RPC2 User Guide and Reference Manual

M. Satyanarayanan (Editor) Richard Draves, James Kistler, Anders Klemets, Qi Lu, Lily Mummert, David Nichols, Larry Raper, Gowthami Rajendran, Jonathan Rosenberg, Ellen Siegel[1]

This manual describes the programming interface and internal design of *RPC2*, a highly portable and extensible remote procedure call package for Unix. RPC2 runs on top of the IP/UDP protocol, and provides support for streaming file transfer, parallel RPC calls, and IP-level multicast. The manual also describes two other packages used by RPC2: *RP2Gen*, a stub generator, and *LWP*, a coroutine-based lightweight process package. All the software described in this manual runs at user-level on the Unix 4.3BSD interface; no kernel modifications are necessary. The software has been ported to a variety of machine architectures (such as IBM-RT, MIPS, Sun2, Sun3, Sparc, and i386) and to variants of the Unix operating system (such as Mach, SunOS and AIX).

Chapter 1

Preface

RPC2 is a portable and extensible remote procedure call package built on top of the IP/UDP network protocol. Since its original use in the Andrew File System@cite[Howard88], it has been used by the Coda File System@cite[Satyanarayanan89b], the Scylla database server and a variety of other applications at Carnegie Mellon University and elsewhere. RPC2 runs on LWP, a lightweight process package that allows multiple non-preemptive threads of control to coexist within one Unix process. RP2Gen is a stub generator that simplifies the use of RPC2. RPC2 and LWP run entirely at user-level on the Unix 4.3BSD interface; no kernel changes are necessary.

The first versions of LWP and RPC2 were operational in early 1984 and mid-1985 respectively. The design of LWP predates that of the Cthreads package in Mach@cite[cthreads]. LWP and Cthreads are conceptually similar, but differ substantially in the details. It should, in principle, be possible to modify RPC2 to run directly on top of Cthreads, but we have not tried to do this so far. We have successfully emulated all of LWP on top of the preemptive and nonpre-emptive versions of Cthreads, and a subset of the non-preemptive version of Cthreads on top of LWP.

Both LWP and RPC2 have evolved over time, resulting in increased functionality and robustness. They have also been ported to a wide variety of machine architectures, such as IBM-RTs, MIPS, Sun2, Sun3, Sparc, and i386, as well as variants of the Unix operating systems such as Mach, SunOS and AIX. The RPC2 extensions include the SFTP side effect mechanism for streaming file transfer, the MultiRPC mechanism for parallel RPC calls, and IP multicast versions of MultiRPC and SFTP. Whenever there has been choice between portability and machine-specific performance, we have always favored portability.

Although this manual may seem formidable, RPC2 is not difficult to use. For simple applications you do not have to know everything described in these pages. Advanced features such as use of side effects, MultiRPC, and use of multicast for parallelism can be ignored initially.

Some day there will be a true tutorial in this manual. Until then the best way to learn RPC2 is as follows:

1. Study the overview and examples in Chapter 2 (XXX). The latter are actual pieces of working code, and you should try running the examples. Appendix **??** (XXX) gives you details of library and include file names, current limitations, and other similar details. Read Chapter 3 (XXX) next. This describes the procedural abstraction provided by RP2Gen, the stub generator.

2. Read Chapter 4 (XXX) and Part @ref[LWPPart], which describe the RPC2 and LWP runtime calls, respectively. You may need to refer to Appendix **??** (XXX), which contains the header files used by these packages.
3. After you have mastered the basics, you may wish to explore the advanced features described in Chapters 5 (XXX), 6 (XXX) and 7 (XXX).
4. If you wish to learn about the internals of RPC2, you may wish to consult the chapters in Part **??** (XXX).

Credits

The original design, implementation and documentation of RPC2 was by M. Satyanarayanan. The MultiRPC implementation and chapter were done by Ellen Siegel. The extensions to use multicast were done by James Kistler. Jonathan Rosenberg contributed to the RP2Gen implementation and chapter. Comments from Robert Sidebotham, David Nichols, Vasilis Apostolides, Drew Perkins, Bradley White, Stephen Hecht and many other users have been helpful in the improvement of RPC2.

The original design and implementation of LWP was done by Larry Raper. Its documentation descends from a manual by Jonathan Rosenberg and Larry Raper, later extended by David Nichols and M. Satyanarayanan. Richard Draves, Qi Lu, Anders Klemets and Gowthami Rajendran helped in revising and improving this document.

RPC2 User Reference

Chapter 2

Overview and Examples

Remote Procedure Call (RPC) is a paradigm of communication between a client and a server. Interactions consist of an alternating sequence of brief requests by the client, and brief replies by the server. Each request consists of an *Operation* and a set of *Parameters*. A reply consists of a *Return Code* and a set of result parameters. Site and communication failures are synchronously detected.

2.1 Components

RPC2 consists of two relatively independent components: a Unix-based run time library written in C, and a stub generator RP2Gen. The run-time system is self contained and usable in the absence of RP2Gen. The RPC2 interface, which describes the characteristics of the procedures provided by the server to the clients is defined in a file with extension `.rpc2` or `.rpc`. RP2Gen takes a description of the interface and automatically produces the client and server stubs, and the code for marshalling.

The RPC2 package is used in conjunction with the Lightweight Process (LWP) Package. The client and server code together with the respective stubs and the RPC2 and LWP packages are all linked together to generate the client and server object binaries. Although RPC2 and LWP are written in C, they have been successfully used by programs used in C++.

2.2 Client-Server Model and Binding

The concepts of “server and “client are quite general. A server may be the client of one or more other servers. A server may have one more *Subsystems* associated with it, each subsystem corresponding to a set of functionally-related procedure calls. Binding by clients is done to a host-portal-subsystem triple. There is no a priori restriction (other than resource limitations) on the number of clients a server may have, or on the number of servers a client may be connected to. Host, portal, and subsystem specifications are discriminated union types, to allow a multiplicity of representations. Thus hosts may be specified either by name or by Internet address.

2.3 How RPC2 uses LWPs

Clients and servers are each assumed to be Unix processes using the LWP package. RPC2 will not work independently of the LWP package. The LWP package makes it possible for a single Unix process to contain multiple threads of control (LWPs). An RPC call is synchronous with respect to an individual LWP, but it does not block the encapsulating Unix process. Although LWPs are non-preemptive, RPC2 internally yields control when an LWP is blocked awaiting a request or reply. Thus more than one LWP can be concurrently making RPC requests. There is no a priori binding of RPC connections to LWPs, or LWPs to subsystems within a client or server. Thus RPC connections, LWPs and subsystems are completely orthogonal concepts.

Since LWPs are non-preemptive, a long-running computation by an LWP will prevent the RPC2 code from getting a chance to run. This will typically manifest itself as a timeout by the client on the RPC in progress. To avoid this, the long running computation should periodically invoke @IOMGR(Poll) followed by @LWP(DispatchProcess).

For similar reasons, Unix system calls such as `read (2)`, `sleep (3)`, and `select (2)` that may block for a long time should be avoided. Instead, use IOMGR_Select.

2.4 Side Effects

RPC connections may be associated with *Side-Effects* to allow application-specific network optimizations to be performed. An example is the use of a specialized protocol for bulk transfer of large files. Detailed information pertinent to each type of side effect is specified in a *Side Effect Descriptor*.

Side effects are explicitly initiated by the server using @RPC2(InitSideEffect) and occur asynchronously. Synchronization occurs on an @RPC2(CheckSideEffect) call by the server. Each client-server connection deals with one kind of side effect; however different connections may use different kinds of side effects.

Adding support for a new type of side effect is analogous to adding a new device driver in Unix. To allow this extensibility, the RPC code has hooks at various points where side-effect routines will be called. Global tables contain pointers to these side effect routines. The basic RPC code itself knows nothing about these side-effect routines.

2.5 Security

A *Security Level* is associated with each connection between a client and a server. Currently four levels of security are supported:

- neither authenticated, nor secure

  authenticated, but not secure

  authenticated, and headers secure

  authenticated, and fully secure

Authenticated in this context means that the client and server start out as mutually suspicious parties and exchange credentials during the establishment of a connection. A secret encryption key, known a priori only to the server and client is used in authentication handshakes. The secret key itself is never sent over the network. The handshake is a variant of the well-known Needham and Schroeder protocol@cite[Needham78].

Secure means that transmitted data is immune to eavesdropping and undetected corruption. This is achieved by encryption, using a session key generated during connection establishment. No attempt is made, however, to guard against traffic analysis attacks.

The RPC package makes no assumptions about the format of client identities or about the mapping between clients, servers and shared secret keys. A server-supplied procedure is invoked during the authentication sequence to validate client identities and obtain keys. The paper on authentication in the Andrew file system@cite[Satyanarayanan89a] gives a detailed example of how this functionality may be used.

2.6 Examples

In this section we walk the reader through examples of increasing complexity, thereby demonstrating the use of RPC2. There are two examples: a relatively simple one using a single LWP and exporting a single subsystem, followed by a more complex one using multiple LWPs and subsystems. There is a Makefile for both these examples in Section **??** (XXX). Neither example uses side effects. An example using side effects can be found in Chapter 5 (XXX).

2.6.1 Example 1: Getting Time of Day

This is a very simple example that has only one subsystem telling the client the time of the day in seconds and micro seconds on the server machine.

The interface is defined in the `rtime.rpc` file. This file is used as input to RP2Gen to generate the client and server stubs `rtime.client.c` and `rtime_server.c.sgml`

The first step at both the client and the server is initialization of the LWP and the RPC2 package. This is done by calling the @LWP(Init) and @RPC2(Init) calls respectively. It is necessary that the LWP package be initialized before the RPC2 package.

The client, after initialization, binds to the server by calling RPC2_NewBinding. The parameters to this call include the identity of the server, the subsytem etc. A connection id is returned to the client. The client is now ready to make RPC calls on this connection.

The server indicates its willingness to accept calls for the `rtime` subsytem by calling RPC2_Export. A filter for the `rtime` subsytem is specified and used in awaiting an RPC request by calling @RPC2(GetRequest). On receiving a request, the server decodes and executes it by calling `rtime_ExecuteRequest. rtime_ExecuteRequest,` rpcwhich is actually defined in the server stub file calls the server code for `GetRTime` routine and sends the response to the client. The server then goes back to waiting for more requests.

Interface Specification (*rtime_ rpc*)

Subsystem "rtime";

#define RTIMESUBSYSID 100

#define RTIMEPORTAL 4000

GetRTime(OUT RPC2_Integer time_sec, OUT RPC2_Integer time_usec) ;

Client Code (*rtime_ clnt.c.sgml*)

```
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <assert.h>
#include "lwp.h"
#include "rpc2.h"
#include "rtime.h"


main(argc, argv)
int argc;
char *argv[];
{
  RPC2_Handle cid;
  RPC2_Integer tv_sec, tv_usec;
  int rc ;
  char msg[200];

  if (argc != 2)
      error_report("usage: rtime <machine name>");

  Init_RPC();

  cid = connect_to_machine(argv[1]);

/* client makes a remote procedure call to get the time on the server machine */
  rc = GetRTime(cid, &tv_sec, &tv_usec) ;
  if (rc != RPC2_SUCCESS) {
      sprintf(msg, "%s\nGet remote time on machine %s failed",
              RPC2_ErrorMsg(rc),argv[1]);
      error_report(msg);
  } else {
      printf("The remote time on machine %s is\n", argv[1]);
      printf("tv_sec = %d and tv_usec = %d\n", tv_sec, tv_usec);
  };

  rc = RPC2_Unbind(cid) ;
  if (rc != RPC2_SUCCESS)
```

```
        error_report("%s\nCant' close the connection!", RPC2_ErrorMsg(rc));
} ;


error_report(message)
char *message;
{
  fprintf(stderr, message);
  fprintf(stderr, "\n");
  exit(1);
}

Init_RPC()
{
  PROCESS mylpid;
  int rc ;

  if (LWP_Init(LWP_VERSION, LWP_NORMAL_PRIORITY, &mylpid) != LWP_SUCCESS)
      error_report("Can't Initialize LWP");   /* Initialize LWP package */

  rc = RPC2_Init(RPC2_VERSION, NULL, NULL, 0, -1, NULL) ;
  if (rc != RPC2_SUCCESS)
      error_report("%s\nCan't Initialize RPC2", RPC2_ErrorMsg(rc));
  /* Initialize RPC2 package */
}

/* This routine tries to establish a connection to the server running on machine machine_name */
RPC2_Handle connect_to_machine(machine_name)
char *machine_name;
{
  RPC2_HostIdent hid;
  RPC2_PortalIdent pid;
  RPC2_SubsysIdent sid;
  RPC2_Handle cid;
  char msg[100];
  int rc;
  RPC2_BindParms bp;

  hid.Tag = RPC2_HOSTBYNAME;
  if (strlen(machine_name) >= 64) {
      sprintf(msg, "Machine name %s too long!", machine_name);
      error_report(msg);
  } ;
  strcpy(hid.Value.Name, machine_name);
  pid.Tag = RPC2_PORTALBYINETNUMBER;
  pid.Value.InetPortNumber = htons(RTIMEPORTAL);
  sid.Tag = RPC2_SUBSYSBYID;
  sid.Value.SubsysId = RTIMESUBSYSID;

  bp.SecurityLevel = RPC2_OPENKIMONO;
  bp.EncryptionType = NULL;
```

```
    bp.SideEffectType = NULL;
    bp.ClientIdent = NULL;
    bp.SharedSecret = NULL;
    rc = RPC2_NewBinding(&hid, &pid, &sid, &bp, &Gcid);
    if (rc != RPC2_SUCCESS) {
        sprintf(msg, "%s\nCan't connect to machine %s", RPC2_ErrorMsg(rc),
                machine_name);
        error_report(msg);
    };
    return cid;
  } ;
```

Server Code (*rtime_srv.c*)

```
  #include <stdio.h>
  #include <sys/types.h>
  #include <netinet/in.h>
  #include <assert.h>
  #include <sys/time.h>
  #include "lwp.h"
  #include "rpc2.h"
  #include "rtime.h"


  main()
  {
    RPC2_Handle cid;
    RPC2_RequestFilter reqfilter;
    RPC2_PacketBuffer *reqbuffer;
    int rc;

    Init_RPC();

    reqfilter.FromWhom = ANY;
    reqfilter.OldOrNew = OLDORNEW;
    reqfilter.ConnOrSubsys.SubsysId = RTIMESUBSYSID;

    /* loop forever, wait for the client to call for service */
    for ( ; ; ) {
        rc = RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL) ;
        if (rc != RPC2_SUCCESS)
            fprintf(stderr, RPC2_ErrorMsg(rc));
        rc = rtime_ExecuteRequest(cid, reqbuffer);
        if (rc != RPC2_SUCCESS)
            fprintf(stderr, RPC2_ErrorMsg(rc));
    };
  };

  error_report(message)
  char *message;
```

```
{
  fprintf(stderr, message);
  fprintf(stderr, "\n");
  exit(1);
}

Init_RPC()
{
  PROCESS mylpid;
  RPC2_PortalIdent pid, *pids;
  RPC2_SubsysIdent sid;
  int rc;
  char msg[100];

  /* Initialize LWP package */
  if (LWP_Init(LWP_VERSION, LWP_NORMAL_PRIORITY, &mylpid) != LWP_SUCCESS)
      error_report("Can't Initialize LWP") ;

  /* Initialize RPC2 package */
  pids = &pid;
  pid.Tag = RPC2_PORTALBYINETNUMBER;
  pid.Value.InetPortNumber = htons(RTIMEPORTAL);
  rc = RPC2_Init(RPC2_VERSION, NULL, &pids, 1, -1, NULL);
  if (rc != RPC2_SUCCESS) {
      sprintf(msg, "%s\nCan't Initialize RPC2", RPC2_ErrorMsg(rc));
      error_report(msg);
  };

  sid.Tag = RPC2_SUBSYSBYID;
  sid.Value.SubsysId = RTIMESUBSYSID;
  rc = RPC2_Export(&sid) != RPC2_SUCCESS ;
  if (rc != RPC2_SUCCESS) {
      sprintf(msg, "%s\nCan't export the rtime subsystem");
      error_report(msg);
  };
}

long GetRTime(_cid, tv_sec, tv_usec)
RPC2_Handle _cid;
int *tv_sec;
int *tv_usec;
{
  struct timeval tp;
  struct timezone tzp;

  gettimeofday(&tp, &tzp);
  *tv_sec = tp.tv_sec;
  *tv_usec = tp.tv_usec;
  return RPC2_SUCCESS;
}
```

2.6.2 Example 2: Authentication and Computation Subsystems

In this example there are two subsystems: an authentication subsystem and a computation subsystem. Each subsystem services a different set of calls.

In this example, two LWP processes are created at the server, one for each subsytem to be serviced. But this is not essential. An alternative would have been for a single LWP to service calls for both subsystems or for a collection of LWPs to service calls for any subsystems.

Interface Specifications for Auth Subsystem (*auth_ rpc.c)*

```
/*
RPC interface specification for a trivial authentication subsystem.
This is only an example: all it does is name to id and id to name conversions.
*/

Server Prefix "S";
Subsystem  "auth";

/*
Internet port number; note that this is really not part of a specific
subsystem, but is part of a server; we should really have a  separate ex.h
file with this constant.  I am being lazy here
*/
#define AUTHPORTAL      5000

#define AUTHSUBSYSID    100     /* The subsysid for auth subsystem */

/*
Return codes from auth server
*/
#define AUTHSUCCESS     0
#define AUTHFAILED      1


typedef
    RPC2_Byte PathName[1024];

typedef
    RPC2_Struct
        {
        RPC2_Integer GroupId;
        PathName HomeDir;
        }
        AuthInfo;


AuthNewConn (IN RPC2_Integer seType, IN RPC2_Integer secLevel, IN RPC2_Integer encType,
                        IN RPC2_CountedBS cIdent) NEW_CONNECTION;
```

```
AuthUserId (IN RPC2_String Username, OUT RPC2_Integer UserId);
                /* Returns AUTHSUCCESS or AUTHFAILED */

AuthUserName (IN RPC2_Integer UserId, IN OUT RPC2_BoundedBS Username);
                 /* Returns AUTHSUCCESS or AUTHFAILED */

AuthUserInfo (IN RPC2_Integer UserId, OUT AuthInfo UInfo);
                /* Returns AUTHSUCCESS or AUTHFAILED */

AuthQuit();
```

Interface Specifications for Comp Subsystem(*comp_ rpc.c)*

```
/*
RPC interface specification for a trivial computational subsystem.
Finds squares and cubes of given numbers.
*/



Server Prefix "S";
Subsystem  "comp";



#define COMPSUBSYSID    200     /* The subsysid for comp subsystem */

#define COMPSUCCESS 1
#define COMPFAILED 2

CompNewConn (IN RPC2_Integer seType, IN RPC2_Integer secLevel, IN RPC2_Integer encType,
                        IN RPC2_CountedBS cIdent) NEW_CONNECTION;


CompSquare (IN RPC2_Integer X); /* returns square of x */

CompCube (IN RPC2_Integer X);   /* returns cube of x */

CompAge();      /* returns the age of this connection in  seconds */

CompQuit();
```

Client Code(*authcomp_ clnt.c*

```
/* exclient.c -- Trivial client to demonstrate basic RPC2 functionality */

#include <stdio.h>
#include <strings.h>
#include <sys/time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <assert.h>
#include <pwd.h>
#include <lwp.h>
#include <rpc2.h>
#include "auth.h"
#include "comp.h"

main()
    {
    int a;
    char buf[100];

    printf("Debug Level? (0) ");
    gets(buf);
    RPC2_DebugLevel = atoi(buf);

    InitRPC();
    while (1)
        {
        printf("Action? (1 = New Conn, 2 = Auth Request, 3 = Comp Request) ");
        gets(buf);
        a = atoi(buf);
        switch(a)
            {
            case 1:     NewConn(); continue;
            case 2:     Auth(); continue;
            case 3:     Comp(); continue;
            default:    continue;
            }
        }
    }



NewConn()
    {
    char hname[100], buf[100];
    int newcid, rc;
    RPC2_HostIdent hident;
    RPC2_PortalIdent pident;
    RPC2_SubsysIdent sident;
```

```
    printf("Remote host name? ");
    gets(hident.Value.Name);

    hident.Tag = RPC2_HOSTBYNAME;
    printf("Subsystem? (Auth = %d, Comp = %d) ", AUTHSUBSYSID, COMPSUBSYSID);
    gets(buf);
    sident.Value.SubsysId = atoi(buf);

    sident.Tag = RPC2_SUBSYSBYID;
    pident.Tag = RPC2_PORTALBYINETNUMBER;
    pident.Value.InetPortNumber = htons(AUTHPORTAL);
                                /* same as COMPPORTAL */
    rc = RPC2_Bind(RPC2_OPENKIMONO, NULL, &hident, &pident, &sident,
                                        NULL, NULL, NULL, &newcid);
    if (rc == RPC2_SUCCESS)
        printf("Binding succeeded, this connection id is %d\n", newcid);
    else
        printf("Binding failed: %s\n", RPC2_ErrorMsg(rc));

    }


Auth()
    {
    RPC2_Handle cid;
    int op, rc, uid;
    char name[100], buf[100];
    AuthInfo ainfo;
    RPC2_BoundedBS bbs;

    printf("Connection id? ");
    gets(buf);
    cid = atoi(buf);
    printf("Operation? (1 = Id, 2 = Name, 3 = Info, 4 = Quit) ");
    gets(buf);
    op = atoi(buf);
    switch(op)
        {
        case 1:
            printf("Name? ");
            gets(name);
            rc = AuthUserId(cid, name, &uid);
            if (rc == AUTHSUCCESS) printf("Id = %d\n", uid);
            else
                if (rc == AUTHFAILED) printf("Bogus user name\n");
                else printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            break;

        case 2:
            printf("Id? ");
            gets(buf);
```

```
            uid = atoi(buf);
            bbs.MaxSeqLen = sizeof(name);
            bbs.SeqLen = 0;
            bbs.SeqBody =  (RPC2_ByteSeq) name;
            rc = AuthUserName(cid, uid, &bbs);
            if (rc == AUTHSUCCESS) printf("Name = %s\n", bbs.SeqBody);
            else
                if (rc == AUTHFAILED) printf("Bogus user id\n");
                else printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            break;

        case 3:
            printf("Id? ");
            gets(buf);
            uid = atoi(buf);
            rc = AuthUserInfo(cid, uid, &ainfo);
            if (rc == AUTHSUCCESS) printf("Group = %d   Home = %s\n", ainfo.GroupId, ainfo.HomeDir);
            else
                if (rc == AUTHFAILED) printf("Bogus user id\n");
                else printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            break;

        case 4:
            rc = AuthQuit(cid);
            if (rc != AUTHSUCCESS)
                printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            RPC2_Unbind(cid);
            break;

        }

    }


Comp()
    {
    RPC2_Handle cid;
    int op, rc, x;
    char buf[100];

    printf("Connection id? ");
    gets(buf);
    cid = atoi(buf);
    printf("Operation? (1 = Square, 2 = Cube, 3 = Age, 4 = Quit) ");
    gets(buf);
    op = atoi(buf);
    switch(op)
        {
        case 1:
            printf("x? ");
            gets(buf);
```

```
            x = atoi(buf);
            rc = CompSquare(cid, x);
            if (rc > 0) printf("x**2 = %d\n", rc);
            else
                printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            break;

        case 2:
            printf("x? ");
            gets(buf);
            x = atoi(buf);
            rc = CompCube(cid, x);
            if (rc > 0) printf("x**3 = %d\n", rc);
            else
                printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            break;

        case 3:
            rc = CompAge(cid);
            if (rc > 0) printf("Age of connection = %d seconds\n", rc);
            else
                printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            break;

        case 4:
            rc = CompQuit(cid);
            if (rc < 0)
                printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            RPC2_Unbind(cid);
            break;

        }


    }
/*
RPC Initialization and Error handling
*/
InitRPC()
    {
    int mylpid = -1;

    assert(LWP_Init(LWP_VERSION, LWP_NORMAL_PRIORITY, &mylpid) == LWP_SUCCESS);
    assert (RPC2_Init(RPC2_VERSION, 0, NULL, 1,  -1, NULL) == RPC2_SUCCESS);
    }



iopen(){}
```

Server Code*authcomp_ srv.c*

```
#include <stdio.h>
#include <strings.h>
#include <sys/signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <assert.h>
#include <pwd.h>
#include <lwp.h>
#include <rpc2.h>
#include "auth.h"
#include "comp.h"


/*
This data structure provides per-connection info.  It is
created on every new connection and ceases to exist after AuthQuit().
*/
struct UserInfo
    {
    int Creation;       /* Time at which this connection was created */
    /* other fields would go here */
    };

int NewCLWP(), AuthLWP(), CompLWP();    /* bodies of LWPs */
void DebugOn(), DebugOff();   /* signal handlers */

main()
    {
    int mypid;

    signal(SIGEMT, DebugOn);
    signal(SIGIOT, DebugOff);

    InitRPC();
    LWP_CreateProcess(AuthLWP, 4096, LWP_NORMAL_PRIORITY, "AuthLWP", NULL, &mypid);
    LWP_CreateProcess(CompLWP, 4096, LWP_NORMAL_PRIORITY, "CompLWP", NULL, &mypid);
    LWP_WaitProcess(main);      /* sleep here forever; no one will ever wake me up */
    }

AuthLWP(p)
    char *p;    /* single parameter passed to LWP_CreateProcess() */
    {
    RPC2_RequestFilter  reqfilter;
    RPC2_PacketBuffer *reqbuffer;
    RPC2_Handle cid;
    int rc;
```

```
    char *pp;

    /*  Set filter to accept  auth requests on new or existing connections */
    reqfilter.FromWhom = ONESUBSYS;
    reqfilter.OldOrNew = OLDORNEW;
    reqfilter.ConnOrSubsys.SubsysId = AUTHSUBSYSID;

    while(1)
        {
        cid = 0;
        rc = RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL);
        if (rc < RPC2_WLIMIT)  HandleRPCError(rc, cid);
        rc = auth_ExecuteRequest(cid, reqbuffer);
        if (rc < RPC2_WLIMIT)  HandleRPCError(rc, cid);
        pp = NULL;
        if (RPC2_GetPrivatePointer(cid, &pp) != RPC2_SUCCESS || pp == NULL)
            RPC2_Unbind(cid);   /* This was almost certainly an AuthQuit() call */
        }
    }

CompLWP(p)
    char *p;    /* single parameter passed to LWP_CreateProcess() */
    {
    RPC2_RequestFilter  reqfilter;
    RPC2_PacketBuffer *reqbuffer;
    RPC2_Handle cid;
    int rc;
    char *pp;

    /* Set filter  to accept  comp requests on new or existing connections */
    reqfilter.FromWhom = ONESUBSYS;
    reqfilter.OldOrNew = OLDORNEW;
    reqfilter.ConnOrSubsys.SubsysId = COMPSUBSYSID;

    while(1)
        {
        cid = 0;
        rc = RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL);
        if (rc < RPC2_WLIMIT) HandleRPCError(rc, cid);

        rc = comp_ExecuteRequest(cid, reqbuffer);
        if (rc < RPC2_WLIMIT) HandleRPCError(rc, cid);

        pp = NULL;
        if (RPC2_GetPrivatePointer(cid, &pp) != RPC2_SUCCESS || pp == NULL)
            RPC2_Unbind(cid);   /* This was almost certainly an CompQuit() call */
        }

    }
```

```
/*
Bodies of Auth RPC routines
*/

S_AuthNewConn(cid, seType, secLevel, encType, cIdent)
    RPC2_Handle cid;
    RPC2_Integer seType, secLevel, encType;
    RPC2_CountedBS *cIdent;
    {
    struct UserInfo *p;

    p = (struct UserInfo *) malloc(sizeof(struct UserInfo));
    RPC2_SetPrivatePointer(cid, p);
    p->Creation = time(0);
    }


S_AuthQuit(cid)
/*
Get rid of user state; note that we do not do RPC2_Unbind() here, because this
        request itself has to complete.  The invoking server LWP therefore checks to see if this
        connection can be unbound.
*/
    {
    struct UserInfo *p;
    RPC2_GetPrivatePointer(cid, &p);
    assert(p != NULL); /* we have a bug then */
    free(p);
    RPC2_SetPrivatePointer(cid, NULL);
    return(AUTHSUCCESS);
    }


S_AuthUserId(cid, userName, userId)
    char *userName;
    int *userId;
    {
    struct passwd *pw;
    if ((pw = getpwnam(userName)) == NULL) return(AUTHFAILED);
    *userId = pw->pw_uid;
    return(AUTHSUCCESS);
    }

S_AuthUserName(cid, userId, userName)
    int userId;
    RPC2_BoundedBS *userName;
    {
    struct passwd *pw;
    if ((pw = getpwuid(userId)) == NULL) return(AUTHFAILED);
    strcpy(userName->SeqBody, pw->pw_name);
                /* we hope the buffer is big enough */
```

```
    userName->SeqLen = 1 + strlen(pw->pw_name);
    return(AUTHSUCCESS);
    }

S_AuthUserInfo(cid, userId, uInfo)
    int userId;
    AuthInfo *uInfo;
    {
    struct passwd *pw;
    if ((pw = getpwuid(userId)) == NULL) return(AUTHFAILED);
    uInfo->GroupId = pw->pw_gid;
    strcpy(uInfo->HomeDir, pw->pw_dir);
    return(AUTHSUCCESS);

    }




/*
Bodies of Comp RPC routines
*/
S_CompNewConn(cid, seType, secLevel, encType, cIdent)
    RPC2_Handle cid;
    RPC2_Integer seType, secLevel, encType;
    RPC2_CountedBS *cIdent;
    {
    struct UserInfo *p;

    p = (struct UserInfo *) malloc(sizeof(struct UserInfo));
    RPC2_SetPrivatePointer(cid, p);
    p->Creation = time(0);
    }

S_CompQuit(cid)
/*
Get rid of user state; note that we do not do RPC2_Unbind() here, because this
        request itself has to complete.  The invoking server LWP therefore checks to see if this
        connection can be unbound.
*/
    {
    struct UserInfo *p;
    RPC2_GetPrivatePointer(cid, &p);
    assert(p != NULL);  /* we have a bug then */
    free(p);
    RPC2_SetPrivatePointer(cid, NULL);
    return(0);
    }


S_CompSquare(cid, x)
    int x;
```

```
    {
    return(x*x);
    }

S_CompCube(cid, x)
    RPC2_Handle cid;
    int x;
    {
    return(x*x*x);
    }

S_CompAge(cid, x)
    RPC2_Handle cid;
    int x;
    {
    struct UserInfo *p;
    assert(RPC2_GetPrivatePointer(cid, &p) == RPC2_SUCCESS);
    return(time(0) - p->Creation);
    }

/*
 iopen() is a system call created at the ITC; put a dummy here for other sites
*/
iopen(){}


/*
RPC Initialization and Error handling
*/
InitRPC()
    {
    int mylpid = -1;
    RPC2_PortalIdent portalid, *portallist[1];
    RPC2_SubsysIdent subsysid;
    long rc;

    assert(LWP_Init(LWP_VERSION, LWP_NORMAL_PRIORITY, &mylpid) == LWP_SUCCESS);

    portalid.Tag = RPC2_PORTALBYINETNUMBER;
    portalid.Value.InetPortNumber = htons(AUTHPORTAL);
    portallist[0] = &portalid;
    rc = RPC2_Init(RPC2_VERSION, 0, portallist, 1, -1, NULL);
    if (rc != RPC2_SUCCESS) {
        fprintf(stderr, "RPC2_Init: failed with %ld\n", rc);
        exit(1);
    }
    subsysid.Tag = RPC2_SUBSYSBYID;
    subsysid.Value.SubsysId = AUTHSUBSYSID;
    assert(RPC2_Export(&subsysid) == RPC2_SUCCESS);
    subsysid.Value.SubsysId = COMPSUBSYSID;
    assert(RPC2_Export(&subsysid) == RPC2_SUCCESS);
```

```
    }


HandleRPCError(rCode, connId)
    int rCode;
    RPC2_Handle connId;
    {
    fprintf(stderr, "exserver: %s\n", RPC2_ErrorMsg(rCode));
    if (rCode < RPC2_FLIMIT && connId != 0) RPC2_Unbind(connId);
    }

void DebugOn()
    {
    RPC2_DebugLevel = 100;
    }

void DebugOff()
    {
    RPC2_DebugLevel = 0;
    }
```

Makefile for All Examples*Makefile*

```
HFILES = $(INCLDIR)/rpc2.h $(INCLDIR)/se.h  $(INCLDIR)/lwp.h

LIBS = -lrpc2 -lse -loldlwp

examples: authcomp_clnt authcomp_srv rtime_clnt rtime_srv rcat_clnt rcat_srv\
                multi_rtime_clnt multi_rcat_clnt

OCLIENT1 = rtime_clnt.o rtime.client.o
OSERVER1 = rtime_srv.o rtime.server.o

OCLIENT2 = rcat_clnt.o rcat.client.o
OSERVER2 = rcat_srv.o rcat.server.o

OCLIENT3 = authcomp_clnt.o  auth.client.o comp.client.o
OSERVER3 = authcomp_srv.o  auth.server.o comp.server.o

OCLIENT4 = multi_rtime_clnt.o rtime.client.o

OCLIENT5 = multi_rcat_clnt.o rcat.client.o


rtime_clnt: $(OCLIENT1)
        $(CC) $(CFLAGS) $(OCLIENT1) $(LIBS) -o rtime_clnt

rtime_srv: $(OSERVER1)
        $(CC) $(CFLAGS) $(OSERVER1) $(LIBS) -o rtime_srv
```

```
rcat_clnt: $(OCLIENT2)
        $(CC) $(CFLAGS) $(OCLIENT2) $(LIBS) -o rcat_clnt

rcat_srv: $(OSERVER2)
        $(CC) $(CFLAGS) $(OSERVER2) $(LIBS) -o rcat_srv

authcomp_clnt: $(OCLIENT3)
        $(CC) $(CFLAGS) $(OCLIENT3) $(LIBS) -o authcomp_clnt

authcomp_srv: $(OSERVER3)
        $(CC) $(CFLAGS) $(OSERVER3) $(LIBS) -o authcomp_srv

multi_rtime_clnt: $(OCLIENT4)
        $(CC) $(CFLAGS) $(OCLIENT4) $(LIBS) -o multi_rtime_clnt

multi_rcat_clnt: $(OCLIENT5)
        $(CC) $(CFLAGS) $(OCLIENT5) $(LIBS) -o multi_rcat_clnt


$(OCLIENT1) $(OSERVER1): rtime.h $(HFILES)

$(OCLIENT2) $(OSERVER2): rcat.h $(HFILES)

$(OCLIENT3) $(OSERVER3): auth.h comp.h $(HFILES)


auth.h auth.server.c auth.client.c: auth.rpc  $(RP2GEN)
        $(RP2GEN) auth.rpc

comp.h comp.server.c comp.client.c: comp.rpc  $(RP2GEN)
        $(RP2GEN) comp.rpc

rtime.h rtime.server.c rtime.client.c: rtime.rpc $(RP2GEN)
        $(RP2GEN) rtime.rpc

rcat.h rcat.server.c rcat.client.c: rcat.rpc $(RP2GEN)
        $(RP2GEN) rcat.rpc
```

In chapter 2, "The RP2Gen Stub Generator," we will see how the stub generator can create such files.

Chapter 3

The RP2Gen Stub Generator

3.1 Introduction

RP2GEN takes a description of a procedure call interface and generates stubs to use the RPC2 package, making the interface available on remote hosts. RP2GEN is designed to work with a number of different languages (C, FORTRAN 77, PASCAL), however, only the C interface is currently implemented.

RP2GEN also defines a set of external data representations for RPC types. These representations are defined at the end of this document in the section entitled **External Data Representations.** Any program wishing to communicate with a remote program using the RP2GEN semantics must obey these representation standards.

3.2 Usage

RP2GEN is invoked as follows:

```
rp2gen file
```

File is the file containing the description of the interface. Normally, these files have the extension *.rpc2*. RPGen creates three files named: *base*.client.c; *base*.server.c; and *base*.h, where *base* is the name of the file without the extension and the pathname prefix. Thus:

```
rp2gen samoan.rpc2
```

would yield the files: *samoan*.client.c; *samoan*.server.c; and *samoan*.h.

A person wanting to provide a package remotely writes his package with a normal interface. The client programmer writes his code to make normal calls on the interface. Then the client program is linked with:

```
ld ... base.client.o /usr/andrew/lib/librpc2.a ...
```

and the server program with

```
ld ...  base.server.o /usr/andrew/lib/librpc2.a ...
```

The server module provides a routine, the *ExecuteRequest* routine, that will decode the parameters of the request and make an appropriate call on the interface. (The routine is described below in the language interface sections.) The client module translates calls on the interface to messages that are sent via the RPC2 package. The `.h` file contains type definitions that RP2GEN generated from the type definitions in the input file, and definitions for the op-codes used by RP2GEN. This file, which is automatically included in the server and client files, may be included by any other module that needs access to these types.

3.3 Format of the Description File

In the syntax of a description file below, non-terminals are represented by *italic* names and literals are represented by **bold** strings.

In addition to the syntax above, text enclosed in /* and */ is treated as a comment and ignored. Appearances of an include statement will be replaced by the contents of the specified file. All numbers are in decimal and may be preceded by a single hyphen: -, character. Identifiers follow C syntax except that the underline character: _, may not begin an identifier. (Note that a particular language interface defines what identifiers may actually be used in various contexts.)

The following are reserved words in RP2GEN: **server**, **client**, **prefix**, **subsystem**, **timeout**, **typedef**, **rpc2_struct**, **rpc2_enum**, **in** and **out**. Case is ignored for reserved words, so that, for example, **subsystem** may be spelled as **SubSystem** if desired. Case is not ignored, however, for identifiers. Note that the predefined type names (RPC2_Integer , RPC2_Byte , etc.) are identifiers and must be written exactly as given above.

The *prefixes* may be used to cause the names of the procedures in the interface to be prefixed with a unique character string. The line:

```
Server Prefix "test";
```

will cause the server file to assume that the name of the server interface procedure *name* is test_*name*. Likewise, the statement:

```
Client Prefix "real";
```

affects the client interface. This feature is useful in case it is necessary to link the client and server interfaces together. Without this feature, name conflicts would occur.

The *header_line* defines the name of this subsystem. The subsystem name is used in generating a unique for the *execute request* routine.

The *default_timeout* is used in both the server and client stubs. Both are specified in seconds. Zero is interpreted as an infinite timeout value. The value specifies the timeout value used on RPC2_MakeRPC () and RPC2_SendResponse () calls in the client and server stubs respectively. The timeout parameter may be overriden for individual procedures by specifying a *timeout_override*. Note that the timeouts apply to each individual Unix blocking system call, not to the entire RPC2 procedure.

file ::= prefixes header_line default_timeout decl_or_proc_list
prefixes ::= empty | prefix | prefix prefix
prefix ::= **Server Prefix** string **;** | **Client Prefix** string **;**
header_line ::= **Subsystem** subsystem_name **;**
subsystem_name ::= string
string ::= **"** zero_or_more_ascii_chars **"**
default_timeout ::= **Timeout (** id_number **) ;** | empty
decl_or_proc_list ::= decl_or_proc | decl_or_proc decl_or_proc_list
decl_or_proc ::= include | define | typedef | procedure_description
include ::= #**include** file_name
define ::= #**define** identifier number
typedef ::= **typedef** rpc2_type identifier array_spec **;**
rpc2_type ::= type_name | rpc2_struct | rpc2_enum
type_name ::= **RPC2_Integer** | **RPC2_Unsigned** | **RPC2_Byte** | **RPC2_String** | **RPC2_Cou**
rpc2_struct ::= **RPC2_Struct** { field_list }
field_list ::= field | field field_list
field ::= type_name identifier_list **;**
identifier_list ::= identifier | identifier **,** identifier_list
rpc2_enum ::= **RPC2_Enum** { enum_list }
enum_list ::= enum **,** enum_list | enum
enum ::= identifier = number
array_spec ::= empty | **[** id_number**]**
id_number ::= number | identifier
procedure_description ::= proc_name **(** formal_list **)**
timeout_override new_connection **;**
proc_name ::= identifier
formal_list ::= empty | formal_parameter | formal_parameter **,** formal_list
formal_parameter ::= usage type_name parameter_name
usage ::= **IN** | **OUT** | **IN OUT**
parameter_name ::= identifier
timeout_override ::= **Timeout (** id_number**)** | empty
new_connection ::= **NEW_CONNECTION** | empty
empty ::=

The *new_connection* is used to designate at most one server procedure that will be called when the subsystem receives the initial RPC2 connection. The new connection procedure must have 4 arguments in the following order with the following usages and types:

```
(IN RPC2_Integer: SideEffectType,
 IN RPC2_Integer: SecurityLevel,
 IN RPC2_Integer: EncryptionType,
 IN RPC2_CountedBS: ClientIdent,
```

where `SideEffectType`, `SecurityLevel`, `EncryptionType`, and `ClientIdent` have the values that were specified on the clients call to RPC2_Bind. Note that RP2Gen will automatically perform an RPC2_Enable call at the end of this routine. If no new connection procedure is specified, then the call to the *execute request* routine with the initial connection request will return RPC2_FAIL.

The *usage* tells whether the data for the parameter is to be copied in, copied out, or copied in both directions. The *usage* and *type_name* specifications together tell how the programmer should declare the parameters in the server code.

3.3.1 Example 1: Common Definitions for Coda File System

```
/*
 * Include file common to callback.rpc2, vice.rpc2 and res.rpc2
 */

typedef RPC2_Unsigned   VolumeId;
typedef VolumeId               VolId;
typedef RPC2_;Unsigned  VnodeId;
typedef RPC2_;Unsigned  Unique;

typedef RPC2_Struct
        {
        VolumeId        Volume;
        VnodeId         Vnode;
        Unique          Unique;
        } ViceFid;

typedef RPC2_Struct
        {
        RPC2_Unsigned   Host;
        RPC2_Unsigned   Uniquifier;
        } ViceStoreId;

typedef RPC2_Struct
        {
        RPC2_Integer    Site0;
        RPC2_Integer    Site1;
        RPC2_Integer    Site2;
        RPC2_Integer    Site3;
        RPC2_Integer    Site4;
```

```
        RPC2_Integer    Site5;
        RPC2_Integer    Site6;
        RPC2_Integer    Site7;
        } ViceVersionArray;

typedef RPC2_Struct
        {
        ViceVersionArray        Versions;
        ViceStoreId     StoreId;
        RPC2_Unsigned   Flags;
        } ViceVersionVector;

typedef RPC2_Unsigned   UserId;
typedef RPC2_Unsigned   FileVersion;
typedef RPC2_Unsigned   Date;
typedef RPC2_Integer    Rights;

typedef RPC2_Enum
        {
        Invalid = 0,
        File = 1,
        Directory = 2,
        SymbolicLink = 3
        } ViceDataType;

typedef RPC2_Enum
        {
        NoCallBack = 0,
        CallBackSet = 1,
        BidFidReleased = 3
        } CallBackStatus;


typedef RPC2_Struct
        {
        RPC2_Unsigned   InterfaceVersion;
        ViceDataType    VnodeType;
        RPC2_Integer    LinkCount;
        RPC2_Unsigned   Length;
        FileVersion     DataVersion;
        ViceVersionVector       VV;
        Date            Date;
        UserId          Author;
        UserId          Owner;
        CallBackStatus  CallBack;
        Rights          MyAccess;
        Rights          AnyAccess;
        RPC2_Unsigned   Mode;
        VnodeId         vparent;
        Unique          uparent;
        } ViceStatus;
```

3.3.2 Example 2: The Coda Resolution Subsystem Interface

```
/* res.rpc2
 * Defines the resolution subsystem interface
 *
 * Created Puneet Kumar, June 1990
 */
server prefix "RS";
client prefix "Res";

Subsystem "resolution";

#define RESPORTAL       1361
#define RESOLUTIONSUBSYSID 5893

/*
Return codes from the servers on resolution subsystem
*/
#define RES_FAILURE     -512
#define RES_SUCCESS     0
#define RES_TIMEDOUT    -513
#define RES_NOTRUNT     -514
#define RES_BADOPLIST   -515

#include "vcrcommon.rpc2"

typedef RPC2_Struct
        {
        RPC2_Integer    status;
        RPC2_Unsigned   Author;
        RPC2_Unsigned   Owner;
        RPC2_Unsigned   Date;
        RPC2_Unsigned   Mode;
        } ResStatus;

typedef RPC2_Struct
        {
        RPC2_Integer            LogSize;
        ViceVersionVector       VV;
        } ResVolParm;

typedef RPC2_Enum
        {
        FetchStatus = 0,
        FetchSData = 1
        } ResFetchType;

typedef RPC2_Enum
        {
        ResStoreStatus = 0,
        ResStoreData = 1
```

```
        } ResStoreType;

COP2     (IN ViceStoreId StoreId,
                IN ViceVersionVector UpdateSet);

NewConnection (IN RPC2_Integer SideEffectType,
                IN RPC2_Integer SecurityLevel,
                IN RPC2_Integer EncryptionType,
                IN RPC2_CountedBS ClientIdent)
                NEW_CONNECTION;

ForceFile    (IN ViceFid Fid,
                IN ResStoreType Request,
                IN RPC2_Integer Length,
                IN ViceVersionVector VV,
                IN ResStatus Status,
                IN OUT SE_Descriptor BD);

LockAndFetch (IN ViceFid Fid,
                IN ResFetchType Request,
                OUT ViceVersionVector VV,
                OUT RPC2_Integer logsize);

UnlockVol    (IN VolumeId Vid);

MarkInc      (IN ViceFid Fid);

FetchFile (IN ViceFid Fid,
                IN RPC2_Unsigned PrimaryHost,
                OUT ResStatus Status,
                IN OUT SE_Descriptor BD);

ForceDirVV (IN ViceFid Fid,
                IN ViceVersionVector VV);

DoForceDirOps (IN ViceFid Fid,
                IN ViceStatus status,
                IN OUT RPC2_CountedBS AccessList,
                OUT RPC2_Integer rstatus,
                IN OUT SE_Descriptor sed);

GetForceDirOps  (IN ViceFid Fid,
                OUT ViceStatus status,
                IN OUT RPC2_CountedBS AccessList,
                IN OUT SE_Descriptor sed);

FetchLog (IN ViceFid Fid,
                OUT RPC2_Integer logsize,
                IN OUT SE_Descriptor sed);

DirResPhase2 (IN ViceFid Fid,
```

```
                 IN ViceStoreId logid,
                 OUT ViceStatus status,
                 IN RPC2_BoundedBS pbinc);

DirResPhase1 (IN ViceFid Fid,
                 IN RPC2_Integer size,
                 IN OUT ViceStatus status,
                 IN OUT RPC2_BoundedBS piggyinc,
                 IN OUT SE_Descriptor sed);

DirResPhase3 (IN ViceFid Fid,
                 IN ViceVersionVector UpdateSet,
                 IN OUT SE_Descriptor sed);
```

3.4 Command Line Parameters

In addition, several command line flags are available to modify the behavior of rp2gen :

-c `file`

Specify the name of the client .c file.

-s `file`

Specify the name of the server .c file.

-h `file`

Specify the name of the header file.

-m `file`

Specify the name of the MultiRPC stub file.

-I `path`

Additional path to look for included files.

-e,-neterrors

Translate system-specfic error codes to caller's system-specfic codes.

-cplusplus

Generate C++ compatible code in .cc files.

3.5 The C Interface

This section describes the **C interface** generated by RP2GEN. The following table shows the relationship between RP2GEN parameter declarations and the corrseponding C parameter declarations.

In all cases it is the caller's responsibility to allocate storage for all parameters. This means that for `IN` and`IN OUT` parameters of a non-fixed type, it is the callee's responsibility to ensure that the value to be copied back to the caller does not exceed the storage allocated by the callee.

RPC2 Type	**in**	**out**	**in out>**
RPC2_Integer	long	long *	long *
RPC2_Unsigned	unsigned long	unsigned long *	unsigned long *
RPC2_Byte	unsigned char	unsigned char *	unsigned char *
RPC2_String	unsigned char *	unsigned char *	unsigned char *
RPC2_CountedBS	RPC2_CountedBS *	RPC2_CountedBS *	RPC2_CountedBS *
RPC2_BoundedBS	RPC2_BoundedBS *	RPC2_BoundedBS *	RPC2_BoundedBS *
RPC2_EncryptionKey	RPC2_EncryptionKey	RPC2_EncryptionKey *	RPC2_EncryptionKey *
SE(Descriptor)	*illegal*	*illegal*	
SE(Descriptor) *			
RPC2_Enum *name*	*name*	*name* *	*name* *
RPC2_Struct *name*	*name* *	*name* *	*name* *
RPC2_Byte *name[...*	*name*	*name*	*name*

Table 3.1: RP2Gen representation of parameters

The caller must call an RPC2 procedure with an initial implicit argument of type RPC2_Handle that indicates the destination address(es) of the target process(es). The callee must declare the C routine that corresponds to an RPC2 procedure with an initial implicit argument of type RPC2_Handle. Upon invocation, this argument will be bound to the address of a handle that indicates the address of the caller.

RP2GEN also generates a routine that serves to decode an RPC2 request. The name of this routine is "*subsystem_ name_ ExecuteRequest*", and it is invoked as follows:

```
int subsystem_name_ExecuteRequest (cid, Request, bd) RPC2_Handle cid;
RPC2_PacketBuffer *Request; SE_Descriptor *bd;
```

This routine will unmarshall the arguments and call the appropriate interface routine. The return value from this routine will be the return value from the interface routine.

The client program is responsible for actually making the connection with the server and must pass the connection id as an additional parameter (the first) on each call to the interface.

3.6 External Data Representations

This section defines the external data representation used by RP2GEN, that is, the representation that is sent out over the wire. Each item sent over on the wire is required to be a multiple of 4 (8-bit) bytes. (Items are padded as necessary to achieve this constraint.) The bytes of an item are numbered 0 through n-1 (where $n \bmod 4 = 0$). The bytes are read and written such that byte m always precedes byte $m+1$.

RPC2_Integer

An RPC2_Integer is a 32-bit item that encodes an integer represented in twos complement notation. The most significant byte of the integer is 0, and the least significant byte is 3.

RPC2_Unsigned

An RPC2_Unsigned is a 32-bit item that encodes an unsigned integer. The most significant byte of the integer is 0, the least significant byte is 3.

RPC2_Byte

An RPC2_Byte is transmitted as a single byte followed by three padding bytes.

RPC2_String

An RPC2_String is a C-style null-terminated character string. It is sent as an RPC2_Integer indicating the number of characters to follow, not counting the null byte, which is, however, sent. This is followed by bytes representing the characters (padded to a multiple of 4), where the first character (i.e., farthest from the null byte) is byte 0. An RPC2_String of length 0 is representing by sending an RPC2_Integer with value 0, followed by a 0 byte and three padding bytes.

RPC2_CountedBS

An RPC2_CountedBS is used to represent a byte string of arbitrary length. The byte string is not terminated by a null byte. An RPC2_CountedBS is sent as an RPC2_Integer representing the number of bytes, followed by the bytes themselves (padded to a multiple of 4 . The byte with the lowest address is sent as byte 0.

RPC2_BoundedBS

An RPC2_BoundedBS is intended to allow you to remotely play the game that C programmers play: allocate a large buffer, fill in some bytes, then call a procedure that takes this buffer as a parameter and replaces its contents by a possibly longer sequence of bytes. An RPC2_BoundedBS is transmitted as two RPC2_Integer s representing the maximum and current lengths of the byte strings. This is followed by the bytes representing the contents of the buffer (padded to a multiple of 4). The byte with the lowest address is byte 0.

RPC2_EncryptionKey

An RPC2_EncryptionKey is used to transmit an encryption key (surprise!). A key is sent as a sequence of RPC2_KEYSIZE bytes, padded to a multiple of 4. Element 0 of the array is byte 0.

SE(Descriptor)

Objects of type @SE(Descriptor) are never transmitted.

RPC2_Struct

An RPC2_Struct is transmitted as a sequence of items representing its fields. The fields are sent in textual order of declaration (i.e., from left to right and top to bottom). Each field is sent using, recursively, its RPC2 representation.

RPC2_Enum

An RPC2_Enum has the same representation has an RPC2_Integer , and the underlying integer used by the compiler is transmitted as the value of an RPC2_Enum. (Note that in C, this underlying value may be specified by the user. This is recommended practice.)

Array

> The total number of bytes transmitted for an array must be a multiple of 4. However, the number of bytes sent for each element depends on the type of the element.

Currently, only arrays of RPC2_Byte are defined. The elements of such an array are each sent as a single byte (no padding), with array element, *n-1,* preceding element, *n*.

Chapter 4

RPC2 Runtime System

The purpose of this section is to describe the calls provided by the RPC2 library. These calls deal with contiguous *Packet Buffers* each of which consists of a:

Prefix.

which is of fixed length, and is used internally by the runtime system. It is NOT transmitted.

Header.

which is also of fixed length, and whose format is understood by the runtime system. The`opcode` associated with the RPC, sequencing information, and the completion code returned by the remote site are the kinds of information found here.

Body.

of arbitrary size. It is NOT interpreted by the runtime system, and is used to transmit the input and output parameters of an RPC.

Appendix **??** (XXX) contains the header file, *rpc2.h*, that defines the packet format as well as numerous other definitions used in the call descriptions that follow. Of course, the actual header files are the authoritative source of these definitions, and will be more up-to-date than this manual.

The following sections define the RPC2 runtime calls. Some of these calls are not relevant to you if you use RP2Gen. Others, such as the initialization and export calls, are pertinent to all users of RPC2.

4.1 Client-related Calls

4.1.1 ROC2_NewBinding – *Create a new connection*

Call:

int RPC2_NewBinding(**in** RPC2_HostIdent*Host;**in** RPC2_PortalIdent *Portal;**in** RPC2_SussysIdent *Subsys;**in** RPC2_BindParms *Bparms;**out** RPC2_Handle *ConnHandle).

Parameters:

Host

The identity of the remote host on which the server to be contacted is located. This may be specified as a string name or as an Internet address. In the former case the RPC runtime system will do the necessary name resolution.

Portal

An identification of the server process to be contacted at the remote site. Portals are unique on a given host. A portal may be specified as a string name or as an Internet port value. In the former case the RPC runtime system will do the necessary name to port number conversion. Support for other kinds of portals (such as Unix domain) may be available in future.

Subsys

Which of the potentially many subsystems supported by the remote server is desired. May be specified as a number or as a name. In the latter case, the RPC runtime system will do the translation from name to number.

Bparms

Type RPC2_BindParms is a struct containing 5 fields. The first field SecurityLevel is one of the constants RPC2_OPENKIMONO, RPC2_ONLYAUTHENTICATE, RPC2_HEADERSONLY or RPC2_SECURE. The second field, EncryptionType, describes what kind of encryption to be used on this connection. For example, RPC2_XOR, RPC2_DES, etc. is ignored if, SecurityLevel, is RPC2_OPENKIMONO. The bind will fail if the remote site does not support the requested type of encryption. The third field, SharedSecret, is an encryption key known by the callback procedure on the server side to be uniquely associated with, ClientIdent. Used by the RPC runtime system in the authentication handshakes. May be NULL if, SecurityLevel, is RPC2_OPENKIMONO. The fourth field, ClientIdent, provides the information for the server to uniquely identify this client and to obtain, SharedKey. Not interpreted by the RPC runtime system. Only the GetKeys callback procedure on the server side need understand the format of ClientIdent. May be NULL if SecurityLevel is RPC2_OPENKIMONO. The fifth field, SideEffectType, describes what kind of side effects are to be associated with this connection. The only side effects intially supported are bulk-transfers of files, identified by type, SMARTFTP. May be 0 if no side effects are ever to be attempted on this connection.

ConnHandle

An unique integer returned by the call, identifying this connection. This is not necessarily a small-valued integer.

Completion Codes:

RPC2_SUCCESS

All went well.

RPC2_NOBINDING

The specified host, server or subsystem could not be contacted.

RPC2_WRONGVERSION

The client and server runtime systems are incompatible. Note that extreme incompatibilty may result in the server being unable to respond even with this error code. In such a case, the server will appear to be down, resulting in an RPC_NOBINDING return code.

RPC2_OLDVERSION

This is a warning. The, RPC2_VERSION, values on client and server sides are different. Normal operation is still possible, but one of you is running an obsolete version of the runtime system. You should obtain the latest copy of the RPC runtime system and recompile your code.

RPC2_NOTAUTHENTICATED

A SecurityLevel other than, RPC2_OPENKIMONO, was specified, and the server did not accept your credentials.

RPC2_SEFAIL1

The associated side effect routine indicated a minor failure. The connection is established, and usable.

RPC2_SEFAIL2

The associated side effect routine indicated a serious failure. The connection is not established.

RPC2_FAIL

Some other mishap occurred.

Description:

Creates a new connection and binds to a remote server on a remote host. The subsystem information is passed on to that server to alert it to the kind of remote procedure calls that it may expect on this connection../

A client-server version check is performed to ensure that the runtime systems are compatible. Note that there are really two version checks. One is for the RPC network protocol and packet formats, and this must succeed. The other check reports a warning if you have a different RPC runtime system from the server. You may also wish to do a higher-level check, to ensure that the client and server application code are compatible../

The SecurityLevel parameter determines the degree to which you can trust this connection. If RPC2_OPENKIMONO is specified, the connection is not authenticated and no encryption is done on future requests and responses. If RPC2_ONLYAUTHENTICATE is specified, an authentication handshake is done to ensure that the client and the server are who they claim to to be (the fact that the server can find SharedSecret from ClientIdent is assumed to be proof of its identity. If RPC2_SECURE is specified, the connection is authenticated and all future transmissions on it are encrypted using a session key generated during the authentication handshake. RPC2_HEADERSONLY is similar to RPC2_SECURE, except that only RPC headers are encrypted.

The kind of encryption used is specified in EncryptionType. The remote site must specify an RPC2_GetRequest with an EncryptionTypeMask that includes this encryption type.

4.1.2 RPC2_Bind – *Obsolete: use RPC2_NewBinding*

Call:

int RPC2_Bind(**in** long SecurityLevel, **in** long EncryptionType, **in** RPC2_HostIdent *Host, **in** RPC2_PortalIdent *Portal, **in** RPC2_SubsysIdent *Subsys, **in** long SideEffectType, **in** RPC2_CountedBS *ClientIdent, **in** RPC2_EncryptionKey *SharedSecret, **out** RPC2_Handle *ConnHandle.

Parameters:

N/A

Completion Codes:

None.

Description:

Obsolete.

4.1.3 RPC2_MakeRPC – *Make a remote procedure call (with possible side-effect)*

Call:

int RPC2_MakeRPC(**in** RPC2_Handle ConnHandle, **in** RPC2_PacketBuffer *Request, **in** Descriptor *SDesc, **out** RPC2_PacketBuffer **Reply, **in** struct timeval *Patience, **in** long EnqueueRequest.

Parameters:

ConnHandle

Identifies the connection on which the call is to be made

Request

A properly formatted request buffer.

SDesc

A side effect descriptor with local fields filled in. May be NULL if no side effects will occur as a result of this call.

Reply

On return, it will point to a response buffer holding the response from the server. You should free this buffer when you are done with it.

Patience

Maximum time to wait for remote site to respond. A NULL pointer indicates infinite patience.

EnqueueRequest.

Specifies whether the caller should be blocked if ConnHandle is already servicing an RPC request from some other lwp. If this variable is 1 the caller is blocked. Otherwise a return code of RPC2_CONNBUSY is returned.

Completion Codes:

RPC2_SUCCESS

All went well.

RPC2_NOCONNECTION

ConnHandle does not refer to a valid connection.

RPC2_TIMEOUT

A response was not received soon enough. Occurs only if the Patience parameter was non-NULL.

RPC2_SEFAIL1

The associated side effect resulted in a minor failure. Future calls on this connection will still work.

RPC2_SEFAIL

The associated side effect resulted in a serious failure. Future calls on this connection will fail.

RPC2_DEAD

The remote site has been deemed dead or unreachable. Note that this is orthogonal to an RPC2_TIMEOUT return code.

RPC2_NAKED

The remote site sent an explicit negative acknowledgement. This can happen if that site thought you were dead, or if someone at that site unbound your connection.

RPC2_CONNBUSY

EnqueueRequest specified 0 and ConnHandle is currently servicing a call. Try again later. .

RPC2_FAIL

Other assorted calamities, such as attempting to use a connection already declared broken.

Description:

The workhorse routine, used to make remote calls after establishing a connection. The call is sequential and the calling lwp is blocked until the call completes. The associated side effect, if any, is finished before the call completes. The listed completion codes are from the local RPC stub. Check the RPC2_ReturnCode fields of the reply and the status fields of SDesc to see what the remote site thought of your request. Without an explicit timeout interval the remote site can take as long as it wishes to perform the requested operation and associated side effects. The RPC protocol checks periodically to ensure that the remote site is alive. If an explicit Patience timeout interval is specified, the call must complete within that time.

4.1.4 RPC2_MultiRPC – *Make a collection of remote procedure calls*

Calls:

int RPC2_MultiRPC(**in** int HowMany, **in** RPC2_Handle ConnHandleList[ ], **out** RPC2_Integer *RCList, **in out** RPC2_Multicast *MCast, **in** RPC2_PacketBuffer *Request, **in** Descriptor SDescList[ ], **in** long (*UnpackMulti (), **in out** ARG(INFO) *ArgInfo, **in** struct timeval *Patience.

Parameters:

HowMany

How many servers to contact.

ConnHandleList.

Array of length HowMany, containing the handles of the connections on which calls are to be made.

Request

A properly formatted request buffer.

RCList

Array of length HowMany, into which RPC2 will place return codes for each of the connections specified in ConnHanleList. May be specified as NULL if return codes will not be examined.

MCast

Pointer to multicast sturcture. Set to NULL for now.

SDescList

Array of length HowMany, containing side effect descriptors for each of the connections specified in ConnHandleList.

UnpackMulti

Pointer to unpacking routine called by RPC2 when each server response as received. If RP2Gen is used, this will be supplied by MRPC(MakeMulti. Otherwise, it must be supplied by the client.

ArgInfo

A pointer to a structure containing argument information. This structure is not examined by RPC2; it is passed untouched to UnpackMulti. If RP2Gen is used, this structure will be supplied by MRPC(MakeMulti. Otherwise, it can be used to pass any structure desired by the client or supplied as NULL.

Patience

Maximum time to wait for remote sites to respond. A NULL pointer indicates infinite patience as long as RPC2 believes that the server is alive. Note that this timeout value is orthogonal to the RPC2 internal timeout for determining connection death.

Completion Codes:

RPC2_SUCCESS.

All servers returned successfully, or all servers until client-initiated abort returned successfully. Individual server response information is supplied via UnpackMulti to the user handler routine supplied in the ArgInfo structure.

RPC2_TIMEOUT.

The user specified timeout expired before all the servers responded.

RPC2_FAIL.

Something other than SUCCESS or TIMEOUT occurred. More detailed information is supplied via UnpackMulti to the user handler routine supplied in the ArgInfo structure.

Description:

Logically identical to iterating through ConnHandleList and making RPC2_MakeRPC calls to each specified connection using Request as the request block, but this call will be considerably faster than explicit iteration. The calling lightweight process blocks until either the client requests that the call abort or one of the following is true about each of the connections specified in ConnHandleList: a reply has been received, a hard error has been detected for that connection, or the specified timeout has elapsed.

The ArgInfo structure exists to supply argument packing and unpacking information in the case where RP2Gen is used. Since its value is not examined by RPC2, it can contain any pointer that a non-RP2Gen generated client wishes to supply.

Similarly, UnpackMulti will point to a specific unpacking routine in the RP2Gen case. If the RP2Gen interface is not used, you should assume that the return codes of the supplied routine must conform to the specifications in section 18.3.3 (XXX).

Side effects are supported as in the standard RPC2 case except that the client must supply a separate **SE[Descriptor]** for each connection. The format for the **SE[Descriptor]**argument is described in Section 6.3 (XXX). It will often be useful to supply connection specific information such as unique file names in the **SE[Descriptor]**.

4.2 Server-related RPC Calls

4.2.1 RPC2_Export – *Indicate willingness to accept calls for a subsystem*

Call:

int RPC2_Export(**in** *RPC2_SubsysIdent *Subsys*)

Parameters:

Subsys

Specifies a subsystem that will be henceforth recognized by this server. This is either an integer or a symbolic name that can be translated to the unique integer identifying this subsystem.

Completion Codes:

RPC2_SUCCESS

All went well.

RPC2_DUPLICATESERVER

Your have already exported Subsys.

RPC2_BADSERVER

Subsys is invalid.

RPC2_FAIL

Something else went wrong.

Description:

Sets up internal tables so that when a remote client performs an RPC2_Bind(operation specifying this host-portal-subsystem triple, the RPC runtime system will accept it. A server may declare itself to be serving more than one subsystem by making more than one RPC2_Export calls.

4.2.2 RPC2_DeExport – *Stop accepting new connections for one or all subsystems*

Call:

int RPC2_Export(**in** RPC2_SubsysIdent *Subsys)

Parameters:

Subsys.

Specifies the subsystem to be deexported. This is either an integer or a symbolic name that can be translated to the unique integer identifying this subsystem. A value of NULL deexports all subsystems.

Completion Codes:

RPC2_SUCCESS

All went well.

RPC2_BADSERVER

Subsys is not a valid subsystem, or has not been previously exported.

RPC2_FAIL

Something else went wrong.

Description:

After this call, no new connections for subsystem Subsys will be accepted. The subsystem may, however, be exported again at a later time. Note that existing connections are not broken by this call.

4.2.3 RPC2_GetRequest – *Wait for an RPC request or a new connection*

Call:

int RPC2_GetRequest(**in**RPC2_RequestFilter *Filter, **out** RPC2_Handle *ConnHandle, **out** RPC2_PacketBuffer **Request, **in** struct timeval *Patience, **in** long (*GetKeys) (), **in** long EncryptionTypeMask, **in** long (*AuthFail) ())

Parameters:

Filter

A filter specifying which requests are acceptable. See description below.

ConnHandle

Specifies the connection on which the request was received.

Request

Value ignored on entry. On return, it will point to a buffer holding the response from the client. Free this buffer after you are done with it.

Patience

A timeout interval specifying how long to wait for a request. If NULL, infinite patience is assumed.

GetKeys

Pointer to a callback procedure to obtain authentication and session keys. See description below. May be NULL if no secure bindings to this server are to be accepted.

EncryptionTypeMask

A bit mask specifying which types of encryption is supported. Binds from clients who request an encryption type not specified in this mask will fail.

AuthFail

Pointer to a callback procedure to be called when an authentication failure occurs. See description below. May be NULL if server does not care to note such failures.

Completion Codes:

RPC2_SUCCESS

I have a request for you in Request. New connections result in a fake request.

RPC2_TIMEOUT

Specified time interval expired.

RPC2_BADFILTER

A nonexistent connection or subsystem was specified in Filter.

RPC2_SEFAIL1

The associated side effect routine indicated a minor failure. Future calls on this connection will still work.

RPC2_SEFAIL2

The associated side effect routine indicated a serious failure. Future calls on this connection will fail too.

RPC2_DEAD

You were waiting for requests on a specific connection and that site has been deemed dead or unreachable.

RPC2_FAIL

Something irrecoverable happened.

Description:

The call blocks the calling lightweight process until a request is available, a new connection is made, or until the specified timeout period has elapsed. The Filter parameter allows a great deal of flexibility in selecting precisely which calls are acceptable. New connections result in a fake request with a body of type RPC2_NewConnection. Do not try to do a RPC2_SendResponse to this call. All other RPC2_GetRequest calls should be eventually matched with a corresponding RPC2_SendResponse call.

The fields of RPC2_NewConnection are self-explanatory. Note that you must invoke RPC2_Enable() after you have handled the new connection packet for further requests to be visible. If you are using RP2Gen, this is done for you automatically by the generated code that deals with new connections.

The callback procedure for key lookup should be defined as follows

```
long GetKeys (in RPC2_CoundedBS *ClientIdent, out RPC2_EncryptionKey *IdentKey,
out RPC2_EncryptionKey *SessionKey)
```

GetKeys (will be called at some point in the authentication handshake. It should return 0 if ClientIdent is successfully looked up, and -1 if the handshake is to be terminated. It should fill IdentKey with the key to be used in the handshake, and SessionKey with an arbitrary key to be used for the duration of this connection. You may, of course, make SessionKey the same as IdentKey.

The callback procedure for noting authentication failure should be defined as follows:

```
long AuthFail (in RPC2_CoundedBS *ClientIdent, in RPC2_Integer EncryType, in
RPC2_HostIdent *PeerHost, in RPC2_PortalIdent *PeerPortal)
```

AuthFail ()will be called after an RPC2_NOTAUTHENTICATED packet has been sent to the client. The parameters give information about the client who was trying to authenticate himself, the type of encryption requested, and the site from which the RPC2_Bind(was attempted. The callback procedure will typically record this in a log file somewhere.

4.2.4 RPC2_Enable – *Allow servicing of requests on a new connection*

Call:

*int32_t RPC2_Enable(***in** *RPC2_Handle ConnHandle)*

Parameters:

ConnHandle.

Which connection is to be enabled.

Completion Codes:

RPC2_SUCCESS

Enabled the connection.

RPC2_NOCONNECTION

A bogus connection was specified.

Description:

Typically invoked by the user at the end of his NewConnection routine, after setting up his higher-level data structures appropriately. Until a connection is enabled, RPC2 guarantees that no requests on that connection will be returned in a RPC2_GetRequest call. Such a request from a client will, however, be held and responded to with RPC2_BUSY signals until the connection is enabled. This call is present primarily to avoid race hazards in higher-level connection establishment. Note that RP2Gen automatically generates this call after a NewConnection routine.

4.2.5 RPC2_SendResponse – *Respond to a request from my client*

Call:

int32_t RPC2_SendResponse(**in** RPC2_Handle ConnHandle, **in** RPC2_PacketBuffer *Reply)

Parameters:

ConnHandle

hich connection the response is to be sent on.,

Reply

A filled-in buffer containing the reply to be sent to the client.

Completion Codes:

RPC2_SUCCESS

I sent your response.

RPC2_NOTWORKER

You were not given a request to service.

RPC2_SEFAIL1

The associated side effect routine indicated a minor failure. Future calls on this connection will still work.

RPC2_SEFAIL2

The associated side effect routine indicated a serious failure. Future calls on this connection will fail too.

RPC2_FAIL

Some irrecoverable failure happened.

Description:

Sends the specified reply to the caller. Any outstanding side effects are completed before Reply is sent. Encryption, if any, is done in place and will clobber the Reply buffer. Note that this call returns immediately after sending the reply; it does not wait for an acknowledgement from the client.

4.2.6 RPC2_InitSideEffect – *Initiate side effect*

Call:

int32_t RPC2_InitSideEffectin RPC2_Handle ConnHandle, ***in*** **SDesc)*

Parameters:

ConnHandle

The connection on which the side effect is to be initiated.,

SDesc

A filled-in side effect descriptor.

Completion Codes:

RPC2_SUCCESS

The side effect has been initiated.

RPC2_NOTSERVER

Only one side effect is allowed per RPC call. This has to be initiated between the GetRequest and SendResponse of that call. You are violating one of these restrictions.

RPC2_SEFAIL1

The associated side effect routine indicated a nonfatal failure. Future calls on this connection will work.

RPC2_SEFAIL2

The associated side effect routine indicated a serious failure. Future calls on this connection will fail too.

RPC2_FAIL

Other assorted calamities,

Description:

Initiates the side effect specified by SDesc on ConnHandle. The call does not wait for the completion of the side effect. If you need to know what happened to the side effect, do a RPC2_CheckSideEffect call with appropriate flags.

4.2.7 RPC2_CheckSideEffect – *Check progress of side effect*

Call: *int32_t RPC2_ CheckSideEffect*(in RPC2_Handle ConnHandle, inout *SDesc, in long Flags)

Parameters:

ConnHandle

The connection on which the side effect has been initiated.

SDesc

The side effect descriptor as it was returned by the previous RPC2_InitSideEffect or RPC2_CheckSideEffect call on ConnHandle. On output, the status fields are filled in.

Flags.

Specifies what status is desired. This call will block until the requested status is available. This is a bit mask, with RPC2_GETLOCALSTATUS and RPC2_GETREMOTESTATUS bits indicating local and remote status. A Flags value of 0 specifies a polling status check: no blocking will occur and the currently known local and remote status will be returned.

Completion Codes:

RPC2_SUCCESS

The requested status fields have been made available.

RPC2_NOTSERVER

No side effect is ongoing on ConnHandle.

RPC2_SEFAIL1

The associated side effect routine indicated a nonfatal failure. Future calls on this connection will work.

RPC2_SEFAIL2

The associated side effect routine indicated a serious failure. Future calls on this connection will fail too.

RPC2_FAIL

Other assorted calamities.

Description:

Checks the status of a previously initiated side effect. This is a (potentially) blocking call, depending on the specified flags.

4.3 Miscellaneous Calls

4.3.1 RPC2_Init – *Perform runtime system initialization*

Call:

int RPC2_Init(**in** char *VersionId, **in** long Options, **in** RPC2_PortalIdent *PortalList[], **in** long HowManyPortals, **in** long RetryCount, **in** struct timeval *KeepAliveInterval)

Parameters:

VersionId

Set this to the constant RPC2_VERSION. The current value of this string constant must be identical to the value at the time the client runtime system was compiled.

Options

Right now there are no options.

PortalList

An array of unique network addresses within this machine, on which requests can be listened for, and to which responses to outgoing calls can be made. In the Internet domain this translates into a port number or a symbolic name that can be mapped to a port number. You need to specify this parameter even if you are only going to be a client and not export any subsystems. A value of NULL will cause RPC2 to select an arbitrary, nonassigned portal.

HowManyPortals

Specifies the number of elements in the array PortalList.

RetryCount

How many times to retransmit a packet before giving up all hope of receiving acknowledgement of its receipt. Should be in the range 1 to 30. Use a value of -1 to obtain the default.

KeepAliveInterval

How often to probe a peer during a long RPC call. This value is also used to calculate the retransmission intervals when packet loss is suspected by the RPC runtime system. Use NULL to obtain the default.

Completion Codes:

RPC2_SUCCESS

All went well,

RPC2_FAIL

Unable to initialize client. Check for bogus parameter values.

RPC2_WRONGVERSION

The header file and the library have different versions. This should never happen in a properly administered system.

RPC2_LWPNOTINIT

The LWP package has not been properly initialized. Be sure to call LWP(Init(before calling RPC2_Init(.

RPC2_BADSERVER

The PortalList field specifies an invalid address.

RPC2_DUPLICATESERVER

An entry in PortalList specifies an address which is already in use on this machine.

RPC2_SEFAIL1

The associated side effect routine indicated a minor failure.

RPC2_SEFAIL2

The associated side effect routine indicated a serious failure.

Description:

Initializes the RPC runtime system in this process. This call should be made before any other call in this package is made. It should be preceded by an initialization call to the LWP package and a call to SE(SetDefaults) with InitialValues as argument. If you get a wrong version indication, obtain a consistent version of the header files and the RPC runtime library and recompile your code. Note that this call incorporates a call to initialize IOMGR. RetryCount and KeepAliveInterval together define what it means for a remote site to be dead or unreachable. Packets are retransmitted at most RetryCount times until positive acknowledgement of their receipt is received. This is usually piggy-packed with useful communication, such as the reply to a request. The KeepAliveInterval is used for two purposes: to determine how often to check a remote site during a long RPC call, and to calculate the intervals between the RetryCount retransmissions of a packet. The RPC runtime system guarantees detection of remote site failure or network partition within a time period in the range KeepAliveInterval to twice KeepAliveInterval. See Chapter 15 (XXX) for further information on the retry algorithm.

Remember to activate each side effect, XXX, that you are interested in by invoking the corresponding XXX()Activate () call, prior to calling RPC2_Init.

You may get a warning about SOux[ ]GREEDY being undefined, if your kernel does not have an ITC bug fix. RPC2 will still work but may be slower and more likely to drop connections during bulk transfer. This is because of insufficient default packet buffer space within the Unix kernel.

4.3.2 RPC2_Unbind – *Terminate a connection by client or server*

Call:

int RPC2_Unbind(**in** RPC2_Handle ConnHandle)

Parameters:

ConnHandle

identifies the connection to be terminated,

Completion Codes:

RPC2_SUCCESS

All went well.

RPC2_NOCONNECTION

ConnHandle is bogus.

RPC2_SEFAIL1

The associated side effect routine indicated a minor failure.

RPC2_SEFAIL2

The associated side effect routine indicated a serious failure.

RPC2_FAIL

Other assorted calamities.

Description

Removes the binding associated with the specified connection. Normally a higher-level disconnection should be done by an RPC just prior to this call. Note that this call may be used both by a server and a client, and that no client/server communication occurs: the unbinding is unilateral.

4.3.3 RPC2_AllocBuffer – *Allocate a packet buffer*

Call:

int RPC2_AllocBuffer(**in** long MinBodySize, **out** RPC2_PacketBuffer **Buff)

Parameters:

MinBodySize

Minimum acceptable body size for the packet buffer.

Buff

Pointer to the allocated buffer.

Completion Codes:

RPC2_SUCCESS

Buffer has been allocated and *Buff points to it.

RPC2_FAIL

Could not allocate a buffer of requested size.

Description:

Allocates a packet buffer of at least the requested size. The BodyLength field in the header of the allocated packet is set to MinBodySize. The RPC runtime system maintains its own free list of buffers. Use this call in preference to malloc ().

4.3.4 RPC2_FreeBuffer – *Free a packet buffer*

Call:

int RPC2_FreeBuffer(**inout** RPC2_PacketBuffer **Buff)

Parameters:

Buff

Pointer to the buffer to be freed. Set to NULL by the call.

Completion Codes:

RPC2_SUCCESS

Buffer has been freed. *Buff has been set to NULL.

RPC2_FAIL

Could not free buffer.

Description:

Returns a packet buffer to the internal free list. Buff is set to NULL specifically to simplify locating bugs in buffer usage.

4.3.5 RPC2_GetPrivatePointer – *Obtain private data mapping for a connection*

Call:

int RPC2_GetPrivatePointer(**in** RPC2_Handle WhichConn, **out** char **PrivatePtr)

Parameters:

WhichConn

Connection whose private data pointer is desired.

PrivatePtr

Set to point to private data.

Completion Codes:

RPC2_SUCCESS

*PrivatePtr now points to the private data associated with this connection.

RPC2_FAIL

Bogus connection specified.

Description:

Returns a pointer to the private data associated with a connection. No attempt is made to validate this pointer.

4.3.6 RPC2_SetPrivatePointer – *Set private data mapping for a connection*

Call:

int RPC2_SetPrivatePointer(**in** RPC2_Handle WhichConn, **in** char *PrivatePtr)

Parameters:

WhichConn

Connection whose private data pointer is to be set.

PrivatePtr

Pointer to private data.

Completion Codes:

RPC2_SUCCESS

Private pointer set for this connection.

RPC2_FAIL

, Bogus connection specified.

Description:

Sets the private data pointer associated with a connection. No attempt is made to validate this pointer.

4.3.7 RPC2_GetSEPointer – *Obtain per-connection side-effect information*

Call:

int RPC2_GetSEPointer(**in** RPC2_Handle WhichConn, **out** char **SEPtr)

Parameters:

WhichConn

Connection whose side-effect data pointer is desired.

SEPtr

Set to point to side-effect data.

Completion Codes:

RPC2_SUCCESS

*SEPtr now points to the side effect data associated with this connection.

RPC2_FAIL

Bogus connection specified.

Description

Returns a pointer to the side effect data associated with a connection. No attempt is made to validate this pointer. This call is should only by the side effect routines, not by clients.

4.3.8 RPC2_SetSEPointer – *Set per-connection side-effect connection*

Call:

int RPC2_SetSEPointer(**in** RPC2_Handle WhichConn, **in** char *SEPtr)

Parameters:

WhichConn

Connection whose side effect pointer is to be set.

SEPtr

Pointer to side effect data.

Completion Codes:

RPC2_SUCCESS

Side effect pointer set for this connection.

RPC2_FAIL

Bogus connection specified.

Description:

Sets the side effect data pointer associated with a connection. No attempt is made to validate this pointer. This call should only be used by the side effect routines, not by clients.

4.3.9 RPC2_GetPeerInfo – *Obtain miscellaneous connection information*

Call:

int RPC2_GetPeerInfo(**in** RPC2_Handle WhichConn, **out** RPC2_PeerInfo *PeerInfo)

Parameters:

WhichConn

Connection whose peer you wish to know about.

PeerInfo

Data structure to be filled.

Completion Codes:

RPC2_SUCCESS

Peer information has been obtained for this connection.

RPC2_FAIL

Bogus connection specified.

Description:

Returns the peer information for a connection. Also returns other miscellaneous connection-related information, such as the securrity level in use. This information may be used by side-effect routines or high-level server code to perform RPC bindings in the opposite direction. The RemoteHandle and Uniquefier information are useful as end-to-end identification between client code and server code.

LamportTime. RPC2_LamportTime, Get Lamport time. , Returns the current Lamport time. Bears no resemblance to the actual time of day. Each call is guaranteed to return a value at least one larger than the preceding call. Every RPC packet sent and received by this Unix process has a Lamport time field in its header. The value returned by this call is guaranteed to be greater than any Lamport time field received or sent before now. Useful for generating unique timestamps in a distributed system.

4.3.10 RPC2_DumpState – *Dump internal RPC state*

Call:

int RPC2_DumpState(**in** FILE *OutFile, **in** long Verbosity)

Parameters:

OutFile

File on which the trace is to be produced. A value of NULL implies stdout.

Verbosity

Controls the amount of information dumped. Right now two values 0 and 1 are meaningfull.

Completion Codes:

RPC2_SUCCESS

The dump has been produced.

Description:

You should typically call this routine after calling RPC(DumpTrace).

4.3.11 RPC2_InitTraceBuffer – *Set trace buffer size*

Call:

int RPC2_InitTraceBuffer(**in** long HowMany)

Parameters:

HowMany

How many entries the trace buffer should have. Set it to zero to delete trace buffer.

Completion Codes:

RPC2_SUCCESS

The trace buffer has been adjusted appropriately.

Description:

Allows you to create and change the trace buffer at runtime. All existing trace entries are lost.

4.3.12 RPC2_DumpTrace – *Print a trace of recent RPC calls and packets received*

Call:

int RPC2_DumpTrace(**in** FILE *OutFile, **in** long HowMany)

Parameters:

OutFile

File on which the trace is to be produced. A value of NULL implies stdout.

HowMany

The HowMany most recent trace entries are printed. A value of NULL implies as many trace entries as possible. Values larger than TraceBufferLength specifed in RPC2_Init are meaningless.

Completion Codes:

RPC2_SUCCESS

The requested trace has been produced.

RPC2_FAIL

The trace buffer had no entries.

Description:

Note that it is not necessary for RPC2_Trace to be currently set. You can collect a trace and defer calling RPC2_DumpTrace until a convenient time. This call does not alter the current value of RPC2_Trace.

4.3.13 RPC2_SetLog – *Set the logfile and debug level to use*

Call:

void RPC2_SetLog(**in** FILE * file, **int level**)

Parameters:

file

File to send trace dumps and log entries to.

level

Value to set the debug level to.

Completion Codes:

None.

Description:

The debug level defines the verbosity of trace and log entries that will be written by a RPC2_DumpTrace.

4.3.14 XXX_SetDefaults – *Set an SE initializer to its default values*

Call:

int XXX_SetDefaults(**out** XXX_Initializer *Initializer)

Parameters:

Initializer

, Initializer for side effect XXX which you wish to set to default values.

Completion Codes:

RPC2_SUCCESS

Initialization Succeeded.

Description:

Each side effect type, XXX, defines an initialization structure type, XXX()Initializer,and an initialization routine, XXX()SetDefaults ().

A typical initialization sequence consists of the following: for each side effect, XXX, that you care about,

1. declare a local variable of type XXX()Initializer.
2. call XXX_SetDefaults () with this local variable as argument.
3. selectively modify those initial values you care about in the local variable, and
4. call XXX_Activate () with this local variable as argument. end(format

Finally call RPC2_Init.

This allows you to selectively set parameters of XXX without having to know the proper values for all of the possible parameters. Alas, if only C allowed initialization in type declarations this routine would be unnecessary.

4.3.15 XXX_Activate – *Activates a side effect type and initializes it*

Call:

int XXX_Activate(**in** XXX_Initializer *Initializer)

Parameters:

Initializer

Initializer for side effect XXX.

Completion Codes:

RPC2_SUCCESS

Activation Succeeded.

Description:

Activates side effect XXX. Code corresponding to this side effect will not be linked in otherwise. See comment for XXX()SetDefaults () for further details.

4.3.16 RPC2_ErrorMsg – *Describe RPC2 error code*

Call:

*char * RPC2_ErrorMsg*(**in** long ReturnCode)

Parameters:

ReturnCode

Error code returned by any RPC2 call.,

Description:

Converts ReturnCode into a string that can be used for printing error messages. Note that this is the only RPC2 call that returns a non-integer value.

4.3.17 RPC2_SetColor – *Set the color of a connection*

Call:

int RPC2_SetColor(**in** RPC2_Handle *Conn, **in** RPC2_Integer NewColor)

Parameters:

Conn

Connection whose color is to be changed.

NewColor

The lowest-order byte of this value is used as the new color for this connection.

Completion Codes:

RPC2_SUCCESS

The connection has been colored as requested.

RPC2_NOCONNECTION

Connection specified is bogus.

Description:

A color is an integer between 0 and 255 that is associated with a connection. When a packet is sent out on a connection, it acquires this color. Colors have no implicit significance to RPC2. But they can be used by other packages such as the Coda failure emulator package cite[CodaManual] to selectively induce failures. For example, in debugging an implementation of a two-phase commit protocol, one needs to test the situation where a failure occurs between the two phases. This situation can be detected by the failure emulator by using packets of different colors for the different phases. A change in color of a connection takes effect with the next packet that is sent out.

4.3.18 RPC2_GetColor – *Obtain current color of a connection*

Call:

int RPC2_ GetColor(**in** RPC2_Handle *Conn, **out** RPC2_Integer *Color)

Parameters:

Conn.

Connection whose color is to be obtained.

Color,

Current color of connection./

Completion Codes:

RPC2_SUCCESS.

Color obtained.

RPC2_NOCONNECTION.

Conn is bogus.

Description:

Useful if a package using RPC2 wants to save current color, set new color, then restore original color.

Chapter 5

SFTP: A Side Effect for Bulk Data Transfer

5.1 General Usage

SFTP is a protocol that allows a byte stream to be transferred efficiently from point A to point B, using optimizations such as streaming for large files and piggybacking for small files. These optimizations are completely transparent to the programmer. The byte stream to be transferred can be a file in the file system or a file in virtual memory. A file in VM at one end need not be transferred into a file in VM at the other end; any combination is possible. An SFTP transfer can take place from either an RPC2 server to an RPC2 client or the other way around. However the request for the transfer is always initiated by the client. To avoid confusion, we will refer to the sending entity *source* and the receiving entity *sink*.

The average user of SFTP need not be aware of the details of SFTP related to acknowledgements, flow control etc. All he needs to specify are the details of the file transfer such as the name of the file, the transmission direction etc. These are specified by the user in a data structure called a *Side Effect Descriptor*. The side effect descriptor (shown in Appendix **??** (XXX) is used both at the client and the server ends to describe the details of the byte stream transfer. The side effect descriptor is not transmitted: each side provides its own local version.

The most relevant of the side effect descriptors fields are given below.

Tag

This describes the type of side effect used. Currently the only side effect that has been implemented is the SFTP file transfer side effect.

Value

This is a union containing descriptor definitions for each type of side effect. The Tag field serves as the discriminant of the union. Currently, there is only one member of the union, corresponding to SFTP.

Value.SmartFTPD.TransmissionDirection

specifies if the transfer is from the server to the client (SERVERTOCLIENT) or from the client to the server (CLIENTTOSERVER).

Value.SmartFTPD.SeekOffset

specifies the initial position from where a file should be read from, or written to. This is useful, for instance, when bytes are appended to a file.

Value.SmartFTPD.ByteQuota

specifies the maximum number of data bytes to be sent or received. A value of -1 implies a limit of infinity. The RPC call fails if an attempt is made to transfer more bytes.

Value.BytesTransferred

is an output parameter. On completion of the file transfer, it specifies the number of bytes actually transferred.

Value.SmartFTPD.Tag

indicates whether the file to be transferred is specified by its name, its inode number, or by its VM address.

Value.SmartFTPD.FileInfo

is a union type, discriminated by the previous Tag field, that specifies the local identity of the file to be transferred. Note that there is no information of the remote identity of the file.

5.2 An Example

An example of an RPC subsystem which uses SFTP is given in the following section. In describing the use of SFTP, we will refer to this example and outline the steps used. The client-server code using SFTP is very similar to that of an RPC which does not use SFTP. Hence , we will only describe the additional steps needed when making an RPC which uses SFTP.

5.2.1 rcat Subsystem Interface (*in file rcat.rpc*)

```
include(rcat.rpc.mss)
```

5.2.2 Server Code (*in file rcat_srv.c*)

1. Initialize the SFTP package. This is done by first declaring a data structure of type SFTP_Initializer.The fields of the SFTP_Initializer specify parameters of the SFTP file transfer protocol, and include items such as the window size, packet size etc. These can be set by the user, or can be set to default values by calling the *SFTP_SetDefaults* primitive. The SFTP_Initializer is then activated using the *SFTP_Activate* primitive. Note that the SFTP package should be activated before the RPC2 package is initialized.

2. The routine which actually transfers the file does so by first declaring a data structure of type SE_Descriptor and filling in the values. It then initiates the file transfer by calling the *RPC2_InitSideEffect* primitive.

3. The *RPC2_CheckSideEffect* primitive can then be used to check the status of the side effect. Note that in the case of SFTP, the actual file transfer only occurs on this call.

```
include(rcat_srv.c.mss)
```

5.2.3 Client Code (*in file rcat_clnt.c*)

1. As with the server, the SFTP package is initialized by declaring an SFTP_Initializer, and by calling the *SFTP_SetDefaults* and *SFTP_Activate* primitives.
2. Set the SideEffectType field in RPC2_BindParms to SMARTFTP.
3. Declare a data structure of type SE_Descriptor. Set the Tag field to SMARTFTP, and fill in the rest of the fields.
4. Make the RPC.

```
include(rcat_clnt.c.mss)
```

Chapter 6

MultiRPC

6.1 Design Issues

The MultiRPC facility **??** ([Satyanarayanan 90]) is an extension to RPC2 that provides a parallel RPC capability for sending a single request to multiple servers and awaiting their individual responses. Although the actual transmission is done sequentially, the resultant concurrent processing by the servers results in a significant increase in time and efficiency over a sequence of standard RPC calls. The RPC2 runtime overhead is also reduced as the number of servers increases. For the purposes of this discussion, the base RPC2 facility will be referred to simply as **RPC2**.

A noteworthy feature of the MultiRPC design is the fact that the entire implementation is contained on the client side of the RPC2 code. The packet which is finally transmitted to the server is identical to a packet generated by an RPC2 call, and the MultiRPC protocol requires only a normal response from a server.

A major design goal was the desire to automatically provide MultiRPC capability for any subsystem without requiring any additional support from the subsytem designer or implementor. This has been achieved through modifications to RP2Gen, the RPC2 stub generation package (see chapter **??** (RP2Gen)). RP2Gen generates an array of argument descriptor structures for each server operation in the specification file, and these arrays are inserted in the beginning of the client side stub file. These structures are made available to the client through definitions in the associated *.h* file, and allow the use of MultiRPC with any routine in any subsystem with RP2Gen generated interfaces.

The orthogonality of the MultiRPC modifications also extends to the side effect mechanism. Side effects for MultiRPC work exactly as in the RPC2 case except that the client must supply a separate **SE_Descriptor** for each connection. Side effects can be omitted on a subset of the connections by specifying a the Tag field value of *OMITSE* for those **SE_Descriptor**s. This used, for example, in the Coda file system where a MultiRPC call is made to obtain file version information from a set of servers, but data from only one of those servers.

Parameter packing and unpacking for MultiRPC is provided in the RPC2 runtime library by a pair of routines. These library routines provide the functionality of the client side interface generated by RP2Gen as well as some additional modifications to support MultiRPC. It was decided to perform the packing and unpacking in RPC2 library routines rather than in individual client side stub routines as in the RPC2 case; this requires some extra processing time, but saves a significant amount of space in the client executable file. This approach has the added advantage of modularity; execution of RPC2 calls will not be affected at all, and even for MultiRPC calls the additional processing time is negligable in comparison to the message transmission overheads imposed by the UNIX kernel.

Another feature of MultiRPC is the client supplied handler routine. Through the handler routine the client is allowed to process each server response as it arrives rather than waiting for the entire MultiRPC call to complete. After processing each response, the client can decide whether to continue accepting server responses or whether to abort the remainder of the call. This facility can be useful if only a subset of responses are required, or if one failed message renders the entire call useless to the client. This capability is discussed further in section 18.1 (XXX).

MultiRPC also provides the same correctness guarantees as RPC2 except in the case where the client exercises his right to terminate the call. RPC2 guarantees that a request (or response) will be processed exactly once in the absence of network and machine crashes; otherwise, it guarantees that it will be processed at most once. If the call completes normally, a return code of *RPC2_Success* guarantees that all messages have been received by the appropriate servers.

6.2 Examples

To illustrate the use of MultiRPC, we revisit Example 2.6.1 (XXX) from Chapter 2 (XXX) and Example 5.2 (XXX) from Chapter 5 (XXX). The interface specification files (*rtime.rpc* and *rcat.rpc*) and the files for the server code (*rtime_srv.c* and *rcat_srv.c*) remain unchanged. Only the client files (*rtime_clnt.c* and *rcat_lowbar;clnt.c*) have to be modified.

6.2.1 MultiRPC Client for Example 2.6.1 (XXX) (*in file multi_rtime_clnt.c*)

This example specifies no handler, and does not perform any side effects. The client will prompt for the number of servers to which the request is to be made, and for their connection ids. Note that *RPC2_MultiRPC* is used even when only one server is requested.

```
@include(multi_rtime_clnt.c.mss)
```

6.2.2 MultiRPC Client for Example 5.2 (XXX) *(in file multi_rcat_clnt.c)*

In this example, a minimal handler routine is supplied for each server operation. It is adequate to demonstrate the format of the routine even though it does little actual processing of the responses. This example also demonstrates the use of MultiRPC in conjunction with side effects.

```
@include(multi_rcat_clnt.c.mss)
```

RPC2 Type	**in**	**out**	**in out**>)RPC2_Integer	long	lon
RPC2_Unsigned	unsigned long	unsigned long *[ ]	unsigned long *[ ]		
RPC2_Byte	unsigned char	unsigned char *[ ]	unsigned char *[ ]		
RPC2_String	unsigned char *	unsigned char **[ ]	unsigned char **[ ]		
RPC2_CountedBS	RPC2_CountedBS) *	RPC2_CountedBS) *[ ]	RPC2_CountedBS) *[ ]		
RPC2_BoundedBS	RPC2_BoundedBS) *	RPC2_BoundedBS) *[ ]	RPC2_BoundedBS) *[ ]		
RPC2_EncryptionKey	RPC2_EncryptionKey *	RPC2_EncryptionKey *[ ]	RPC2_EncryptionKey *[ ]		
SE_Descriptor	*illegal*	*illegal*	SE_Descriptor *[ ]		
RPC2_Enum *name*	*name*	*name* *[ ]	*name* *[ ]		
RPC2_Struct *name*	*name* *	*name* *[ ]	*name* *[ ]		
RPC2_Byte *name[...]*	*name*	*name* *[ ]	*name* *[ ]		

Table 6.1: RP2Gen representation of MultiRPC parameters

6.3 C Interface Specification

The following table shows the C type interface between the client routine and *MRPC_MakeMulti* for all the possible combinations of legal parameter declarations and types. In all cases it is the clients responsibility to allocate storage for all parameters, just as in the RPC2 case. For all types except RPC2_EncryptionKey, IN parameters are handled the same as in the single MakeRPC case; RPC2_EncryptionKeys must be passed as pointers in the MultiRPC case. For OUT and INOUT parameters, arrays of pointers to parameters must be supplied in order to hold the multiple server responses. The array for each parameter must contain the same number of items as the number of servers contacted, and they must be filled sequentially starting from element zero. For all INOUT parameters except for **SE_Descriptors**, only the first element of the array need be filled in. For **SE_Descriptors**, all elements must be filled in.

The following table should be consulted for specific formats.

The client is only responsible for understanding the parameter type interface to the *MakeMulti* and *HandleResult* routines, and for allocating all necessary storage. *MRPC_MakeMulti* and *MRPC_UnpackMulti* are included in the RPC2 libraries.

@newpage

6.4 Runtime Calls

6.4.1 MRPC_MakeMulti – *Pack arguments and initialize state for RPC2_MultiRPC*

Call:

MRPC_MakeMulti (**in** long Server **in** ARG ArgTypes[], **in** long HowMany, **in** RPC2_Handle CIDList[], **out** RPC2_Integer RCList[], **in out** RPC2_Multicast *MCast, **in** long (*HandleResult) (), **in** struct timeval *Timeout, Variable Length Argument List[]*)*

Parameters:

ServerOp

For server routine foo, "foo". RP2GEN generated opcode, defined in include file. Note that subsystems with overlapping routine names may cause problems in a MakeMulti call.,

ArgTypes

For server routine foo, "foo;PTR". RP2GEN generated array of argument type specifiers. A pointer to this array is located in the generated include file *foo.h.*,

HowMany

How many servers are being called,

CIDList

Array of HowMany connection handles, one for each of the servers,

RCList

Array of length HowMany, into which RPC2 will place return codes for each of the connections specified in ConnHanleList. May be specified as NULL if return codes will not be examined.,

MCast

Pointer to multicast sturcture. Set to NULL for now.,

HandleResult

User procedure to be called after each server response. Responses are processed as they come in. Client can indicate when he has received sufficient responses (see below). @MRPC(MakeMulti) will not return the server responses.,

Timeout

User specified timeout. Note that the default timeout set in the .rpc file will not be active here: a NULL value will be passed through to MultiRPC, where it will indicate infinite patience as long as RPC2 believes that the server is alive. Note that this timeout value is orthogonal to the RPC2 internal timeout for determining connection death.,

<Variable Length Argument List>

This is just the list of the server arguments as they are declared in the *.rpc2* file. It is represented in this form since each call will have a different argument list.,

Completion Codes:

@RPC2(SUCCESS)

All went well,

@RPC2(TIMEOUT)

The user specified timeout expired before all the server responses were received,

@RPC(FAIL)

Something other than SUCCESS or TIMEOUT occurred. Individual server response is supplied via UnpackMulti to the client handler routine.

Description:

Text=‘For all **in** or **in out** parameters, an array of HowMany of the appropriate type should be allocated and supplied by the client. For example, if one argument is an **out** integer, an array of HowMany integers (i.e. int foo[HowMany]) should be used. For structures, an array of structures and NOT an array of pointers to structures should be used. **in** arguments are treated as in the @RPC2(MakeRPC) case.)

6.4.2 RPC2_MultiRPC – *Make a collection of remote procedure calls*

Call:

RPC2_MultiRPC(**in** long HowMany **in** RPC2_Handle ConnHandleList[], **in** RPC2_PacketBuffer *Request, **in** SE_Descriptor SDescList[], **in** long (*UnpackMulti) (), **in** **out** @ARG(INFO) *ArgInfo, **in** struct timeval *Patience *)*

Parameters:

HowMany

How many servers to contact,

ConnHandleList

List of HowMany connection handles for the connections on which calls are to be made.,

Request

A properly formatted request buffer.,

SDescList

List of HowMany side effect descriptors,

UnpackMulti

Pointer to unpacking routine called by RPC2 when each server response as received. If RP2Gen is used, this will be supplied by @MRPC(MakeMulti). Otherwise, it must be supplied by the client.,

ArgInfo

A pointer to a structure containing argument information. This structure is not examined by RPC2; it is passed untouched to UnpackMulti. If RP2Gen is used, this structure will be supplied by @MRPC(MakeMulti). Otherwise, it can be used to pass any structure desired by the client or supplied as NULL.,

Patience

Maximum time to wait for remote sites to respond. A NULL pointer indicates infinite patience as long as RPC2 believes that the server is alive. Note that this timeout value is orthogonal to the RPC2 internal timeout for determining connection death.,

Completion Codes:

RPC2_SUCCESS)

All servers returned successfully, or all servers until client-initiated abort returned successfully. Individual server response information is supplied via UnpackMulti to the user handler routine supplied in the ArgInfo structure.,

RPC2_TIMEOUT)

The user specified timeout expired before all the servers responded.,

RPC2_FAIL)

Something other than SUCCESS or TIMEOUT occurred. More detailed information is supplied via UnpackMulti to the user handler routine supplied in the ArgInfo structure.,

Description:

Logically identical to iterating through ConnHandleList and making RPC2_MakeRPC) calls to each specified connection using Request as the request block, but this call will be considerably faster than explicit iteration. The calling lightweight process blocks until either the client requests that the call abort or one of the following is true about each of the connections specified in ConnHandleList: a reply has been received, a hard error has been detected for that connection, or the specified timeout has elapsed.

The ArgInfo structure exists to supply argument packing and unpacking information in the case where RP2Gen is used. Since its value is not examined by RPC2, it can contain any pointer that a non-RP2Gen generated client wishes to supply.

Similarly, UnpackMulti will point to a specific unpacking routine in the RP2Gen case. If the RP2Gen interface is not used, you should assume that the return codes of the supplied routine must conform to the specifications in section **??** (XXX).

Side effects are supported as in the RPC2 case except that the client must supply a separate **@SE[Descriptor]** for each connection. The format for the **@SE[Descriptor]** argument is described in section 6.3 (XXX). It will often be useful to supply connection specific information such as unique file names in the **@SE[Descriptor]**.

6.4.3 MRPC_UnpackMulti – *Unpack server arguments and call client handler routine*

Call:

MRPC_UnpackMulti(**in** long HowMany, **in** RPC2_Handle ConnHandleList[], **in out** @ARG(INFO) *ArgInfo, **in** RPC2_PacketBuffer *Response, **in** long rpcval, **in** long *)*

Parameters:

HowMany

How many servers were included in the MultiRPC call,

ConnHandleList

Array of HowMany connection ids,

ArgInfo

Pointer to argument information structure. This pointer is the same one passed in to MultiRPC, so for the non-RP2Gen case its type is determined by the client.,

Response

RPC2 response buffer,

rpcval

Individual connection error code or server response code,

thishost

Index into ConnHandleList to identify the returning connection,

Completion Codes:

0

Continue accepting and processing server responses,

-1

Abort MultiRPC call and return,

Description:

This routine is fixed in the RP2Gen case, and can be ignored by the client. For the non-RP2Gen case, a pointer to a routine with the argument structure described must be supplied as an argument to RPC2_MultiRPC). The functionality of such a client-supplied routine is unconstrained, but note that the return codes have an important effect on the process of the MultiRPC call.

6.4.4 HandleResult, – *Process incoming server replies as they arrive*

Call:

HandleResult(**in** long HowMany, **in** RPC2_Handle ConnArray[], **in** long WhichHost, **in** long rpcval, [Variable Length Argument List] *)*

Parameters:

HowMany

number of servers from @MRPC(MakeMulti) call,

ConnArray

array of connection ids as supplied to @MRPC(MakeMulti),

WhichHost

this is an offset into ConnArray and into any OUT or IN;OUT parameters. Using this to index the arrays will yield the responding server and its corresponding argument values.,

rpcval

this is the RPC2 return code from the specified server, p5=‘<Variable Length Argument L

Completion Codes:

0

Continue processing server responses,

-1

Terminate @MRPC(MakeMulti) call and return,

Description:

This routine must return either 0 or -1. A return value of zero indicates that the client wants to continue receiving server responses as they come in (normal case). A return value of -1 indicates that the client has received enough responses and wants to terminate the MakeMulti call (in which the client is still blocked). This allows the client to call a large number or servers and terminate after the first *n* responses are received.

Note that the name of this routine is arbitrary and may be determined by the client. RPC2_MultiRPC) sees it only as a pointer supplied as an argument to @MRPC(MakeMulti). The parameter list is predefined, however, and the client must follow the structure specified here in writing the routine.

Chapter 7

Multicast

7.1 General Usage

@center(*Not written yet*)

7.2 Runtime Calls

7.2.1 RPC2_CreateMgrp – *Create a new RPC2 mgroup ID*

Call:

RPC2_CreateMgrp(**out** RPC2_Handle *MgroupHandle, **in** RPC2_McastIdent *McastAddr, **in** RPC2_PortalIdent *McastPortal, **in** RPC2_SubsysIdent *Subsys,**in** RPC2_Integer SecurityLevel,**in** RPC2_EncryptionKey SessionKey,**in** RPC2_Integer EncryptionType,**in** long SideEffectType *)*

Parameters:

MgroupHandle

An integer, unique to a specific client, returned by the call. When combined with the clients network address, it uniquely identifies this mgroup connection. This is not necessarily a small-valued integer.,

McastAddr

The IP multicast address of the desired multicast group. This is the address that a server must use in a joingroup () system call. Only MGRPBYINETADDR is supported at present.,

McastPortal

The portal at which server (s) will be listening. All servers in the multicast group must be listening at the same portal.,

Subsys

The RPC2 subsystem corresponding to this multicast group. Each server in the multicast group must have exported this subsystem.,

SecurityLevel

same as in RPC2_Bind().,

SessionKey

Encryption key to be used for all communication on this multicast channel. RPC2 will send this (securely) to each server that is added via a RPC2_AddToMgrp)() call.,

EncryptionType

same as in RPC2_Bind)().,

Completion Codes:

RPC2_SUCCESS)

All went well,

RPC2_SEFAIL1)

Side effect routines reported failure.,

RPC2_SEFAIL2)

Side effect routines reported failure.,

RPC2_FAIL),

Some other mishap occurred.,

Description:

Creates a new RPC2 mgroup identifier, unique to a particular <multicastaddr, portal, subsystem> combination, and returns it to the caller. Once the identifier has been assigned, RPC2 mgroup members may be added and deleted via the RPC2_AddToMgrp) and RPC2_RemoveFromMgrp calls. The security level, encryption type and side effect types of each added connection must match the corresponding parameters of this RPC2_CreateMgrp call.

7.2.2 RPC2_AddToMgrp – *Add a connection ID (server) to an RPC2 mgroup*

Call:

RPC2_AddToMgrpAdd(**in** RPC2_Handle MgroupHandle,**in** RPC2_Handle) ConnHandle*)*

Parameters:

MgroupHandle

identifies the mgroup to which the new server should be added,

ConnHandle

identifies the connection to be added to the mgroup,

Completion Codes:

RPC2_SUCCESS

All went well,

RPC2_NOMGROUP

MgroupHandle is not a valid mgroup,

RPC2_NOCONNECTION

ConnHandle is not a valid connection.,

RPC2_BADSECURITY

SecurityLevel or EncryptionType of ConnHandle does not match that of Mgroup,

RPC2_DUPLICATEMEMBER

ConnHandle is already a member of Mgroup,

RPC2_MGRPBUSY

A call is in progress on MgroupHandle.,

RPC2_CONNBUSY

A call is in progress on ConnHandle.,

RPC2_SEFAIL1

Error code returned by side effect routine.,

RPC2_SEFAIL2

Error code returned by side effect routine.,

RPC2_NAKED

The remote site corresponding to ConnHandle sent an explicit negative acknowledgement. This can happen if that site thought you were dead, or if someone at that site unbound your connection.,

RPC2_DEAD

The remote site corresponding to ConnHandle has been deemed dead or unreachable,

RPC2_FAIL

Some strange mishap occurred.,

Description:

Adds ConnHandle to the mgroup associated with MgroupHandle. RPC2 contacts the remote site to initialize its mgroup connection information. The security level, encryption type and side effect type of the connection being added must match that specified when Mgroup was defined.

7.2.3 RPC2_RemoveFromMgrp – *Remove a connection from an RPC2 mgroup*

Call:

RPC2_RemoveFromMgrp(**in** RPC2_Handle MgroupHandle, **in** ConnHandle *)*

Parameters:

MgroupHandle

the mgroup to be shrunk,

ConnHandle

the connection to be removed from MgroupHandle,

Completion Codes:

RPC2_NOMGROUP

MgroupHandle does not refer to a valid mgroup connection,

RPC2_NOCONNECTION

ConnHandle is bogus.,

RPC2_NOTGROUPMEMBER

ConnHandle is not a member of MgroupHandle.,

RPC2_MGRPBUSY

A call is in progress on MgroupHandle,

RPC2_FAIL

Some strange mishap occurred,

Description:

‘Removes ConnHandle from the mgroup associated with MgroupHandle

7.2.4 RPC2_DeleteMgrp – Delete a RPC2 mgroup

Call:

*RPC2_DeleteMgrp (***in** RPC2_Handle MgroupHandle *)*

Parameters:

MgroupHandle

the mgroup to be deleted,

Completion Codes:

RPC2_SUCCESS

All went well,

RPC2_NOMGROUP

MgroupHandle is bogus.,

RPC2_MGRPBUSY

A call is in progress on MgroupHandle.,

RPC2_FAIL

Some other mishap occurred.,

Description:

Deletes an mgroup, removing any existing members before deletion

Chapter 8

Chapter(The Basic LWP Package)

The LWP package implements primitive functions providing basic facilities that enable procedures written in C, to proceed in an unsynchronized fashion. These separate threads of control may effectively progress in parallel, and more or less independently of each other. This facility is meant to be general purpose with a heavy emphasis on simplicity. Interprocess communication facilities can be built on top of this basic mechanism, and, in fact, many different IPC mechanisms could be implemented. The RPC2 remote procedure call package (also described in this manual) is one such IPC mechanism.

The LWP package makes the following key design choices:

- The package should be small and fast;
- All processes are assumed to be trustworthy – processes are not protected from each others actions;
- There is no time slicing or preemption – the processor must be yielded explicitly.

In order to set up the environment needed by the lightweight process support, a one-time invocation of the LWP_Init function must precede the use of the facilities described here. The initialization function carves an initial process out of the currently executing C procedure. The process id of this initial process is returned as the result of the LWP_Init function. For symmetry a LWP_TerminateProcessSupport function may be used explicitly to release any storage allocated by its initial counterpart. If used, it must be issued from the process created by the LWP_Init function.

Upon completion of any of the lightweight process functions, an integer value is returned to indicate whether any error conditions were encountered.

Macros, typedefs, and manifest constants for error codes needed by the lightweight process mechanism reside in the file `<lwp.h>` (shown in Appendix **??** (XXX). A process is identified by an object of type PROCESS, which is defined in the include file.

The process model supported by the operations described here is based on a non-preemptive priority dispatching scheme. (A priority is an integer in the range [0..LWP_MAX_PRIORITY], where 0 is the lowest priority.) Once a given lightweight process is selected and dispatched, it remains in control until it voluntarily relinquishes its claim on the CPU. Relinquishment may be either explicit (LWP_DispatchProcess) or implicit (through the use of certain other LWP operations). In general, all LWP operations that may cause a higher priority process to become ready for dispatching, preempt the process requesting the service. When this occurs, the priority dispatching mechanism takes over and dispatches the highest priority process automatically. Services in this category (where the scheduler is guaranteed to be invoked in the absence of errors) are

- LWP_CreateProcess
- LWP_WaitProcess
- LWP_MwaitProcess
- LWP_SignalProcess
- LWP_DispatchProcess
- LWP_DestroyProcess

The following services are guaranteed not to cause preemption (and so may be issued with no fear of losing control to another lightweight process):

- LWP_Init
- LWP_NoYieldSignal
- LWP_CurrentProcess
- LWP_StackUsed
- LWP_NewRock
- LWP_GetRock

The symbol LWP_NORMAL_PRIORITY provides a good default value to use for process priorities.

8.1 A word about initialization

The LWP, IOMGR, Fast Time, and RPC2 packages have routines that perform global initialization for the package. Each of these routines may be called more than once, and only the parameters from the first invocation will be used. In addition, each routine calls any of the others that it needs for proper operation.

The result is that if you only use one package directly, you need only call the initialization routine for that package. You may call the initialization routines for other packages anyway in order to set the initialization parameters yourself. If you wish to initialize all of these packages yourself, you must call the initialization routines in this order: Fast Time, LWP, IOMGR, RPC2. The RPC2 package does not call the LWP packages initialization routine. You must call LWP_Init explicitly before calling RPC2_Init.

In contrast, the Preemption package initialization routine may be called multiple times to change the value of the preemption time slice.

The Lock and Timer packages have initialization routines that initialize objects instead of global data. The only restriction on the order of the initialization calls to these packages is that calls to TM_Init must follow your call to FT_Init, if you have one.

8.2 A Simple Example

```
#include <lwp.h>

static read_process (id)
    int *id;
{
    LWP_DispatchProcess ();             /* Just relinquish control for now */

    for (;;) {
        /* Wait until there is something in the queue */
        while (empty(q)) LWP_WaitProcess (q);
        /* Process queue entry */
        LWP_DispatchProcess ();
   }
}

static write_process ()
{
    . . .

    /* Loop & write data to queue */
    for (mesg=messages; *mesg!=0; mesg++) {
        insert (q, *mesg);
        LWP_SignalProcess (q);
    }
}

main (argc, argv)
   int argc; char **argv;
{
    PROCESS *id;

    LWP_Init (LWP_VERSION, 0, &id);
    /* Now create readers */
    for (i=0; i < nreaders; i++)
        LWP_CreateProcess (read_process, STACK_SIZE, 0, i, "Reader", &readers[i]);
    LWP_CreateProcess (write_process, STACK_SIZE, 1, 0, "Writer", &writer);
    /* Wait for processes to terminate */
    LWP_WaitProcess (&done);
    for (i=nreaders-1; i>=0; i--) LWP_DestroyProcess (readers[i]);
}
@end(program)
main (argc, argv)
   int argc; char **argv;
{
    PROCESS *id;

    LWP_Init (LWP_VERSION, 0, &id);
    /* Now create readers */
```

```
        for (i=0; i< nreaders; i++)
            LWP_CreateProcess (read_process, STACK_SIZE, 0, i, "Reader", &readers[i]);
        LWP_CreateProcess (write_process, STACK_SIZE, 1, 0, "Writer", &writer);
        /* Wait for processes to terminate */
        LWP_WaitProcess (&done);
        for (i=nreaders-1; i>=0; i--) LWP_DestroyProcess (readers[i]);
}
```

8.3 LWP Runtime Calls

8.3.1 LWP_Init – *Initialize LWP support and start initial process*

Call:

*int LWP_Init(***in** char *VersionId>, @w<**in** int priority>, @w<**out** PROCESS *pid>)

Parameters:

VersionId

Set this to the constant LWP_VERSION. The current value of this string constant must be identical to the value at the time the client runtime system was compiled.

priority

Priority at which initial process is to run.

pid

The process id of the initial process will be returned in this parameter.

Completion Codes:

LWP_SUCCESS

All went well

LWP_EBADPRI

Illegal priority specified (< 0 or too large)

Description:

Initializes the LWP package. In addition, this routine turns the current thread of control into the initial process with the specified priority. The process id of this initial process will be returned in parameter pid. This routine must be called to ensure proper initialization of the LWP routines. This routine will not cause the scheduler to be invoked.)

8.3.2 LWP_TerminateProcessSupport – Terminate process support and clean up

Call:

int LWP_ TerminateProcessSupport()

Parameters:

none.

Completion Codes:

N/A

Description:

This routine will terminate the LWP process support and clean up by freeing any auxiliary storage used. This routine must be called from within the procedure and process that invoked LWP_Init. After LWP_TerminateProcessSupport has been called, LWP_Init may be called again to resume LWP process support.)

8.3.3 LWP_CreateProcess – Create and start a light-weight process

Call:

*int LWP_ CreateProcess(***in** int (*ep) (),**in** int stacksize,**in** int priority, **in** char *parm **in** char *name, **out** PROCESS *pid)

Parameters:

ep

This is the address of the code that is to execute the function of this process. This parameter should be the address of a C routine with a single parameter.,

stacksize

This is the size (in bytes) to make the stack for the newly-created process. The stack cannot be shrunk or expanded, it is fixed for the life of the process.,

priority

This is the priority to assign to the new process.,

parm

This is the single argument that will be passed to the new process. Note that this argument is a pointer and, in general, will be used to pass the address of a structure containing further "parameters".,

name

This is an ASCII string that will be used for debugging purposes to identify the process. The name may be a maximum of 32 characters.,

pid

The process id of the new process will be returned in this parameter,

Completion Codes:

LWP_SUCCESS

Process created successfully,

LWP_ENOMEM

Not enough free space to create process,

LWP_EBADPRI

Illegal priority specified (< 0 or too large),

LWP_EINIT

LWP_Init has not been called

This routine is used to create and mark as runnable a new light-weight process. This routine will cause the scheduler to be called. Note that the new process will begin execution before this call returns only if the priority of the new process is greater than or equal to the priority of the creating process.)

8.3.4 LWP_DestroyProcess – Destroy a light-weight process

Call:

*int LWP_DestroyProcess(***in** PROCESS pid)

Parameters:

pid

The process id of the process to be destroyed.

Completion Codes:

LWP_SUCCESS

Process destroyed successfully

LWP_EINIT

LWP_Init has not been called

Description:

This routine will destroy the specified process. The specified process will be terminated immediately and its internal storage will be freed. A process is allowed to destroy itself (of course, it will only get to see the return code if the destroy fails). Note a process may also destroy itself by executing a **return** from the C routine. This routine calls the scheduler.

8.3.5 LWP_WaitProcess – Wait for event

Call:

*int LWP_ WaitProcess(***in** char *event)

Parameters:

event

The event to wait for. This can be any memory address. But, 0 is an illegal event.

Completion Codes:

LWP_SUCCESS

The event has occurred

LWP_EINIT

LWP_Init has not been called

LWP_EBADEVENT

The specified event was illegal (0)

Description:

This routine will put the calling process to sleep until another process does a call of LWP_SignalProcess or LWP_NoYieldSignal with the specified event. Note that signals of events are not queued: if a signal occurs and no process is woken up, the signal is lost. This routine invokes the scheduler.)

8.3.6 LWP_MwaitProcess – Wait for a specified number of a group of signals

Call:

*int LWP_MwaitProcess(***in** int wcount, **in** char *evlist)

Parameters:

wcount

Is the number of events that must be signaled to wake up this process

evlist

This a null-terminated list of events (remember that 0 is not a legal event). There may be at most LWP_MAX_EVENTS events

Completion Codes:

LWP_SUCCESS

The specified number of appropriate signals has occurred

LWP_EBADCOUNT

There are too few events (0) or wcount > the number of events in evlist

LWP_EINIT

LWP_Init has not been called

Description:

This routine allows a process to wait for wcount signals of any of the signals in evlist. Any number of signals of a particular event is only counted once. The scheduler will be invoked.)

8.3.7 LWP_SignalProcess – Signal an event

Call:

*int LWP_SignalProcess(***in** char *event)

Parameters:

event

The event to be signaled. An event is any memory address except 0

Completion Codes:

LWP_SUCCESS

The signal was a success (a process was waiting for it)

LWP_EBADEVENT

The specified event was illegal (0),

LWP_EINIT

LWP_Init was not called,

LWP_ENOWAIT

No process was waiting for this signal,

Description:

This routine causes event to be signaled. This will mark all processes waiting for only this event as runnable. The scheduler will be invoked. Signals are not queued: if no process is waiting for this event, the signal will be lost and LWP_ENOWAIT will be returned.)

8.3.8 What call is thore for.

Parameters:

event

The event to be signaled. An event is any memory address except 0,

Completion Codes:

LWP_SUCCESS

The signal was a success (a process was waiting for it),

LWP_EBADEVENT

The specified event was illegal (0),

LWP_EINIT

LWP_Init was not called,

Description:

Text=‘This routine causes event to be signaled. This will mark all processes waiting for only this event as runnable. This call is identical to LWP_SignalProcess except that the scheduler will not be invoked – control will remain with the signalling process. Signals are not queued: if no process is waiting for this event, the signal will be lost and LWP_ENOWAIT will be returned.)

8.3.9 LWP_XXXFIXME

Parameters:

Well??

Completion Codes:

LWP_SUCCESS

All went well,

LWP_EINIT

LWP_Init has not been called,

Description:

Text='This routine is a voluntary yield to the LWP scheduler.

8.3.10 LWP_CurrentProcess – Get the current process id

Call:

*int LWP_CurrentProcess(***out** PROCESS *pid)

Parameters:

pid,

The current process id will be returned in this parameter

Completion Codes:

LWP_SUCCESS

The current process id has been returned,

LWP_EINIT

LWP_Init has not been called,

Description:

Text='This routine will place the current process id in the parameter pid.)

8.3.11 LWP_StackUsed – Get information about stack usage for a process

Call:

*int LWP_StackUsed(***in** PROCESS pid**out** int *max**out** int *used)

Parameters:

pid

The target process

max

Max stack size given at process creation time

used

Stack used so far

Completion Codes:

LWP_SUCCESS

No problem,

LWP_NO_STACK

Stack counting was not enabled for this process.,

Description:

Text='This routine returns the amount of stack space allocated to the process and the amount actually used by the process so far. It works by initializing the stack to a special pattern at process creation time and checking to see how much of the pattern is still there when **LWP_StackUsed** is called. The stack of the process is only initialized to the special pattern if the global variable **lwp_stackUseEnabled** is true when the process is created. This variable is initially true. If **lwp_stackUseEnabled** was false at the time the process was created, then ***used** will be set to zero and the routine will return **LWP_NO_STACK**.)

8.3.12 LWP_NewRock – Find a rock under which private information can be hidden

***int LWP_NewRock(*in int Tag in char *Value)**

Parameters:

Tag

A unique integer identifying this rock.

Value

A value (usually a pointer to some data structure) to be associated with the current LWP and identified by Tag.

Completion Codes:

LWP_SUCCESS

No problem.

LWP_EBADROCK

Rock called Tag already exists for this LWP.

LWP_ENOROCKS

All rocks are in use.

Description:

Text=‘The rock is exactly what its name implies: a place to squirrel away application-specific information associated with an LWP. The Tag is any unique integer. Users of the LWP package must coordinate their choice of Tag values. Note that you cannot change the value associated with Tag. To obtain a mutable data structure use one level of indirection.)

8.3.13 LWP_GetRock – Obtain information hidden under a rock

Call:

*int LWP_ GetRock(***in** int Tag, **out** char **Value)

Parameters:

Tag

Rock under which to look.,

Value

The current value (usually a pointer to some data structure) hidden under this rock.

Completion Codes:

LWP_SUCCESS

Value has been filled.,

LWP_EBADROCK

Specified rock does not exist.,

Description:

Text=‘Recovers information hidden by a LWP_NewRock call.

Chapter 9

The Lock Package

The lock package contains a number of routines and macros that allow C programs that utilize the LWP abstraction to place read and write locks on data structures shared by several light-weight processes. Like the LWP package, the lock package was written with simplicity in mind – there is no protection inherent in the model.

In order to use the locking mechanism for an object, an object of type `struct Lock` must be associated with the object. After being initialized, with a call to @Lock(Init), the lock is used in invocations of the macros ObtainReadLock, ObtainWriteLock, ReleaseReadLock and ReleaseWriteLock.

The semantics of a lock is such that any number of readers may hold a lock. But only a single writer (and no readers) may hold the clock at any time. The lock package guarantees fairness: each reader and writer will eventually have a chance to obtain a given lock. However, this fairness is only guaranteed if the priorities of the competing processes are identical. Note that no ordering is guaranteed by the package.

In addition, it is illegal for a process to request a particular lock more than once, without first releasing it. Failure to obey this restriction may cause deadlock.

9.1 Key Design Choices

- The package must be simple and @u[fast]: in the case that a lock can be
- obtained immediately, it should require a minimum of instructions;
- All the processes using a lock are trustworthy;
- The lock routines ignore priorities;

9.1.1 A Simple Example

@Begin(program) #include "lock.h"

struct Vnode { . . . struct Lock lock; /* Used to lock this vnode */ . . . };

#define READ 0 #define WRITE 1

struct Vnode *get_vnode (name, how) char *name; int how; { struct Vnode *v;

v = lookup (name); if (how == READ) ObtainReadLock (&v-> lock); else ObtainWriteLock (&v-> lock); } @End(program)

9.2 Lock Primitives

9.2.1 Lock_Init – Initialize a lock

Call:

void Lock_Init(@w<**out** struct Lock *lock> *)*

Parameters:

lock

The (address of the) lock to be initialized,

Completion Codes:

N/A

Description:

This routine must be called to initialize a lock before it is used.

9.2.2 ObtainReadLock – Obtain a read lock

Call:

ObtainReadLock(@w<**in out** struct Lock *lock> *)*

Parameters:

lock

The lock to be read-locked,

Completion Codes:

N/A

Description:

A read lock will be obtained on the specified lock. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.

9.2.3 ObtainWriteLock – Obtain a write lock

Call:

ObtainWriteLock(@w<**in out** struct Lock *lock> *)*

Parameters:

lock

The lock to be write-locked

Completion Codes:

N/A

Description:

A write lock will be obtained on the specified lock. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.)

9.2.4 ReleaseReadLock – Release a read lock

Call:

ReleaseReadLock(@w<**in out** struct Lock *lock> *)*

Parameters:

lock

The lock to be released

Completion Codes:

N/A

Description:

The specified lock will be released. This macro requires that the lock must have been previously read-locked. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.

9.2.5 ReleaseWriteLock – Release a write lock

Call:

ReleaseWriteLock(@w<**in out** struct Lock *lock> *)*

Parameters:

lock

The lock to be released

Completion Codes:

N/A

Description:

The specified lock will be released. This macro requires that the lock must have been previously write-locked. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.)

9.2.6 CheckLock – Check the status of a lock

Call:

CheckLock(@w<**in** struct Lock *lock> *)*

Parameters:

lock

The lock to be checked

Completion Codes:

N/A

Description:

This macro yields an integer that specifies the status of the indicated lock. The value will be -1 if the lock is write-locked 0 if unlocked, or a positive integer that indicates the numer of readers with read locks. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.)

Chapter 10

The IOMGR Package

The IOMGR package allows light-weight processes to wait on various Unix events. IOMGR_Select allows a light-weight process to wait on the same set of events that the Unix `select` call waits on. The parameters to these routines are the same. IOMGR_Select puts the caller to sleep until no user processes are active. At this time the IOMGR process, which runs at the lowest priority, wakes up and coaleses all of the select request together. It then performs a single `select` and wakes up all processes affected by the result.

The IOMGR_Signal call allows a light-weight process to wait on delivery of a Unix signal. The IOMGR installs a signal handler to catch all deliveries of the Unix signal. This signal handler posts information about the signal delivery to a global data structure. The next time that the IOMGR process runs, it delivers the signal to any waiting light-weight processes.

10.1 Key Design Choices

- The meanings of the parameters to IOMGR_Select, both before and after the call, should be identical to those of the Unix `select`;
- A blocking select should only be done if no other processes are runnable.

10.2 A Simple Example

```
void rpc2_SocketListener ()
{
    int ReadfdMask, WritefdMask, ExceptfdMask, rc;
    struct timeval *tvp;

    while (TRUE) {
        . . .
        ExceptfdMask = ReadfdMask = (1 << rpc2_RequestSocket);
        WritefdMask = 0;
        rc = IOMGR_Select (8*sizeof(int), &ReadfdMask, &WritefdMask, &ExceptfdMask, tvp);
```

```
            switch (rc) {
                case 0:     /* timeout */
                        continue;   /* main while loop */

                case -1:    /* error */
                        SystemError ("IOMGR_Select");
                        exit (-1);

                case 1:     /* packet on rpc2_RequestSocket */
                        . . . process packet . . .
                        break;

                default:    /* should never occur */
            }
        }
    }
```

10.3 IOMGR Primitives

10.3.1 IOMGR_Initialize – Initialize the IOMGR package

Call:

void IOMGR_Initialize( )

Parameters:

None.

Completion Codes:

LWP_SUCCESS

All went well,

LWP_ENOMEM

Not enough free space to create the IOMGR process,

-1

Something went wrong with other init calls,

Description:

This call will initialize the IOMGR package. Its main task is to create the IOMGR process, which runs at priority 0, the lowest priority. The remainder of the processes must be running at priority 1 or greater for the IOMGR package to function correctly.)

10.3.2 IOMGR_Finalize – Clean up after IOMGR package

Call:

void IOMGR_Finalize(,)

Parameters:

None.

Completion Codes:

LWP_SUCCESS

Package finalized okay,

Description:

This call cleans up when the IOMGR package is no longer needed. It releases all storage and destroys the IOMGR process.

10.3.3 –

Call:

( )

Parameters:

Completion Codes:

Description:

10.3.4 IOMGR_Select – Perform an LWP select operation

Call:

IOMGR_Select(@w<**in** int fds>, @w<**in out** int *readfds>, @w<**in out** *writefds>, *)*

Parameters:

fds

Maximum number of bits to consider in masks,

readfds

Mask of file descriptors that process wants notification of when ready to be read,

writefds

Mask of file descriptors that process wants notification of when ready to be written,

exceptfds

Mask of file descriptors that process wants notification of when exceptional condition occurs,

timeout

Timeout for use on this selectrip>

Completion Codes:

Description:

This function performs an LWP version of Unix **select**. The parameters have the same meanings as the Unix call. However, the return value will only be -1 (an error occurred), 0 (a timeout occurred), or 1 (some number of file descriptors are ready). If this is a polling select, it is done and IOMGR_;Select returns to the user with the results. Otherwise, the calling process is put to sleep. If at some point, the IOMGR process is the only runnable process, it will awaken and collect all select requests. It will then perform a single select and awaken those processes the appropriate processes – this will cause return from the affected IOMGR_;selects.

10.3.5 IOMGR_Signal – Convert Unix signals to LWP signals

Call:

IOMGR_Signal(@w<**in** int signo>, @w<**in** char *event>, *)*

Parameters:

signo

The unix signal number, as defined in signal.h,

event

The light-weight process event that should be signaled whenever *signo* is delivered

Completion Codes:

LWP_;SUCCESS

No problems.,

LWP_;EBADSIG

signo was out of range,

LWP_;EBADEVENT

event was zero,

Description:

This function associates an LWP signal with a Unix signal. When the Unix signal *signo* is delivered to the process, the IOMGR process will deliver an LWP signal to the event *event* via LWP_;NoYieldSignal, waking any light-weight processes waiting on that event. Multiple deliveries of the signal may be coalesed into one LWP wakeup. The call to LWP_;NoYieldSignal will happen synchronously. It is safe for an LWP to check for some condition and then go to sleep waiting for a Unix signal without having to worry about delivery of the signal happening between the check and the call to LWP_;WaitProcess.

10.3.6 IOMGR_;CancelSignal – Cancel association between Unix signal and LWP event

Call:

IOMGR_;CancelSignal(@w<**in** int signo>, *)*

Parameters:

signo

The Unix signal that should no longer be converted.

Completion Codes:

LWP_;SUCCESS

The association was cancelled,

LWP_;EBADSIG

signo is out of range or LWP_Signal has not been called on it,

Description:

This function cancels the association of a Unix signal and an LWP event. After calling this function, the Unix signal *signo* will be handled however it was handled before the corresponding call to LWP_Signal.

Chapter 11

The Timer Package

The timer package contains a number of routines that assist in manipulating lists of objects of type `struct TM_Elem`. TM_Elems (timers) are assigned a timeout value by the user and inserted in a package-maintained list. The time remaining to timeout for each timer is kept up to date by the package under user control. There are routines to remove a timer from its list, to return an expired timer from a list and to return the next timer to expire. This specialized package is currently used by the IOMGR package and by the implementation of RPC2. A timer is used commonly by inserting a field of type `struct TM_Elem` into a structure. After inserting the desired timeout value the structure is inserted into a list, by means of its timer field.

11.1 A Simple Example

```
static struct TM_Elem *requests;

. . .

    TM_Init (&requests);        /* Initialize timer list */
    . . .
    for (;;) {
        TM_Rescan (requests);   /* Update the timers */
        expired = TM_GetExpired (requests);
        if (expired == 0) break;
        . . . process expired element . . .
    }
```

11.1.1 Timer Primitives

11.1.2 TM_Init – Initialize a timer list

Call:

void TM_Init(@w<**out** struct TM_Elem **list>, *)*

Parameters:

list

The list to be initialized

Completion Codes:

0

ok,

-1

not enough free storage

Description:

The specified list will be initialized so that it is an empty timer list. This routine must be called before any other operations are applied to the list.)

11.1.3 TM_Final – Finalize a timer list

Call:

void TM_Final(@w<**out** struct TM_Elem **list>, *)*

Parameters:

list

The list to be finalized

Completion Codes:

0

ok,

-1

*list was 0 or list was never initialized

Description:

Call this routine when you are finished with a timer list and the list is empty. This routine releases any auxiliary storage associated with the list.)

11.1.4 TM_Insert – Release a read lock,

Call:

void TM_Insert(**in** struct TM_Elem *list>, @w<**in out** struct TM_Elem *elem *)*

Parameters:

list

The list into which the element is to be inserted

elem

The element to be initialized and inserted

Completion Codes:

The element *elem* is initialized so that the *TimeLeft* field is equal to the *TotalTime* field. (The *TimeLeft* field may be kept current by use of TM_Rescan.) The element is then inserted into the list.

Description:

The element *elem* is initialized so that the *TimeLeft* field is equal to the *TotalTime* field. (The *TimeLeft* field may be kept current by use of TM_Rescan.) The element is then inserted into the list.)

11.1.5 TM_Rescan – Update *TimeLeft* fields of entries on a timer list and look for expired elements

Call:

void TM_Rescan(@w<**out** struct TM_Elem **list>, *)*

Parameters:

list

The list to be updated

Completion Codes:

Description:

This routine will update the *TimeLeft* fields of all timers on *list*. (This is done by checking the time of day clock in Unix.) This routine returns a count of the number of expired timers on the list. This is the only routine (besides TM_Init that updates the *TimeLeft* field.)

@Pascall(Tag=<TMGetExpired>, Callname=‘TM_GetExpired, Abstract=‘Return an expired timer from a list, Parmlist=‘@w<**in** struct TM_Elem *list>, Type=‘struct TM_Elem *,

list

The list to be searched, Text=‘The specified list will be searched and a pointer to an expired timer will be returned. 0 is returned if there are no expired timers. An expired timer is one whose *TimeLeft* field is less than or equal to 0.)

11.1.6 TM_GetEarliest – Return the earliest timer on a list

Call:

void TM_ GetEarliest(@w<**out** struct TM_Elem **list>, *)*

Parameters:

list

The list to be searched

Completion Codes:

Description:

This routine returns a pointer to the timer that will be next to expire – that with a smallest *TimeLeft* field. If there are no timers on the list, 0 is returned.)

11.1.7 TM_eql – See if 2 timevals are equal

Call:

*void TM_ eql(***in** struct timeval *t1, **in** struct timeval *t2 *)*

Parameters:

t1

a timeval

t2

Another timeval

Completion Codes:

Description:

This routine returns 0 if and only if *t1* and *t2* are not equal.

Chapter 12

The Preemption Package

The preemption package provides a mechanism by which control can pass between light-weight processes without the need for explicit calls to LWP_DispatchProcess. This effect is achieved by periodically interrupting the normal flow of control to check if other (higher priority) procesess are ready to run.

The package makes use of the *interval timer* facilities provided by 4.2BSD, and so will cause programs that make their own use of these facilities to malfunction. In particular, use of *alarm (3)* or explicit handling of *SIGALRM* is disallowed. Also, calls to *sleep (3)* may return prematurely.

Care should be taken that routines are re-entrant where necessary. In particular, note that stdio (3) is not re-entrant in general, and hence light-weight processes performing I/O on the same FILE structure may function incorrectly.

12.1 Key Design Choices

- The package should not affect the nonpreemptive scheduling behaviour of processes which do not use it;

  It must be simple and @u[fast], with a minimum of extra system overhead;

  It must support nested critical regions;

  Processes using the package are assumed to be *co-operating*.

12.2 A Simple Example

```
#include <sys/time.h>
#include "preempt.h"

        ...

        struct timeval tv;

        LWP_Init (LWP_VERSION, ... );
```

```
tv.tv_sec = 10;
tv.tv_usec = 0;
PRE_InitPreempt (&tv);
PRE_PreemptMe ();

...

PRE_BeginCritical ();

...

PRE_EndCritical ();

...

PRE_EndPreempt ();
```

12.3 Preemption Primitives

12.3.1 PRE_InitPreempt – Initialize the preemption package,

Call:

int PRE_InitPreempt(@w<**in** struct timeval *slice> *)*

Parameters:

slice

The period of the implicit scheduler calls. Use NULL to obtain a useful default.

Completion Codes:

LWP_SUCCESS

All went well

LWP_EINIT

LWP_Init was not called

LWP_ESYSTEM

A system call failed

Description:

This routine must be called to initialize the package (after the call to LWP_Init).

12.3.2 PRE_EndPreempt – Finalize the preemption package,

Call:

int PRE_EndPreempt( )

Parameters:

None

Completion Codes:

LWP_SUCCESS

All went well,

LWP_EINIT

LWP_Init was not called,

LWP_ESYSTEM

A system call failed

Description:

This routine finalizes use of the preemption package. No further preemptions will be made. Note that is not necessary to make this call before exit - it is provided only for those applications that wish to continue after turning off preemption.)

12.3.3 PRE_PreemptMe – Mark a process as a candidate for preemption,

Call:

PRE_PreemptMe( )

Parameters:

Completion Codes:

Description:

Text=‘This is a macro that marks the current process as a candidate for preemption. It is erroneous to invoke PRE_PreemptMe () if LWP_Init has not been called.)

12.3.4 PRE_BeginCritical – Enter a critical section

Call:

PRE_BeginCritical( )

Parameters:

Completion Codes:

Description:

This routine places the current LWP in a *Critical Section*. Upon return, involunatry preemptions of this process will no longer occur. Note that this is a macro and that LWP_Init must have been

12.3.5 PRE_EndCritical – Leave a critcal section,

Call:

PRE_EndCritical(,)

Parameters:

Completion Codes:

Description:

This routine leaves a critical section previously entered with PRE_BeginCritical (). If involuntary preemptions were possible before the matching PRE_BeginCritical (), they are once again possible. Note that this is a macro and that LWP_Init must have been previously invoked.)

Chapter 13

The Fast Time Package

The Fast Time package allows the caller to find out the current time of day without incurring the expense of a kernel call. It works by mapping the page of the kernel that has the kernels time-of-day variable and examining it directly. Currently, this package only works on Suns. You may call the routines on other machines, but they will run more slowly.

The initialization routine for this package is fairly expensive since it does a lookup of a kernel symbol via nlist (). If you have a program which runs for only a short time, you may wish to call @FT(Init) with the `notReally` parameter true to prevent the lookup from taking place. This is useful if you are using another package that uses Fast Time (such as RPC2).

13.1 Fast Time Primitives

13.1.1 FT_Init – Initialize the Fast Time package

Call:

*int FT_Init(***in** int printErrors>, @w<**in** int notReally *)*

Parameters:

printErrors

Print error messages on stderr if somethingn goes wrong

notReally

Dont really do the memory mapping. Make **FT_GetTimeOfDay**

Completion Codes:

0

No problems

-1

Error in initialization

Description:

This call mmaps the kernel page with the time of day variable. If the routine returns -1, calls to **FT_GetTimeOfDay** will still work properly, but will make a kernel call.)

13.1.2 FTGetTimeOfDay – Get the time of day from the mapped kernel pages

Call:

int FTGetTimeOfDay( )

Parameters:

tv

Where to put the time of day

tz

Where to put the time zone information

Completion Codes:

0

No problem

-1

Something went wrong

Description:

This function has the same calling sequence as the kernels **gettimeofday** routine. If the **tz** parameter is not zero, or if initialization failed, or if the **notReally** parameter in the initialization was true, then this routine calls **gettimeofday**. Otherwise, it looks in the mapped page of the kernel to get the time of day.>

Chapter 14

RPC2 Internals

14.1 Background

This chapter describes the internal structure of RPC2. It describes the most relevant data structures and routines used in RPC2. This chapter will not be of interest to the average user of RPC2. It is intended for system programmers who are interested in knowing the internal workings of RPC2. It is not a substitute for source code, but is an annotated guide for the RPC2 source code. As previously mentioned, the RPC2 package is used in conjunction with the LWP Package and its components, the IOMGR package and the TM package. This document refers to these packages only in relation to RPC2. Discussion of side effects is deferred until Chapters 16 (XXX) and 17 (XXX).

RPC2 consists of two relatively independent components: a Unix-based run time library written in C, and a stub generator RP2Gen. The RPC2 runtime consists of the base RPC2 code and a set of routines for each of the side effects described. The run-time system is self-contained and usable in the absence of RP2Gen. In this chapter we will describe some of the routines that make up the run time library.

The LWP package supports multiple nonpreemptive threads of control within a single Unix process. When a remote procedure is called, the calling LWP is suspended until the response is received. While the LWP is suspended, the RPC2 runtime system internally yields control and other LWPs can make concurrent RPC requests. Hence multiple RPCs can be in progress simultaneously. This becomes particularly important when long running side effects occur.

14.2 The LWP package

14.2.1 Basic Routines

The LWP routines that are most relevant to RPC2 are the following.

@begin(itemize) *LWP_Init* - initializes the LWP package, and turns the calling process into the initial thread with the specified priority.

LWP_ CreateProcess - creates a new LWP thread. This call causes the scheduler to be invoked.

LWP_ DispatchProcess - the calling thread yields to the LWP scheduler.

LWP_ NoYieldSignal - signals an event but does not yield to the scheduler. @end(itemize)

Both the RPC2 and the LWP package are entirely outside the Unix Kernel, and depend only on the 4.3 BSD interface. At present RPC2 runs on the DARPA IP/UDP protocol via Unix sockets.

14.2.2 IOMGR Routines

Routines in the IOMGR package allows LWPs to wait on various Unix events. When the package is initialized, an IOMGR thread is created. This thread runs at the lowest priority. In order for the IOMGR package to function correctly, all other threads must run at a higher priority.

A common mistake in user code is to assume that yielding by a user LWP (via LWP_DispatchProcess LEFTPAREN) RIGHTPAREN will allow the IOMGR LWP to run; this will be true **only** if the user LWP runs at the lowest priority. In all other cases, the routine LWP_Poll () must be called prior to the LWP_DispatchProcess. The call to LWP_Poll () executes the same code that would have been executed by the IOMGR LWP, had it been given a chance to run. The LWP_DispathProcess () allows LWPs waiting on IOMGR events to run, if the LWP_Poll () detects such events.

The IOMGR call that is most relevant to the RPC2 package is the *IOMGR_ Select*. The *IOMGR_ Select* allows a light-weight process to wait on the same set of events that the Unix select waits on. The parameters are the same. The *IOMGR_ Select* puts the caller to sleep until no user processes are active. At this time the IOMGR thread, which runs at the lowest priority, wakes up, coaleses all the LWPs blocked by a select performs a single *select* and wakes up all processes affected by the result.

14.2.3 TM Routines

The Timer package is used to create and maintain timers for different events associated with RPC2 calls (for instance timeouts). This package contains a number of routines that assist in manipulating lists of timers. The timers are assigned a timeout value by the user and inserted in a list that is maintained by the package. There are routines to check for expired timers, to update timers etc. The TM package routines that are relevant to RPC2 are the following.

@begin(itemize) *TM_ Insert* - initializes a timer element and inserts it into a timer list.

TM_ Rescan - updates the timer elements in the specified list and looks for expired elements.

TM_ GetExpired - returns an expired timer from a list.

TM_ GetEarliest - returns the earliest timer on a list. @end(itemize)

14.2.4 The RPC2 package

Chapter 2 (XXX) of this manual contains two examples of RPC2 subsystems. We consider the example in Section 2.6.1 (XXX). For this example, RP2Gen generates the server and client stubs shown below.

```
<sect2>Client Stub for Example <ref id="Example-rtime<p>" name="XXX">
@begin(program)
```

```
@include(rtime.client.c.mss)
@end(program)

<sect2>Server Stub for Example <ref id="Example-rtime<p>" name="XXX">
@begin(program)
@include(rtime.server.c.mss)
@end(program)
```

14.2.5 Initialization and Thread Creation

RPC2 provides logical connections for the client to communicate with the server. The client and server communicate via internet sockets. Both at the client and at the server end, the sockets are monitored by a LWP called the SocketListener thread. The SocketListener thread which is created during initialization is an integral part of the RPC2 package. In addition to monitoring the socket, it receives and decodes the packet and then notifies the appropriate LWP which is awaiting a packet. Given that multiple LWPs can be waiting for packets, the job of the SocketListener thread is to demultiplex incoming packets among multiple waiting LWPs.

Besides the SocketListener thread, there is an IOMGR thread. Both these threads are created during initialization of the RPC2 package, via the *RPC2_Init* primitive. Besides these internal threads, clients and servers may have any number of other threads limited only by the amount of memory. A server may be organized in many ways: one thread per subsystem, a pool of threads servicing any of a number of subsystems, and so on.

The interplay between threads is as follows. The client makes an RPC and goes to sleep, thereby handing control to the SocketListener thread. This thread then blocks on an *IOMGR_Select* transferring control to the IOMGR thread, which then blocks on a UNIX select. On receiving a response, the IOMGR thread unblocks from the select, wakes up the SocketListener thread, which receives and decodes the response and then hands control back to the client thread.

At the server, the server thread blocks waiting for a request, hence transferring control to the SocketListener. As with the client, the SocketListener calls *IOMGR_Select* and blocks, thereby handing control to the IOMGR thread. When a request arrives, the IOMGR thread unblocks from the select, hands control to the SocketListener which after receiving and decoding the packet transfers control back to the server thread.

The IOMGR thread has the lowest priority. All other threads are usually at (the same) higher priority level.

RPC2_Init

@begin(itemize) By calling *LWP_Init*, this routine converts the calling process into the initial thread thereby creating the client thread at the client end and the server thread at the server end.

Creates and sets up an Internet socket through which requests are sent and responses received.

Creates the SocketListener thread using the *LWP_ CreateProcess* primitive.

Initializes the IOMGR package, thereby creating the IOMGR thread.

By calling *TM_Init* initializes the rpc2_TimerQueue @foot(RPC2_xxx are the routines that are externally visible and can be used by the user. rpc2_xxx are routines that are internal to the package.) list which is then used to maintain all the timers for the RPC calls.

Initializes the random number generator used for generating sequence numbers for packets.

Sets the retry parameters according to the algorithm described in Chapter 15 (XXX). @end(itemize)

14.2.6 Data Structures used in RPC2

A number of data structures are used in RPC2, the most important of which are the Connection Entry (CEntry) the RPC2_PacketBuffer, and the SocketListener Entry (SLEntry). We consider each one of these in turn.

@paragraph(CEntry) RPC2 provides logical connections for the client to communicate with the server. Associated with each logical connection are two data structures of type CEntry; one at the client end and the other at the server end. All information relevant to the connection is entered into these data structures. This includes information about the security of the connection, the state and identity of the connection itself, the remote site that it is connected to, the sequence number of the next packet expected on the connection and so on. The CEntry is allocated using the *rpc2_AllocConn* call.

@paragraph(RPC2_PacketBuffer) The RPC2_PacketBuffer shown in rpc2.h, defines the structure of the request-response packets exchanged during the RPC calls. The RPC2_PacketBuffer consists of a Prefix, Header and Body. The Prefix is of fixed length and contains information about the PacketBuffer. It is used internally by the runtime system, and is not transmitted.

The Header is also of fixed length, and contains a number of fields some of which are described below. The fields pertaining to the connection are the RemoteHandle and LocalHandle. The RemoteHandle and the LocalHandle refer to the connection ids on the peer host and the local host respectively. The LocalHandle field corresponds to the local connection id on which the packet is sent. In this case, the LocalHandle in the request packet corresponds the clients local connection id. In the response packet this field corresponds to the servers local connection id.

Other fields pertain to the packet itself including the length of the packet, the sequence number, the type of packet (determined by the Opcode), the length of the packet. These fields are encrypted on secure connections. The packet length field is used by the peer host to efficiently allocate packet buffers. The Body is of arbitrary size, and is used to transmit the input and output parameters of the RPC.

The packet buffer is allocated by calling *RPC2_AllocBuffer* and is freed by *RPC2_FreeBuffer*.

@paragraph(SLEntry) SLEntries are data structures used for communication between the user LWPs and the SocketListener. There are three types of SLEntry data structures; REPLY, REQ, OTHER. Each of these are described below.

@begin(itemize) REQ: The SLEntry of type REQ is used by the server in the *RPC2_GetRequest* call (which is described later). In the *RPC2_GetRequest* the server thread allocates a SLEntry, associates a filter and timeout with it, activates it and goes to sleep until the SLEntry is modified. The SLEntry is modified by the SocketListener when a packet arrives or a timeout occurs. A return code associated with the SLEntry is used to indicate if the timeout occurred or a request arrived.

OTHER: The SLEntry of type OTHER is associated with a specific connection. It is created and destroyed by the user LWP. Typically it is used by a client making a request on a particular connection. The client allocates a SLEntry of this type. A timeout equal to the retry parameter is associated with the SLEntry. The client then activates it and goes to sleep until the SLEntry is modified. As with the REQ type SLEntry this modification is done by the SocketListener when either a packet arrives, a BUSY is received or a timeout occurs. The SLEntry is then deactivated with the appropriate return code. If there is an overall timeout associated with the call (as specified by the user in the *RPC2_MakeRPC* call), before the client goes to sleep, it allocates, sets this timeout value and activates another SLEntry of type OTHER.

REPLY: The SLEntry of type REPLY is associated with a specific connection, is created by the user (server) LWP in the *RPC2_SendResponse* call, and is deactivated by the SocketListener in the *rpc2_ExpireEvents* call. @end(itemize)

The type of SLEntry is specified in the field *Type* of the SLEntry structure.

The SLEntries are allocated using the *rpc2_AllocSle* call. They are activated and deactivated using the *rpc2_ ActivateSle* and *rpc2_DeactivateSle* calls respectively. They are freed using the *rpc2_FreeSle* call.

@paragraph(Creating Data Structures) To avoid the cost of mallocing a structure each time it is required, lists are maintained for each of the three mentioned data structures. These lists are called FreeLists (SLFreeList for SLEntries, ConnFreeList for CEntry, rpc2_PBList for PacketBuffers), and contain the addresses of entries which were created previously, but are no longer in use. These lists are first checked for a free entry before mallocing space. When an entry is no longer required it is returned to the respective FreeList.

14.2.7 Binding

In order for the client to communicate with the server, it is first necessary to establish a logical connection between them. These connections are established using the RPC2_Bind primitive.

RPC2_Bind @foot(Here we only consider the unauthenticated and unencrypted security level RPC2_OPENKIMONO.)

@begin(itemize) Creates a new connection and binds to a server on a remote host. The identification of the server is specified by a hostname, port number and subsystem. This along with the required security level is mapped into a unique integer which we will refer to as the *Local Handle*. The Local Handle can also be thought of as the connection identifier, and is used by the client for future references to this connection.

Creates a data structure of type CEntry by calling *rpc2_AllocConn*. The Local Handle is entered into CEntry field *UniqueCID*.

Allocates and packs into a packet buffer, the information required by the server to set up the connection.

Allocates and activates a SLEntry data structure of type OTHER. @end(itemize)

Next, the RPC2_Bind packet is sent to the server. The details of the calls required to send a packet will be discussed later while describing the *RPC2_MakeRPC* call. After sending the packet, the client thread goes to sleep.

The server responds to this call by creating a corresponding connection entry. When the SocketListener thread detects a packet, it receives it and processes it. Processing involves sanity checking and decoding the packet. This packet is decoded as a Bind request. The SocketListener thread allocates a CEntry data structure, and a unique connection identifier (using the *rpc2_ AllocConn* call). Information relevant to this connection such as the clients id, subsytem id etc is entered into CEntry by calling *rpc2_MakeConn*. The server then allocates a Packet Buffer, enters the connection identifier in the Local Handle field of the Packet Header, and sends the response to the client. The connection identifier is used by the client in subsequent requests to the server on this connection. All calls associated with the server will be discussed later.

At the client end, the SocketListener thread detects a packet, receives and processes it. Processing involves sanity checking and decoding the packet. This packet is decoded as the response to the Bind packet, and the connection identifier sent by the server is entered into the CEntrys field PeerHandle. Now the connection has been set up.

14.2.8 Clien-Related RPC Calls

A client wishing to make a remote procedure call on this connection makes a local procedure call to the client stub and specifies the connection (or Local Handle) on which it wishes to make a call. The client stub allocates a packet buffer, marshalls the arguments into a packet and calls the *RPC2_MakeRPC primitive*.

RPC2_MakeRPC

@begin(itemize) Obtains the connection entry associated with the corresponding Local Handle. Initializes the request packet buffer by filling in the appropriate header information using *rpc2_InitPacket*.

Allocates a SLEntry of type OTHER.

@Begin(Multiple) Sends the request packet by calling the *rpc2_SendReliably* primitive.

rpc2_SendReliably @End(Multiple) @begin(itemize) Transmits the request packet using the *sendto* system call, and activates the SLEntry. The SLEntry is activated using *rpc2_ActivateSle* where the timeout value is set to the retransmission timeout @g(b)1. The entry is placed in the timer list managed by the TM package.

The client thread then blocks until either a response is received or a retransmission timeout occurs. When the client thread unblocks, it first checks to see if the overall timeout of the call has expired (i.e the timeout specified in *RPC2_MakeRPC*). If this occurs, then this timer is deactivated and removed from the list and the call returns to *RPC2_MakeRPC*. If the call has not timed out, the return code on the SLEntry is used to find out the outcome of the call. There are four possibilities: @begin(itemize) A negative acknowledgement (NAK) is received in which case control is transferred back to the *RPC2_MakeRPC call*.

A response is received in which case control is transferred back to the *RPC2_MakeRPC call*.

A BUSY response is received. A BUSY is sent to a client by the server in response to clients retries. This is to indicate to the client that the server is still alive and connected to the network. In this case the SLEntry is again activated with the retransmission timeout set to @g(b)0.

A retransmission timeout occurs. The client calls *rpc2_CancelRetry* to check if a side effect has heard from the server during the interval. If this is the case, then the client behaves as if it received a BUSY at the time of the last server response in the side effect. In other words, suppose while the thread was blocked the client received a packet from the server as part of the execution of a side effect. Let that time be t. Suppose the time is now n. Then the SLEntry is reactivated with the retransmission timeout set to @g(b)0-(n-t). If the client has not heard from a side effect in the interval, then the SLEntry is again reactivated with the retransmission timeout set to the next value of @g(b), until N retries have been attempted. After N retransmissions, the server is assumed dead and the call transfers control back to *RPC2_MakeRPC*.

@end(itemize) @end(itemize) @end(itemize) On returning to the *Make_RPC* call, the return code indicates: @begin(itemize) If an overall timeout occurred (i.e. the return value of *rpc2_SendReliably* is RPC2_TIMEOUT), *RPC2_MakeRPC* frees the corresponding SLEntry and returns control to the client stub with the outcome of the RPC call (i.e. the return code) specified as RPC2_TIMEOUT.

If an overall timeout did not occur (i.e. the return value of *rpc2_SendReliably* is RPC2_SUCCESS), then the response to the RPC call could be one of three. The ReturnCode field in the SLEntry indicates which of the three cases occured. @begin(itemize) The response was received, indicated by return code ARRIVED. Control is transferred to the client stub with the return code of the RPC call set to RPC2_SUCCESS.

A negative acknowledgment, indicated by return code NAKED. Control is transferred to the client stub with the return code of the RPC call set to RPC2_NAKED.

No response because of either the server being dead or the network being down, indicated by return code TIMEOUT. Control is transferred to the client stub with the return code of the RPC call set to RPC2_DEAD.

@end(itemize) In all cases the SLEntry is freed and control is transferred back to the client stub. The arguments of the call are unmarshalled and returned to the calling program.

@end(itemize)

14.2.9 Server Related RPC Calls

The server program is given in Section 2.1.2. After initialization, in order for the client to bind to a server, the server must first export the subsytem. This is done using the *RPC2_Export* primitive. The server now ready to receive requests, makes a call to *RPC2_ GetRequest*.

RPC2_GetRequest

@begin(itemize) First, checks to see if there are any *HeldRequests*. When a request arrives, and there are no threads available to service it, it is added to a *HoldList*. We refer to these requests as HeldRequests. If there are no HeldRequests, it looks for new requests. This is done by making a call to *GetNewRequest*, which allocates and activates a SLEntry of type REQ, and blocks until either a timeout (specified in *RPC2_ GetRequest)* occurs or a request arrives.

If the request is a Bind request, the SocketListener thread after allocating a CEntry using the *Make_ Conn* call hands control to the server thread. The server then checks to see that it is a bind request. It then allocates a PacketBuffer and transmits the Bind packet (called the Init2 packet) to the client. This is done by calling the *SendOKInit2* primitive.

If a request arrives, on an already existing connection, control is transferred back to the main server program which then transfers control to the server stub. The request is then executed, a PacketBuffer is allocated and the response is sent to the client using the *RPC2_SendResponse* call.

@end(itemize)

RPC2_SendResponse

@begin(itemize) Takes as input the Local Handle from which it obtains the corresponding CEntry using the *rpc2_GetConn* call.

Records the relevant information into the response Packet Header and transmits the response packet using *rpc2_XmitPacket*.

Allocates and saves a packet for retry which is sent to the client if a duplicate request is received. @end(itemize)

@subsection (TM related RPC2 calls) In this section we describe some of the frequently used RPC2 primitives that deal with the calls in the TM package.

rpc2_ActivateSle

@begin(itemize) Sets the fields of the timer element TM_Elem (Section 12.2); sets the TotalTime field with the timeout value, sets the BackPointer field with the address of the SLEntry. If there isnt a timeout associated with the SLEntry, the TotalTime field is set to -1.

Checks to see if the timeout of the new TM_Elem is less than any of those already in the list (the timeout field of the TM_Elem is compared with the TimeLeft fields of all the other TM_Elems in the list). If so, the SocketListener thread is made to wait on this TM_Elem.

Inserts the timer element into the timer list rpc2_TimerQueue.

@end(itemize)

rpc2_DeactivateSle

@begin(itemize) @begin(multiple) Enters the specified return code to ReturnCode field of the SLEntry. The ReturnCode can be one of of the following.

- TIMEOUT : if the TimeLeft field of the TM_Elem is less than or equal to 0.

- ARRIVED : if the response has arrived.

- KEPTALIVE : if a BUSY response is received.

- NAKED : if a negative acknowledgement is received.

If TM_Elem is timed (i.e TotalTime field is not equal to -1), it removes it from Timer list. @end(multiple) @end(itemize) Note that all SLEntries are deactivated by the SocketListener.

rpc2_ExpireEvents

This routine is called by the SocketListener. @begin(itemize)

Using *TM_Rescan*, it updates the TimeLeft fields on all the timers in the rpc2_TimerQueue. It then checks for expired timers and deactivates the SLEntries with the expired timers.

@end(itemize)

14.2.10 SocketListener

An integral part of the RPC2 code is the SocketListener. As previously mentioned, the SocketListener thread monitors the IP Socket through which packets are sent and received. A SocketListener thread is created at the client and the server when RPC2 is initialized. Although the functionality of the SocketListener at the client and at the server are similar, some differences do exist. We first discuss the section of the code that is common to both.

@begin(itemize) On creation, the SocketListener thread yields to the main thread (the client thread for the client and server thread for the server). It gains control when the main thread goes to sleep.

It first polls the IP socket (polling select) to see if any packets are in the socket. This is done using the *MorePackets* primitive.

If there are no packets waiting in the socket, by calling *rpc2_ExpireEvents* it checks the Timer list to see if there are any expired events. Expired events refer to timeouts that might have occurred since it was last checked.

It then finds the earliest event on the Timer list, and blocks on an *IOMGR_Select* until either a packet arrives or the event times out. This is done by calling *PacketCame*.

@begin(multiple) If a packet is present in the socket, the SocketListener processes the packet by calling *rpc2_ProcessPacket*. The *rpc2_ProcessPacket* routine calls on a number of primitives and is described below.

rpc2_ProcessPacket @end(multiple) @begin(itemize) The *PullPacket* primitive allocates a Packet Buffer for the incoming packet. The packet is then received using the *rpc2_RecvPacket* call which in turn uses the *recvfrom* system call.

PoisonPacket is called to sanity check the packet. This entails checking the version of RPC2 used, and the packet header.

The RemoteHandle field of the packet is used to determine if the packet is a new bind request. If not, the CEntry is obtained using the *rpc2_ GetConn* call. Once the CEntry is obtained the packet is decoded by calling *DecodePacket*.

@end(itemize) @end(itemize)

DecodePacket

Here is where the differences arise between the client and the server. We first consider the server and then the client.

@begin(itemize) @begin(multiple) For a server, the packet could be a request for a new connection, a new request, or a retransmission . A retransmission could occur because the response packet was lost or the server is busy. In the first case, the response is retransmitted. If the server is busy, a BUSY packet is sent out. The reason for having the BUSY mechanism is the following.

When the server is heavily loaded, or a long side effect (such as a large file transfer) is occurring, responses to clients requests may take an arbitrary time. In the meantime however the client might timeout and retransmit. After a number of retransmissions, if no response is received, the connection is declared to be dead. To prevent this from happening, the server sends out a BUSY if it sees a retransmission, thereby informing the client that it is still alive. When the client receives a BUSY it backs off from retransmitting for time @g(b)0. Another advantage of the BUSY is that it prevents clients from flooding the server with retransmissions, and hence decreases unnecessary load on the server.

If the packet is a bind request, a CEntry is created. The SocketListener thread then checks if any server threads are available. If so it deactivates the corresponding SLEntry. If not it puts the request in the HeldRequest queue. Similarly for the new request, it checks for available server threads and depending on the availability it either deactivates a SLEntry or puts the request on a HoldList.

At this point the SocketListener thread does not yield control. It polls to see if any packets are waiting in the socket. If so, it processes them, if not it checks on expired events and blocks on the *IOMGR_Select* and then yields to the server thread. The rationale for the server thread not yielding control immediately after deactivating the SLEntry is as follows. If retransmissions are present, it is generally better to send out the BUSY responses as soon as possible. Hence the SocketListener thread decodes all the packets waiting in the socket and responds to the retransmissions before yielding control.

@end(multiple) For a client, the packet could be a response to a bind request, a response to a request or a BUSY from the server or a NAK. For each of these, the SLEntry waiting on the event is found and deactivated by calling *rpc2_DeactivateSle*. As in the case of the server, at this point the SocketListener does not yield control. It polls the socket to see if any packets are waiting in the socket. If a packet is waiting it processes the packet, otherwise it checks on expired events, blocks on an *IOMGR_Select* and then yields control to the client thread. @end(itemize)

Chapter 15

Failure Detection

Three hazards face the user of an RPC package: @begin(Enumerate) The communication medium may fail.

The peer process at a remote site may crash. @end(Enumerate)

A key problem in RPC is reliably detecting either of these events when an RPC call is in progress. Detection of failures in the absence of RPC calls in progress is an orthogonal issue, and can be reduced to this issue by generating artificial keepalive RPC calls.

Ideally, the detection of these failures should be independent of the specific RPC call in progress. In other words, as long as we are sure that communication medium is not broken and that the remote server process is alive, we should not care how long it takes to receive the reply to an RPC request. At the same time failures should be detected as soon as possible, so that suitable recovery actions can be performed. The following paragraphs show this goal is achieved in RPC2.

15.1 Retransmission Algorithm

When the RPC2 runtime system receives a retry packet for a request it is already working on, it responds with a **Busy** packet. There are two constants *B@-[total]* and *N*. These constants are set in @RPC2(Init)()], with suitable defaults built in. These semantics of these two constants are: @begin(Enumerate) Communication failure is declared if *N* successive retries of a packet fail to provoke any kind of response. The response may be a reply, a **Busy** packet, an acknowledgement if the packet being sent is a reply, or an implicit piggy-backed acknowledgement.

Site failure is declared if silence is observed for a total period of time in the range *B*@-[*total*] to *2B@-[total*]. @end(Enumerate)

RPC2 does not try to accurately distinguish between site failure and communication failure: one may masquerade as the other, and a single failure @RPC2(DEAD) reflects both cases. Loosely speaking, *N* characterises the probability of packet loss in the communication medium, while *B*@-[*total*] characterises how sluggish a server may get before it is declared dead.

Given *B@-[total]* and *N*, we can determine *B@-[1], B@-[2], ... B@-[N]* such that @w{*B@-[1] + B@-[2] + B@-[3] ... B@-[N]*} = @w{*B@-[total] and B@-[i] < B@-[i+1.]*} Each *B@-[i]* is a retry interval and the progressive lengthening of these intervals is to allow for transient overloads at remote sites. In RPC2, *B*@-[*i+i*] = *2B*@-[*i*]. In practise we may place a minimum bound on the values for *B*@-[*i*]s, to avoid send out packets too close to each other.

The RPC2 packet transmission algorithm is based on these concepts and is outlined as follows: @begin(ProgramExample)

while (TRUE) { for (i = 0; i < N; i++) { send (packet); awaitresponse (B@-[i]); if (reply or lastack arrived) quit; if (BUSY arrived) break; } if (i >= N) goto TimeOut; sleep (B@-[total]); }

TimeOut: mark connection @RPC2(DEAD); mark dead all other connections to this (host, portal) pair; @end(ProgramExample)

Failure is detected in time *B]@-[total]* if the remote site dies just after the *sleep ()* call ends. If the failure occurs immediately after the remote site sends a **Busy** packet, failure is detected after a total of*2B@-[total.*] These cases bound the time it takes to detect failure. Failure is also declared if all *N* of the retries are lost due to communication failure. This will occur in a time exactly equal to *B@-[total]*.

How does this mesh with side effects? The above algorithm will work regardless of the duration of a side effect as long as there is some response from the server in RPC (a **Busy packet**) or the side effect (any packet) at intervals of *B*@-[*total*]. Note that it is immaterial whether the side effect involves asynchronous Unix processes or not. If such processes are involved their failure will be detected (perhaps as @RPC2(DEAD) failures or in other ways) and reported by the remote server explicitly as @RPC2(SEFAIL2). Only if the remote server is itself dead or unreachable is the RPC return code @RPC2(DEAD) and this will occur no later than *2B*@-[*total*] after the failure. In SFTP or where TCP or other protocols are being used for side effects, the failure detection mechanisms of these protocols will be relied upon to detect side effect failure.

The tables at the end of this chapter show how the *N* retransmissions take place within *B@-[total]*, for typical values of *N* and *B@-[total]*. The original attempt is at time 0. The numbers in parentheses indicate the time (@i{B@-[N]}) that RPC2 waits after the transmission of the last retry, before declaring failure. A lower limit of 300 milliseconds for the retry interval is assumed.

15.1.1 Adjusting Retransmission Intervals

Retransmission intervals in RPC are maintained on a connection-specific basis. They are initialized with the values in the tables, given the number of retries and total timeout specified in *RPC2_Init*. There is a minimum bound of 300 milliseconds for the *B@-[i]* so that packets are not sent out too close to each other. For slower networks, this may still be too short an interval. The minimum interval should be at least a packet round trip time (RTT). To estimate the minimum interval, an RPC client collects RTT observations on bind1-bind2, bind3-bind4, initmulticast-reply, and request-reply pairs. A "smoothed RTT" is calculated (a la TCP) and used to set the lower limit on RPC retransmission intervals. Whenever the RTT estimate is changed, the retransmission intervals are recalculated, always maintaining the given timeout. The maximum value for the lower limit is @g(b)0, so the client has time for at least one retry.

RTT observations are collected by timestamping packets as described for TCP in @cite(RFC1323). The client stamps an outgoing packet with the current time. The server echoes that timestamp on the next packet back to that client. The client computes the RTT observation by subtracting the echoed stamp from the current time.

Two longwords in the packet header are used to carry timestamps. @footnote(RPC uses one for collecting RTT observations. It uses the other field to send the time for the bind sequence to the server. The bind time is used to initialize side effects. SFTP uses both timestamp fields. This is described further in chapter 16 (XXX).) Time units must be meaningful for the range 10Mbps - 1200bps. To keep the stamp small, time is measured relative to connection creation. The resolution of the stamp is 10 msec. Though it may be simpler to maintain a finer granularity unit (the maximum packet size is 2900 bytes, which over Ethernet takes 2.32 ms to transmit), the clock resolution on most machines is 10-20 ms. The 10 msec unit is still finer than TCP, which uses 500 ms ticks. A connection can stay open over a year before the time "wraps around" using 10 msec units.

The RTT and RTT variance are stored scaled, as in TCP, for greater precision and ease of computation. We need a longword for each, unlike TCP (which uses a short). If we used a short, and scaled by 8, we would have only 12 bits for time (40.950 sec). This is roughly the RTT of a packet over a 1200 bps link: it takes 19.33 sec to transmit a 2900 byte packet.

For compatibility, if a packet arrives with a 0 timestamp, it means the sender of the packet does not maintain or use RTT estimates. RPC simply skips the RTT update, and the retransmission timeout stays at its original setting.

Timestamps should be set as late as possible and collected as early as possible. Packets leaving the client are stamped in *rpc2_SendReliably* (for multicast, in *mrpc_SendPacketReliably*). Retries are restamped. The RTT is updated and retransmission intervals adjusted from the socket listener, upon receipt of a bind2, bind4, or response.

Packets arriving at the server are checked in the *SocketListener*. If the request is a "good" init1, init3, or request, the timestamp is stashed in the connection entry and the request time is taken. The reply is stamped with the original timestamp plus the service time.

Requests that provoke busies are stamped with the incoming timestamp – the service time is assumed to be 0. The client gets additional observations this way. Erroneous requests and binds that will be rejected are sent back with null timestamps. In such cases, there is usually no connection state to update.

RPC does not currently take request and reply length into account in its RTT estimate. Because of this, the RTT has high variance at low bandwidths. Ideally, RPC would derive an RTT independent of length, then apply it given the length of the request and probable length of the reply.

@newpage @string(Included="Yes") @include(retry.mss)

Chapter 16

SFTP Internals

16.1 Background

An SFTP file transfer can take place from either an RPC2 server or a RPC2 client. To avoid confusion we will refer to the transmitting entity as the *source* and the receiver entity as the *sink*. RPC2 clients and servers are not regarded as peers. While an RPC2 client might be someones personal workstation, an RPC2 server could be serving a large user community. In an effort to improve scalability when more clients are added to the system, the servers will handle all SFTP flow control, irrespective if they are the source or the sink.

An RPC2 client can use SFTP to transfer a file simultaneously to more than one RPC2 server using IP multicasting. Multicast file transfers are only possible when the source is an RPC2 client. The sinks will send flow control information to the source, and it will adapt to the requirements of the slowest sink.

An SFTP file transfer is basically a cyclic exchange of data and acknowledgements. At the beginning of each cycle, the source will send a block of data packets. It will then wait for an acknowledgement to arrive. The acknowledgement will specify which packets the sink has received. The cycle then repeats. The source will now retransmit any packets that it knows that the sink did not receive, followed by a block of new packets.

When the source has transmitted a block of data packets, it will wait for the arrival of an acknowledgement. If the source is an RPC2 server and the acknowledgement does not arrive after a predetermined time, the source will retransmit the block of data packets. It basically acts as if it received an acknowledgement that indicated that the entire block of data packets had been lost.

If the source is an RPC2 client, however, it will wait passively for an acknowledgement to arrive. If the sink does not receive more data packets after a predetermined period of time, it will conclude that acknowledgement was lost in transit and retransmit it.

16.2 SFTP Code Structure

In this section we describe the SFTP code structure. Our description assumes that the reader is already familiar with the description of basic RPC2 internals in Chapter 14 (XXX).

16.2.1 Thread creation and Initialization

In the base RPC2, the RPC2 client and server communicate via Internet Sockets. Both at the client and at the server, the socket is created during initialization by calling RPC2_Init. A SocketListener thread is present at both ends to monitor these sockets.

When using SFTP, in addition to the above, there is another set of sockets created; one at the client and another at the server. These sockets are monitored by the SFTP_Listener. Both of these are created during the iniailization of the SFTP package.

Note that there are two independent channels of communication between the client and the server. The first channel (which we will refer to as the *RPC2 channel*) that is associated with the base RPC2 is used for making simple RPCs, for making RPCs requesting for the file transfers, for retransmissions and BUSYs. All other exchanges related to the file transfer are handled by the second channel which will be referred to in future as the *sftp channel*.

As previously mentioned, a file can be transferred from a client to a server or from a server to a client. In the first case, the server is the sink. In the second case, the client is the sink. The code however is not symmetric; i.e., the code executed when a client is the sink is slightly different from the code executed when a server is the sink. We describe both cases below.

16.2.2 Data Structures used in SFTP

In addition to the data structures used in RPC2 (on the RPC2 channel), SFTP uses a data structure called SFTP_Entry which is given in sftp.h. It contains fields relevant to the sftp channel such as the LocalHandle. Other fields include the state of the file transfer, the packet size, the window size and a number of others. It is created by the call sftp_AllocSEntry.

16.2.3 File transfer from server to client

This is the case in which the server is the source and the client is the sink. The client makes a request for the file by doing a *RPC2_MakeRPC* on the RPC2 channel. This is received by the servers SocketListener which wakes up a suitable LWP blocked on a *RPC2_GetRequest*. The LWP then calls the routine that is meant to handle this request. This routine contains calls to two routines, *RPC2_InitSE* and *RPC2_CheckSE*. *RPC2_InitSE* initializes certain internal data structures. *RPC2_CheckSE* handles the actual file transfer. The main procedure in *RPC2_CheckSE* which deals with the file transfer from SERVERTOCLIENT is the *PutFile* routine.

The *PutFile* routine sets some of the fields of the data structure SEntry, sets the transfer state of SEntry to *XferInProgress* (transfer in progress) and calls *sftp_SendStrategy*. This routine sends a set of packets, using a strategy described in the next section. After sending the first set of packets, a *while* loop is entered and executed as long as the transfer state of SEntry is still in *XferInProgress*. In the *while* loop, *AwaitPacket* and *sftp_SendStrategy* are called alternately. The *AwaitPacket* routine waits for either an ACK, NAK or for a timeout. If a timeout occurs, the packets are retransmitted using *sftp_SendStrategy*. If an ACK is received, the *sftp_AckArrived* routine is called. This routine advances the transmission window and checks to see if the transfer is complete. If so, it sets the SEntrys transfer state to *XferCompleted*, and the *while* loop is exited. Otherwise, the next set of packets is transmitted, after which control is yielded. Note that all these packets are sent on the SFTP channel, not the main RPC2 channel.

At the client end, the sftp_Listener detects a packet in the socket, receives it and processes it by calling *sftp_ProcessPacket*. This routine after receiving the packet calls the *ExaminePacket* routine. This routine sanity checks the packet, and identifies it as a DATA packet. It then calls *sftp_DataArrived* which sends the requested ACKs and writes the data to disk by calling the *WriteStrategy* routine. The sftp_listener yields after each packet it processes.

The packet from the client is received by the sftp_listener at the server which then calls the *ServerPacket* routine which modifies the appropriate SEntry. It then does an *IOMGR_Select*, and yields control. Control is then transferred to the LWP waiting on this packet, and the cycle continues.

16.2.4 File transfer from client to server

This is the case in which the server is the sink and the client is the source. As in previous case, the client makes a request for the file by doing a *RPC2_MakeRPC* on the RPC2 channel. This is received by the servers SocketListener which wakes up a suitable LWP blocked on a *RPC2_ GetRequest*. The LWP then calls the routine that is meant to handle this request. This routine contains calls to the two routines *RPC2_InitSE* and *RPC2_ CheckSE*. The *RPC2_InitSE* initializes some of the fields of the data structure. The main routine in *RPC2_ CheckSE* which deals with the file transfer from CLIENTTOSERVER is the *GetFile* routine.

The *GetFile* routine sets some of the fields of the data structure SEntry, sets the transfer state of SEntry to *XferInProgress* and sends a *START* packet to the client to tell the client that the server is ready to receive the file. It then enters a *while* loop which is executed as long as the transfer state of SEntry is still in *XferInProgress*. In the *while* loop, *AwaitPacket* and *sftp_DataArrived* are called alternately. The *AwaitPacket* routine waits for either a packet to arrive or for a timeout. If a timeout occurs, the ACK is retransmitted. If a DATA packet is received, the *sftp_DataArrived* routine is called. This routine in turn calls the *sftp_ WriteStrategy*. When the file transfer is eventually completed, the transfer state of the SEntry is set to *XferCompleted*, and the loop is exited.

The sftp_Listener at the client end receives the packet and decodes it, and calls the *ClientPacket* routine which in turn identifies the packet as an *SFTP_ START* packet. It then calls *sftp_ StartArrived* which sets some of the fields in the SEntry data structure and calls the *sftp_ SendStrategy* descirbed above. The sftp_Listener then block on an IOMGR_Select. Note that it patiently waits for an ACK from the server, and does not retransmit if it does not receive an ACK within a given time. What prevents the client from waiting forever is that communication exists between the client and the server on the RPC2 channel in the form of retransmissions and BUSYs. When an ACK arrives, it transmits the next set of packets.

The sftp_Listener at the server end receives the packet and calls the *ServerPacket* routine. This routine wakes up the appropriate LWP (which is blocked in the *AwaitPacket* call). @foot(Note that although the client sends a number of packets, the sftp_Listener receives and processes them one at a time; yielding control after each one. The same applies at the server end.)

Note that the role of the sftp_listener is different at the client end and at the server end. At the client end, the whole sftp transfer is handled by the sftp_listener. At the server end the sftp_listener receives and decodes the packet. Most of the sftp transfer is handled by the LWP thread.

16.3 Packet formats

All packets carry 32 bit sequence numbers. Data packets and control packets have independent sequence numbers. The sequence number series of the source and sink (s) are also independent of each other.

There are thus at least 4 sequences in a connection:

- Source to Sink, Data Source to Sink, Control Sink to Source, Data Sink to Source, Control

The Sink to Source sequence space is currently never used. When doing a multicast file transfer each sink will have independent sequence number series.

The sequence number for a particular packet type is incremented by one for each new packet of its type that is sent.

The *MOREDATA* flag will be set in each data packet except for the very last one. This is to facilitate end of file detection. If the *ACKME* flag is set on a data packet it requests an acknowledgement, *ACK*, from all of the servers.

Each *ACK* packet describes which packets have been received by the particular server. There should be little or no need to transmit an acknowledgement packet for each data packet. It is of particular benefit to limit the number of *ACK* packets given our single channel operating environment. The acknowledgement packets will contend with data packets going in the other direction.

Each acknowledgement packet has a 64-bit wide bitmask and an offset counter, *GotEmAll*. This counter is the highest sequence number of a data packet such that it and all preceding data packets have been received. The bitmask indicates which of the data packets with sequence numbers greater than *GotEmAll* that have been received. Each bit in the bitmask represents a single packet.

16.3.1 Protocol details

If the source is an RPC2 client it must first wait for permission from the sink (s) before it can transmit. This permission is granted by a special *START* packet.

The following counters are of relevance to the SFTP source protocol machine. *SendLastContig*, which is the sequence number of the latest packet to be moved out of the transmission window, and *SendMostRecent*, which is the sequence number of the data packet last sent. There are also three important transmission parameters: the transmission window size, *AckPoint*, and size of the *SendAhead* set.

When an SFTP source begins the transfer, *SendLastContig* and *SendMostRecent* will be equal. The packets in the *SendAhead* set are transmitted, and *SendMostRecent* is increased by the size of *SendAhead*. Only one of these packets will have the *ACKME* flag set. The relative position of this packet in the *SendAhead* set is given by *AckPoint*. *AckPoint* must thus be less than or equal to the size of the *SendAhead* set.

Packets which have been sent and for which an *ACK* has been requested but not yet received fall into two categories: the *NeedAck* set and the *Worried* set. They are distinguished by whether or not an retranmission timeout has occurred since they were sent. Packets in the *NeedAck* set have been sent and an *ACK* has been requested, but not enough time has passed to be worried about the fact that an *ACK* has not been received. Packets which have been sent for which an *ACK* has not yet been requested, if any, are called the *InTransit* set. The *InTransit* set will always be empty if *AckPoint* equals the *SendAhead* size.

The source then waits for an *ACK* packet from the sink. Our implementation uses the waiting time to prefetch more data from the disk. During ideal conditions the source will proceed only after having received the *ACK* it is waiting for. In practice, however, it may timeout and retransmit data packets if it is operating as an RPC2 server.

At this point, the source will revise the *Worried* set. Any packets that have been acknowledged will be taken off the *Worried* set. The transmit window is shifted by increasing the *SendLastContig* counter. It will be set to one less the smallest sequence number of a member in the *Worried* set.

All the packets that are in the *Worried* set are retransmitted followed by *SendAhead* new packets. No packets will have the *ACKME* flag set, except for the member of *SendAhead* set whose index is given by *AckPoint*. The packets in the new *SendAhead* set are then either added to the *NeedAck* set or to the *InTransit* set, depending upon the *AckPoint* value, as described above. Packets are placed in the *Worried* set only after a retransmission interval has expired. The procedure is repeated until the file has been completely transfered. At no point, however, will the protocol have more packets outstanding than what is given by the transmit window size. Whenever the sum of the number of packets in the various sets is greater than the transmit window size, only the first packet in the *Worried* set will be sent.

16.3.2 Sink side operation

The sink keeps two counters, *RecvLastContig* and *RecvMostRecent*, which are similar to their counterparts at the source side. *RecvLastContig* is the sequence number of the last data packet where it and all previous data packets have been received. It is used as the *GotEmAll* counter when sending an *ACK* packet. *RecvMostRecent* is the highest sequence number of a packet received so far.

Before the file transfer takes place, the source will inform the sink about the parameters *RetryInterval*, *RetryCount* and *DupThreshold*. If the sink is an RPC2 server it will grant the source permission to transmit data packets by sending a *START* packet. It will then start waiting for data packets. If the sink is an RPC2 server and no data packet has arrived after the time specified by *RetryInterval*, it will send an *ACK* (or *START*) packet, trying to cause the source to retransmit its data. If this fails *RetryCount* number of times, without any valid data packet being received, the sink will consider the connection unusable.

The sink keeps track of the number of duplicate data packets that have arrived since the last time an *ACK* was sent. If that number exceed *DupThreshold*, the sink will send an *ACK* in an attempt to inform the source about the situation.

16.3.3 Client and server invariants

The state of the counters at the source and sink can be summarized by the following invariant relations, where SendAckLimit and SendWorriedLimit are upper bounds of the NeedAck and Worried sets, respectively.

Invariants when transfer is in progress

1. SendLastContig <= SendWorriedLimit <= SendAckLimit <= SendMostRecent (SendMostRecent - SendLastContig) <= WindowSize (SendMostRecent - SendAckLimit) <= SendAhead
2. RecvLastContig <= RecvMostRecent (RecvMostRecent - RecvLastContig) <= WindowSize

Invariants when transfer is completed, aborted or not started

1. SendLastContig (at source) = SendMostRecent (at source)
2. RecvLastContig (at sink) = RecvMostRecent (at sink)
3. SendLastContig (at source) = RecvLastContig (at sink)

16.4 Adjusting the Retransmission Interval

SFTP uses the retranmission interval to determine when it should be worried about packets for which it has not received an *ACK*. Initially, the retranmission interval is set to *SFTP_RetryInterval* (2 seconds), but varies depending on RTT observations collected during file transfers. When a timeout occurs, the retranmission timer is backed off. The backed off timer is independent of the RTT state in the *sEntry*.

Like RPC2, SFTP collects RTT observations by using packet timestamps. The timestamps and storage of RTT state is the same as presented in chapter 15 (XXX). In SFTP, timestamping is *two-way*, namely, both source and sink collect RTT observations during a transfer. Both timestamp fields in the packet header are used: one for the current timestamp, and one for the timestamp being echoed back to the other side.

The source collects observations as follows: it timestamps outgoing *DATA* packets. The sink echos a timestamp back on the *ACK* packet. When the *ACK* arrives at the source, the source computes the RTT for that send-ahead set, and updates its RTO accordingly.

The sink collects observations as follows: it timestamps *START*, *ACK*, and *TRIGGER* packets. (A trigger packet is an *ACK* that is being used by the server because it has timed out on the client.) The source echos the timestamp back on the first *DATA* packet sent in response to such a packet. When that *DATA* packet arrives at the sink, the sink computes the RTT and updates its RTO. If the first packet gets lost, no update is performed. If it is delayed, the update is performed when it arrives.

All that is needed for state in the *sEntry* is a single word, *TimeEcho*, to hold the timestamp that will next be echoed on a packet. Each packet may carry up to two timestamps - one is the time at which the sender sent it, and the other is the echoed timestamp. (Only *ACK* packets and certain *DATA* packets actually use both fields.) The spare2 and spare3 fields of the packet header are used for these fields, called *TimeStamp* and *TimeEcho*. These fields were previously reserved for bitmask fields, but were not being used.

Packets are timestamped as they are sent out in the following routines:

- sftp_SendSendAhead, sftp_ResendWorried, sftp_SendFirstUnacked (data)

- sftp_SendAck
- send_SendStart

The packet TimeStamp field is stashed in sEntry->TimeEcho as appropriate when a packet with a timestamp is received. This is the timestamp that will be echoed back to the other side eventually. This occurs in:

- sftp_DataArrived, on the sink, if the packet advances the left edge of the window (Header.SeqNumber == sEntry->RecvLastContig+1).
- sftp_StartArrived, on the source, whether the transfer has started or not. Data will be sent in response to the *START* packet either way.
- sftp_AckArrived, on the source. If there is more data to send, the source will send it in response to this packet.

The value in sEntry->TimeEcho is then placed in the Header.TimeEcho field in the following routines:

- sftp_SendAck, on the sink.
- sftp_SendSendAhead, sftp_ResendWorried, or sftp_SendFirstUnacked, on the source. The timestamp is echoed on the *first* packet sent out by these collectively (the one corresponding to sEntry->SendLastContig+1). All other *DATA* packets carry a TimeEcho of 0. A special case occurs in the first set of *DATA* packets on a server-to-client transfer, from PutFile. In this case there is no timestamp to echo, because the source does not hear from the sink before sending data. In this case, sEntry->TimeEcho is set to 0 at the top of PutFile. A second special case also occurs in PutFile, when the server times out. Again, there is no timestamp to echo, because the data is not being sent in response to a packet from the sink.

RTT measurements are computed from Header.TimeEcho in the following routines:

- sftp_AckArrived, on the source, if the *ACK* is not a trigger. Triggers are sent when the server times out during a client-to-server transfer. They do not represent real observations because there was no transmission from the source that caused them. Triggers are marked so that the source can distinguish them from real *ACK*s.
- sftp_DataArrived, on the sink.

Any zero TimeStamp or TimeEcho is ignored, and the RTO remains unchanged. This is chiefly for compatibility with versions of SFTP that do not use packet timestamps. RTT state in the sEntry is initialized on the client in *SFTP_Bind2*, using the BindTime supplied by RPC2. On the server, it is initialized in *SFTP_ GetRequest*, using the same BindTime shipped to the server on the first request on the connection.

16.5 Performance

RPC2 and SFTP perform well over a wide range of network speeds. Figure **??** (XXX) compares the performance of SFTP and TCP over three different networks: Ethernet, a WaveLan wireless network, and a modem over a phone line. In almost all cases, SFTPs performance equals or exceeds that of TCP.

Chapter 17

Adding New Kinds of Side Effects

This chapter is not intended for the average user. Rather, it is meant for a system programmer who intends to add support for a new kind of side effect. To do this, the system programmer must do the following:

1. Define an appropriate side effect descriptor, add it to the header file se.h and to the discriminated union in the definition of SE_Descriptor.

   Define an appropriate Initializer structure and a corresponding component in the SE_Initializer structure in file se.h.

   Write a set of routines corresponding to each of the SE_XXX routines described in the following pages. This includes a SE_Activate() routine to enlarge the table in file se.c, and a SE_SetDefaults() routine to deal with SE_Initializer structures.

17.1 Usage Notes

1. You will modify two RPC2 files (se.h and se.c), and add one more file containing the code implementing your new side effect. Also modify the Makefile to compile and link in your new file.

   Client and server programs will cause the appropriate side effect routines to be linked in by calling the appropriate SE_Activate() for each side effect they are interested in. Note that these calls must precede @RPC(Init)().

   None of these procedures will be called for a connection, if the RPC2_Bind that created the connection specified NULL for the SideEffectType parameter.

   In each of the calls, ConnHandle is the handle identifying the connection on which the side effect is desired. It is not likely to be a small integer. Since you cannot access the internal data structures of the RPC2 runtime system, you cannot use this for much. It is passed to you primarily for identification.

   You can use RPC2_GetSEPointer() and RPC2_SetSEPointer() to associate per-connection side effect data structures.

   Use RPC2_GetPeerInfo)() to get the identity of a connections peer.

Three return codes:RPC2_SUCCESS and RPC2_SEFAIL1 and RPC2_SEFAIL2 are recognized for each of the calls. The successful return causes the RPC runtime system to resume normal execution from the point at which the side effect routine was invoked. The failure returns abort the call at that point and returns RPC2_SEFAIL1 or RPC_SEFAIL2 to the client or server code that invoked the RPC system call. RPC2_SEFAIL1 is an error, but not a fatal error. Future RPC calls on this connection will still work. RPC2_SEFAIL2 is a fatal error.

17.2 Entry Point Templates

17.2.1 SE_Init –

Call:

SE_Init( )

Parameters:

None.

Completion Codes:

CC1=‘RPC2_SUCCESS,CCD1=‘, CC2=‘RPC2_SEFAIL1, CCD2=‘, CC3=‘RPC2_SEFAIL2, CCD3=‘,

Description:

Called just prior to return from RPC2_Init

17.2.2 SE_Bind1 –

Call:

SE_Bind1(**in** RPC2_Handle ConnHandle, **in** RPC2_CountedBS *ClientIdent *)*

Parameters:

P1=‘ConnHandle, PD1=‘, P2=‘ClientIdent, PD2=‘,

Completion Codes:

CC1=‘RPC2_SUCCESS, CC2=‘RPC2_SEFAIL1, CC3=‘RPC2_SEFAIL2,

Description:

Called on RPC2_Bind on client side. The call is made just prior to sending the first connection-establishment packet to the server. The connection establishment is continued only if RPC2_SUCCESS) is returned.

17.2.3 SE_Bind2 –

Call:

SE_Bind2(**in** RPC2_Handle) ConnHandle, **in** struct timeval *BindTime *)*

Parameters:

P1=‘ConnHandle, PD1=‘, P2=‘BindTime, PD2=‘,

Completion Codes:

CC1=‘RPC2_SUCCESS, CC2=‘RPC2_SEFAIL1, CC3=‘RPC2_SEFAIL2,

Description:

Called on RPC2_Bind on client side. The call is made just after the connection is successfully established, before control is returned to the caller. BindTime is the amount of time the bind took because of the network. If SE_Bind2) returns RPC2_SEFAIL1) or RPC2_SEFAIL2, that code is returned as the result of the RPC2_Bind). Otherwise the usual code is returned.

17.2.4 SE_Unbind –

Call:

*SE_ Unbind(***in** RPC2_Handle ConnHandle *)*

Parameters:

P1=‘ConnHandle, PD1=‘,

Completion Codes:

CC1=‘RPC2_SUCCESS, CC2=‘RPC2_SEFAIL1, CC3=‘RPC2_SEFAIL2,

Description:

Called when RPC2_Unbind) is executed on the client or server side. You are expected to free any side effect storage you associated with this connection, and to do whatever cleanup is necessary. Note that the connection state is available to you and is not destroyed until you return RPC2_SUCCESS).

17.2.5 SE_NewConnection –

Call:

*SE_NewConnection(***in** RPC2_Handle ConnHandle, **in** RPC2_CountedBS *ClientIdent *)*

Parameters:

P1=‘ConnHandle, PD1=‘, P2=‘ClientIdent, PD2=‘,

Completion Codes:

CC1=‘RPC2_SUCCESS, CC2=‘RPC2_SEFAIL1, CC3=‘RPC2_SEFAIL2,

Description:

Called after a request has been completely filled, just prior to network ordering of header fields, encryption and transmission. You may use the Prefix information to determine the actual size of the buffer corresponding to *RequestPtr. If you add data, remember to update the BodyLength field of the header in *RequestPtr. You also probably wish to update the SideEffectFlags and SideEffectDataOffset fields of the header. SDesc points to the side effect descriptor passed in by the client.

If you need more space than available in the buffer passed to you, you may allocate a larger packet, copy the current contents and add additional data. Return a pointer to the packet you allocated in RequestPtr: this is the packet that will actually get sent over the wire. DO NOT free the buffer pointed to by RequestPtr initially. If you allocate a packet, it will be freed immediately after successful transmission.

17.2.6 SE_MakeRPC2 –

Call:

SE_MakeRPC2( )

Parameters:

ConnHandle

.

SDesc

.

Reply

.

Completion Codes:

CC1=‘RPC2_SUCCESS/. CC2=‘RPC2_SEFAIL1/. CC3=‘RPC2_SEFAIL2/.

Description:

Called just after Reply has been received, after decryption and host ordering of header fields. Examine the SideEffectFlags and SideEffectDataOffset fields to determine if there is piggy-backed side effect data for you in Reply. If you remove data, remember to update the BodyLength field of the header in Reply. SDesc points to the side effect descriptor. You will probably wish to fill in the status fields of this descriptor. If the MakeRPC call fails for some reason, this routine will be called with a Reply of NULL. This allows you to take suitable cleanup action.

17.2.7 SE_GetRequest –

Call:

*SE_ GetRequest(***in** RPC2_Handle ConnHandle, **inout** RPC2_PacketBuffer *Request *)*

Parameters:

P1=‘ConnHandle, PD1=‘, P2=‘Request, PD2=‘,

Completion Codes:

CC1=‘RPC2_SUCCESS, CC2=‘RPC2_SEFAIL1, CC3=‘RPC2_SEFAIL2,

Description:

Called just prior to successful return of Request to the server. You should look at Request, extract side effect data if any, modify the header fields appropriately.

17.2.8 SE_InitSideEffect –

Call:

*SE_InitSideEffect(***in** RPC2_Handle ConnHandle, **inout** SE_Descriptor *SDesc *)*

Parameters:

P1=‘ConnHandle, PD1=‘, P2=‘SDesc, PD2=‘,

Completion Codes:

CC1=‘RPC2_SUCCESS, CC2=‘RPC2_SEFAIL1, CC3=‘RPC2_SEFAIL2,

Description:

Called when the server does an RPC2_InitSideEffect) call. You will probably want to examine some fields of SDesc and fill in some status-related fields. Note that there is no requirement that you should actually initiate any side effect action. You may choose to piggy back the side effect with the reply later.

17.2.9 SE_CheckSideEffect –

Call:

SE_CheckSideEffect(**in** RPC2_Handle ConnHandle, **inout** SE_Descriptor *SDesc, **in** long Flags *)*

Parameters:

P1=‘ConnHandle, PD1=‘, P2=‘SDesc/. P3=‘Flags/.

Completion Codes:

CC1=‘RPC2_SUCCESS/. CC2=‘RPC2_SEFAIL1/. CC3=‘RPC2_SEFAIL2/.

Description:

Called when the server does an RPC2_CheckSideEffect) call. The Flags parameter will specify what status is desired. You may have to actually initiate the side effect, depending on the circumstances.

17.2.10 SE_SendResponse –

Call:

SE_SendResponse(**in** RPC2_Handle ConnHandle, **in** RPC2_PacketBuffer **ReplyPtr *)*

Parameters:

P1=‘ConnHandle, PD1=‘, P2=‘ReplyPtr, PD2=‘,

Completion Codes:

CC1=‘RPC2_SUCCESS, CC2=‘RPC2_SEFAIL1, CC3=‘RPC2_SEFAIL2,

Description:

Text=‘Called just before the reply packet is network-ordered, encrypted and transmitted. You may wish to add piggy-back data to the reply; modify the BodyLength field in that case. If you are not piggybacking data, make sure that the side effect is complete before returning from this call.

If you need more space than available in the buffer passed to you, you may allocate a larger packet, copy the current contents and add additional data. Return a pointer to the packet you allocated in ReplyPtr: this is the packet that will actually get sent over the wire. DO NOT free the buffer pointed to by ReplyPtr initially. If you allocate a packet, it will be freed immediately after successful transmission.,

17.2.11 SE_PrintSEDescriptor –

Call:

SE_PrintSEDescriptor(**in** SE_Descriptor *SDesc, **in** FILE *outFile *)*

Parameters:

> P1=‘SDesc, PD1=‘Guaranteed to refer to your type of side effect. P2=‘outFile, PD2=‘Already open and ready to receive bytes.

Completion Codes:

RPC2_SUCCESS

> .

Description:

Called when printing debugging information. You should print out SDesc, suitably formatted, on outFile.

17.2.12 SE_SetDefaults –

Call:

SE_SetDefaults(**out** XXX()Initializer *SInit *)*

Parameters:

> P1=‘SInit, PD1=‘An initializer for this side effect, XXX.,

Completion Codes:

RPC2_SUCCESS

.

Description:

Called to set SInit to appropriate default values.

17.2.13 SE_Activate –

Call:

SE_Activate(**in** XXX()Initializer *SInit *)*

Parameters:

SInit

Initialization values to be used for this side effect, XXX.

Completion Codes:

RPC2_SUCCESS

.

Description:

Called to activate this side effect type. The body of this procedure should allocate and fill in a routine vector in the side effect table in file se.c. It should also obtain its initialization parameters from SInit.

<

Chapter 18

Implementation of MultiRPC

Support for MultiRPC exists both at the language level and at the runtime level. The runtime level support includes the MultiRPC routines themselves along with the associated library routines which perform argument packing and unpacking. The language level support consists mainly of the argument descriptor information supplied by RP2Gen for each subsystem. The client may choose to interface directly with the runtime MultiRPC system without taking advantage of the RP2Gen simplifications, but the discussion in the following sections assumes the existence of the RP2Gen interface except where explicitly noted otherwise.

The procedure for making a MultiRPC call is very similar to that for making an RPC2 call. The subsystem is designed and the specification is written into a *<subsys>.rpc2* file (the specification format is described in Section @ref[RP2GenChapter]). RP2Gen is then invoked on the specification file, and it generates both the standard server and client side interfaces as well as the MultiRPC argument descriptor structures and definitions for each server operation. The relevant descriptor pointers are made available to the client through the associated *<subsys>.h* file.

Once the interface has been specified, the subsystem implementor is responsible for writing the server main loop and the procedures to perform the server operations. This implementation is completely independent of any considerations relating to MultiRPC; MultiRPC is completely transparent to the server side of a subsystem.

From the clients perspective, making a MultiRPC call is slightly different from the RPC2 case. Instead of the procedure-like client side interface supplied by the stub routines, the single library routine @MRPC(MakeMulti) is used to interface to @RPC2(MultiRPC). The use of the library routine represents a large space savings in the executable files, but requires some additional information from the client making the call (see Sections @ref[MakeMultiInfo] and @ref[MakeMulti]). The client is also responsible for supplying a handler routine for any server operation which is used in a MultiRPC call. This handler routine is called by RPC2 as each individual server response arrives; it is used both for providing individual server return codes to the client and for giving the client control over the continuation or termination of the MultiRPC call. The handler routine is discussed in greater detail in the following section, and its interface is described in Section **??** (XXX).

18.1 The Client Handler

The client handler routine is intended to give the client control and flexibility in handling the incoming server responses from the MultiRPC call. For each connection specified in a @RPC2(MultiRPC) call, the client handler is called either when a connection error is detected or when the server response for that connection arrives. This allows the client to examine the replies as they arrive, and provides the opportunity to perform incremental bookeeping and analysis of the responses. The handler also has the ability to abort the MultiRPC call at any time. A more detailed discussion of the handler specifications can be found in Section **??** (XXX).

Since a MultiRPC call could potentially last a long time, it is crucial to provide the client with some measure of control over the progress and termination of the call. With many server responses, there are many variables that the client might wish to monitor in order to evaluate the progress of the call. In particular, the server responses and return codes themselves have a significant effect on the clients perception of the progress of the call. To address these requirements, RPC2 periodically passes control to the client during execution of the MultiRPC call. A client supplied routine designed to be called as each server response arrives provides access to complete current information about the status of the call; it also gives the client the ability to perform any incremental processing he considers necessary or useful. The client then indicates his decision to either continue accepting server responses or to terminate the MultiRPC call via the handler return code.

The value of client control over the progress of the MultiRPC call can best be illustrated with some specific examples. One example is in the case of connection errors. If the client requires responses on all of the designated connections and one of them returns an error, then the final result of the MultiRPC call will be useless and the remainder of the processing time will have been wasted. With the client handler routine the client has the ability to notice the connection error. He then has the ability to abort the call, or even to use the handler routine as an opportunity to rebind to the failed site and make an RPC2 call on that connection.

Another example is in the implementation of a replicated server. A useful way to deal with operation quorums (specified as some subset *n) of the total number of replicated servers* is to send messages out to all or many of the available servers and abort the call as soon as the first n responses arrive. This has the advantage of supplying the fastest possible execution for the replicated call; furthermore, since the n members of the quorum need not be chosen explicitly, the call will rarely have to be repeated if one of the servers is busy or inoperational.

The handler receives full sets of arguments each time it is called, along with an index identifying the current connection. The types of the server arguments to the client handler are identical to the types in the original MakeMulti call: the argument list is in fact passed through RPC2 and returned to the handler. Any processing is permissible in the handler routine, although it should be noted that since @RPC2(MultiRPC) does not support enqueueing of server requests any call made on a connection already active in a MultiRPC call will generate a return code of @RPC2(BUSY). Also, for lengthy blocking computations the same cautions with respect to lightweight processes apply as for RPC2.

It should also be noted that the use of the abort facility of the client handler carries with it some risks. These are discussed in more detail in Section **??** (XXX).

18.2 Flow of Control in MultiRPC

The flow of control in MultiRPC is much the same as for RPC2 except for the iterative calling of the client handler. The client initiates the MultiRPC call by calling the library routine @MRPC(MakeMulti). MakeMulti packs the client arguments into a request buffer, and calls @RPC2(MultiRPC) with the request buffer, some argument packing information, and a pointer to @MRPC(UnpackMulti), the library unpacking routine.

@RPC2(MultiRPC) sets up the processing environment, initializes the request packet headers for all the designated servers, and performs any necessary side effect initialization. It then calls an internal routine to perform the transmission of the request packets. This transmission routine does not return until either the client supplied timeout expires or until it has received responses from all of the designated servers. Once the request packets have been transmitted, the routine settles into a loop waiting for server responses to arrive. As each response arrives, some preliminary processing is performed, and any remaining side effect processing is completed. Then RPC2 calls @MRPC(UnpackMulti) to unpack the response buffer into the clients original arguments. @MRPC(UnpackMulti) unpacks the buffer and calls the client handler routine with the current serverss information. The client then performs whatever processing he wishes, and returns with his instructions to continue or terminate the call. If he wishes to continue, the internal loop continues until all the server responses have been received. Otherwise, the loop terminates and the transmission routine cleans up any loose ends caused by the termination.

Control then returns to @RPC2(MultiRPC), which checks the return code and returns to @MRPC(MakeMulti). MakeMulti simply passes the supplied return code back to the client as it returns.

Since side effects are completely determined by the **@SE[Descriptor]** and the connection, extending the side effect mechanism to MultiRPC requires nothing more than supplying a unique **@SE[Descriptor]** for each connection.

18.3 MultiRPC Related Calls

18.3.1 @RPC2[MultiRPC]

@RPC2(MultiRPC) is the RPC2 runtime routine responsible for setting up the internal state properly for sending the request packets to the specified servers. It is called via the RPC2 library routine @MRPC(MakeMulti). One of the arguments to MultiRPC is the ArgInfo structure. This structure is never examined by RPC2, but is simply passed through UnpackMulti. If the RP2Gen interface is used, this argument is supplied by @MRPC(MakeMulti) and need not concern the client. If the RP2Gen interface is not used, this can point to any structure needed by the clients unpacking routine.

The UnpackMulti argument is also related to the RP2Gen interface. If the RP2Gen interface is used, this argument is automatically supplied by @MRPC(MakeMulti) and will point to the RPC2 library unpacking routine. If the RP2Gen interface is not used, the client is responsible for supplying a pointer to a routine matching the UnpackMulti specification (see Section @ref[UnpackMulti]).

18.3.2 @MRPC[MakeMulti]

@MRPC(MakeMulti) is the library routine which provides the parameter packing interface to @RPC2(MultiRPC). It takes the place of the individual client side stub routines generated by RP2Gen. In additon to the usual information supplied in an RPC2 call, it takes as arguments RP2Gen generated argument and operation descriptors, the number of servers to be called, and a pointer to a client supplied handler routine (see Section @ref[MakeMulti] for more detailed information). Using the argument descriptors, @MRPC(MakeMulti) packs the supplied server arguments into an RPC2 request buffer and creates a data structure containing call specific information and a pointer to the client handler routine. It then makes the MultiRPC call, and passes the final return code back to the client when the call terminates.

OUT and INOUT parameters must be supplied in the form of arrays of pointers to the appropriate argument types. The parameter interface specifications are discussed in sectin 6.3 (XXX). The size of the array is dependent on the number of servers designated by the client. For INOUT parameters it is only necessary to actually fill in a value for the first element of the array, although storage must be properly allocated for all of the elements.

18.3.3 @MRPC[UnpackMulti]

appendix

Chapter 19

Usage and Implementation Notes

19.1 Current Limitations

- Only one portal is allowed in @RPC2(Init). This should be a list eventually.
- SmartFTP is the only side-effect type currently supported.
- getsubsysbyname() is a fake routine. It knows about "Vice2-FileServer", and "Vice2-CallBack".
- At present the multicast routines work only on RTs running Mach. The non-multicast MultiRPC routines work on all architectures.

19.2 Where to Find Files at Carnegie Mellon

The LWP release consists of a set of include files (cargs.h, lwp.h, lock.h, timer.h, and preempt.h) and a library (liblwp.a).

The RPC2 release consists of a set of include files (rpc2.h, se.h), a set of libraries (librpc2.a and libse.a), and the RP2Gen binary (rp2gen).

The LWP and RPC2 releases are available for RTs, Sun-3s, Vaxen, Pmaxen running Andrew and Mach, and Next running Andrew. (Systypes rt_r3, rt_mach, sun3_35, sun3_mach, vax_3, vax_mach, pmax_3, pmax_mach, next_mach20)

On Coda, these files may be found in `/coda/project/coda/alpha/{include,lib,bin}`.

On Andrew, these files are located in `/afs/andrew/scs/cs/15-612{include,lib,bin}`.

iopen () is a system call created at Carnegie Mellon. Put an empty dummy function if your C library doesn't have it.

19.3 Debugging

The following external RPC2 variables may be set for debugging:

@RPC2(DebugLevel)

values of 0, 1, 10 and 100 are meaningful. Initial value is 0.

@RPC2(Perror)

set to 1 to see Unix error messages on stderr. Initial value is 1.

@RPC2(Trace)

set to 1 to enable tracing. 0 turns off tracing. Initial value is 0. Also see description of @RPC2(InitTrace) and @RPC2(DumpTrace) calls.

A global variable **lwp_debug** can be set to activate or deactivate debugging messages tracing the flow of control within the LWP routines. To activate debugging messages, set **lwp_debug** to a non-zero value. To deactivate, reset it to zero. All debugging output from the LWP routines is sent to stdout.

The LWP package checks for stack overflows at each context switch. The **lwp_overflowAction** variable controls the action of the package when an overflow occurs. If it is set to **LWP_SOMESSAGE**, then LWP will print a message on stderr telling of the overflow and will then be quiet. If it is set to **LWP_SOABORT**, LWP will call the abort () subroutine. Finally, if it is set to **LWP_SOQUIET**, LWP will ignore the errors. The **LWP_SOABORT** setting is the default.