data. This is different from [D-Bus](#), for example, which uses a compact and self-describing encoding. In practice, this is sufficient for a wide range of applications.

GNU Guile-RPC provides an easy access to the ONC RPC protocol for the Guile Scheme programmer. In particular, it allows standard Scheme types to be mapped to XDR data types, so that Scheme objects are easily encoded to or decoded from XDR.

Please send bug reports, comments and patches to the [Guile-RPC mailing list](#).

2 Obtaining and Installing GNU Guile-RPC

The GNU Guile-RPC web page is located at <http://www.gnu.org/software/guile-rpc/>. You can obtain copy of GNU Guile-RPC from <ftp://alpha.gnu.org/gnu/guile-rpc/>.

In order to use GNU Guile-RPC, all that is needed is **GNU Guile 1.8** along with the R6RS library package for Guile 1.8, known as **Guile-R6RS-Libs**¹.

Before version 1.8.3, Guile 1.8 did not contain an implementation of **SRFI-35**. Thus, if you are using one of these versions of Guile, you need to install **Guile-Lib**, which contains the relevant module.

GNU Guile-RPC comes with a compiler for the XDR/RPC interface description language (see [Section 5.1 \[Invoking grpc-compile\]](#), page 27). To use that, **Guile-Lib** version 0.1.7 or later is needed².

¹ The source code repository is located at <http://repo.or.cz/w/guile-r6rs-libs.git>. Hopefully, this package will eventually be integrated into core Guile.

² Guile-Lib 0.1.6 can be used but will yield inaccurate error reporting from the compiler in some cases.

3 Quick Start

This section illustrates how ONC RPC clients and servers can be implemented using Guile-RPC. ONC RPC defines a language to describe RPCs and the data types they depend on (see [Chapter 6 \[References\], page 31](#)). This language, which we refer to as the *XDR/RPC language* or simply *RPC language*, is essentially an *interface description language* (IDL). It resembles the C programming language and borrows C's type definition constructs and adds the `program` and `version` constructs that introduce RPC definitions.

Consider the following RPC definition, written using the XDR/RPC language:

```
struct result_t
{
    int          integer_part;
    unsigned int decimal_part;
};

program ARITHMETIC_PROGRAM
{
    version ARITHMETIC_VERSION
    {
        /* Return the integer part and the 1000th of the
           given double-precision floating point number. */
        result_t split_number (double) = 1;
    } = 0;
} = 80000;
```

It defines a simple RPC *interface* named `ARITHMETIC` which contains only one procedure called `split_number ()`. The interface itself has a *program number* that identifies it (here, 80000). Normally, program numbers below 20000000 (hexadecimal) are assigned by Sun Microsystems, Inc. and thus should not be used unless the number has been properly registered (see [Chapter 6 \[References\], page 31](#), for details). It also has a *version number* (here, 0) that is user-defined and should be increased when the interface changes (e.g., when procedures are added, modified, etc.). Finally, the procedure `split_number ()` has a procedure number (here, 1) that allows it to be distinguished from other procedures.

People vaguely familiar with the C programming language should have guessed by now that this simple interface defines a procedure that takes a double-precision floating-point number and returns a structure that contains two fields.

Client and server creation are two-step. Since the first step—data type definition—is the same for both, that leaves us with a total of three steps, described below. Nicely, each of these steps can be automated using the XDR/RPC language compiler (see [Section 5.1 \[Invoking grpc-compile\], page 27](#)).

More details about the XDR type definition as well as client and server creation are available in the API reference (see [Chapter 4 \[API Reference\], page 11](#)).

3.0.1 Defining Data Types

Before actually creating a client or server for this interface, one must define the types it uses. The simplest way to define one or several data types is to pipe their definition in XDR/RPC language through the compiler (see [Section 5.1 \[Invoking grpc-compile\], page 27](#)):

```
$ grpc-compile --xdr --constants < input.x > types.scm
```

Given the description from ‘input.x’ in RPC language, this command generates code that provides access to the constants and data types defined therein. The resulting Scheme code, ‘types.scm’, can then be loaded in other Scheme files (see [section “Loading” in *The GNU Guile Reference Manual*](#)).

In addition, code in ‘types.scm’ depends on Guile-RPC modules that it uses at run-time. Thus, it must first import the relevant modules:

```
(use-modules (rpc xdr)
             (rpc xdr types))
```

Then, the `result_t` type is defined (this is the code generated by the compiler but it can also be written “by hand”):

```
(define result-type
  (make-xdr-struct-type (list xdr-integer          ;; ‘integer_part’
                              xdr-unsigned-integer) ;; ‘decimal_part’))
```

3.0.2 Creating the Client

Producing a client to invoke `split_number ()` is as simple as this:

```
(use-modules (rpc rpc))

(define invoke-split-number
  (make-synchronous-rpc-call 80000 0      ;; program and version
                             1           ;; procedure number
                             xdr-double ;; argument type
                             result-type))
```

Again, this definition, minus the `use-modules` clause, can alternatively be generated by the compiler from the RPC description in XDR/RPC language (see [Section 5.1 \[Invoking `grpc-compile`\], page 27](#)):

```
$ grpc-compile --client < input.x > client.scm
```

Once this is done, invoking the procedure is as simple as this:

```
(invoke-split-number 3.14 #x7777 socket)
```

The first argument to `invoke-split-number` is the argument of `split_number ()`; the second argument is a transaction ID, i.e., an arbitrarily chosen number that identifies the remote procedure call; the third argument should be an output port, typically one bound to a connection to the RPC server:

```
(define socket (socket PF_INET SOCK_STREAM 0))
(connect socket AF_INET INADDR_LOOPBACK 6666)
```

This example creates an IPv4 connection to the local host on port 6666 (see [section “Network Sockets and Communication” in *The GNU Guile Reference Manual*](#)).

On success, `invoke-split-number` returns a two-element list where the first element corresponds to the `integer_part` field of the result and the second element correspond to the `decimal_part` field of the result, both represented as Scheme exact integers.

3.0.3 Creating the Server

Creating a TCP server for our RPC interface should be quite easy as well. We can re-use our previous type definitions (see [Chapter 3 \[Defining Data Types\]](#), page 7). Then, all we have to do is to create a definition for our program.

```
(use-modules (rpc rpc server))

(define (split-number-handler number)
  ;; Handle a 'split-number' request.
  (let* ((int (floor number))
         (dec (floor (* 1000 (- number int))))))
    (list (inexact->exact int)
          (inexact->exact dec))))

(define my-rpc-program
  ;; Define our RPC program.
  (let* ((proc (make-rpc-procedure 1 xdr-double result-type
                                  split-number-handler))
         (version (make-rpc-program-version 0 (list proc))))
    (make-rpc-program 80000 (list version))))
```

Alternatively, using the compiler allows the above definition of *my-rpc-program* to be automatically generated from the XDR/RPC definition file (see [Section 5.1 \[Invoking grpc-compile\]](#), page 27):

```
$ grpc-compile --server < input.x > server.scm
```

However, there is a slight difference compared to the above “hand-written” approach: ‘*server.scm*’ does not contain the actual definition of *my-rpc-program* since it does not know about your *split-number-handler* procedure. Instead, given the RPC/XDR definition given earlier, it contains a *make-ARITHMETIC-PROGRAM-server* procedure; this procedure can be passed a list associating RPC names to Scheme procedures, and returns the resulting RPC program object:

```
(define my-rpc-program
  (make-ARITHMETIC-PROGRAM-server
   ‘(("ARITHMETIC_VERSION" ;; specify the target version

     ;; list all supported RPCs for this version
     ("split_number" . ,split-number-handler))))
```

As can be seen, using the compiler-generated server stub, one doesn’t have to deal explicitly with program, version and RPC numbers, which clarifies the code.

Finally, we can make the server listen for incoming connections and handle client requests, using [section “Network Sockets and Communication”](#) in *The GNU Guile Reference Manual*.

```
;; Creating a listening TCP socket.
(define server-socket (socket PF_INET SOCK_STREAM 0))

;; Listen for connections on 127.0.0.1, port 6666.
(bind server-socket AF_INET INADDR_LOOPBACK 6666)
```

```
(listen server-socket 1024)

;; Go ahead and serve requests.
(run-stream-rpc-server (list (cons server-socket my-rpc-program))
  1000000 ;; a one-second timeout
  #f      ;; we don't care about closed connections
  (lambda () ;; our idle thunk
    (format #t "one second passed~%")))
```

And now we're all set: We have a working TCP client and server for this wonderful RPC interface! This would work similarly for other stream-oriented transports such as Unix-domain sockets: only the `socket` and `bind` calls need to be adapted.

4 API Reference

This section first details facilities available to manipulate XDR types. It then describes Scheme procedures that should be used to build ONC RPC clients and servers.

4.1 Usage Notes

Guile-RPC makes use of the [SRFI-34 exception mechanism](#) along with [SRFI-35 error conditions](#) to deal with the various protocol errors¹. Thus, users are expected to use these mechanisms to handle the error conditions described in the following sections. Hopefully, in most cases, error conditions raised by Guile-RPC code provide users with detailed information about the error.

4.2 Implementation of XDR

This section describes how XDR types are represented in Guile-RPC, as well as how one can encode Scheme values to XDR and decode XDR binary data to Scheme values.

4.2.1 XDR Type Representations

The XDR standard defines various basic data types and allows for the definition of compound data types (“structs”), fixed-size and variable-size arrays as well as “unions”. Fixed-size arrays and structs can actually be thought as the same type: Their size is known in advance and they are encoded as the succession of the data they contain. Thus, those types can be summarized as 4 great classes: “basic” types, variable-length arrays, structs and unions.

The (`rpc xdr`) module provides facilities to create Scheme objects representing XDR data types and to manipulate them. These Scheme objects, described below, are all immutable, i.e., they cannot be modified after creation.

`make-xdr-basic-type` *name size type-pred encoder decoder* [Scheme Procedure]
 [*vector-encoder* [*vector-decoder*]]

This returns an `<xdr-basic-type>` object describing a type whose encoding fits on *size* octets, and which is encoded/decoded using the *encoder* and *decoder* procedures. *type-pred* should be a predicate checking the validity of an input Scheme value for encoding into this type.

Optionally, *vector-encoder* and *vector-decoder* can be passed and should be procedures that efficiently encode/decode sequences of data items of this type (for instance, the vector decoder could use the `bytevector->int-list` procedure of the (`r6rs bytevector`) module to speed up decoding). The vector encoder is invoked as `(vector-encoder type value bv index)` while the vector decoder is invoked as `(vector-decoder type count port)`.

Users should normally not need to define new basic types since all the basic types defined by XDR are already available in (`rpc xdr types`) (see [Section 4.2.2 \[XDR Standard Data Types\]](#), page 13). Thus, we will not describe its use further.

¹ Guile 1.8 provides an implementation of the former in the (`srfi srfi-34`) module, while the latter is currently provided by the [guile-lib package](#).

make-xdr-struct-type *base-types* [Scheme Procedure]

Return a new XDR struct type made of a sequence of XDR data items whose types are listed in *base-types*.

Struct types encode from/decode to Scheme lists whose length is that of *base-types*.

make-xdr-vector-type *base-type max-element-count* [Scheme Procedure]

Return an object describing an XDR variable-length array of elements of types *base-type* (again, an XDR type object). If *max-element-count* is a number, then it describes the maximum number of items of type *base-type* that are allowed in actual arrays of this type. If *base-type* is **#f**, then arrays of this type may contain up to $2^{32} - 1$ items of type *base-type*.

Vector types are encoded from *generalized vectors* such as Scheme vectors, SRFI-4 vectors or strings (see [section “Generalized Vectors”](#) in *The GNU Guile Reference Manual*). By default, vector types decode to vectors, but any other kind of generalized vector can be used: it only needs to be specified as the *vector-decoder* argument of **make-xdr-basic-type** for the corresponding base type. Of course, SRFI-4 vectors, for example, may only be used when an XDR integer vector with a matching integer range is expected.

If *max-element-count* is specified and a vector to be encoded contains more than *max-element-count* elements, then an **&xdr-vector-size-exceeded-error** error condition is raised. Likewise, if XDR data to be decoded describes vectors larger than *max-element-count*, this error condition is raised.

make-xdr-union-type *discr-type discr/type-alist default-type* [Scheme Procedure]

Return a new XDR discriminated union type, using *discr-type* as the discriminant type (which must be a 32-bit basic type) and *discr/type-alist* to select the “arm” type depending on the discriminant value. If no suitable value is found in *discr/type-alist* and *default-type* is not **#f**, then default type is used as the arm type.

Union types encode from/decode to Scheme pairs whose **car** is the discriminant’s value and whose **cdr** is the actual union value.

xdr-union-arm-type *union discriminant* [Scheme Procedure]

Return the type that should be used for *union*’s arm given *discriminant* (a Scheme value).

Sometimes, one may want to define *recursive types*, i.e., types that refer to themselves. This is particularly useful to implement lists. For example, in XDR language, a list of integers can be defined as follows:

```
struct integer_list_t
{
  int x;
  integer_list_t *next;
};
```

This notation is a shortcut for the following structure:

```
struct integer_list_t
{
  int x;
```

```

union switch (bool opted)
{
  case TRUE:
    integer_list_t value;
  case FALSE:
    void;
} next;
};

```

The `integer_list_t` type references itself. Defining it using our API seems impossible at first: one cannot pass a self-reference to `make-xdr-struct-type` (since the object is not yet created!), and the self-reference cannot be added after the fact since objects returned by `make-xdr-struct-type` are immutable.

The API addresses this problem by allowing *thunks* (zero-argument procedures) to be used as types. Together with Scheme’s `letrec` recursive binding construct or a top-level `define` (see section “Binding constructs” in *Revised⁵ Report on the Algorithmic Language Scheme*), it makes it possible to create such recursive types:

```

(letrec ((integer-list
         (make-xdr-struct-type
          (list xdr-integer
               (make-xdr-union-type xdr-boolean
                                   '((TRUE . ,(lambda ()
                                                integer-list))
                                       (FALSE . ,xdr-void))
                                       #f))))))
  integer-list)

```

The trick here is that using the thunk effectively *defers* the evaluation of the self-reference².

It is often useful to know the size in octets it takes to encode a given value according to an XDR type. However, as we just saw, the size of some XDR types (variable-length arrays and unions) cannot be known in advance: The encoding size depends on the actual value to encode. The following procedure allow the computation of the size of the XDR representation of some value.

xdr-type-size *type value* [Scheme Procedure]
 Return the size (in octets) of *type* when applied to *value*. *type* must be an XDR type object returned by one of the above procedures, while *value* should be a Scheme value suitable for encoding with *type*.

The following section lists the standard XDR data types.

4.2.2 XDR Standard Data Types

All the basic data types defined by XDR are defined in the `(rpc xdr types)` module.

² This idea was inspired by Oleg Kiselyov’s description of *thunked parent pointers* in SXML, which may be found at <http://okmij.org/ftp/Scheme/parent-pointers.txt>.

xdr-integer [Scheme Variable]
xdr-unsigned-integer [Scheme Variable]
xdr-hyper-integer [Scheme Variable]
xdr-unsigned-hyper-integer [Scheme Variable]
 XDR's 32-bit and 64-bit signed and unsigned integer types. This type decodes to/encodes from Scheme exact numbers.

xdr-float [Scheme Variable]
xdr-double [Scheme Variable]
 32-bit and 64-bit IEEE-754 floating point numbers. This type decodes to/encodes from Scheme inexact numbers. Note that XDR also defines a “quadruple-precision floating point type” (i.e., 128-bit long) that is currently not available (FIXME).

xdr-void [Scheme Variable]
 The “void” type that yields zero bits. Any Scheme value is suitable as an input when encoding with this type. When decoding this type, the %void value (which may be compared via `eq?`) is returned.

XDR provides support for “enumerations”, similar to that found in C. An enumeration type maps symbols to integers and are actually encoded as 32-bit integers.

make-xdr-enumeration *name enum-alist* [Scheme Procedure]
 Return an enumeration type that obeys the symbol-integer mapping provided in *enum-alist* which should be a list of symbol-integer pairs. The returned type decodes to/encodes from Scheme symbols, as provided in *enum-alist*. Upon decoding/encoding of an enumeration, an `&xdr-enumeration-error` is raised if an incorrect value (i.e., one not listed in *enum-alist*) is found.

xdr-boolean [Scheme Variable]
 XDR's boolean type which is an enumeration. It encodes to/decodes from Scheme symbols `TRUE` and `FALSE`.

Several fixed-size and variable-size are predefined in the standard.

make-xdr-fixed-length-opaque-array *size* [Scheme Procedure]
 Return a fixed-length “opaque” array of *size* octets. An opaque array is simply a sequence of octets.
 The returned XDR type object is actually an `<xdr-struct-type>` object. Thus, it encodes from/decodes to Scheme lists of exact integers. Conversion to a Scheme string, if needed, is left to the user.

make-xdr-variable-length-opaque-array *limit* [Scheme Procedure]
 Return a variable-length opaque array. As for `make-xdr-vector-type` (see [Section 4.2.1 \[XDR Type Representations\], page 11](#)), *limit* can be either a number specifying the maximum number of elements that can be held by the created type, or `#f` in which case the variable-length array can hold up to $2^{32} - 1$ octets.
 The returned XDR type object is actually an `<xdr-vector-type>` object. Thus, it encodes from/decodes to Scheme vectors of exact integers.

`make-xdr-string` *limit* [Scheme Procedure]

This is a synonym of `make-xdr-variable-length-opaque-array` since XDR's string type actually represents ASCII strings, i.e., sequences of octets.

`xdr-variable-length-opaque-array` [Scheme Variable]

`xdr-string` [Scheme Variable]

These convenience variables contain the unlimited variable-length opaque array.

4.2.3 XDR Encoding and Decoding

The following primitives are exported by the `(rpc xdr)` module. They implement the encoding of Scheme values to XDR data types, and the decoding of binary XDR data to Scheme values. The exact mapping between XDR data types and Scheme data types has been discussed earlier.

`xdr-encode!` *bv index type value* [Scheme Procedure]

Encode *value* (a suitable Scheme value), using XDR type *type*, into bytevector *bv* at *index*. Return the index where encoding ended.

bv should be an R6RS bytevector large enough to hold the XDR representation of value according to *type*. To that end, users may rely on `xdr-type-size` (see [Section 4.2.1 \[XDR Type Representations\]](#), page 11).

Error conditions sub-classing `&xdr-error` may be raised during encoding. The exact exceptions that may be raised are type-dependent and have been discussed in the previous sections.

`xdr-decode` *type port* [Scheme Procedure]

Decode from *port* (a binary input port) a value of XDR type *type*. Return the decoded value.

Error conditions sub-classing `&xdr-error` may be raised during encoding. The exact exceptions that may be raised are type-dependent and have been discussed in the previous sections.

4.3 Implementation of ONC RPC

This section describes facilities available for the creation of ONC RPC clients and servers, as well as lower-level details about raw RPC messages.

4.3.1 Building an RPC Client

Basic building blocks for the creation of RPC clients are provided by the `(rpc rpc)` module. The easiest way to build an RPC client is through `make-synchronous-rpc-call`.

`make-synchronous-rpc-call` *program version procedure* [Scheme Procedure]

arg-type result-type

Return a procedure that may be applied to a list of arguments, transaction ID (any unsigned number representable on 32 bits), and I/O port, to make a synchronous RPC call to the remote procedure numbered *procedure* in *program*, version *version*. On success, the invocation result is eventually returned. Otherwise, an error condition is raised. *arg-type* and *result-type* should be XDR type objects (see [Section 4.2.1 \[XDR Type Representations\]](#), page 11).

Error conditions that may be raised include those related to XDR encoding and decoding (see [Section 4.2.3 \[XDR Encoding and Decoding\]](#), page 15), as well as RPC-specific error conditions inheriting from `&rpc-error` (i.e., conditions that pass the `rpc-error?` predicate). These are detailed in [Section 4.3.3 \[ONC RPC Message Types\]](#), page 19.

For an example, see [Chapter 3 \[Creating the Client\]](#), page 8.

It is also possible to create “one-way” calls, i.e., RPC calls that do not expect a reply (i.e., no return value, not even `void`). This is useful, for instance, to implement batched calls where clients do not wait for the server to reply (see [Chapter 6 \[References\]](#), page 31). Asynchronous calls can be implemented in terms of this, too.

`make-one-way-rpc-call` *program version procedure arg-type* [Scheme Procedure]
result-type

Similar to `make-synchronous-rpc-call`, except that the returned procedure does *not* wait for a reply.

4.3.2 Building an RPC Server

The (`rpc rpc server`) module provides helpful facilities for building an ONC RPC server. In particular, it provides tools to decode RPC call messages, as well as an event loop mechanisms that allows RPC calls to be automatically dispatched to the corresponding Scheme handlers.

`procedure-call-information` *call-msg* [Scheme Procedure]

Return an `<rpc-call>` object that denotes the procedure call requested in *call-msg* (the result of an `(xdr-decode rpc-message port)` operation). If *call-msg* is not an appropriate RPC call message, an error condition is raised.

The error condition raised may be either `onc-rpc-version-mismatch-error?` or `rpc-invalid-call-message-error?`.

The returned object can be queried using the `rpc-call-` procedures described below.

`rpc-call-xid` *call* [Scheme Procedure]

Return the transaction ID (“xid” for short) of *call*.

`rpc-call-program` *call* [Scheme Procedure]

`rpc-call-version` *call* [Scheme Procedure]

`rpc-call-procedure` *call* [Scheme Procedure]

Return the program, version or procedure number of *call*.

`rpc-call-credentials` *call* [Scheme Procedure]

`rpc-call-verifier` *call* [Scheme Procedure]

Return the credentials and verifier provided by the client for *call*. **FIXME:** As of version 0.3, this information is not usable.

The following procedures allow the description of RPC “programs”. Such descriptions can then be readily used to produced a full-blown RPC processing loop.

`make-rpc-program` *number versions* [Scheme Procedure]

Return a new object describing the RPC program identified by *number* and consisting of the versions listed in *versions*.

make-rpc-program-version *number procedures* [Scheme Procedure]
 Return a new object describing the RPC program version identified by *number* and consisting of the procedures listed in *procedures*.

make-rpc-procedure *number argument-xdr-type* [Scheme Procedure]
result-xdr-type handler [one-way?]
 Return a new object describing a procedure whose number is *number*, whose argument type is *argument-xdr-type* and whose result type is *result-xdr-type* (see [Section 4.2.1 \[XDR Type Representations\]](#), page 11). *handler* should be a procedure that will be invoked upon reception of an RPC call for that procedure.

If synchronous RPC processing is used, i.e., through **serve-one-stream-request**, then *handler* is passed the decoded argument and should return a result type that may be encoded as *result-xdr-type*. If asynchronous processing is used, i.e., through **serve-one-stream-request/asynchronous**, then *handler* is passed the decoded argument along with a continuation that must be invoked to actually return the result.

If *one-way?* is passed and is true, then the returned procedure is marked as “one-way” (see [Section 4.3.1 \[Building an RPC Client\]](#), page 15). For one-way procedures, **run-stream-rpc-server** and similar procedures ignores the return value of *handler* and don’t send any reply when procedure *number* is invoked.

Once a program, its versions and procedures have been defined, an RPC server for that program (and possibly others) can be run using the following procedures.

run-stream-rpc-server *sockets+rpc-programs timeout* [Scheme Procedure]
close-connection-proc idle-thunk
 Run a full-blown connection-oriented (i.e., SOCK_STREAM, be it PF_UNIX or PF_INET) RPC server for the given listening sockets and RPC programs. *sockets+rpc-programs* should be a list of listening socket-RPC program pairs (where “RPC programs” are objects as returned by **make-rpc-program**). *timeout* should be a number of microseconds that the loop should wait for input; when no input is available, *idle-thunk* is invoked, thus at most every *timeout* microseconds. If *close-connection-proc* is a procedure, it is called when a connection is being closed is passed the corresponding `<stream-connection>` object.

While an RPC server is running over a stream-oriented transport such as TCP using **run-stream-rpc-server**, its procedure handlers can get information about the current connection and client:

current-stream-connection [Scheme Procedure]
 This procedure returns a `<stream-connection>` object describing the current TCP connection (when within a **run-stream-rpc-server** invocation). This object can be queried with the procedures described below.

stream-connection? *obj* [Scheme Procedure]
 Return `#t` if *obj* is a `<stream-connection>` object.

stream-connection-port *connection* [Scheme Procedure]
 Return the I/O port (*not* the TCP port) for *connection*.

stream-connection-peer-address *connection* [Scheme Procedure]
 Return the IP address of the peer/client of *connection* (see section “Network Socket Address” in *The GNU Guile Reference Manual*).

stream-connection-rpc-program *connection* [Scheme Procedure]
 Return the RPC program object corresponding to *connection*.

For a complete RPC server example, Chapter 3 [Creating the Server], page 9.

The **run-stream-rpc-server** mechanism is limited to servers managing only RPC connections, and only over stream-oriented transports. Should your server need to handle other input sources, a more general event handling mechanism is available. This works by first creating a set of *I/O managers* and then passing **run-input-event-loop** a list of I/O manager-file descriptor pairs to actually handle I/O events.

make-i/o-manager *exception-handler read-handler* [Scheme Procedure]
 Return an I/O manager. When data is available for reading, *read-handler* will be called and passed a port to read from; when an exception occurs on a port, *exception-handler* is called and passed the failing port.

i/o-manager? *obj* [Scheme Procedure]
 Return #t if *obj* is an I/O manager.

i/o-manager-exception-handler *manager* [Scheme Procedure]

i/o-manager-read-handler *manager* [Scheme Procedure]

Return, respectively, the exception handler and the read handler of *manager*.

run-input-event-loop *fd+manager-list timeout idle-thunk* [Scheme Procedure]

Run an input event loop based on *fd+manager-list*, a list of pairs of input ports (or file descriptors) and I/O managers. I/O managers are invoked and passed the corresponding port when data becomes readable or when an exception occurs. I/O manager handlers can:

- return #f, in which case the port and I/O manager are removed from the list of watched ports;
- return a pair containing an input port and I/O manager, in which case this pair is added to the list of watched ports;
- return true, in which case the list of watched ports remains unchanged.

When *timeout* (a number of microseconds) is reached, *idle-thunk* is invoked. If timeout is #f, then an infinite timeout is assumed and *idle-thunk* is never run. The event loop returns when no watched port is left.

The event loop provided by **run-input-event-loop** should cover a wide range of applications. However, it will turn out to be insufficient in situations where tasks must be executed at specific times, and where the interval between consecutive executions varies over the program’s lifetime.

Finally, a lower-level mechanism is available to handle a single incoming RPC:

serve-one-stream-request *program port* [Scheme Procedure]
 Serve one RPC for *program*, reading the RPC from *port* (using the record-marking protocol) and writing the reply to *port*. If *port* is closed or the end-of-file was reached, an **&rpc-connection-lost-error** is raised.

serve-one-stream-request/asynchronous *program port* [Scheme Procedure]
 Same as **serve-one-stream-request** except that the RPC is to be handled in an asynchronous fashion.

Concretely, the procedure handler passed to **make-rpc-procedure** is called with two arguments instead of one: the first one is the actual procedure argument, and the second one is a one-argument procedure that must be invoked to return the procedure’s result—in other words, procedure call processing is decoupled from procedure call return using *continuation-passing style*.

4.3.3 ONC RPC Message Types

The (**rpc rpc types**) module provides a representation of the various XDR types defined in the standard to represent RPC messages (see [Chapter 6 \[References\]](#), page 31). We only describe the most important ones as well as procedures from the (**rpc rpc**) module that help use it.

rpc-message [Scheme Variable]
 This variable contains a XDR struct type representing all possible RPC messages—the **rpc_msg** struct type defined in RFC 1831. By “rpc message” we mean the header that is transmitted before the actual procedure argument to describe the procedure call being made.

Roughly, this header contains a transaction ID allowing clients to match call/reply pairs, plus information describing either the call or the reply being made. Calls essentially contain a program, version and procedure numbers. Replies, on the other hand, can be more complex since they can describe a large class of errors.

rpc-message-type [Scheme Variable]
 This variable is bound to an XDR enumeration. Its two possible values are **CALL** and **REPLY** (both represented in Scheme using symbols), denoting a procedure call and a reply to a procedure call, respectively.

make-rpc-message *xid type args ...* [Scheme Procedure]
 Return an **rpc-message** datum. *type* should be either **CALL** or **REPLY** (the two values of the **rpc-message-type** enumeration). The arguments *args* are message-type-specific. For example, a message denoting a procedure call to procedure number 5 of version 1 of program 77 can be created as follows:

```
(define my-call-msg
  (make-rpc-message #x123 ;; the transaction ID
                   'CALL ;; the message type
                   77 1 5))
```

It can then be encoded in the usual way (see [Section 4.2.3 \[XDR Encoding and Decoding\]](#), page 15):

```
(let* ((size (xdr-type-size rpc-message my-call-msg))
      (bv (make-bytevector size)))
  (xdr-encode! bv 0 rpc-message my-call-msg)
  ;;; ...
)
```

Likewise, a reply message denoting a successful RPC call can be produced as follows:

```
(make-rpc-message xid 'REPLY 'MSG_ACCEPTED 'SUCCESS)
```

It is worth noting that in practice, “messages” of type *rpc-message* are followed by additional data representing either the procedure call arguments (if the message is a **CALL** message) or the procedure return value (if the message is a **REPLY** message).

assert-successful-reply *rpc-msg xid* [Scheme Procedure]

Return true if *rpc-msg* (an RPC message as returned by a previous (**xdr-decode-rpc-message** *port*) call) is a valid reply for the invocation labeled with transaction ID *xid* indicating that it was accepted. If *xid* is **#t**, any reply transaction ID is accepted and it is returned (provided the rest of the message denotes an accepted message). On failure, an appropriate error condition is raised.

The error conditions that may be raised obey **rpc-error?** and **rpc-call-error?**. More precisely, error conditions include the following:

rpc-program-unavailable-error?

If *rpc-msg* denotes the fact that the program requested by the corresponding RPC call is not available.

rpc-program-mismatch-error?

If the corresponding RPC call requested a program version that is not available. The procedures **rpc-program-mismatch-error:low-version** and **rpc-program-mismatch-error:high-version** return, respectively, the lowest and highest version numbers supported by the remote server.

rpc-procedure-unavailable-error?

If the corresponding RPC call requested a procedure that is not available.

rpc-garbage-arguments-error?

If the remote server failed to decode the procedure arguments.

rpc-system-error?

If the remote server failed to allocate enough memory for argument decoding, for instance.

4.3.4 Record Marking Standard

The ONC RPC standard defines a *record-marking protocol* for stream-oriented transport layers such as TCP whereby (1) each RPC message is sent out as a single *record* and (2) where records may be split into several *fragments*. This allows implementations to “delimit one message from another in order to detect and possibly recover from protocol errors” (see Chapter 6 [References], page 31).

This protocol is implemented by the (**rpc rpc transports**) module. It is automatically used by the high-level client and server facilities, namely Section 4.3.1 [Building an RPC Client], page 15 and Section 4.3.2 [Building an RPC Server], page 16. The available facilities are described below.

send-rpc-record *port bv offset len* [Scheme Procedure]

Send the RPC message of *len* octets encoded at offset *offset* in *bv* (a bytevector) to *port*. This procedure sends the *len* octets of the record without fragmenting them.

make-rpc-record-sender *fragment-len* [Scheme Procedure]

This procedure is a generalization of **send-rpc-record**.

Return a procedure that sends data according to the record marking standard, chopping its input bytevector into fragments of size *fragment-len* octets.

rpc-record-marking-input-port *port* [Scheme Procedure]

Return a binary input port that proxies *port* in order to implement decoding of the record marking standard (RFC 1831, Section 10).

4.4 Standard RPC Programs

GNU Guile-RPC provides client-side and/or server-side of some commonly found ONC RPC program, which are described below. Currently, this is limited to the portmapper interface, but other interfaces (e.g., “mount”, NFSv2) may follow.

4.4.1 The Portmapper Program

The (`rpc rpc portmap`) module implements the widespread *portmapper* RPC program defined in RFC 1833 (see [Chapter 6 \[References\]](#), page 31). As the name suggests, the portmapper interface allows servers to be queried for the association between an RPC service and the port it is listening to. It also allows clients to query the list of services registered.

In practice, most machines run a system-wide `portmap` daemon on port 111 (TCP or UDP), and it is this server that is queried for information about locally hosted RPC programs. The `grpc-rpcinfo` program is a portmapper client that can be used to query a portmapper server (see [Section 5.2 \[Invoking grpc-rpcinfo\]](#), page 28)

Note that registering RPC programs with the portmapper is optional: it is basically a directory mechanism that allows servers to be located quite easily, but other existing mechanisms could be used for that purpose, e.g., decentralized *service discovery* (see [section “service discovery with DNS-SD in Guile-Avahi” in *Using Avahi in Guile Scheme Programs*](#)).

The module exports client-side procedures, as returned by **make-synchronous-rpc-call** (see [Section 4.3.1 \[Building an RPC Client\]](#), page 15), for the various portmapper procedures. They are listed below.

portmapper-null *arg xid port* [Scheme Procedure]

Invoke the `null` RPC over *port*, ignoring *arg*, and return `%void`.

portmapper-set *arg xid port* [Scheme Procedure]

Invoke the `set` RPC over *port* with argument *arg*. The invoked server should register the RPC program specified by *arg*, where *arg* must be an XDR struct (i.e., a Scheme list) containing these four elements: the RPC program number, its version number, its protocol and its port. The protocol number should be one of `IPPROTO_TCP` or `IPPROTO_UDP` (see [section “Network Sockets and Communication” in *The GNU Guile Reference Manual*](#)). An XDR boolean is returned, indicating whether the request successful.

portmapper-unset *arg xid port* [Scheme Procedure]

Invoke the `unset` RPC over *port* with argument *arg*. The invoked server should unregister the RPC program specified by *arg*, where *arg* must have the same form

as for `portmapper-set`. Again, an XDR boolean is returned, indicating whether the request was successful.

`portmapper-get-port` *arg xid port* [Scheme Procedure]

Invoke the `get-port` RPC over *port* with argument *arg*, which must have the same form as previously mentioned (except that its port number is ignored). The invoked server returns an unsigned integer indicating the port of that RPC program.

`portmapper-dump` *arg xid port* [Scheme Procedure]

Invoke the `dump` RPC over *port*, ignoring *arg*. The invoked server should return a list of 4-element lists describing the registered RPC programs. Those four element list are the same as for `portmapper-set` and `portmapper-get`, namely the RPC program number and version, its protocol and its port.

`portmapper-call-it` *arg xid port* [Scheme Procedure]

Invoke the `call-it` procedure over *port*. Quoting RFC 1833, this procedure “allows a client to call another remote procedure on the same machine without knowing the remote procedure’s port number”. Concretely, it makes the portmapper invoke over UDP the procedure of the program matching the description in *arg*, where *arg* is an XDR struct (i.e., a Scheme list) containing an RPC program and version number, a procedure number, and an opaque array denoting the procedure arguments (an `xdr-variable-length-opaque-array`).

On success, it returns a struct consisting of the port number of the matching program and an opaque array representing the RPC reply. On failure, it does not return. Therefore, this synchronous call version may be inappropriate. We recommend that you do not use it.

The `portmap` module also provides convenience functions to retrieve the symbolic name associated with common RPC program numbers. The association between program numbers and their name is usually stored in `/etc/rpc` on Unix systems and it can be parsed using the `read-rpc-service-list` procedure.

`read-rpc-service-list` *port* [Scheme Procedure]

Return a list of name-program pairs read from *port* (e.g., the `/etc/rpc` file), showing the connection between an RPC program human-readable name and its program number.

`lookup-rpc-service-name` *service-list program* [Scheme Procedure]

Lookup RPC program numbered *program* in *service-list* (a list as returned by `read-rpc-service-list`) and return its human-readable name.

`lookup-rpc-service-number` *service-list program* [Scheme Procedure]

Lookup RPC program named *program* in *service-list* (a list as returned by `read-rpc-service-list`) and return its RPC program number.

4.5 XDR/RPC Language Compiler

This section describes the compiler's *programming interface*. Most of the time, its command-line interface is all that is needed; it is described in [Section 5.1 \[Invoking `grpc-compile`\]](#), page 27. This section is intended for users who need more flexibility than is provided by `grpc-compile`.

The compiler consists of a lexer, a parser and two *compiler back-ends*. The lexer separates input data into valid XDR/RPC language tokens; the parser then validates the input syntax and produces an abstract syntax tree of the input. Finally, the back-ends are responsible for actually “compiling” the input into something usable by the programmer. The back-end used by the `grpc-compile` command is the *code generation back-end*. In addition, an experimental *run-time compiler back-end* is available, making it possible to compile dynamically definitions in the XDR/RPC language *at run-time*; this technology paves the way for a wide range of crazy distributed applications, the programmer's imagination being the only real limitation³.

4.5.1 Parser

The parser is available under the (`rpc compiler parser`) module. The main procedure, `rpc-language->sexp`, reads XDR/RPC language descriptions and returns the abstract syntax tree (AST) in the form of an S-expression. The AST can be shown using the `--intermediate` option of the `grpc-compile` command-line tool (see [Section 5.1 \[Invoking `grpc-compile`\]](#), page 27). Below is an illustration of the mapping between the XDR/RPC language and the S-exp representation.

```

const SIZE = 10;
struct foo
{
  int x;
  enum { NO = 0, YES = 1 } y;
  float z[SIZE];
};
... yields:
(define-constant "SIZE" 10)
(define-type
  "foo"
  (struct
    ("x" "int")
    ("y" (enum ("NO" 0) ("YES" 1)))
    ("z" (fixed-length-array "float" "SIZE"))))

```

`rpc-language->sexp port` [Scheme Procedure]

Read a specification written in the XDR Language from *port* and return the corresponding sexp-based representation. This procedure can raise a `&compiler-error` exception (see below).

The behavior of the parser can be controlled using the **parser-options** parameter object:

³ Finding useful applications leveraging the flexibility offered by the run-time compiler back-end is left as an exercise to the reader.

parser-options [Scheme Variable]

This [SRFI-39 parameter object](#) must be a list of symbols or the empty list. Each symbol describes an option. For instance, `allow-unsigned` instructs the parser to recognize `unsigned` as if it were `unsigned int` (see [Section 5.1 \[Invoking `grpc-compile`\]](#), page 27).

Source location information is attached to the S-expressions returned by `rpc-language->sexp`. It can be queried using the procedures below. Note that not-only top-level S-expressions (such as `define-type` or `define-constant` expressions) can be queried, but also sub-expressions, e.g., the `enum` S-expression above.

`sexp-location sexp` [Scheme Procedure]

Return the source location associated with `sexp` or `#f` if no source location information is available.

`location-line loc` [Scheme Procedure]

`location-column loc` [Scheme Procedure]

`location-file loc` [Scheme Procedure]

Return the line number, column number or file name from location `loc`, an object returned by `sexp-location`.

In case of parse errors or other compiler errors, a `&compiler-error` error condition (or an instance of a sub-type thereof) may be raise.

`&compiler-error` [Scheme Variable]

The “compiler error” error condition type.

`compiler-error? c` [Scheme Procedure]

Return `#t` if `c` is a compiler error.

`compiler-error:location c` [Scheme Procedure]

Return the source location information associated with `c`, or `#f` if that information is not available.

4.5.2 Code Generation Compiler Back-End

The code generation back-end is provided by the `(rpc compiler)` module. Given an XDR/RPC description, it returns a list of S-expressions, each of which is a top-level Scheme expression implementing an element of the input description. These expressions are meant to be dumped to a Scheme file; this is what the command-line interface of the compiler does (see [Section 5.1 \[Invoking `grpc-compile`\]](#), page 27).

Here is an example XDR/RPC description and the resulting client code, as obtained, e.g., with `grpc-compile --xdr --constants --client`:

```
const max = 010;

struct foo
{
  int    x;
  float  y<max>;
};
```

```
=>

(define max 8)
(define foo
  (make-xdr-struct-type
    (list xdr-integer
          (make-xdr-vector-type xdr-float max))))
```

As can be seen here, the generated code uses the run-time support routines described earlier (see [Section 4.2 \[Implementation of XDR\]](#), page 11); an optimization would consist in generating specialized code that does not depend on the run-time support, but it is not implemented yet.

This front-end consists of two procedures:

```
rpc-language->scheme-client input type-defs?           [Scheme Procedure]
  constant-defs?
rpc-language->scheme-server input type-defs?         [Scheme Procedure]
  constant-defs?
```

These procedures return a list of top-level Scheme expressions implementing *input* for an RPC client or, respectively, a server.

input can be either an input port, a string, or an AST as returned by `rpc-language->sexp` (see [Section 4.5.1 \[Parser\]](#), page 23). If *type-defs?* is `#t`, then type definition code is produced; if *constant-defs?* is `#t`, then constant definition code is produced.

Both procedures can raise error conditions having a sub-type of `&compiler-error`.

4.5.3 Run-Time Compiler Back-End

The run-time compiler back-end is also provided by the `(rpc compiler)` module. It reads XDR/RPC definitions and returns data structures readily usable to deal with the XDR data types or RPC programs described, *at run-time*. Actually, as of version 0.3, it does not have an API to deal with RPC programs, only with XDR data types.

```
rpc-language->xdr-types input                         [Scheme Procedure]
  Read XDR type definitions from input and return an alist; element of the returned
  alist is a pair whose car is a string naming an XDR data type and whose cdr is an
  XDR data type object (see Section 4.2.1 \[XDR Type Representations\], page 11). input
  can be either an input port, a string, or an AST as returned by rpc-language->sexp
  (see Section 4.5.1 \[Parser\], page 23).
```

This procedure can raise error conditions having a sub-type of `&compiler-error`.

Here is an example of two procedures that, given XDR type definitions, decode (respectively encode) an object of that type:

```
(use-modules (rpc compiler)
             (rpc xdr)
             (r6rs bytevector)
             (r6rs io ports))
```

```

(define (decode-data type-defs type-name port)
  ;; Read binary data from PORT as an object of type
  ;; TYPE-NAME whose definition is given in TYPE-DEFS.
  (let* ((types (rpc-language->xdr-types type-defs))
         (type (cdr (assoc type-name types))))
    (xdr-decode type port)))

(define (encode-data type-defs type-name object)
  ;; Encode OBJECT as XDR data type named TYPE-NAME from
  ;; the XDR type definitions in TYPE-DEFS.
  (let* ((types (rpc-language->xdr-types type-defs))
         (type (cdr (assoc type-name types)))
         (size (xdr-type-size type object))
         (bv (make-bytevector size)))
    (xdr-encode! bv 0 type object)
    (open-bytevector-input-port bv)))

```

These procedures can then be used as follows:

```

(let ((type-defs (string-append "typedef hyper chbouib<>;"
                                "struct foo { "
                                "  int x; float y; chbouib z;"
                                "};")))
  (type-name "foo")
  (object '(1 2.0 #(3 4 5))))
(equal? (decode-data type-defs type-name
                    (encode-data type-defs type-name
                                object))
        object))

```

=>

#t

Note that in this example *type-defs* contains two type definitions, which is why the *type-name* argument is absolutely needed.

5 Stand-Alone Tools

GNU Guile-RPC comes with stand-alone tools that can be used from the command-line.

5.1 Invoking `grpc-compile`

The `grpc-compile` command provides a simple command-line interface to the XDR/RPC language compiler (see [Section 4.5 \[Compiler\]](#), page 23). It reads a RPC definitions written in the XDR/RPC language on the standard input and, depending on the options, write Scheme code containing client, server, data type or constant definitions on the standard output.

`--help` Print a summary of the command-line options and exit.

`--version`
Print the version number of GNU Guile-RPC and exit.

`--xdr`
`-x` Compile XDR type definitions.

`--constants`
`-C` Compile XDR constant definitions.

`--client`
`-c` Compile client RPC stubs.

`--server`
`-s` Compile server RPC stubs.

`--strict`
`-S`

Use strict XDR standard compliance per [Chapter 6 \[References\]](#), page 31. By default, the compiler recognizes extensions implemented by Sun Microsystems, Inc., and also available in the GNU C Library's `rpcgen`. These extensions include:

- support for % line comments; these are actually treated as special directives by `rpcgen` but they are simply ignored by `grpc-compile`;
- support for the `char` type, equivalent to `int`;
- support for the `unsigned` type, equivalent to `unsigned int`;
- the ability to use `struct` in type specifiers.

Also note that some XDR/RPC definition files (`.x` files) originally designed to be used in C programs with `rpcgen` include C preprocessor directives. Unlike `rpcgen`, which automatically invokes `cpp`, such input files need to be piped through `cpp -P` before being fed to `grpc-compile`.

`--intermediate`
Output the intermediate form produced by the parser (see [Section 4.5.1 \[Parser\]](#), page 23).

Code generation options can be combined. For instance, the command line below writes data type and constant definitions as well as client stubs in a single file:

```
$ grpc-compile --xdr --constants --client < input.x > client-stubs.scm
```

The various pieces of generated code can also be stored in separate files. The following example shows how to create one file containing constant and type definitions, another one containing client stubs, and a third one containing server stubs. Since the two last files depend on the first one, care must be taken to load them beforehand.

```
$ grpc-compile --xdr --constants < input.x > types+constants.scm
$ echo '(load "types+constants.scm")' > client-stubs.scm
$ grpc-compile --client < input.x >> client-stubs.scm
$ echo '(load "types+constants.scm")' > server-stubs.scm
$ grpc-compile --server < input.x >> server-stubs.scm
```

In the future, there may be additional `--use-module` and `--define-module` options to make it easier to use Guile's module system in generated code.

5.2 Invoking `grpc-rpcinfo`

This program is equivalent to the `rpcinfo` program available on most Unix systems and notably provided by the GNU C Library. In is a client of the portmapper RPC program (see [Section 4.4.1 \[The Portmapper Program\], page 21](#)). Among the options supported by `rpcinfo`, only a few of them are supported at this moment:

```
'--help'   Print a summary of the command-line options and exit.
'--version'
            Print the version number of GNU Guile-RPC and exit.
'--dump'
'-p'       Query the portmapper and list the registered RPC services.
'--delete'
'-d'       Unregister the RPC program with the given RPC program and version numbers
            from the portmapper.
```

Note that the host where the portmapper lives can be specified as the last argument to `grpc-rpcinfo`:

```
# Query the portmapper at host 'klimt'.

$ grpc-rpcinfo -p klimt
program vers  proto  port  name
100000  2      tcp    111   portmapper
100000  2      udp    111   portmapper
$ grpc-rpcinfo -d 100000 2 klimt
ERROR: 'portmapper-unset' failed FALSE
```

5.3 Invoking `grpc-nfs-export`

Guile-RPC comes with an example NFS (Network File System) server, provided by the `grpc-nfs-export` command. More precisely, it implements NFS version 2, i.e., the `NFS_PROGRAM` RPC program version 2 along with the `MOUNTPROG` program version 1, which are closely related (see [Chapter 6 \[References\], page 31](#)). It is a TCP server.

Enough technical details. The important thing about `grpc-nfs-export` is this: although it's of little use in one's everyday life, this NFS server is nothing less than life-changing. It's different from any file system you've seen before. It's the ultimate debugging aid for any good Guile hacker.

The “file hierarchy” served by `grpc-nfs-export` is—guess what?—Guile's module hierarchy! In other words, when mounting the file system exported by `grpc-nfs-export`, the available files are *bindings*, while directories represent *modules* (see [section “The Guile module system”](#) in *The GNU Guile Reference Manual*). The module hierarchy can also be browsed from the REPL using Guile's `nested-ref` procedure. Here's a sample session:

```
$ ./grpc-nfs-export &

$ sudo mount -t nfs -o nfsvers=2,tcp,port=2049 localhost: /nfs/

$ ls /nfs/%app/modules/
guile/  guile-rpc/  guile-user/  ice-9/  r6rs/  rpc/  srfi/

$ ls /nfs/%app/modules/rpc/rpc/portmap/%module-public-interface/
lookup-rpc-service-name    %portmapper-port
lookup-rpc-service-number  %portmapper-program-number
portmapper-call-it        portmapper-set
portmapper-dump            portmapper-unset
portmapper-get-port        %portmapper-version-number
portmapper-null            read-rpc-service-list

$ cat /nfs/%app/modules/rpc/xdr/xdr-decode
#<procedure xdr-decode (type port)>

$ cat /nfs/%app/modules/rpc/xdr/%xdr-endianness
big
```

Here is the option reference:

```
'--help'    Print a summary of the command-line options and exit.
'--version'
            Print the version number of GNU Guile-RPC and exit.
'--nfs-port=port'
'-p port'   Listen for NFS connections on port (default: 2049).
'--mount-port=mount-port'
'-P mount-port'
            Listen for mount connections on mount-port (default: 6666).
'--debug'   Produce debugging messages.
```

In addition, `grpc-nfs-export` can be passed the name of a Scheme source file, in which case it will load that file in a separate thread while still serving NFS and `mount` requests. This allows the program's global variables to be monitored via the NFS mount.

As of version 0.3, this toy server exhibits poor performance, notably when running `ls` (which translates into a few `readdir` and many `lookup` RPCs, the latter being costly) in

directories containing a lot of files. This is probably partly due to the use of TCP, and partly due to other inefficiencies that we hope to fix soon.

6 References

- RFC 1831 R. Srinivasan et al., “[RPC: Remote Procedure Call Protocol Specification Version 2](#)”, August 1995.
- RFC 4506 M. Eisler et al., “[XDR: External Data Representation Standard](#)”, May 2006.
- RFC 1833 R. Srinivasan et al., “[Binding Protocols for ONC RPC Version 2](#)”, August 2005.
- RFC 1094 B. Nowicki, “[NFS: Network File System Protocol Specification](#)”, March 1989.

Appendix A Portability Notes

This appendix is about Guile-RPC’s portability. Of course, Guile-RPC can be ported to any OS/architecture Guile runs on. What this section deals with is portability among Scheme implementations.

Although implemented on top of GNU Guile, Guile-RPC uses mostly portable APIs such as **SRFIs**. Thus, it should be relatively easy to port to other Scheme implementations or to systems like **Snow**. Below are a few notes on portability, listing APIs and tools Guile-RPC depends on.

- **Modules**. Guile-RPC uses Guile’s hierarchical modules system (see [section “The Guile module system” in *The GNU Guile Reference Manual*](#)). Nevertheless, many Scheme systems have a similar module system, e.g., Snow and Bigloo, and also R6RS. Thus, it should be easy to translate the `define-module` and `use-module` clauses to some other Scheme.
- **R6RS APIs**. While Guile-RPC is *not* R6RS code, it employs two R6RS APIs: bytevectors and related binary I/O primitives. These are provided by the `(r6rs bytevector)` and `(r6rs io ports)` modules of Guile-R6RS-Libs; the official R6RS name of these modules is the same with `rnrs` in lieu of `r6rs`. MzScheme, Larceny, Ikarus (among others) provide these APIs.
- **Generalized vectors**. The `(rpc xdr)` module uses Guile’s generalized vectors API (see [section “Generalized Vectors” in *The GNU Guile Reference Manual*](#)). This allows applications to use regular vectors, SRFI-4 homogeneous vectors, arrays, etc., to represent XDR variable-length arrays (see [Section 4.2.1 \[XDR Type Representations\], page 11](#)). On Scheme implementations that do not support generalized vectors, regular vectors can be used instead.
- **Pattern matching**. The `(rpc compiler)` module uses [Andrew K. Wright’s pattern matcher](#), known as `(ice-9 match)` in Guile. This pattern matcher is portable and available in many Scheme implementations; alternative, compatible pattern matchers are also available sometimes, e.g., in MzScheme.
- **LALR parsing**. The `(rpc compiler parser)` module uses [Dominique Boucher’s LALR parser generator](#), known as `(text parse-lalr)` in [Guile-Lib](#). This package is available on most Scheme implementations and as a “snowball”.
- **Lexer**. The `(rpc compiler lexer)` module was automatically generated using [Danny Dubé’s SILex](#), a portable lexer generator.
- **Testing**. The test suite uses the [SRFI-64 API for test suites](#), whose reference implementation runs on most Schemes.
- **Networking**. Guile’s own networking primitives are only used in the `(rpc rpc server)`, which contains a server event loop. This part of the module would need porting to the target system, but it would be quite easy to isolate the few features it depends on.

Portability patches can be posted to the [Guile-RPC mailing list](#) where they will be warmly welcomed!

Appendix B GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none. The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called

an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts. A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

A

abstract syntax tree 23

B

batched calls 16, 17

C

C preprocessor 27
 compiler back-ends 23
 compiler errors 24
 compiler invocation 27

D

distributed programming 3

E

event loop 16

G

`grpc-compile` 27
`grpc-nfs-export` 28
`grpc-rpcinfo` 28

I

I/O manager 18
 IDL 7
 interface description language 7

M

marshalling 3

N

Network File System 28
 NFS 28

O

one-way calls 16, 17

P

parser 23
 portability 33
`portmap` daemon 21
 portmapper 21

R

R6RS 33
 record-marking protocol 20
 recursive types 12
`rpc_msg` struct type 19
`rpcgen` 27
`rpcinfo` 28

S

self-referencing types 12
 service discovery 21
 Sun XDR/RPC language extensions 27

T

transaction ID (`xid`) 8, 16, 20

X

XDR type objects 11
 XDR/RPC language 7

Function Index

A

assert-successful-reply 20

C

compiler-error:location 24
 compiler-error? 24
 current-stream-connection 17

I

i/o-manager-exception-handler 18
 i/o-manager-read-handler 18
 i/o-manager? 18

L

location-column 24
 location-file 24
 location-line 24
 lookup-rpc-service-name 22
 lookup-rpc-service-number 22

M

make-i/o-manager 18
 make-one-way-rpc-call 16
 make-rpc-message 19
 make-rpc-procedure 17
 make-rpc-program 16
 make-rpc-program-version 17
 make-rpc-record-sender 21
 make-synchronous-rpc-call 15
 make-xdr-basic-type 11
 make-xdr-enumeration 14
 make-xdr-fixed-length-opaque-array 14
 make-xdr-string 15
 make-xdr-struct-type 12
 make-xdr-union-type 12
 make-xdr-variable-length-opaque-array 14
 make-xdr-vector-type 12

P

portmapper-call-it 22
 portmapper-dump 22
 portmapper-get-port 22

portmapper-null 21
 portmapper-set 21
 portmapper-unset 21
 procedure-call-information 16

R

read-rpc-service-list 22
 rpc-call-credentials 16
 rpc-call-error? 20
 rpc-call-procedure 16
 rpc-call-program 16
 rpc-call-verifier 16
 rpc-call-version 16
 rpc-call-xid 16
 rpc-error? 20
 rpc-garbage-arguments-error? 20
 rpc-language->scheme-client 25
 rpc-language->scheme-server 25
 rpc-language->sexp 23
 rpc-language->xdr-types 25
 rpc-procedure-unavailable-error? 20
 rpc-program-mismatch-error? 20
 rpc-program-unavailable-error? 20
 rpc-record-marking-input-port 21
 rpc-system-error? 20
 run-input-event-loop 18
 run-stream-rpc-server 17

S

send-rpc-record 20
 serve-one-stream-request 18
 serve-one-stream-request/asynchronous 19
 sexp-location 24
 stream-connection-peer-address 18
 stream-connection-port 17
 stream-connection-rpc-program 18
 stream-connection? 17

X

xdr-boolean 14
 xdr-decode 15
 xdr-encode! 15
 xdr-type-size 13
 xdr-union-arm-type 12

Variable Index

%

`%void` 14

&

`&compiler-error` 24

*

`*parser-options*` 24

R

`rpc-message` 19

`rpc-message-type` 19

S

`SOCK_STREAM` 17

X

`xdr-double` 14

`xdr-float` 14

`xdr-hyper-integer` 14

`xdr-integer` 14

`xdr-string` 15

`xdr-unsigned-hyper-integer` 14

`xdr-unsigned-integer` 14

`xdr-variable-length-opaque-array` 15

`xdr-void` 14

