RPC Extensions Developer's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–3576–10
February 2002

011115@2870

Contents

Preface

The *RPC Extensions Developer's Guide* describes additions made to Sun's RPC (remote procedure call) library in Solaris™ 8 update 7.

Who Should Use This Book

This book is intended for the use of application developers using remote procedure calls in a Solaris™ development environment, and includes detailed information on:

- One-way messaging
- Non-blocking I/O
- Client connection closure callback
- User file descriptor callbacks

Use of this guide assumes basic competence in programming, a working familiarity with the C programming language, and a working familiarity with the UNIX® operating system. Previous experience in network programming is helpful, but is not required to use this manual.

For more information on distributed services, see the *ONC+ Developer's Guide*.

How This Book Is Organized

Chapter 1, discusses the additions made to the Sun RPC library in Solaris™ 8 update 7.

Related Books

For more information on developing with RPC, and distributed systems in general, see the *ONC+ Developer's Guide*.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Extensions to The Sun RPC Library

New features have been added to the Sun RPC library which are integrated into the standard Solaris™ 9 product.

New and altered man pages are available to describe the functionality added to the Sun RPC library.

The additions to the Sun RPC library are:

New Features

The new features added to the Sun RPC library and the advantage of these features are as follows:

- One-way messaging—To reduce the time a client thread waits before continuing processing.
- Non-blocking I/O—To enable a client to send requests without being blocked.
- Client connection closure callback—To allow a server to detect client disconnection and to take corrective action.
- Callback user file descriptor—To extend the RPC server to handle non-RPC descriptors.

Enhancement of The Sun RPC Library

In previous versions of the Sun RPC library, most requests were sent by two-way messaging. In two-way messaging, the client thread waits until it gets an answer from the server before continuing processing. If the client thread does not receive a reply from the server within a certain period of time, a time-out occurs. This client thread cannot send a second request until the first request is executed or until a time-out occurs. This messaging method is illustrated in Figure 1–1.

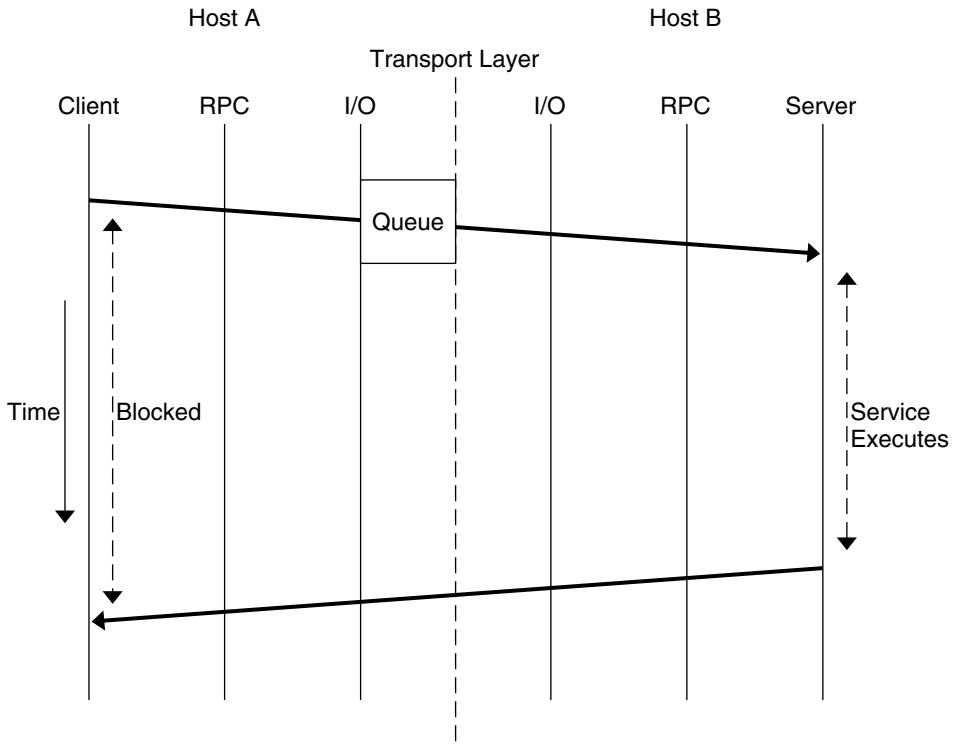

FIGURE 1–1 Two-Way Messaging

Note – Previous versions of the Sun RPC library contain a second method of messaging called batching. In this method, client requests are held in a queue until a group of requests can be processed at the same time. This is a form of one-way messaging. See "The Programmer's Interface to RPC" in *ONC+ Developer's Guide* for further details.

The new features of the Sun RPC library enhance the library's performance as follows:

1. One-way messaging, where when a client sends a request to a server, the client does not wait for a reply from the server and is free to continue processing when the transport layer accepts the request. If you choose to send a message by one-way messaging, rather than two-way messaging, you gain processing time. See Figure 1–2 for an illustration of one-way messaging.

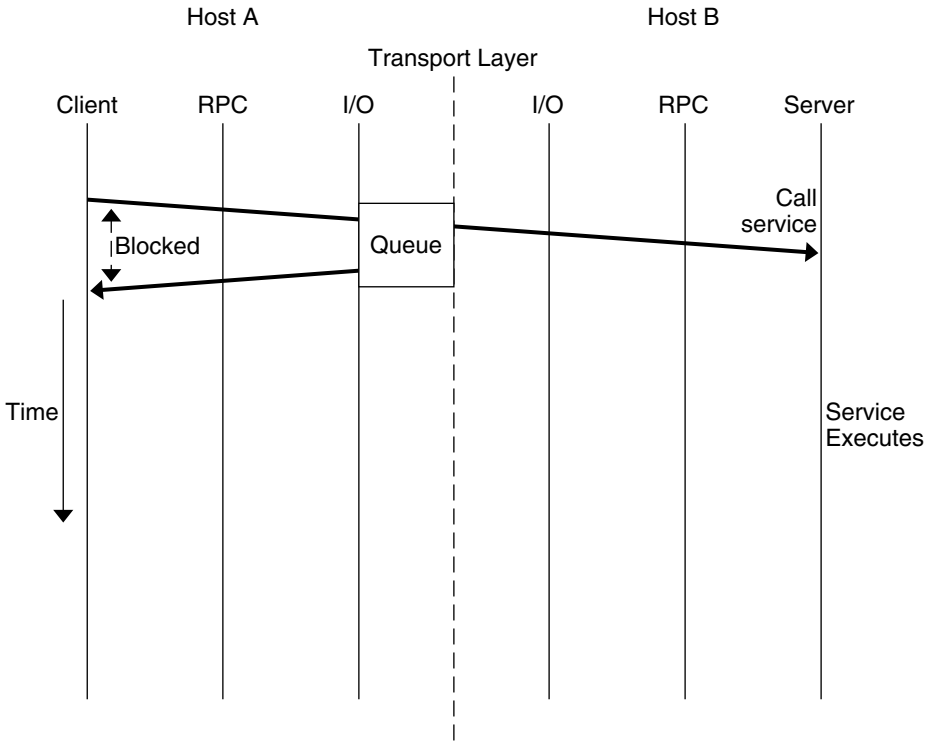

FIGURE 1–2 One-Way Messaging

2. Non-blocking I/O, where there is an additional buffer between the client and the transport layer. Figure 1–3 describes the case where you choose the non-blocking argument of the I/O mode and the transport layer queue is full. In this situation,

requests cannot go into the transport layer queue and are placed in the buffer. The client does not wait for the transport layer to accept the request but is free to continue processing when the request is accepted by the buffer. By using non-blocking I/O you gain further processing time as compared to two-way and one-way messaging. The client can send requests in succession without being blocked from processing.

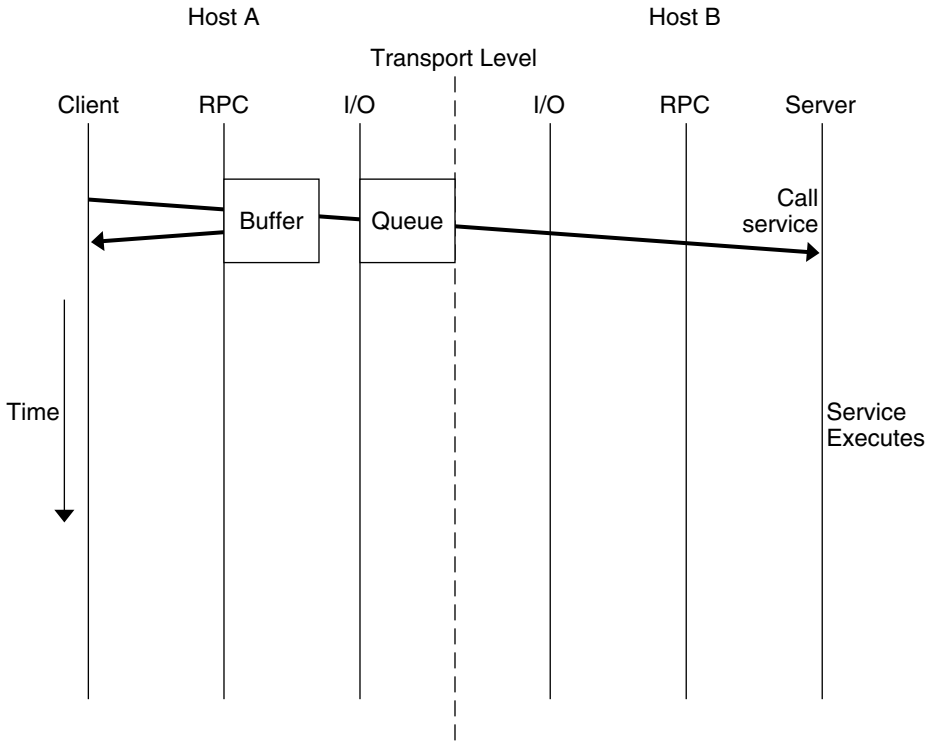

FIGURE 1–3 Non-Blocking Messaging

3. Client connection closure callback, where, for connection-oriented transport the server can detect that the client has disconnected, and the server can take the necessary action to recover from transport errors. Transport errors occur when a request arrives at the server, or when the server is waiting for a request and the connection is closed.
4. User file descriptor callbacks, which enable an RPC server to receive non-RPC requests as well as RPC requests. With previous versions of the Sun RPC library you can use a server to receive both RPC calls and non-RPC file descriptors only if you write your own server loop, or use a separate thread to contact the socket API.

   With user file descriptor callbacks, RPC servers handle client requests through a run loop using `svc_run()`. See the `svc_run`(3NSL) man page for further

information. The server enters the run loop at the start of the server connection, and exits the run loop at the end of the connection. When you make an RPC call but do not have user file descriptor callbacks you are blocked from receiving non-RPC requests.

One-Way Messaging

In one-way messaging the client thread sends a request containing a message to the server. The client thread does not wait for a reply from the server and is free to continue processing when the request has been accepted by the transport layer. The request is not always sent immediately to the server by the transport layer, but waits in a queue until the transport layer sends it. The server executes the request received by processing the message contained in the request.

After the transport layer accepts a request, the client is not notified of failures in transmission and does not receive a receipt from the server from the request. For example, if the server refuses the request due to an authentication problem, the client is not notified of this problem. If the transport layer does not accept the request, the sending operation returns an immediate error to the client.

Note – If you need to check that the server is functioning correctly, you can send a two-way request to the server. Such a request can determine whether the server is still available and if it has received the one-way requests sent by the client.

For one-way messaging, the `clnt_send()` function, has been added to the Sun RPC library, and the `oneway` attribute has been added to the RPC grammar.

`clnt_send()`

In previous versions of the Sun RPC library, you use the `clnt_call()` function to send a remote procedure call. With the extended one-way messaging service, the `clnt_send()` function sends one-way remote procedure calls.

When the client calls `clnt_send()`, the client sends a request to the server and continues processing. When the request arrives at the server, the server calls a dispatch routine to process the incoming request.

Like `clnt_call()`, the `clnt_send()` function uses a client handle to access a service. See the `clnt_send`(3NSL) and `clnt_call`(3NSL) man pages for further information.

If you don't provide the correct version number to `clnt_create()`, `clnt_call()` fails. In the same circumstances, `clnt_send()` does not report the failure, as the server does not return a status.

The `oneway` Attribute

To use one-way messaging, add the `oneway` keyword to the XDR definition of a server function. When you use the `oneway` keyword, the stubs generated by `rpcgen` use `clnt_send()`. You can either:

- Use a simplified interface as outlined in "Introduction to TI-RPC" in *ONC+ Developer's Guide*. The stubs used by the simplified interface must call `clnt_send()`.
- Call the `clnt_send()` function directly as described in the `clnt_send`(3NSL) man page.

For one-way messaging, use version 1.1 of the `rpcgen` command.

When declaring the `oneway` keyword, follow the RPC language specification using the following syntax:

```
"oneway" function-ident "(" type-ident-list ")" "=" value;
```

See "RPC Protocol and Language Specification" in *ONC+ Developer's Guide* for details on RPC language specifications.

When you declare the `oneway` attribute for an operation, no result is created on the server side and no message is returned to the client.

Information on the `oneway` attribute must be added to the RPC Language Definition Table in "The RPC Language Specification" in *ONC+ Developer's Guide* as shown in Table 1–1:

TABLE 1–1 RPC Language Definitions

Term	Definition
procedure	`type-ident procedure-ident (type-ident) = value` `oneway procedure-ident (type-ident) = value`

EXAMPLE 1–1 One-way call using a simple counter service

This example illustrates how to use a one-way procedure on a simple counter service. In this counter service the `ADD()` function is the only function available. Each remote call sends an integer and this integer is added to a global counter managed by the server. For this service, you must declare the `oneway` attribute in the RPC language definition, as described in Table 1–1.

EXAMPLE 1–1 One-way call using a simple counter service *(Continued)*

In this example, you generate stubs using the `-M`, `-N` and `-C` `rpcgen` options. These options ensure that the stubs are multithread safe, accept multiple input parameters and that generated headers are ANSI C++ compatible. Use these `rpcgen` options even if the client and server applications are mono-threaded as the semantic to pass arguments is clearer and adding threads in applications is easier since the stubs do not change.

1. Write the service description in the `counter.x` file.

```
   /* counter.x: Remote counter protocol */
program COUNTERPROG {
     version COUNTERVERS {
       oneway ADD(int) = 1;
     } = 1;
} = 0x20000001;
```

   The service has a program number, `(COUNTERPROG) 0x200000001`, and a version number, `(COUNTERVERS) 1`.

2. Call `rpcgen` on the `counter.x` file.

```
rpcgen -M -N -C counter.x
```

   This generates client and server stubs, `counter.h`, `counter_clnt.c` and `counter_svc.c`.

3. As shown in the `server.c` file, write the service handler for the server side and the `counterprog_1_freeresult()` function used to free memory areas allocated to the handler. The RPC library calls this function when the server sends a reply to the client.

```
      #include <stdio.h>
#include "counter.h"

int counter = 0;

       bool_t
add_1_svc(int number, struct svc_req *rqstp)
{
      bool_t retval = TRUE;

       counter = counter + number;

       return retval;
}
      int
counterprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result, caddr_t
                         result)
{
      (void) xdr_free(xdr_result, result);

       /*
          * Insert additional freeing code here, if needed
```

EXAMPLE 1–1 One-way call using a simple counter service *(Continued)*

```
           */

        return TRUE;
}
```

Build the server by compiling and linking the service handler to the `counter_svc.c` stub. This stub contains information on the initialization and handling of TI-RPC.

4. Write the client application, `client.c`.

```
      #include <stdio.h>
#include "counter.h"

main(int argc, char *argv[])
{
      CLIENT *clnt;
      enum clnt_stat result;
      char *server;
      int number;

       if(argc !=3) {
           fprintf(stderr, "usage: %s server_name number\n", argv[0];
              exit(1);
           }
      server = argv[1];
      number = atoi(argv[2]);

       /*
          * Create client handle
          */
      clnt = clnt_create(server, COUNTERPROG, COUNTERVERS, "tcp");

       if(clnt == (CLIENT *)NULL) {
         /*
          * Couldn't establish connection
          */
               clnt_pcreateerror(server);
           exit(1);
      }

       result = add_1(number, clnt);
      if (result !=RPC_SUCCESS) {
           clnt_perror(clnt, "call failed");
      }

       clnt_destroy(clnt);
      exit(0);
}
```

The `add_1()` client function is the `counter_clnt.c` stub generated for the remote function.

EXAMPLE 1–1 One-way call using a simple counter service *(Continued)*

To build the client, compile and link the client main and the `counter_clnt.c`.

5. Launch the server that you built.

```
% ./server
```

6. Invoke the service in another shell.

```
% ./client servername 23
```

23 is the number being added to the global counter.

Non-Blocking I/O

Non-blocking I/O avoids the client being blocked while waiting for a request to be accepted by the transport layer during one-way messaging for connection-oriented protocols.

For connection-oriented protocols, there is a limit to the amount of data that can be put in a network protocol queue. The limit depends on the transport protocols used. When a client sending a request reaches the data limit, this client is blocked from processing until its request has entered the queue. You cannot determine how long a message will wait before being added to the queue.

In non-blocking I/O, when the transport queue is full, there is an additional buffer available between the client and the transport layer. As requests not accepted by the transport queue can be stored in this buffer, the client is not blocked. The client is free to continue processing as soon as it has put the request in the buffer. The client does not wait until the request is put in the queue and does not receive information on the status of the request after the buffer accepts the request.

Using Non-Blocking I/O

To use non-blocking I/O, configure a client handle using the `CLSET_IO_MODE` *rpciomode_t** option of the `clnt_control()` function with the `RPC_CL_NONBLOCKING` argument. See the `clnt_control`(3NSL) man page for further information.

When the transport queue is full, the buffer is used. The buffer continues to be used until two criteria are fulfilled:

- The buffer is empty.

- The queue can immediately accept a request.

Then requests go directly to the transport queue until the queue is full. The default size of the buffer is 16 kilobytes.

Note that the buffer is not emptied automatically. You must flush the buffer when it contains data.

When you chose the `RPC_CL_NONBLOCKING` argument of `CLSET_IO_MODE`, you have a choice of flush modes. You can specify either the `RPC_CLBESTEFFORT_FLUSH` or `RPC_CL_BLOCKING_FLUSH` argument to `CLSET_FLUSH_MODE`. You can also empty the buffer by sending a synchronous call, such as `clnt_call()`. See the `clnt_control`(3NSL) man page for further information.

If the buffer is full, an `RPC_CANTSTORE` error is returned to the client and the request is not sent. The client must re-send the message later. You can find out or change the size of the buffer by using the `CLSET_CONNMAXREC` and `CLGET_CONNMAXREC` commands. To determine the size of all pending request stored in the buffer, use the `CLGET_CURRENT_REC_SIZE` command. For further information on these commands see the `clnt_control`(3NSL) man page.

The server does not confirm whether the request is received or processed. After a request enters a buffer, you can use `clnt_control()` to obtain information on the status of the request.

EXAMPLE 1–2 Using a simple counter with non-blocking I/O

The `client.c` file given in Example 1–1 has been modified to demonstrate how to use the non-blocking I/O mode. In this new file, `client_nonblo.c`, the I/O mode is specified as non-blocking with the RPC_CL_NONBLOCKING argument, the flush mode is chosen to be blocking by use of the `RPC_CL_BLOCKING_FLUSH`. The I/O mode and flush mode are invoked with `CLSET_IO_MODE`. When an error occurs, `RPC_CANT_STORE` is returned to the client and the program tries to flush the buffer. To choose a different method of flush, consult the `clnt_control`(3NSL) man page.

```
#include <stdio.h>
#include "counter.h"

main(int argc, char *argv[])
{
    CLIENT* clnt;
    enum clnt_stat result;
    char *server;
    int number;
    bool_t bres;
        /*
         * Choose the I/O mode and flush method to be used.
         * The non-blocking I/O mode and blocking flush are
         * chosen in this example.
         */
    int mode = RPC_CL_NONBLOCKING;
```

EXAMPLE 1–2 Using a simple counter with non-blocking I/O *(Continued)*

```
int flushMode = RPC_CL_BLOCKING_FLUSH;

if (argc != 3) {
          fprintf(stderr, "usage: %s server_name number\n", argv[0]);
          exit(1);
}
server = argv[1];
number = atoi(argv[2]);

clnt= clnt_create(server, COUNTERPROG, COUNTERVERS, "tcp");
if (clnt == (CLIENT*) NULL) {
          clnt_pcreateerror(server);
          exit(1);
}

    /*
  * Use clnt_control to set the I/O mode.
  * The non-blocking I/O mode is
  * chosen for this example.
  */
bres = clnt_control(clnt, CLSET_IO_MODE, (char*)&mode);
if (bres)
    /*
      * Set flush mode to blocking
      */
    bres = clnt_control(clnt, CLSET_FLUSH_MODE, (char*)&flushMode);

if (!bres) {
    clnt_perror(clnt, "clnt_control");
    exit(1);
}
     /*
   * Call the RPC services.
   */
result = add_1(number, clnt);

switch (result) {
case RPC_SUCCESS:
    fprintf(stdout,"Success\n");
    break;
    /*
  * RPC_CANTSTORE is a new value returned to the
     * client when the buffer cannot store a request.
     */
case RPC_CANTSTORE:
    fprintf(stdout,"RPC_CANTSTORE error. Flushing ... \n");
    /*
     * The buffer is flushed using the blocking flush method
     */
          bres = clnt_control(clnt, CLFLUSH, NULL);
    if (!bres) {
               clnt_perror(clnt, "clnt_control");
```

EXAMPLE 1–2 Using a simple counter with non-blocking I/O *(Continued)*

```
                }
            break;
        default:
                clnt_perror(clnt, "call failed");
            break;
        }

        /* Flush */
        bres = clnt_control(clnt, CLFLUSH, NULL);
        if (!bres) {
                clnt_perror(clnt, "clnt_control");
        }

        clnt_destroy(clnt);
        exit(0);
}
```

`clnt_call()` Configured as Non-Blocking

For a one-way message, use the `clnt_send()` function. No time-out is applied as the client sends a request to a server and does not wait for a reply.

For two-way messaging, use `clnt_call()`. The client remains blocked until the server sends a reply or an error status message, or until a time-out occurs at the client side.

The non-blocking feature enables you to send two-way and one-way calls together. If you use `clnt_call()` on the client side configured as non-blocking, that is using the `RPC_CL_NONBLOCKING` I/O mode, you get the following modified behavior. When a two-way request is sent to the buffer, all one-way requests already in the buffer are sent through the transport layer before the two-way request is processed. The time taken to empty the buffer is not counted in the two-way call time-out. For further information, see the `clnt_control`(3NSL) man page.

Client Connection Closure Callback

Client connection closure callback enables the server to detect that the client is no longer available and to recover from errors that result from the client connection being closed.

The connection closure callback is called when no requests are currently being executed on the connection. If the client connection is closed when a request is being executed, the server executes the request but a reply may not be sent to the client. The connection closure callback is called when all pending request are completed.

When a connection closure occurs, the transport layer sends an error message to the client. The handler is attached to a service using `svc_control()` for example as follows:

```
svc_control(service, SVCSET_RECVERRHANDLER, handler);
```

The arguments of `svc_control()` are:

1. A service or an instance of this service. When this argument is a service, any new connection to the service inherits the error handler. When this argument is an instance of the service, only this connection gets the error handler.
2. The error handler callback. The prototype of this callback function is:

   ```
   void handler(const SVCXPRT *svc, const boot_t IsAConnection);
   ```

For further information, see the `svc_control`(3NSL) man page.

Note – For XDR unmarshalling errors, if the server is unable to unmarshal a request, the message is destroyed and an error is returned directly to the client.

EXAMPLE 1–3 Example using client connection closure callback

This example implements a message log server. A client can use this server to open a log (actually a text file), to store message log, and then to close the log.

The `log.x` file describes the log program interface.

```
enum log_severity { LOG_EMERG=0, LOG_ALERT=1, LOG_CRIT=2, LOG_ERR=3,
            LOG_WARNING=4, LOG_NOTICE=5, LOG_INFO=6 };

program LOG {
              version LOG_VERS1 {
                          int OPENLOG(string ident) = 1;

                          int CLOSELOG(int logID) = 2;

                          oneway WRITELOG(int logID, log_severity severity,
                  string message) = 3;
                } = 1;
} = 0x20001971;
```

The two procedures (`OPENLOG` and `CLOSELOG`) open and close a log that is specified by its logID. The `WRITELOG()` procedure, declared as `oneway` for the example, logs a message in an opened log. A log message contains a severity attribute and a text message.

EXAMPLE 1–3 Example using client connection closure callback *(Continued)*

This is the Makefile for the log server. Use this Makefile to call the `log.x` file.

```
RPCGEN = rpcgen

CLIENT = logClient
CLIENT_SRC = logClient.c log_clnt.c log_xdr.c
CLIENT_OBJ = $(CLIENT_SRC:.c=.o)

SERVER = logServer
SERVER_SRC = logServer.c log_svc.c log_xdr.c
SERVER_OBJ = $(SERVER_SRC:.c=.o)

RPCGEN_FILES = log_clnt.c log_svc.c log_xdr.c log.h

CFLAGS += -I.

RPCGEN_FLAGS    = -N -C
LIBS = -lsocket -lnsl

all: log.h ./$(CLIENT) ./$(SERVER)


$(CLIENT): log.h $(CLIENT_OBJ)
              cc -o $(CLIENT) $(LIBS) $(CLIENT_OBJ)

$(SERVER): log.h $(SERVER_OBJ)
              cc -o $(SERVER)  $(LIBS) $(SERVER_OBJ)

$(RPCGEN_FILES): log.x
              $(RPCGEN) $(RPCGEN_FLAGS) log.x

clean:
              rm -f $(CLIENT_OBJ) $(SERVER_OBJ) $(RPCGEN_FILES)
```

`logServer.c` shows the implementation of the log server. As the log server opens a file to store the log messages, it registers a closure connection callback in `openlog_1_svc()`. This callback is used to close the file descriptor even if the client program forgets to call the `closelog()` procedure (or crashes before doing so). This example demonstrates the use of the connection closure callback feature to free up resources associated to a client in an RPC server.

```
#include "log.h"
#include <stdio.h>
#include <string.h>

#define NR_LOGS 3

typedef struct {
    SVCXPRT* handle;
    FILE* filp;
    char* ident;
```

EXAMPLE 1–3 Example using client connection closure callback *(Continued)*

```
} logreg_t;


static logreg_t logreg[NR_LOGS];
static char* severityname[] = {"Emergency", "Alert", "Critical", "Error",
                               "Warning", "Notice", "Information"};

    static void
close_handler(const SVCXPRT* handle, const bool_t);


    static int
get_slot(SVCXPRT* handle)
{
    int i;

    for (i = 0; i < NR_LOGS; ++i) {
        if (handle == logreg[i].handle) return i;
    }
    return -1;
}

    static FILE*
_openlog(char* logname)
/*
 * Open a log file
 */
{
    FILE* filp = fopen(logname, "a");
    time_t t;

    if (NULL == filp) return NULL;

    time(&t);
    fprintf(filp, "Log opened at %s\n", ctime(&t));

    return filp;
}

    static void
_closelog(FILE* filp)
{
    time_t t;

    time(&t);
    fprintf(filp, "Log close at %s\n", ctime(&t));
        /*
      * Close a log file
      */
    fclose(filp);
}
```

EXAMPLE 1–3 Example using client connection closure callback *(Continued)*

```
    int*
openlog_1_svc(char* ident, struct svc_req* req)
{
    int slot = get_slot(NULL);
    FILE* filp;
    static int res;
    time_t t;

    if (-1 != slot) {
        FILE* filp = _openlog(ident);
        if (NULL != filp) {
            logreg[slot].filp = filp;
            logreg[slot].handle = req->rq_xprt;
            logreg[slot].ident = strdup(ident);

        /*
      * When the client calls clnt_destroy, or when the
      * client dies and clnt_destroy is called automatically,
      * the server executes the close_handler callback
      */
            if (!svc_control(req->rq_xprt, SVCSET_RECVERRHANDLER,
                                            (void*)close_handler)) {
                        puts("Server: Cannot register a connection closure
                    callback");
                        exit(1);
                    }
        }

    }
    res = slot;
    return &res;
}

    int*
closelog_1_svc(int logid, struct svc_req* req)
{
    static int res;

    if ((logid >= NR_LOGS) || (logreg[logid].handle != req->rq_xprt)) {
        res = -1;
        return &res;
    }
    logreg[logid].handle = NULL;
    _closelog(logreg[logid].filp);
    res = 0;
    return &res;
}

/*
 * When there is a request to write a message to the log,
 * write_log_1_svc is called
 */
```

EXAMPLE 1–3 Example using client connection closure callback *(Continued)*

```
    void*
writelog_1_svc(int logid, log_severity severity, char* message,
               struct svc_req* req)
{
    if ((logid >= NR_LOGS) || (logreg[logid].handle != req->rq_xprt)) {
        return NULL;
    }
        /*
         * Write message to file
         */
    fprintf(logreg[logid].filp, "%s (%s): %s\n",
            logreg[logid].ident, severityname[severity], message);
    return NULL;
}

    static void
close_handler(const SVCXPRT* handle, const bool_t dummy)
{
    int i;

        /*
         * When the client dies, the log is closed with closelog
         */
    for (i = 0; i < NR_LOGS; ++i) {
        if (handle == logreg[i].handle) {
            logreg[i].handle = NULL;
            _closelog(logreg[i].filp);
        }
    }
}
```

The `logClient.c` file shows a client using the log server.

```
#include "log.h"
#include <stdio.h>

#define MSG_SIZE 128

    void
usage()
{
    puts("Usage: logClient <logserver_addr>");
    exit(2);
}

    void
runClient(CLIENT* clnt)
{
    char msg[MSG_SIZE];
    int logID;
    int* result;
```

EXAMPLE 1–3 Example using client connection closure callback *(Continued)*

```
        /*
       * client opens a log
       */
    result = openlog_1("client", clnt);
    if (NULL == result) {
        clnt_perror(clnt, "openlog");
        return;
    }
    logID = *result;
    if (-1 == logID) {
        puts("Cannot open the log.");
        return;
    }

    while(1) {
        struct rpc_err e;

             /*
               * Client writes a message in the log
               */
        puts("Enter a message in the log (\".\" to quit):");
        fgets(msg, MSG_SIZE, stdin);
        /*
                * Remove trailing CR
                */
        msg[strlen(msg)-1] = 0;

        if (!strcmp(msg, ".")) break;

        if (writelog_1(logID, LOG_INFO, msg, clnt) == NULL) {
            clnt_perror(clnt, "writelog");
            return;
        }
    }
         /*
          * Client closes the log
          */
    result = closelog_1(logID, clnt);
    if (NULL == result) {
        clnt_perror(clnt, "closelog");
        return;
    }
    logID = *result;
    if (-1 == logID) {
        puts("Cannot close the log.");
        return;
    }
}

    int
main(int argc, char* argv[])
{
```

EXAMPLE 1–3 Example using client connection closure callback *(Continued)*

```
    char* serv_addr;
    CLIENT* clnt;

    if (argc != 2) usage();

    serv_addr = argv[1];

    clnt = clnt_create(serv_addr, LOG, LOG_VERS1, "tcp");

    if (NULL == clnt) {
        clnt_pcreateerror("Cannot connect to log server");
        exit(1);
    }
    runClient(clnt);

    clnt_destroy(clnt);
}
```

User File Descriptor Callbacks

User file descriptor callbacks enable you to register file descriptors with callbacks, specifying one or more event types. Now you can use an RPC server to handle file descriptors that were not written for the Sun RPC library.

For user file descriptor callbacks, two new functions have been added to the Sun RPC library, `svc_add_input`(3NSL) and `svc_remove_input`(3NSL), to implement user file descriptor callbacks. These functions declare or remove a callback on a file descriptor.

When using this new callback feature you must:

1. Create your `callback()` function by writing user code with the following syntax:

   ```
   typedef void (*svc_callback_t) (svc_input_id_t id, int fd, \
   unsigned int revents, void* cookie);
   ```

   The four parameters passed to the `callback()` function are:

   - *id*—provides an identifier for each callback. This identifier can be used to remove a callback.
   - *fd*—the file descriptor that your callback is waiting for.
   - *revents*—an unsigned integer representing the events that have occurred. This set of events is a subset of the list given when the callback is registered.
   - *cookie*—the cookie given when the callback is registered. This cookie can be a pointer to specific data the server needs during the callback.

2. Call `svc_add_input()` to register file descriptors and associated events, such as read or write, that the server must be aware of.

   ```
   svc_input_id_t svc_add_input (int fd, unsigned int revents, \
   svc_callback_t callback, void* cookie);
   ```

   A list of the events that can be specified is given in`poll`(2).
3. Specify a file descriptor. This file descriptor can be an entity such as a socket or a file.

When one of the specified events occurs, the standard server loop calls the user code through `svc_run()` and your callback performs the required operation on the file descriptor (socket or file).

When you no longer need a particular callback, call `svc_remove_input()` with the corresponding identifier to remove the callback.

EXAMPLE 1–4 How to register user file descriptors on an RPC server.

This example shows you how to register a user file descriptor on an RPC server and how to provide user defined callbacks. With this example you can monitor the time of day on both the server and the client.

1. The `Makefile` for this example.

```
   RPCGEN = rpcgen

CLIENT = todClient
CLIENT_SRC = todClient.c timeofday_clnt.c
CLIENT_OBJ = $(CLIENT_SRC:.c=.o)

SERVER = todServer
SERVER_SRC = todServer.c timeofday_svc.c
SERVER_OBJ = $(SERVER_SRC:.c=.o)

RPCGEN_FILES = timeofday_clnt.c timeofday_svc.c  timeofday.h

CFLAGS += -I.

RPCGEN_FLAGS = -N -C
LIBS = -lsocket -lnsl

all: ./$(CLIENT) ./$(SERVER)


$(CLIENT): timeofday.h $(CLIENT_OBJ)
              cc -o $(CLIENT) $(LIBS) $(CLIENT_OBJ)

$(SERVER): timeofday.h $(SERVER_OBJ)
              cc -o $(SERVER)  $(LIBS) $(SERVER_OBJ)

timeofday_clnt.c: timeofday.x
                $(RPCGEN) -l $(RPCGEN_FLAGS) timeofday.x > timeofday_clnt.c
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
timeofday_svc.c: timeofday.x
              $(RPCGEN) -m $(RPCGEN_FLAGS) timeofday.x > timeofday_svc.c

timeofday.h: timeofday.x
              $(RPCGEN) -h $(RPCGEN_FLAGS) timeofday.x > timeofday.h


clean:
              rm -f $(CLIENT_OBJ) $(SERVER_OBJ) $(RPCGEN_FILES)
```

2. The `timeofday.x` file defines the RPC services offered by the server in this examples. The services in this examples are `gettimeofday()` and `settimeofday()`.

```
program TIMEOFDAY {
              version VERS1 {
                           int SENDTIMEOFDAY(string tod) = 1;

                           string GETTIMEOFDAY() = 2;
              } = 1;
} = 0x20000090;
```

3. The `userfdServer.h` file defines the structure of messages sent on the sockets in this example.

```
   #include "timeofday.h"
#define PORT_NUMBER 1971

/*
 * Structure used to store data for a connection.
     *    (user fds test).
 */
typedef struct {
    /*
 * Ids of the callback registration for this link.
 */
    svc_input_id_t in_id;
    svc_input_id_t out_id;

    /*
          * Data read from this connection.
          */
    char in_data[128];

    /*
          * Data to be written on this connection.
          */
    char out_data[128];
    char* out_ptr;

} Link;

    void
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
socket_read_callback(svc_input_id_t id, int fd, unsigned int events,
                                    void* cookie);
    void
socket_write_callback(svc_input_id_t id, int fd, unsigned int events,
                      void* cookie);
    void
socket_new_connection(svc_input_id_t id, int fd, unsigned int events,
                      void* cookie);

    void
timeofday_1(struct svc_req *rqstp, register SVCXPRT *transp);
```

4. The `todClient.c` file shows how the time of day is set on the client. In this file RPC is used with and without sockets.

```
   #include "timeofday.h"

#include <stdio.h>
#include <netdb.h>
#define PORT_NUMBER 1971

    void
runClient();
    void
runSocketClient();


char* serv_addr;

    void
usage()
{
    puts("Usage: todClient [-socket] <server_addr>");
    exit(2);
}

    int
main(int argc, char* argv[])
{
    CLIENT* clnt;
    int sockClient;

    if ((argc != 2) && (argc != 3))
        usage();

    sockClient = (strcmp(argv[1], "-socket") == 0);

        /*
         * Choose to use sockets (sockClient).
         * If sockets are not available,
         * use RPC without sockets (runClient).
         */
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
    if (sockClient && (argc != 3))
        usage();

    serv_addr = argv[sockClient? 2:1];

    if (sockClient) {
        runSocketClient();
    } else {
        runClient();
    }

    return 0;
}
/*
     * Use RPC without sockets
 */
    void
runClient()
{
    CLIENT* clnt;
    char* pts;
    char** serverTime;

    time_t now;

    clnt = clnt_create(serv_addr, TIMEOFDAY, VERS1, "tcp");
    if (NULL == clnt) {
        clnt_pcreateerror("Cannot connect to log server");
        exit(1);
    }

    time(&now);
    pts = ctime(&now);

    printf("Send local time to server\n");

    /*
          * Set the local time and send this time to the server.
          */
    sendtimeofday_1(pts, clnt);

    /*
          * Ask the server for the current time.
          */
    serverTime = gettimeofday_1(clnt);

    printf("Time received from server: %s\n", *serverTime);

    clnt_destroy(clnt);
}

/*
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
 * Use RPC with sockets
 */
   void
runSocketClient()
/*
 * Create a socket
 */
{
    int s = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in sin;
    char* pts;
    char buffer[80];
    int len;

    time_t now;
    struct hostent* hent;
    unsigned long serverAddr;

    if (-1 == s) {
        perror("cannot allocate socket.");
        return;
    }

    hent = gethostbyname(serv_addr);
    if (NULL == hent) {
        if ((int)(serverAddr = inet_addr(serv_addr)) == -1) {
            puts("Bad server address");
            return;
        }
    } else {
        memcpy(&serverAddr, hent->h_addr_list[0], sizeof(serverAddr));
    }

    sin.sin_port = htons(PORT_NUMBER);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = serverAddr;

         /*
          * Connect the socket
          */
    if (-1 == connect(s, (struct sockaddr*)(&sin),
                      sizeof(struct sockaddr_in))) {
        perror("cannot connect the socket.");
        return;
    }

    time(&now);
    pts = ctime(&now);

        /*
         * Write a message on the socket.
         * The message is the current time of the client.
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
         */
        puts("Send the local time to the server.");
        if (-1 == write(s, pts, strlen(pts)+1)) {
            perror("Cannot write the socket");
            return;
        }

         /*
          * Read the message on the socket.
          * The message is the current time of the server
          */
        puts("Get the local time from the server.");
        len = read(s, buffer, sizeof(buffer));

        if (len == -1) {
            perror("Cannot read the socket");
            return;
        }
        puts(buffer);

        puts("Close the socket.");
        close(s);
}
```

5. The `todServer.c` file shows the use of the `timeofday` service from the server side.

```
   #include "timeofday.h"
#include "userfdServer.h"
#include <stdio.h>
#include <errno.h>
#define PORT_NUMBER 1971

int listenSocket;

/*
 * Implementation of the RPC server.
 */

    int*
sendtimeofday_1_svc(char* time, struct svc_req* req)
{
    static int result = 0;

    printf("Server: Receive local time from client  %s\n", time);
    return &result;
}

    char **
gettimeofday_1_svc(struct svc_req* req)
{
    static char buff[80];
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
    char* pts;
    time_t now;
    static char* result = &(buff[0]);

    time(&now);
    strcpy(result, ctime(&now));

    return &result;
}

/*
     * Implementation of the socket server.
 */

    int
create_connection_socket()
{
    struct sockaddr_in sin;
    int size = sizeof(struct sockaddr_in);
    unsigned int port;

         /*
          * Create a socket
          */
    listenSocket = socket(PF_INET, SOCK_STREAM, 0);

    if (-1 == listenSocket) {
        perror("cannot allocate socket.");
        return -1;
    }

    sin.sin_port = htons(PORT_NUMBER);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenSocket, (struct sockaddr*)&sin, sizeof(sin)) == -1) {
        perror("cannot bind the socket.");
        close(listenSocket);
        return -1;
    }
         /*
          * The server waits for the client
          * connection to be created
          */
    if (listen(listenSocket, 1)) {
        perror("cannot listen.");
        close(listenSocket);
        listenSocket = -1;
        return -1;
    }

         /*
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
            * svc_add_input registers a read callback,
            * socket_new_connection, on the listening socket.
            * This callback is invoked when a new connection
            * is pending. */
    if (svc_add_input(listenSocket, POLLIN,
                      socket_new_connection, (void*) NULL) == -1) {
        puts("Cannot register callback");
        close(listenSocket);
        listenSocket = -1;
        return -1;
    }

    return 0;
}

/*
 * Define the socket_new_connection callback function
 */
    void
socket_new_connection(svc_input_id_t id, int fd,
                      unsigned int events, void* cookie)
{
    Link* lnk;
    int connSocket;

         /*
          * The server is called when a connection is
          * pending on the socket. Accept this connection now.
          * The call is non-blocking.
          * Create a socket to treat the call.
          */
    connSocket = accept(listenSocket, NULL, NULL);
    if (-1 == connSocket) {
        perror("Server: Error: Cannot accept a connection.");
        return;
    }

    lnk = (Link*)malloc(sizeof(Link));
    lnk->in_data[0] = 0;

         /*
          * New callback created, socket_read_callback.
          */
    lnk->in_id = svc_add_input(connSocket, POLLIN,
                   socket_read_callback, (void*)lnk);
}
/*
 * New callback, socket_read_callback, is defined
 */
    void
socket_read_callback(svc_input_id_t id, int fd, unsigned int events,
                     void* cookie)
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
{
    char buffer[128];
    int len;
    Link* lnk = (Link*)cookie;

    /*
          * Read the message. This read call does not block.
          */
    len = read(fd, buffer, sizeof(buffer));

    if (len > 0) {
          /*
                * Got some data. Copy it in the buffer
                * associated with this socket connection.
                */
        strncat (lnk->in_data, buffer, len);

        /*
                * Test if we receive the complete data.
         * Otherwise, this is only a partial read.
                */
        if (buffer[len-1] == 0) {
            char* pts;
            time_t now;

            /*
                     * Print the time of day you received.
                     */
            printf("Server: Got time of day from the client: \n %s",
                   lnk->in_data);

            /*
                     * Setup the reply data
                     * (server current time of day).
                     */
            time(&now);
            pts = ctime(&now);

            strcpy(lnk->out_data, pts);
            lnk->out_ptr = &(lnk->out_data[0]);

                    /*
                     * Register a write callback (socket_write_callback)
                     * that does not block when writing a reply.
                     * You can use POLLOUT when you have write
                     * access to the socket
                     */
            lnk->out_id = svc_add_input(fd, POLLOUT,
                                        socket_write_callback, (void*)lnk);

        }
    } else if (len == 0) {
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
        /*
         * Socket closed in peer.  Closing the socket.
         */
        close(fd);
        } else {
            /*
                    * Has the socket been closed by peer?
                    */
            if (errno != ECONNRESET) {
                /*
                         * If no, this is an error.
                         */
                perror("Server: error in reading the socket");
                printf("%d\n", errno);
            }
            close(fd);
        }
    }

    /*
     * Define the socket_write_callback.
     * This callback is called when you have write
     * access to the socket.
     */
        void
    socket_write_callback(svc_input_id_t id, int fd, unsigned int events,
                          void* cookie)
    {
        Link* lnk = (Link*)cookie;

        /*
              * Compute the length of remaining data to write.
              */
        int len = strlen(lnk->out_ptr)+1;

    /*
     * Send the time to the client
     */
        if (write(fd, lnk->out_ptr, len) == len) {
        /*
              * All data sent.
              */

            /*
                   * Unregister the two callbacks. This unregistration
              * is demonstrated here as the registration is
                   * removed automatically when the file descriptor
                 * is closed.
                 */
            svc_remove_input(lnk->in_id);
            svc_remove_input(lnk->out_id);
```

EXAMPLE 1–4 How to register user file descriptors on an RPC server. *(Continued)*

```
            /*
                    * Close the socket.
                    */
            close(fd);
        }
    }

        void
    main()
    {
        int res;

             /*
              * Create the timeofday service and a socket
              */
        res = create_connection_socket();
        if (-1 == res) {
            puts("server: unable to create the connection socket.\n");
            exit(-1);
        }

        res = svc_create(timeofday_1, TIMEOFDAY, VERS1, "tcp");
        if (-1 == res) {
            puts("server: unable to create RPC service.\n");
            exit(-1);
        }

             /*
              Poll the user file descriptors.
              */
        svc_run();
    }
```

Index