CMU-ITC-85-038

RPC2 User Manual

M. Satyanarayanan

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

NOTE: Reference manual only; tutorial in preparation

Table of Contents

Preface

This document is a programmer's reference manual for RPC2, the ITC remote procedure call package. This package is being used at the present time for a variety of distributed applications such as file servers, authentication servers, and database servers.

Considerable effort has gone into making this mechanism flexible and robust. In particular, it works well even under conditions of heavy server load. However the package is simple enough to be used by relatively unsophisticated applications. Do not let the size of this user manual scare you! A tutorial introduction to this manual and procedures to simplify RPC initialization are in preparation.

Until the tutorial introduction is available the best way to learn RPC2 is as follows:

1. Study the example in Chapter 1. This is an actual piece of working code, and you should try running the example.

2. Read Chapter 4 next. This describes the procedural abstraction provided by RP2Gen, the stub generator for RPC2.

3. Read Chapter 2, which describes the RPC2 runtime system. 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. This material will make more sense in conjunction with the example of Chapter 1.

4. Read Chapter 3 to get an idea of how to add new kinds of side effects to RPC2. You will probably not need this material unless you intend to extend RPC2, but an overview of this material will probably be useful.

5. At all times keep available a copy of the LWP reference manual [1] and refer to it as needed.

Some key features of this package are:

- Clients and servers are each assumed to be using the ITC lightweight process package [1]. The RPC2 package 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.

- There is no a priori binding of RPC connections to LWPs within a client or server. RPC connections and threads of control are orthogonal concepts.

- 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.

- A server sends and receives requests via many different ***Portals*** and may service many different ***Subsystems***. A good analogue to a server supporting many subsystems is the ***Inet*** daemon in Unix 4.2, which is the rendezvous point for the FTP, Telnet, and Mail subsystems. Binding by clients is done to a host-portal-subsystem triple.

- Host, portal, subsystem, and side effect descriptor specifications are discriminated union types, to allow a multiplicity of representations. For example, hosts may be specified either by name or by Internet address. Files may be specified by a file name or a low-level identifier (or in future, perhaps even a file descriptor).

- 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 and occur asynchronously. Synchronization occurs due to an explicit RPC2_CheckSideEffect() call by the server.

- 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.

- RPC2 has builtin mechanisms to allow authentication of mutually suspicious clients and servers and to provide encrypted transmissions after connection establishment. Multiple levels of security are available and may be specified on an individual basis for each RPC connection. Multiple encryption types are also supported, to allow servers to deal with various types of clients.

- This is a completely revised implementation of an earlier RPC package [2], used in Vice-I. The earlier implementation is no longer supported.

1. Design Concepts

1.1. Introduction

<<<<<< to be written >>>>>>>

1.2. An Example

<<<<< intro to be written >>>>>>

1.2.1. Auth Subsystem .rpc file

M. Satyanarayanan Information Technology Center Carnegie-Mellon University

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();
```

1.2.2. Comp Subsystem .rpc file

M. Satyanarayanan Information Technology Center Carnegie-Mellon University

(c) IBM Corporation November 1985

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

CompExec(IN RPC2_String Command, IN OUT SE_Descriptor Sed);
                                     Executes a command and ships back the result in a file. Returns
                                     COMPSUCCESS or COMPFAILED

CompQuit();
```

1.3. Server for Auth and Comp Subsystems

exserver.c -- Trivial server to demonstrate basic RPC2 functionality Exports two subsystems: auth and comp, each with a dedicated LWP.

M. Satyanarayanan Information Technology Center Carnegie-Mellon University


```
static char IBMid[] = "(c) Copyright IBM Corporation November 1985";


#include <stdio.h>
#include <potpourri.h>
#include <strings.h>
#include <sys/signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <pwd.h>
#include <lwp.h>
#include <rpc2.h>
#include <se.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(TRUE)
  {
  cid = 0;
  if ((rc = RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL)) < RPC2_WLIMIT)
    HandleRPCError(rc, cid);
  if ((rc = auth-ExecuteRequest(cid, reqbuffer)) < 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(TRUE)
    {
    cid = 0;
    if ((rc = RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL)) < RPC2_WLIMIT)
      HandleRPCError(rc, cid);
    if ((rc = comp-ExecuteRequest(cid, reqbuffer)) < 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);
  }

S - CompExec(cid, cmd)
  RPC2_Handle cid;
  char *cmd;
                                  We should really have a formal of type SE_Descriptor at the end;
                                  but it is a dummy anyway
  {
  SE_Descriptor sed;
  char mycmd[100];
  sprintf(mycmd, "%s > /tmp/answer 2>&1", cmd);
  system(mycmd);                  beware; if this takes too long, client will get RPC2_DEAD!

  bzero(&sed, sizeof(sed));
  sed.Tag = DUMBFTP;
  sed.Value.DumbFTPD.Tag = FILEBYNAME;How I wish C had a "with" clause like Pascal
  sed.Value.DumbFTPD.TransmissionDirection = SERVERTOCLIENT;
  sed.Value.DumbFTPD.ByteQuota = -1;
  strcpy(sed.Value.DumbFTPD.FileInfo.ByName.LocalFileName, "/tmp/answer");
  if (RPC2_InitSideEffect(cid, &sed) != RPC2_SUCCESS) return(COMPFAILED);
  if (RPC2_CheckSideEffect(cid, &sed, SE_AWAITLOCALSTATUS) != RPC2_SUCCESS)
```

```
                return(COMPFAILED);
        return(COMPSUCCESS);
        }
```

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;
    DFTP_Initializer dftpi;
    RPC2_PortalIdent portalid, *portallist[1];
    RPC2_SubsysIdent subsysid;
    struct timeval tout;

    assert(LWP_InitializeProcessSupport(LWP_NORMAL - PRIORITY, &mylpid) = = LWP_SUCCESS);

    portalid.Tag = RPC2_PORTALBYINETNUMBER;
    portalid.Value.InetPortNumber = htons(AUTHPORTAL);
    portallist[0] = &portalid;
    tout.tv - sec = 240;
    tout.tv - usec = 0;
    DFTP_SetDefaults(&dftpi);
    DFTP_Activate(&dftpi);
    assert (RPC2_Init(RPC2_VERSION, 0, portallist, 1, -1, &tout) = = RPC2_SUCCESS);
    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;
    }
```

1.4. Client using Auth and Comp Subsystems

exclient.c -- Trivial client to demonstrate basic RPC2 functionality

M. Satyanarayanan Information Technology Center Carnegie-Mellon University


```
static char IBMid[] = "(c) Copyright IBM Corporation November 1985";


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

#define dgets(p) (LWP_DispatchProcess(), gets(p))
                                    allow RPC to get control periodically


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

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

  InitRPC();
  while (TRUE)
    {
    LWP_DispatchProcess();          otherwise we get RPC2_DEADs
    printf("Action? (1 = New Conn, 2 = Auth Request, 3 = Comp Request) ");
    dgets(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? ");
    dgets(hident.Value.Name);

    hident.Tag = RPC2_HOSTBYNAME;
    printf("Subsystem? (Auth = %d, Comp = %d) ", AUTHSUBSYSID, COMPSUBSYSID);
    dgets(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,
                   DUMBFTP, 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? ");
  dgets(buf);
  cid = atoi(buf);
  printf("Operation? (1 = Id, 2 = Name, 3 = Info, 4 = Quit) ");
  dgets(buf);
  op = atoi(buf);
  switch(op)
    {
    case 1:
      printf("Name? ");
      dgets(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? ");
      dgets(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? ");
        dgets(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;
  SE_Descriptor sed;
  char cmd[100], buf[100];

  printf("Connection id? ");
  dgets(buf);
  cid = atoi(buf);
  printf("Operation? (1 = Square, 2 = Cube, 3 = Age, 4 = Exec, 5 = Quit) ");
  dgets(buf);
  op = atoi(buf);
  switch(op)
    {
    case 1:
        printf("x? ");
        dgets(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? ");
        dgets(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:
            printf("Remote command: ");
            gets(cmd);
            bzero(&sed, sizeof(sed));
                                          How I wish C had a "with" clause like Pascal
            sed.Tag = DUMBFTP;
            sed.Value.DumbFTPD.Tag = FILEBYNAME;
            sed.Value.DumbFTPD.FileInfo.ByName.ProtectionBits = 0644;
            sed.Value.DumbFTPD.TransmissionDirection = SERVERTOCLIENT;
            sed.Value.DumbFTPD.ByteQuota = -1;
            strcpy(sed.Value.DumbFTPD.FileInfo.ByName.LocalFileName, "/tmp/result");

            rc = CompExec(cid, cmd, &sed);
            if (rc = = COMPSUCCESS) system("echo Result of remote exec:;cat /tmp/result");
            else
              if (rc = = COMPFAILED) printf("Could not do remote exec\n");
              else
                printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
            break;

          case 5:
            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;
        DFTP_Initializer dftpi;
        struct timeval tout;

        assert(LWP_InitializeProcessSupport(LWP_NORMAL - PRIORITY, &mylpid) = = LWP_SUCCESS);

        DFTP_SetDefaults(&dftpi);
        dftpi.ChunkSize = 1024;              2K and 4K give much better performance
        DFTP_Activate(&dftpi);
        tout.tv - sec = 240;
        tout.tv - usec = 0;
        assert (RPC2_Init(RPC2_VERSION, 0, NULL, 1, -1, &tout) = = RPC2_SUCCESS);
        }
```

iopen(){}

2. The RPC2 Runtime System

The purpose of this section is to describe the physical layout of data in transmissions between client and server RPC runtime systems. The runtime system deals 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.

a ***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.

a ***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.

The actual header files are the authoritative source of these definitions, and will be more up-to-date than this manual.

2.1. Constants, Types, and Globals (from file rpc2.h)

M. Satyanarayanan Information Technology Center Carnegie-Mellon University


```
#ifndef -RPC2-
#define -RPC2-
```

```
#define RPC2_VERSION "Version 7.0: Satya, 9 April 1986, 11:30"
```

This string is used in RPC initialization calls to ensure that the runtime system and the header files are mutually consistent. Also passed across on RPC2_Bind for advisory information to other side. Changes to this string may cause RPC2_OLDVERSION to be returned on RPC2_Bind()s. For really minor changes alter RPC2_LastEdit in globals.c.

```
#define RPC2_PROTOVERSION 6
```

Found as the first 4 bytes of EVERY packet. Change this if you change any aspect of the protocol sequence, or if you change the packet header, or the body formats of the initialization packets. Used in inital packet exchange to verify that the client and server speak exactly the same protocol. Orthogonal to RPC2_VERSION. We need this in the header at the very beginning, else we cannot change packet formats in a detectable manner.

The following constants are used to indicate the security-level of RPC connections.

```
#define RPC2_OPENKIMONO      98    Neither authenticated nor encrypted
#define RPC2_AUTHONLY        12    Authenticated but not encrypted
#define RPC2_HEADERSONLY     73    Authenticated but only headers encrypted
#define RPC2_SECURE          66    Authenticated and fully encrypted
```

RPC2 supports multiple encryption types; the key length is fixed, and you must always supply a field of RPC2_KEYSIZE bytes wherever an encryption key is called for. However, individual algorithms can choose to ignore excess bytes in the keys.

The encryption types are specified as integer bit positions so that the EncryptionTypesMask field of RPC2_GetRequest() can be a mask of these types. The required type must also be specified in RPC2_Bind().

To add support for other encryption types only the constants below and the internal runtime procedures RPC2_Encrypt() and RPC2_Decrypt() have to be modified.

```
#define RPC2_DES 1
#define RPC2_XOR 2
#define RPC2_ENCRYPTIONTYPES (RPC2_DES | RPC2_XOR)
                                        union of all supported types
#define RPC2_KEYSIZE 8                  Size in bytes of the encryption keys
```

RPC procedure return codes:

These may also occur in the RPC2_ReturnCode field of reply headers: Values of 0 and below in those fields are reserved for RPC stub use. Codes greater than 0 are assigned and managed by subsystems.

There are three levels of errors: Warning, Error, and Fatal Error. RPC2_SUCCESS > RPC2_WLIMIT > warning codes > RPC2_ELIMIT > error codes > RPC2_FLIMIT > fatal error codes

The semantics of these codes are:

RPC2_SUCCESS: Everything was perfect.

Warning: Advisory information.

Error: Something went wrong, but the connection (if any) is still usable.

Fatal: The connection (if any) has been marked unusable.

Note that the routine RPC2_ErrorMsg() will translate return codes into printable strings.

```
#define RPC2_SUCCESS 0

#define RPC2_WLIMIT -1
#define RPC2_ELIMIT -1000
#define RPC2_FLIMIT -2000
```

Warnings

```
#define RPC2_OLDVERSION     RPC2_WLIMIT-1
#define RPC2_INVALIDOPCODE  RPC2_WLIMIT-2
                            Never returned by RPC2 itself; Used by higher levels, such as
                            rp2gen
#define RPC2_BADDATA        RPC2_WLIMIT-3
                            Never used by RPC2 itself; used by rp2gen or higher levels to
                            indicate bogus data
```

Errors

```
#define RPC2_CONNBUSY       RPC2_ELIMIT-1
#define RPC2_SEFAIL1        RPC2_ELIMIT-2
#define RPC2_TOOLONG        RPC2_ELIMIT-3
```

Fatal Errors

```
#define RPC2_FAIL             RPC2_FLIMIT-1
#define RPC2_NOCONNECTION     RPC2_FLIMIT-2
#define RPC2_TIMEOUT          RPC2_FLIMIT-3
#define RPC2_NOBINDING        RPC2_FLIMIT-4
#define RPC2_DUPLICATESERVER  RPC2_FLIMIT-5
```

```
#define RPC2_NOTWORKER           RPC2_FLIMIT-6
#define RPC2_NOTCLIENT           RPC2_FLIMIT-7
#define RPC2_WRONGVERSION        RPC2_FLIMIT-8
#define RPC2_NOTAUTHENTICATED    RPC2_FLIMIT-9
#define RPC2_CLOSECONNECTION     RPC2_FLIMIT-10
#define RPC2_BADFILTER           RPC2_FLIMIT-11
#define RPC2_LWPNOTINIT          RPC2_FLIMIT-12
#define RPC2_BADSERVER           RPC2_FLIMIT-13
#define RPC2_SEFAIL2             RPC2_FLIMIT-14
#define RPC2_DEAD                RPC2_FLIMIT-15
#define RPC2_NAKED               RPC2_FLIMIT-16
```

Universal opcode values: opcode values equal to or less than 0 are reserved. Values greater than 0 are usable by mutual agreement between clients and servers.

#define RPC2_INIT1OPENKIMONO -2	*Begin a new connection with security level RPC2_OPENKIMONO*
#define RPC2_INIT1AUTHONLY -3	*Begin a new connection with security level RPC2_AUTHONLY*
#define RPC2_INIT1HEADERSONLY -4	*Begin a new connection with security level RPC2_HEADERSONLY*
#define RPC2_INIT1SECURE -5	*Begin a new connection with security level RPC2_SECURE*
#define RPC2_LASTACK -6	*Packet that acknowledges a reply*
#define RPC2_REPLY -8	*Reply packet*
#define RPC2_INIT2 -10	*Phase 2 of bind handshake*
#define RPC2_INIT3 -11	*Phase 3 of bind handshake*
#define RPC2_INIT4 -12	*Phase 4 of bind handshake*
#define RPC2_NEWCONNECTION -13	*opcode of fake request generated by RPC2_GetRequest() on new connection*
#define RPC2_BUSY -14	*keep alive packet*

System Limits

#define RPC2_MAXPACKETSIZE 10000	*size of the largest acceptable packet buffer in bytes (includes prefix and header)*

Global variables for debugging:

RPC2_DebugLevel controls the level of debugging output produced on stdout. A value of 0 turns off the output altogether; values of 1, 10, and 100 are currently meaningful. The default value of this variable is 0.

RPC2_Perror controls the printing of Unix error messages on stdout. A value of 1 turns on the printing, while 0 turns it off. The default value for this variable is 1.

RPC2_Trace controls the tracing of RPC calls, packet transmissions and packet reception. Set it to 1 for tracing. Set to zero for stopping tracing. The internal circular trace buffer can be printed out by calling RPC2_DumpTrace().

```
extern long RPC2_DebugLevel;
extern long RPC2_Perror;
extern long RPC2_Trace;
```

************************** Data Types known to RPGen ********************************

typedef long RPC2_Integer;	*32-bit, 2's complement representation. On other machines, an explicit conversion may be needed.*
typedef unsigned long RPC2_Unsigned;	*32-bits.*

```
typedef
  unsigned char RPC2_Byte;                  A single 8-bit byte.
```

```
typedef
  RPC2_Byte *RPC2_ByteSeq;
```

A contiguous sequence of bytes. In the C implementation this is a pointer. RPC2Gen knows how to allocate and transform the pointer values on transmission. Beware if you are not dealing via RPC2Gen. May be differently represented in other languages.

```
typedef
  RPC2_ByteSeq RPC2_String;                 no nulls except last byte
```

A null-terminated sequence of characters. Identical to the C language string definition.

```
typedef
  struct
    {
    RPC2_Integer SeqLen;                    length of SeqBody
    RPC2_ByteSeq SeqBody;                   no restrictions on contents
    }
  RPC2_CountedBS;
```

A means of transmitting binary data.

```
typedef
  struct
    {
    RPC2_Integer MaxSeqLen;                 max size of buffer represented by SeqBody
    RPC2_Integer SeqLen;                    number of interesting bytes in SeqBody
    RPC2_ByteSeq SeqBody;                   No restrictions on contents
    }
  RPC2_BoundedBS;
```

RPC2_BoundedBS is intended to allow you to remotely play the game that C programmers play all the time: allocate a large buffer, fill in some bytes, then call a procedure which takes this buffer as a parameter and replaces its contents by a possibly longer sequence of bytes. Example: strcat().

```
typedef
  RPC2_Byte RPC2_EncryptionKey[RPC2_KEYSIZE];
```

Keys used for encryption are fixed length byte sequences

************************ *Data Types used only in runtime calls* ******************************

```
typedef RPC2_Integer RPC2_Handle;           actually a pointer in the remote machine's addr space
                                            NOT a small integer!!!

typedef
  struct
    {
    enum HostType {RPC2_HOSTBYINETADDR = 17, RPC2_HOSTBYNAME = 39} Tag;
                                            dbx bogosity if anonymous enum
    union
      {
      unsigned long InetAddress;            NOTE: in network order, not host order
      char Name[20];                        this is a pretty arbitrary length
      }
      Value;
    }
  RPC2_HostIdent;
```

```
typedef
  struct
    {
    enum PortalType {RPC2_PORTALBYINETNUMBER = 53, RPC2_PORTALBYNAME = 64} Tag;
                                               dbx bogosity if anonymous enum
    union
      {
      unsigned short InetPortNumber;          NOTE: in network order, not host order
      char Name[20];                          this is a pretty arbitrary length
      }
      Value;
    }
  RPC2_PortalIdent;

typedef
  struct
    {
    enum SubsysType {RPC2_SUBSYSBYID = 71, RPC2_SUBSYSBYNAME = 84} Tag;
                                               dbx bogosity if anonymous enum
    union
      {
      long  SubsysId;
      char Name[20];                          this is a pretty arbitrary length
      }
      Value;
    }
  RPC2_SubsysIdent;


typedef
  struct                                      data structure filled by RPC2_GetPeerInfo() call
    {
    RPC2_HostIdent       RemoteHost;
    RPC2_PortalIdent RemotePortal;
    RPC2_SubsysIdent RemoteSubsys;
    RPC2_Handle   RemoteHandle;
    RPC2_Integer  SecurityLevel;
    RPC2_Integer  EncryptionType;
    RPC2_Integer  Uniquefier;
    RPC2_EncryptionKey   SessionKey;
    }
  RPC2_PeerInfo;
```

The RPC2_PacketBuffer definition below deals with both requests and replies. The runtime system provides efficient buffer storage management routines --- use them!

```
typedef
  struct RPC2_PacketBuffer
    {
    struct RPC2_PacketBufferPrefix
      {
```

NOTE: The Prefix is only used by the runtime system on the local machine. Neither clients nor servers ever deal with it. It is never transmitted.

```
      struct RPC2_PacketBuffer *Next;       pointer to next element in buffer chain
      struct RPC2_PacketBuffer *Prev;       pointer to prev element in buffer chain
      long  MagicNumber;                    to detect storage corruption
      long  LEState;                        to detect buffer chain addling
      struct RPC2_PacketBuffer *Qname;      name of queue this packet is on
      long  BufferSize;                     Set at malloc() time; size of entire packet, including prefix.
      long  LengthOfPacket;                 size of data actually transmitted: header + body
      }
```

```
        Prefix;

The transmitted packet begins here.
    struct RPC2_PacketHeader
        {
                                              The first four fields are never encrypted
        RPC2_Integer  ProtoVersion;           Set by runtime system
        RPC2_Integer  RemoteHandle;           Set by runtime system; -1 indicates unencrypted error packet
        RPC2_Integer  LocalHandle;            Set by runtime system
        RPC2_Integer  Flags;                  Used by runtime system only

                                              Everything below here can be encrypted
        RPC2_Unsigned  BodyLength;            of the portion after the header. Set by client.
        RPC2_Unsigned  SeqNumber;               unique identifier for this message on this connection; set by
                                              runtime system; odd on packets from client to server; even on
                                              packets from server to client
        RPC2_Integer  Opcode;                   Values greater than 0 are subsystem-specific: set by client.
                                              Values less than 0 reserved: set by runtime system. Type of
                                              packet determined by Opcode value: > 0 = => request packet.
                                              Values of RPC2_REPLY = => reply packet, RPC2_ACK = =>
                                              ack packet, and so on
        RPC2_Unsigned SEFlags;                Bits for use by side effect routines
        RPC2_Unsigned SEDataOffset;           Offset of piggy-backed side effect data, from the start of Body
        RPC2_Unsigned SubsysId;               Subsystem identifier. Filled by runtime system.
        RPC2_Integer  ReturnCode;             Set by server on replies; meaningless on request packets
        RPC2_Unsigned Lamport;                For distributed clock mechanism
        RPC2_Integer  Uniquefier;             Used only in Init1 packets; truly unique random number
        RPC2_Integer  Spare2;
        RPC2_Integer  Spare3;
        }
        Header;

    RPC2_Byte Body[1];                         Arbitrary length body. For requests: IN and INOUT parameters;
                                              For replies: OUT and INOUT parameters; Header.BodyLength
                                              gives the length of this field
    }
  RPC2_PacketBuffer;                          The second and third fields actually get sent over the wire
```

Meaning of Flags field in RPC2 packet header

```
    #define RPC2_RETRY  0x1                   set by runtime system
    #define RPC2_ENCRYPTED      0x2           set by runtime system
```

Leftmost byte of Flags field is reserved for use by side effect routines. This is in addition to the SEFlags field. Flags is not encrypted, but SEFLAGS is.

Format of filter used in RPC2_GetRequest

```
typedef
  struct
    {
    enum E1 {ANY = 12, ONECONN = 37, ONESUBSYS = 43} FromWhom;
    enum E2 {OLD = 27, NEW = 38, OLDORNEW = 69} OldOrNew;
    union
      {
      RPC2_Handle WhichConn;                  if FromWhom = = ONECONN
      long SubsysId;                          if FromWhom = = ONESUBSYS
      }
      ConnOrSubsys;
    }
```

```
RPC2_RequestFilter;                    Type of Filter parameter in RPC2_GetRequest()
```

The following data structure is the body of the packet synthesised by the runtime system on a new connection, and returned as the result of an RPC2_GetRequest().

```
typedef
  struct
    {
    RPC2_Integer SideEffectType;
    RPC2_Integer SecurityLevel;
    RPC2_Integer EncryptionType;
    RPC2_CountedBS ClientIdent;
    }
  RPC2_NewConnectionBody;
```

RPC2 runtime routines:

```
extern long RPC2_Init();
extern long RPC2_Export();
extern long RPC2_DeExport();
extern long RPC2_AllocBuffer();
extern long RPC2_FreeBuffer();
extern long RPC2_SendResponse();
extern long RPC2_GetRequest();
extern long RPC2_MakeRPC();
extern long RPC2_MultiRPC();
extern long RPC2_Bind ();
extern long RPC2_InitSideEffect();
extern long RPC2_CheckSideEffect();
extern long RPC2_Unbind();
extern long RPC2_GetPrivatePointer();
extern long RPC2_SetPrivatePointer();
extern long RPC2_GetSEPointer();
extern long RPC2_SetSEPointer();
extern long RPC2_GetPeerInfo();
extern char *RPC2_ErrorMsg();               NOT long !!!!
extern long RPC2_DumpTrace();
extern long RPC2_DumpState();
extern long RPC2_InitTraceBuffer();
extern long RPC2_LamportTime();
extern long RPC2_Enable();
#endif
```

2.2. Client-related Calls

RPC2_Bind

Create a new connection

Call:

long 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:

SecurityLevel
One of the constants RPC2_OPENKIMONO, RPC2_ONLYAUTHENTICATE, RPC2_HEADERSONLY or RPC2_SECURE

EncryptionType
The kind of encryption to be used on this connection. For example, RPC2_XOR, RPC2_DES, etc. Ignored if SecurityLevel is RPC2_OPENKIMONO. The bind will fail if the remote site does not support the requested type of encryption.

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.

SideEffectType
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 DUMBFTP or SMARTFTP. May be 0 if no side effects are ever to be attempted on this connection.

ClientIdent
Adequate 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

SharedSecret

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.

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 a 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 run time 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.

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.

RPC2_MakeRPC

Make a remote procedure call (with possible side-effect)

Call:

long RPC2_MakeRPC(**in** *RPC2_Handle ConnHandle,* **in** *RPC2_PacketBuffer *Request,*
in *SE_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_SEFAIL2
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.

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.

RPC2_MultiRPC

Make a collection of remote procedure calls

Call:

long 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.

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 5.4.1.

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 5.4. It will often be useful to supply connection specific information such as unique file names in the **SE_Descriptor**.

A further discussion of the MultiRPC facility can be found in chapter 5.

2.3. Server-related RPC Calls

RPC2_Export

Indicate willingness to accept calls for a subsystem

Call:

long 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.

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.

RPC2_DeExport

Stop accepting new connections for one or all subsystems.

Call:

long RPC2_DeExport(**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.

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.

RPC2_GetRequest

Wait for an RPC request or a new connection

Call:

long 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.

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 look like this:

*long GetKeys(***in** *ClientIdent,* **out** *IdentKey,* **out** *SessionKey)*

*RPC2_CoundedBS *ClientIdent;*

*RPC2_EncryptionKey *IdentKey;*

*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 look like this:

*long AuthFail(***in** *ClientIdent,* **in** *EncrType,* **in** *PeerHost,* **in** *PeerPortal)*

*RPC2_CoundedBS *ClientIdent;*

RPC2_Integer EncryType;

*RPC2_HostIdent *PeerHost;*

*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.

RPC2_Enable

Allow servicing of requests on a new connection

Call:

long 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.

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 at after a NewConnection routine.

RPC2_SendResponse

Respond to a request from my client

Call:

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

Parameters:

ConnHandle
Which 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_DEAD
The remote site is dead or unreachable.

RPC2_NAKED
The remote site sent an explict negative acknowlegment.

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.

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.

RPC2_InitSideEffect

Initiate side effect

Call:

long RPC2_InitSideEffect(**in** *RPC2_Handle ConnHandle,* **in** *SE_Descriptor *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

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.

RPC2_CheckSideEffect

Check progress of side effect

Call:

long RPC2_CheckSideEffect(**in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *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

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

2.4. Miscellaneous Routines

RPC2_Init

Perform runtime system initialization

Call:

long 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_InitializeProcessSupport() 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.

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 Appendix II 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 SO_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.

RPC2_Unbind

Terminate a connection by client or server

Call:

long 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

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.

RPC2_AllocBuffer

Allocate a packet buffer

Call:

long 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.

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().

RPC2_FreeBuffer

Free a packet buffer

Call:

long 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.

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

RPC2_GetPrivatePointer

Obtain private data mapping for a connection.

Call:

long 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.

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

RPC2_SetPrivatePointer

Set private data mapping for a connection.

Call:

long 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.

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

RPC2_GetSEPointer

Obtain per-connection side-effect information..

Call:

long 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.

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.

RPC2_SetSEPointer

Set per-connection side-effect connection.

Call:

long 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.

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.

RPC2_GetPeerInfo

Obtain miscellaneous connection information.

Call:

long 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.

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.

RPC2_LamportTime

Get Lamport time

Call:

long RPC2_LamportTime( )

Parameters:

None

Completion Codes:

None

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.

RPC2_DumpState

Dump internal RPC state.

Call:

long 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.

You should typically call this routine after calling RPC_DumpTrace.

RPC2_InitTraceBuffer

Set trace buffer size.

Call:

long 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.

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

RPC2_DumpTrace

Print a trace of recent RPC calls and packets received.

Call:

long 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.

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.

XXX_SetDefaults

Set an SE initializer to its default values

Call:

long XXX_SetDefaults(**in** *XXX_Initializer *Initializer)*

Parameters:

Initializer

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

Completion Codes:

RPC2_SUCCESS

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.

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.

XXX_Activate

Activates a side effect type and initializes it

Call:

long XXX_Activate(**in** *XXX_Initializer *Initializer)*

Parameters:

Initializer

Initializer for side effect XXX.

Completion Codes:

RPC2_SUCCESS

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

3. Side Effects

3.1. Constants and Globals (from file se.h)

M. Satyanarayanan Information Technology Center Carnegie-Mellon University


```
#ifndef -SE-
#define -SE-

struct SE_Definition
  {
  long SideEffectType;                    what kind of side effect am I?
  long (*SE_Init)();                      on both client & server side
  long (*SE_Bind1)();                     on client side
  long (*SE_Bind2)();                     on client side
  long (*SE_Unbind)();                    on client and server side
  long (*SE_NewConnection)();             on server side
  long (*SE_MakeRPC1)();                  on client side
  long (*SE_MakeRPC2)();                  on client side
  long (*SE_GetRequest)();                on server side
  long (*SE_InitSideEffect)();            on server side
  long (*SE_CheckSideEffect)();           on server side
  long (*SE_SendResponse)();              on server side
  long (*SE_PrintSEDescriptor)();         for debugging
  long (*SE_SetDefaults)();               for initialization
  };
```

Types of side effects: use this in the RPC2_Bind() call

```
#define DUMBFTP 231
#define SMARTFTP 1189


enum WhichWay {CLIENTTOSERVER = 93, SERVERTOCLIENT = 87};
enum FileInfoTag {FILEBYNAME = 33, FILEBYINODE = 58};

struct DFTP_Descriptor
  {
  enum WhichWay TransmissionDirection;
                                          IN
  char hashmark;                          IN: 0 for non-verbose transfer
  long SeekOffset;                        IN: >= 0; position to seek to before first read or write
  long BytesTransferred;                  OUT: value after RPC2_CheckSideEffect() meaningful
  long ByteQuota;                         IN: maximum number of data bytes to be sent or received.
                                          SE_FAIL1 is returned and the transfer aborted if this limit would
                                          be exceeded. EnforceQuota in DFTP_Initializer must be
                                          specified as 1 at RPC initialization for the quota enforcement to
                                          take place. A value of -1 implies a limit of infinity.
  enum FileInfoTag Tag;                   IN
  union
    {
    struct
```

```
        {
        long ProtectionBits;              Unix mode bits to be set for created files
        char LocalFileName[256];
        }
        ByName;                           if (Tag = = FILEBYNAME); standard Unix open()

      struct
        {
        long Device;                      device on which file resides
        long Inode;                       inode number of file (inode MUST exist already)
        }
        ByInode;                          if (Tag = = FILEBYINODE); ITC inode-open
      }
      FileInfo;                           everything is IN

    };


   #define SFTP_Descriptor DFTP_Descriptor

   enum SE_Status {SE_NOTSTARTED = 33, SE_INPROGRESS = 24, SE_SUCCESS = 57, SE_FAILURE = 36};

   typedef
     struct SE_SideEffectDescriptor
       {
       enum SE_Status LocalStatus;
       enum SE_Status RemoteStatus;
       long Tag;                          DUMBFTP or SMARTFTP or ASYNCFTP
       union
         {
         struct DFTP_Descriptor DumbFTPD;
         struct SFTP_Descriptor SmartFTPD;
         }
         Value;
       }
       SE_Descriptor;


   typedef struct DFTPI
     {
     long NoOfBulkLWPs;
     long ChunkSize;
     long SupportedEncryptionTypes;       Mask
     long EnforceQuota;
     } DFTP_Initializer;

   typedef struct SFTPI
     {
     long PacketSize;                     bytes in data packet
     long WindowSize;                     max number of outstanding unacknowledged packets
     long RetryCount;
     long RetryInterval;                  in milliseconds
     long SendAhead;                      number of packets to read and send ahead
     long AckPoint;                       when to send ack
     long EnforceQuota;                   0 = => don't
     } SFTP_Initializer;
```

Flag options in RPC2_CheckSEStatus(): OR these together as needed

```
#define SE_AWAITLOCALSTATUS  1
#define SE_AWAITREMOTESTATUS 2

extern struct SE_Definition *SE_DefSpecs;   array
extern long SE_DefCount;                    how many are there?
extern void SE_SetDefaults();
#endif
```

3.2. Adding New Kinds of Side Effects

The rest of this chapter is not intended for the average user. Only a system programmer who intends to add support for a new kind of side effect needs to understand the semantics of the calls described here. The normal user need only concern himself with the format of the side effect descriptor, described above.

3.2.1. 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.

2. 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().

3. None of these procedures will be called for a connection, if the RPC2_Bind that created the connection specified NULL for the SideEffectType parameter.

4. 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.

5. You can use RPC2_GetSEPointer() and RPC2_SetSEPointer() to associate per-connection side effect data structures.

6. Use RPC2_GetPeerInfo() to get the identity of a connection's peer.

7. 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.

8. To add a new type of side effect do the following:

a. 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.

b. Define an appropriate Initializer structure and a corresponding component in the SE_Initializer structure in file se.h.

c. 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.

SE_Init

Call:

long SE_Init( )

Parameters:

None

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

Called just prior to return from RPC2_Init.

SE_Bind1

Call:

long SE_Bind1(**in** *RPC2_Handle ConnHandle,* **in** *RPC2_CountedBS *ClientIdent)*

Parameters:

ConnHandle

ClientIdent

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_Bind2

Call:

long SE_Bind2(**in** *RPC2_Handle ConnHandle)*

Parameters:

ConnHandle

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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. 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.

SE_Unbind

Call:

long SE_Unbind(**in** *RPC2_Handle ConnHandle)*

Parameters:

ConnHandle

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_NewConnection

Call:

long SE_NewConnection(**in** *RPC2_Handle ConnHandle,* **in** *RPC2_CountedBS *ClientIdent)*

Parameters:

ConnHandle

ClientIdent

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

Called on server side when a new connection is established, just prior to exit from the corresponding RPC2_GetRequest().

SE_MakeRPC1

Call:

long SE_MakeRPC1(**in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc,*
inout *RPC2_PacketBuffer **RequestPtr)*

Parameters:

ConnHandle

SDesc

RequestPtr

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_MakeRPC2

Call:

long SE_MakeRPC2(**in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc,*
inout *RPC2_PacketBuffer *Reply)*

Parameters:

ConnHandle

SDesc

Reply

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_GetRequest

Call:

long SE_GetRequest(**in** *RPC2_Handle ConnHandle,* **inout** *RPC2_PacketBuffer *Request)*

Parameters:

ConnHandle

Request

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_InitSideEffect

Call:

long SE_InitSideEffect(**in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc)*

Parameters:

ConnHandle

SDesc

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_CheckSideEffect

Call:

long SE_CheckSideEffect(**in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc,*
in *long Flags)*

Parameters:

ConnHandle

SDesc

Flags

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_SendResponse

Call:

long SE_SendResponse(**in** *RPC2_Handle ConnHandle,* **in** *RPC2_PacketBuffer **ReplyPtr)*

Parameters:

ConnHandle

ReplyPtr

Completion Codes:

RPC2_SUCCESS

RPC2_SEFAIL1

RPC2_SEFAIL2

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.

SE_PrintSEDescriptor

Call:

long SE_PrintSEDescriptor(**in** *SE_Descriptor *SDesc,* **in** *FILE *outFile)*

Parameters:

SDesc

Guaranteed to refer to your type of side effect.

outFile

Already open and ready to receive bytes.

Completion Codes:

RPC2_SUCCESS

Called when printing debugging information. You should print out SDesc, suitably formatted, on outFile.

SE_SetDefaults

Call:

*long SE_SetDefaults(XXX_Initializer *SInit)*

Parameters:

SInit An initializer for this side effect, XXX.

Completion Codes:

RPC2_SUCCESS

Called to set SInit to appropriate default values.

SE_Activate

Call:

long SE_Activate(**in** *XXX_Initializer *SInit)*

Parameters:

SInit Initialization values to be used for this side effect, XXX.

Completion Codes:

RPC2_SUCCESS

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.

4. RP2Gen: A Stub Generator for RPC2

NOTE

This chapter is derived from the original documents by Jon Rosenberg, David Nichols and M. Satyanarayanan. RP2Gen was written by Jon Rosenberg.

")

4.1. 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.

4.2. Usage

RP2GEN is invoked as follows:

`rp2gen [`*server language*`] [`*client language*`]` *file*

Where *server language* is the language to be used for the server interface and *client language* is the language for the client interface. The possibilities for these fields are

-c	C
-f	FORTRAN 77
-p	PASCAL

If only one language option is specified, the same language is used for both the server and the client. The default options are -c -c. Note that a particular language option may not support all of the data types.

File is the file containing the description of the interface. Normally, these files have the extension *.rpc2*. RPGen creates three files named *base*.client.*ext*, *base*.server.*ext* and *base*.h, where *base* is the name of the file without the extension and the pathname prefix, and *ext* is the appropriate language-specific extension. The options indicate the target language for the generated output. The default is -c. 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 -lrpc2 ...
```

and the server program with

```
ld ... base.server.o -lrpc2 ...
```

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.

4.3. 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.**

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_CountedBS | RPC2_BoundedBS
              | SE_Descriptor RPC2_EncryptionKey | identifier
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 ::=
```

In addition to the syntax above, text inclosed 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 - sign. 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.

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 client's 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.

An Example

```
Subsystem "fs2";

typedef RPC2_Unsigned VolumeId;
typedef RPC2_Unsigned VnodeId;
typedef RPC2_Unsigned Unique;

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

ViceConnectFS(IN RPC2_String UserName,
              IN RPC2_String WorkStationName,
              IN RPC2_String VenusName);

ViceRemoveCallBack (IN ViceFid Fid);
```

4.4. 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.

RPC2 Type	C Declaration		
	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*

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.

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.

The ExecuteRequest Routine

RP2GEN generates another routine that serves to interpret and execute an RPC2 request. The name of this routine is "*subsystem_name*_ExecuteRequest", and its header is

```
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.

Programming rules for the server and client

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.

4.5. 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 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 two's 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. A 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*.

5. MultiRPC

5.1. Design Issues

The MultiRPC facility 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 4). 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 (see appropriate chapter). Side effects for MultiRPC work exactly as in the RPC2 case except that the client must supply a separate **SE_Descriptor** for each connection.

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 5.3.1.

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.

5.2. An Example

The following example is the same as the one in section 1.2, but here it has been converted to use MultiRPC. Comparison of the two examples will illustrate the differences in the client code necessary to use the MultiRPC facility. Only the code in the file *exclient.c* has been changed; *exserver.c* and both of the *.rpc2* files were unaffected by the modifications.

This example illustrates the MultiRPC interface to a simple system. The system exports two subsystems, an authentication server and a computation server. The authentication operations include looking up either a user name or a user id given the complementary information, or looking up some user statistics given the user id. The computation server operations include squaring a number, cubing a number, requesting the age of a given connection, and causing the remote host to *exec* a specified command and return the results as a side effect in a file.

A user can create a new connection or make a request to either the authentication or computation subsystem. The new connection choice results in an RPC2_Bind to the subsystem specified; subsystem requests cannot be made until a new connection has been created. The bind returns a connection id which can be used to identify the connection when making server requests.

Once a connection has been established to a subsystem, a subsystem request can be made. The client will prompt for the number of servers to which the request is to be made, and for their connection ids. In each case except the Bind, the call is made using MultiRPC using the MRPC_MakeMulti library routine interface. Note that RPC2_MultiRPC is used even when only one server is requested.

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. The handler corresponds to the HandleResult routine described in sections 5.4.1 and 5.3.3.4.

5.2.1. Auth Subsystem .rpc file

M. Satyanarayanan Information Technology Center Carnegie-Mellon University

(c) IBM Corporation November 1985

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 cldent) 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();
```

5.2.2. Comp Subsystem .rpc file

M. Satyanarayanan Information Technology Center Carnegie-Mellon University

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

CompExec(IN RPC2_String Command, IN OUT SE_Descriptor Sed);
                                        Executes a command and ships back the result in a file. Returns
                                        COMPSUCCESS or COMPFAILED

CompQuit();
```

5.2.3. Server for Auth and Comp Subsystems

exserver.c -- Trivial server to demonstrate basic RPC2 functionality Exports two subsystems: auth and comp, each with a dedicated LWP.

M. Satyanarayanan Information Technology Center Carnegie-Mellon University


```
static char IBMid[] = "(c) Copyright IBM Corporation November 1985";
```

```
#include <stdio.h>
#include <potpourri.h>
#include <strings.h>
#include <sys/signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <pwd.h>
#include <lwp.h>
#include <rpc2.h>
#include <se.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(TRUE)
    {
    cid = 0;
    if ((rc = RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL)) < RPC2_WLIMIT)
      HandleRPCError(rc, cid);
    if ((rc = auth - ExecuteRequest(cid, reqbuffer)) < 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(TRUE)
    {
    cid = 0;
    if ((rc = RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL)) < RPC2_WLIMIT)
      HandleRPCError(rc, cid);
    if ((rc = comp-ExecuteRequest(cid, reqbuffer)) < 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);
}

S - CompExec(cid, cmd)
RPC2_Handle cid;
char *cmd;
                                        We should really have a formal of type SE_Descriptor at the end;
                                        but it is a dummy anyway
{
SE_Descriptor sed;
char mycmd[100];
sprintf(mycmd, "%s > /tmp/answer 2>&1", cmd);
system(mycmd);                          beware; if this takes too long, client will get RPC2_DEAD!

bzero(&sed, sizeof(sed));
sed.Tag = DUMBFTP;
sed.Value.DumbFTPD.Tag = FILEBYNAME;How I wish C had a "with" clause like Pascal
sed.Value.DumbFTPD.TransmissionDirection = SERVERTOCLIENT;
sed.Value.DumbFTPD.ByteQuota = -1;
strcpy(sed.Value.DumbFTPD.FileInfo.ByName.LocalFileName, "/tmp/answer");
if (RPC2_InitSideEffect(cid, &sed) != RPC2_SUCCESS) return(COMPFAILED);
if (RPC2_CheckSideEffect(cid, &sed, SE_AWAITLOCALSTATUS) != RPC2_SUCCESS)
            return(COMPFAILED);
return(COMPSUCCESS);
}

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;
```

```
DFTP_Initializer dftpi;
RPC2_PortalIdent portalid, *portallist[1];
RPC2_SubsysIdent subsysid;
struct timeval tout;

assert(LWP_InitializeProcessSupport(LWP_NORMAL - PRIORITY, &mylpid) = = LWP_SUCCESS);

portalid.Tag = RPC2_PORTALBYINETNUMBER;
portalid.Value.InetPortNumber = htons(AUTHPORTAL);
portallist[0] = &portalid;
tout.tv - sec = 240;
tout.tv - usec = 0;
DFTP_SetDefaults(&dftpi);
DFTP_Activate(&dftpi);
assert (RPC2_Init(RPC2_VERSION, 0, portallist, 1, -1, &tout) = = RPC2_SUCCESS);
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;
  }
```

5.2.4. Client using Auth and Comp Subsystems

exclient.c -- Trivial client to demonstrate RPC2 - MultiRPC() functionality

M. Satyanarayanan and E. Siegel Information Technology Center Carnegie-Mellon University

(c) Copyright IBM Corporation November 1985

```
static char IBMid[] = "(c) Copyright IBM Corporation November 1985";


#include <stdio.h>
#include <potpourri.h>
#include <strings.h>
#include <sys/time.h>
```

```
#include <sys/types.h>
#include <netinet/in.h>
#include <pwd.h>
#include <lwp.h>
#include <rpc2.h>
#include <se.h>
#include <preempt.h>
#include "auth.h"
#include "comp.h"

long Handle - AuthUserId(), Handle - AuthUserName();
long Handle - AuthUserInfo(), Handle - AuthQuit();
long Handle - CompSquare(), Handle - CompCube();
long Handle - CompAge(), Handle - CompExec(), Handle - CompQuit();
int returns;

#define MAXCONNS 10
#define dgets(p) {if (gets(p) = = NULL) {perror("stdin");abort();}}
                                                  allow RPC to get control periodically


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

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

  InitRPC();
  while (TRUE)
    {
    LWP_DispatchProcess();              otherwise we get RPC2_DEADs
    printf("Action? (1 = New Conn, 2 = Auth Request, 3 = Comp Request) ");
    dgets(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? ");
  dgets(hident.Value.Name);
```

```
hident.Tag = RPC2_HOSTBYNAME;
printf("Subsystem? (Auth = %d, Comp = %d) ", AUTHSUBSYSID, COMPSUBSYSID);
dgets(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,
                SMARTFTP, 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[MAXCONNS];
 int op, rc, uid[MAXCONNS], howmany, i;
 char name[100], buf[100];
 AuthInfo ainfo[MAXCONNS];
 RPC2_BoundedBS bbs[MAXCONNS];

 while (1) {
  printf("How many servers? ");
  dgets(buf);
  howmany = atoi(buf);
  if (howmany <= 10 && howmany > 0) break;
 }
 for (i = 0; i < howmany; i + +) {
  printf("Connection id? ");
  dgets(buf);
  cid[i] = atoi(buf);
 }
 printf("Operation? (1 = Id, 2 = Name, 3 = Info, 4 = Quit) ");
 dgets(buf);
 op = atoi(buf);
 returns = 0;                                Zero return counter
 switch(op)
   {
   case 1:
     printf("Name? ");
     dgets(name);
     rc = MakeMulti(AuthUserId - OP, AuthUserId - PTR, howmany, cid, Handle - AuthUserId,
                                          NULL, name, uid);
     if (rc != RPC2_SUCCESS) printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
     break;

   case 2:
     printf("Id? ");
     dgets(buf);
     uid[0] = atoi(buf);
     bbs[0].MaxSeqLen = sizeof(name);
     bbs[0].SeqLen = 0;
     bbs[0].SeqBody = (RPC2_ByteSeq) name;
     for(i = 1; i < howmany; i + +) {
```

```
            bbs[i].MaxSeqLen = sizeof(name);
            bbs[i].SeqLen = 0;
            bbs[i].SeqBody = (RPC2_ByteSeq) malloc(sizeof(name));
        }
        rc = MakeMulti(AuthUserName - OP, AuthUserName - PTR, howmany, cid,
                            Handle - AuthUserName, NULL, uid[0], bbs);
        if (rc != RPC2_SUCCESS) printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
        for(i = 1; i < howmany; i++) {
            free(bbs[i].SeqBody);
        }
        break;

      case 3:
        printf("Id? ");
        dgets(buf);
        uid[0] = atoi(buf);
        rc = MakeMulti(AuthUserInfo - OP, AuthUserInfo - PTR, howmany, cid,
                            Handle - AuthUserInfo, NULL, uid[0], ainfo);
        if (rc != RPC2_SUCCESS) printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
        break;

      case 4:
        rc = MakeMulti(AuthQuit - OP, AuthQuit - PTR, howmany, cid, Handle - AuthQuit, NULL);
        if (rc != RPC2_SUCCESS) printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
        break;

      }

    }


long Handle - AuthUserId(HowMany, cid, thishost, rpcval, name, uid)
int HowMany, thishost, rpcval, uid[];
RPC2_Handle cid[];
char name[];
{
        printf("received reply from connection %d:\n", cid[thishost]);
        if (rpcval == AUTHSUCCESS) printf("Id = %d\n", uid[thishost]);
        else
            if (rpcval == AUTHFAILED) printf("Bogus user name\n");
        if (++returns > HowMany) return 1;  /* wait for all returns */
        else return 0;
}

long Handle - AuthUserName(HowMany, cid, thishost, rpcval, uid, bbs)
int HowMany, thishost, rpcval, uid;
RPC2_BoundedBS bbs[];
RPC2_Handle cid[];
{
        printf("received reply from connection %d:\n", cid[thishost]);
        if (rpcval == AUTHSUCCESS) printf("Name = %s\n", bbs[thishost].SeqBody);
        else
            if (rpcval == AUTHFAILED) printf("Bogus user id\n");
            else printf("Call failed --> %s\n", RPC2_ErrorMsg(rpcval));
        if (++returns > HowMany) return 1;  /* wait for all returns */
        return 0;
}


long Handle - AuthUserInfo(HowMany, cid, thishost, rc, uid, ainfo)
```

```
int HowMany, thishost, rc, uid;
AuthInfo ainfo[];
RPC2_Handle cid[];
{
        printf("received reply from connection %d:\n", cid[thishost]);
        if (rc = = AUTHSUCCESS) printf("Group = %d  Home = %s\n",
                      ainfo[thishost].GroupId, ainfo[thishost].HomeDir);
        else
           if (rc = = AUTHFAILED) printf("Bogus user id\n");
           else printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
        if (+ +returns > HowMany) return 1;  /* wait for all returns */
        return 0;
}

long Handle - AuthQuit(HowMany, cid, thishost, rc)
int HowMany, thishost, rc;
RPC2_Handle cid[];
{
        printf("received reply from connection %d:\n", cid[thishost]);
        if (rc != AUTHSUCCESS)
           printf("Call failed for connection %d --> %s\n", cid[thishost], RPC2_ErrorMsg(rc));
        RPC2_Unbind(cid[thishost]);
        if (+ +returns > HowMany) return 1;  /* wait for all returns */
        return 0;
}

Comp()
  {
  RPC2_Handle cid[MAXCONNS];
  int op, rc, x, howmany, i;
  SE - Descriptor sed[MAXCONNS];
  char cmd[100], buf[100], fname[30];

  while (1) {
   printf("How many servers? ");
   dgets(buf);
   howmany = atoi(buf);
   if (howmany <= 10 && howmany > 0) break;
  }
  for (i = 0; i < howmany; i+ +) {
   printf("Connection id? ");
   dgets(buf);
   cid[i] = atoi(buf);
  }
  printf("Operation? (1 = Square, 2 = Cube, 3 = Age, 4 = Exec, 5 = Quit) ");
  dgets(buf);
  op = atoi(buf);
  returns = 0;                              Zero return counter
  switch(op)
    {
    case 1:
       printf("x? ");
       dgets(buf);
       x = atoi(buf);
       rc = MakeMulti(CompSquare - OP, CompSquare - PTR, howmany, cid,
                                 Handle - CompSquare, NULL, x);
       if (rc != RPC2_SUCCESS) printf("MakeMulti call failed --> %s\n", RPC2_ErrorMsg(rc));
       break;

    case 2:
```

```
        printf("x? ");
        dgets(buf);
        x = atoi(buf);
        rc = MakeMulti(CompCube - OP, CompCube - PTR, howmany, cid, Handle - CompCube, NULL, x);
        if (rc != RPC2_SUCCESS) printf("MakeMulti call failed --> %s\n", RPC2_ErrorMsg(rc));
        break;

      case 3:
        rc = MakeMulti(CompAge - OP, CompAge - PTR, howmany, cid, Handle - CompAge, NULL);
        if (rc != RPC2_SUCCESS) printf("MakeMulti call failed --> %s\n", RPC2_ErrorMsg(rc));
        break;

      case 4:
        printf("Remote command: ");
        gets(cmd);
        for (i = 0; i < howmany; i++) {
        bzero(&(sed[i]), sizeof(sed));
                                          How I wish C had a "with" clause like Pascal
        sed[i].Tag = SMARTFTP;
        sed[i].Value.DumbFTPD.Tag = FILEBYNAME;
        sed[i].Value.DumbFTPD.FileInfo.ByName.ProtectionBits = 0644;
        sed[i].Value.DumbFTPD.TransmissionDirection = SERVERTOCLIENT;
        sed[i].Value.DumbFTPD.ByteQuota = -1;
        sprintf(fname, "/tmp/result - %d", cid[i]); tag filename with connection id
        strcpy(sed[i].Value.DumbFTPD.FileInfo.ByName.LocalFileName, fname);
        }
        rc = MakeMulti(CompExec - OP, CompExec - PTR, howmany, cid,
                          Handle - CompExec, NULL, cmd, sed);
        if (rc != RPC2_SUCCESS) printf("MakeMulti call failed --> %s\n", RPC2_ErrorMsg(rc));
        break;

      case 5:
        rc = MakeMulti(CompQuit - OP, CompQuit - PTR, howmany, cid, Handle - CompQuit, NULL);
    }
  }

long Handle - CompSquare(HowMany, cid, thishost, rc, x)
int HowMany, thishost, rc, x;
RPC2_Handle cid[];
{
        printf("received reply from connection %d:\n", cid[thishost]);
        if (rc != 0) printf("x**2 = %d\n", rc);
        else
           printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
        if (++returns > HowMany) return 1;  /* wait for all returns */
        return 0;
}


long Handle - CompCube(HowMany, cid, thishost, rc, x)
int HowMany, thishost, rc, x;
RPC2_Handle cid[];
{
        printf("received reply from connection %d:\n", cid[thishost]);
        if (rc > 0) printf("x**3 = %d\n", rc);
        else
           printf("Call failed --> %s\n", CompCube\n");
        if (++returns > HowMany) return 1;  /* wait for all returns */
        return 0;
}
```

```
long Handle - CompAge(HowMany, cid, thishost, rc)
int HowMany, thishost, rc;
RPC2_Handle cid[];
{
        printf("received reply from connection %d:\n", cid[thishost]);
        if (rc > 0) printf("Age of connection = %d seconds\n", rc);
        else
           printf("Call failed --> %s\n", CompAge\n");
        if (+ + returns > HowMany) return 1; /* wait for all returns */
        return 0;
}

long Handle - CompExec(HowMany, cid, thishost, rc, cmd, sed)
int HowMany, thishost, rc;
RPC2_Handle cid[];
char cmd[];
SE - Descriptor sed[];
{
 char ucmd[100];

        printf("received reply from connection %d:\n", cid[thishost]);
        sprintf(ucmd, "echo Result of remote exec:;cat /tmp/result - %d", cid[thishost]);
        if (rc = = COMPSUCCESS) system(ucmd);
        else
           if (rc = = COMPFAILED) printf("Could not do remote exec\n");
           else
              printf("Call failed --> %s\n", CompExec\n");
        if (+ + returns > HowMany) return 1; /* wait for all returns */
        return 0;
}

long Handle - CompQuit(HowMany, cid, thishost, rc)
int HowMany, thishost, rc;
RPC2_Handle cid[];
{
        if (rc < 0)
           printf("Call failed --> %s\n", RPC2_ErrorMsg(rc));
        RPC2_Unbind(cid);
        if (+ + returns > HowMany) return 1; /* wait for all returns */
        return 0;
}


= = = = = = = = = = = = = = = = = = = = = RPC Initialization and Error handling = = = = = = = = = = = = = = = = = = = =
InitRPC()
  {
  int mylpid = -1;
  struct timeval t;

  DFTP - Initializer dftpi;
  SFTP - Initializer sftpi;
  struct timeval tout;

  assert(LWP_InitializeProcessSupport(0, &mylpid) = = LWP_SUCCESS);
  t.tv - sec = 1;
  t.tv - usec = 0;
  assert(PRE - InitPreempt(&t) = = LWP_SUCCESS);
  PRE - PreemptMe();
```

```
DFTP - SetDefaults(&dftpi);
dftpi.ChunkSize = 1024;                    2K and 4K give much better performance
DFTP - Activate(&dftpi);
SFTP - SetDefaults(&sftpi);
SFTP - Activate(&sftpi);
tout.tv - sec = 30;
tout.tv - usec = 0;
assert (RPC2_Init(RPC2_VERSION, 0, NULL, 1, -1, &tout) = = RPC2_SUCCESS);
}
```

```
iopen(){}
```

5.3. Usage

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 4). 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 client's 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 5.3.3.2 and 5.4.1). 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 5.4.1.

5.3.1. 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 5.4.1.

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 client's 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 5.3.4.

5.3.2. 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 client's original arguments. MRPC_UnpackMulti unpacks the buffer and calls the client handler routine with the current servers's 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.

5.3.3. MultiRPC Related Calls

5.3.3.1. 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 client's 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 5.4.1).

5.3.3.2. 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 5.4.1 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 IN - OUT parameters must be supplied in the form of arrays of pointers to the appropriate argument types. The parameter interface specifications are discussed in sectin 5.4. The size of the array is dependent on the number of servers designated by the client. For IN - OUT 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.

5.3.3.3. MRPC_UnpackMulti

MRPC_UnpackMulti is a RPC2 library routine which functions as the other half of MRPC_MakeMulti. It unpacks the contents of the response buffer into their appropriate places in the client's arguments, and calls the client handler routine. It returns with the return code supplied by the client handler routine. If the RP2Gen interface is not used, the client must supply a pointer to a routine with the specified interface (see section 5.4.1) to RPC2_MultiRPC.

5.3.3.4. HandleResult

HandleResult is a place holder used to refer to the client-supplied handler routine. It is called once for each connection by MRPC_UnpackMulti with the newly arrived server reply. It can perform as much or as little processing as the client deems necessary, and controls the continuation or termination of the MultiRPC call with its return code. The argument specifications of this routine are explained in detail in section 5.4.1.

5.3.4. Error Cases and Abnormal Behavior

The semantics for errors in the MultiRPC case are somewhat different from those in the RPC2 case. Since several messages are being transmitted in the same call, an error on one connection should not necessarily cause the call to terminate. The client does, however, need to be informed of error states on any of his connections. The handler routine will be called at most once for each connection submitted to the MultiRPC call, either with an error condition or with the server response. No packet will actually be sent on any connection for which an error was detected in the course of processing.

As mentioned earlier, the additional flexibility provided by the client handler routine incurs some risks. RPC2 makes no guarantees as to the state of the connections which are not examined because of an abort by the client. When the client returns an abort code, there may still be some outstanding server replies. RPC2_MultiRPC increments the connection sequence number and resets the connection state, thus pretending that the response in question was actually received. This allows the system to continue with normal operation.

The risks of this approach can be illustrated with some examples. A client makes a MultiRPC request R1 to 3 servers, and terminates the call after two of the server responses have been received. At server S3, the request has been queued because the server was busy with a previous request. The client then decides to make another MultiRPC request R2 on a set of servers that includes server S3 from the first call. S3 then receives R2, tagged with the next logical sequence number, on the same connection as R1. If S3 has not yet begun processing R1, then it will throw R2 away because it recognizes that its sequence number is too high. S3 will then proceed to process R1 and send the response back to the client; the client, however, will promptly throw the response away as a retry because the semantics of his abort command was to pretend that the response to R1 from S3 had already arrived.

Now, assuming that the client chooses to terminate his second call before S3 returns, the client and S3 are completely out of synch. S3, having thrown away R2, will always be expecting a packet with R2's sequence number; the client, however, has already incremented the connection at the termination of R2. In order to keep the connection from hanging around uselessly, S3 will send a RPC2_NAK return code if it ever receives a request R3 on the same connection with a sequence number greater than R2. This will kill the connection, forcing the client to rebind if he wants to continue communicating with S3.

Another risk associated with the use of abort is the risk of not identifying dead connections. If a server S2 is dead but the client always chooses to abort his MultiRPC call before a response from S2 arrives, RPC2 may not have time to notice that the connection is dead.

These problems are a result of the client's ability to ignore the responses on some connections in a MultiRPC call, and will generally only manifest themselves in a case where a server is forced to queue a request because it is busy processing an earlier request. This means that the MultiRPC call should be used with caution in cases where simultaneous binding to a single site might result, although the severity of the problem can be lessened by providing a greater number of LWPs at the single site. It is important to note that these problems arise only in the case where the client chooses to abort the call before all replies have been received. However, the explicit NAK by the server at least gives the client the opportunity to learn that something has gone wrong with the connection and act accordingly.

5.4. 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 client's responsibility to allocate storage for all parameters, just as in the RPC2 case. For all types, IN parameters are handled the same as in the single MakeRPC case. For OUT and IN - OUT 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 IN - OUT 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.

RPC2 Type	C Declaration		
	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* *[ ]

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.

5.4.1. MultiRPC Call Specifications

MRPC_MakeMulti

Pack arguments and initialize state for RPC2_MultiRPC

Call:

long MRPC_MakeMulti(**in** *long ServerOp,* **in** *ARG ArgTypes[],* **in** *long HowMany,*
in *RPC2_Handle CIDList[],* **in** *long (*HandleResult)(),*
in *struct timeval *Timeout, <Variable Length Argument List>)*

Parameters:

ServerOp

For server routine foo, "foo - OP". 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 connection handles, one for each of the servers

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

For all OUT 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.

MRPC_UnpackMulti

Unpack server arguments and call client handler routine

Call:

long MRPC_UnpackMulti(**in** *long HowMany,* **in** *RPC_Handle ConnHandleList,*
in out *ARG_INFO *ArgInfo,* **in** *RPC_PacketBuffer *Response,*
in *long rpcval,* **in** *long thishost)*

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

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.

HandleResult

Process incoming server replies as they arrive

Call:

long 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

<Variable Length Argument List>
These should be specified as described above for MRPC_MakeMulti

Completion Codes:

0 Continue processing server responses

-1 Terminate MRPC_MakeMulti call and return

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.

Appendix I
Usage Notes for the ITC

The .h files (rpc2.h, se.h) are in /cmu/itc/nfs/include.

There are actually two versions of the library: and the normal one, librpc2.a, and one with debugging completely turned off librpc2_s.a. Using librpc2_s.a will make your final load module considerably smaller, but will produce no debugging information at all[1]. For the Suns, these libraries are in /cmu/itc/nfs/lib. For any other supported *machine* the libraries will be in /cmu/itc/nfs/*machine*/lib.

Rp2gen is in /cmu/itc/nfs/bin for the Suns and in /cmu/itc/nfs/*machine*/bin for any other supported *machine*.

The currently supported machines are Suns, Vaxes, and the IBM PC-RT.

The directory /cmu/itc/nfs/release/rpc2 contains a copy of the sources used to build the current version of RPC2. Use this in conjunction with dbx, or if you just wish to examine the source corresponding to the released version. The sources of the immediately preceding released version of RPC2 are in /cmu/itc/nfs/oldv/rpc2.

Compile thus:

```
NFS = /cmu/itc/nfs

cc -g -I$(NFS)/include <<your files>> $(NFS)/lib/librpc2.a $(NFS)/lib/lwp.o\
    $(NFS)/lib/timer.o $(NFS)/lib/iomgr.o -o <<output file>>
```

Stack checking is possible. Refer to the LWP manual for details.

The following external 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.

Setting the hashmark variable to a non-zero character in DumbFTP descriptors will allow
you to watch the progress of file transfers.
```

[1] Tracing will still work.

Appendix II
Remote Site and Communication Failures

Two hazards face the user of an RPC package:

1. The communication medium may fail.

2. The peer process at a remote site may crash.

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.

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:

1. 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.

2. Site failure is declared if silence is observed for a total period of time in the range B_{total} to $2B_{total}$.

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 $B_1 + B_2 + B_3 \ldots B_N = B_{total}$ 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_is, 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:

```
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 all other connections to this (host, portal) pair as RPC2_DEAD;
```

Failure is detected in time BJ_{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 **Busy** packets are sent out by that server 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 DUMBFTP, side effect failure is detected because it is implemented using RPC2. In cases 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.

Tables II-1 and II-2 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 (B_N) that RPC2 waits after the transmission of the last retry, before declaring failure. A lower limit of 500 milliseconds for the retry interval is assumed.

")

	15 secs	30 secs	45 secs	60 secs
1 retries	5.00 (10.00)	10.00 (20.00)	15.00 (30.00)	20.00 (40.00)
2 retries	2.14 4.29 (8.57)	4.29 8.57 (17.14)	6.43 12.86 (25.71)	8.57 17.14 (34.29)
3 retries	1.00 2.00 4.00 (8.00)	2.00 4.00 8.00 (16.00)	3.00 6.00 12.00 (24.00)	4.00 8.00 16.00 (32.00)
4 retries	0.50 0.97 1.94 3.87 (7.73)	0.97 1.94 3.87 7.74 (15.48)	1.45 2.90 5.81 11.61 (23.23)	1.94 3.87 7.74 15.48 (30.97)
5 retries	0.50 0.50 0.95 1.90 3.81 (7.33)	0.50 0.95 1.90 3.81 7.62 (15.21)	0.71 1.43 2.86 5.71 11.43 (22.86)	0.95 1.90 3.81 7.62 15.24 (30.48)
6 retries	0.50 0.50 0.50 0.94 1.89 3.78 (6.89)	0.50 0.50 0.94 1.89 3.78 7.56 (14.83)	0.50 0.71 1.42 2.83 5.67 11.34 (22.53)	0.50 0.94 1.89 3.78 7.56 15.12 (30.21)
7 retries	0.50 0.50 0.50 0.50 0.94 1.88 3.76 (6.41)	0.50 0.50 0.50 0.94 1.88 3.76 7.53 (14.38)	0.50 0.50 0.71 1.41 2.82 5.65 11.29 (22.12)	0.50 0.50 0.94 1.88 3.76 7.53 15.06 (29.82)
8 retries	0.50 0.50 0.50 0.50 0.50 0.94 1.88 3.76 (5.92)	0.50 0.50 0.50 0.50 0.94 1.88 3.76 7.51 (13.91)	0.50 0.50 0.50 0.70 1.41 2.82 5.64 11.27 (21.66)	0.50 0.50 0.50 0.94 1.88 3.76 7.51 15.03 (29.38)
9 retries	0.50 0.50 0.50 0.50 0.50 0.50 0.94 1.88 3.75 (5.43)	0.50 0.50 0.50 0.50 0.50 0.94 1.88 3.75 7.51 (13.42)	0.50 0.50 0.50 0.50 0.70 1.41 2.82 5.63 11.26 (21.18)	0.50 0.50 0.50 0.50 0.94 1.88 3.75 7.51 15.01 (28.91)
10 retries	0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.94 1.88 3.75 (4.93)	0.50 0.50 0.50 0.50 0.50 0.50 0.94 1.88 3.75 7.50 (12.93)	0.50 0.50 0.50 0.50 0.50 0.70 1.41 2.81 5.63 11.26 (20.69)	0.50 0.50 0.50 0.50 0.50 0.94 1.88 3.75 7.50 15.01 (28.42)

Table II-1: Retry Constants (B_{total} = 15 to 60 seconds (0.50 secs lower limit))

	90 secs	120 secs	240 secs	300 secs
1 retries	30.00 (60.00)	40.00 (80.00)	80.00 (160.00)	100.00 (200.00)
2 retries	12.86 25.71 (51.43)	17.14 34.29 (68.57)	34.29 68.57 (137.14)	42.86 85.71 (171.43)
3 retries	6.00 12.00 24.00 (48.00)	8.00 16.00 32.00 (64.00)	16.00 32.00 64.00 (128.00)	20.00 40.00 80.00 (160.00)
4 retries	2.90 5.81 11.61 23.23 (46.45)	3.87 7.74 15.48 30.97 (61.94)	7.74 15.48 30.97 61.94 (123.87)	9.68 19.35 38.71 77.42 (154.84)
5 retries	1.43 2.86 5.71 11.43 22.86 (45.71)	1.90 3.81 7.62 15.24 30.48 (60.95)	3.81 7.62 15.24 30.48 60.95 (121.90)	4.76 9.52 19.05 38.10 76.19 (152.38)
6 retries	0.71 1.42 2.83 5.67 11.34 22.68 (45.35)	0.94 1.89 3.78 7.56 15.12 30.24 (60.47)	1.89 3.78 7.56 15.12 30.24 60.47 (120.94)	2.36 4.72 9.45 18.90 37.80 75.59 (151.18)
7 retries	0.50 0.71 1.41 2.82 5.65 11.29 22.59 (45.03)	0.50 0.94 1.88 3.76 7.53 15.06 30.12 (60.21)	0.94 1.88 3.76 7.53 15.06 30.12 60.24 (120.47)	1.18 2.35 4.71 9.41 18.82 37.65 75.29 (150.59)
8 retries	0.50 0.50 0.70 1.41 2.82 5.64 11.27 22.54 (44.62)	0.50 0.50 0.94 1.88 3.76 7.51 15.03 30.06 (59.82)	0.50 0.94 1.88 3.76 7.51 15.03 30.06 60.12 (120.20)	0.59 1.17 2.35 4.70 9.39 18.79 37.57 75.15 (150.29)
9 retries	0.50 0.50 0.50 0.70 1.41 2.82 5.63 11.26 22.52 (44.16)	0.50 0.50 0.50 0.94 1.88 3.75 7.51 15.01 30.03 (59.38)	0.50 0.50 0.94 1.88 3.75 7.51 15.01 30.03 60.06 (119.82)	0.50 0.59 1.17 2.35 4.69 9.38 18.77 37.54 75.07 (149.94)
10 retries	0.50 0.50 0.50 0.50 0.70 1.41 2.81 5.63 11.26 22.51 (43.68)	0.50 0.50 0.50 0.50 0.94 1.88 3.75 7.50 15.01 30.01 (58.91)	0.50 0.50 0.50 0.94 1.88 3.75 7.50 15.01 30.01 60.03 (119.38)	0.50 0.50 0.59 1.17 2.34 4.69 9.38 18.76 37.52 75.04 (149.51)

Table II-2: Retry Constants (B_{total} = 90 to 300 seconds (0.50 secs lower limit))

Appendix III
Implementation Notes

Some of these refer to bugs, others to restrictions, still others to random useful observations. These are specific to the current state of the RPC2 implementation and are very likely to change in the near future, as refinements are made to RPC2

1. RPC2 runs on Suns, Vaxen and the IBM PC-RT machines.

2. Only one portal in RPC2_Init.

3. Only DumbFTPD currently supported.

4. getsubsysbyname() is a fake routine. It knows about "Vice2-FileServer" and "DumbFTP-Server" and "Vice2-CallBack".

5. RPC2_MultiRPC not implemented yet.

Appendix IV
Recent Changes

This appendix summarizes the differences between the latest release of RPC (i.e. corresponding to this manual) and the previous release.

This is release 7 (Version 7.0). The immediately preceding release was 6 (Version 6.2).

Changes visible to the user are:

1. There is a new call RPC2_Enable() which you must use on the server side to enable connections after they are established. This is done for you by RP2Gen if you use it.

2. You must now call XXX_Activate() to activate each type of side effect XXX. If you do not call this routine code for that side effect will not be linked in. For example you must call DFTP_Activate() to enable the dumb file transfer protocol.

3. Each side effect XXX now has a XXX_SetDefaults() routine which sets defaults initialization values on a variable of type XXX_Initializer.

4. RPC2_GetPeerInfo() now returns information in a structure rather than as a long sequence of arguments.

5. RPC2_SendResponse no longer has a SE_Descriptor argument.

6. You no longer have to include dftp.h if you are using the DFTP side effect routines.

Changes internal to RPC2 and invisible to the user:

1. Support is being added for SFTP, the faster file transfer protocol. However, it will not be enabled by default. The next release will have it enabled.

Appendix V
Summary of RPC-related Calls

Note: The numbers in square brackets indicate the page on which the call is described.

References

[1] Jonathan Rosenberg, Larry Raper, David Nichols, M. Satyanarayanan.
LWP Manual
Information Technology Center, CMU-ITC-037, 1985.

[2] M.Satyanarayanan.
RPC Manual
Information Technology Center, CMU-ITC-011, 1984.

List of Tables

[22]

*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)*

[25]

*RPC2_MakeRPC(***in** *RPC2_Handle ConnHandle,* **in** *RPC2_PacketBuffer *Request,*
in *SE_Descriptor *SDesc,* **out** *RPC2_PacketBuffer **Reply,*
in *struct timeval *Patience,* **in** *long EnqueueRequest)*

[27]

*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)*

[29]

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

[30]

*RPC2_DeExport(***in** *RPC2_SubsysIdent *Subsys)*

[31]

*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)())*

[34]

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

[35]

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

[36]

*RPC2_InitSideEffect(***in** *RPC2_Handle ConnHandle,* **in** *SE_Descriptor *SDesc)*

[37]

*RPC2_CheckSideEffect(***in** *RPC2_Handle ConnHandle,*
inout *SE_Descriptor *SDesc,* **in** *long Flags)*

[38]

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

[40]

*RPC2_Unbind(***in** *RPC2_Handle ConnHandle)*

[41]

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

[42]

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

[43]

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

[44]

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

[45]

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

[46]

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

[47]

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

[48]

RPC2_LamportTime()

[49]

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

[50]

*RPC2_InitTraceBuffer(***in** *long HowMany)*

[51]

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

[52]

*XXX_SetDefaults(***in** *XXX_Initializer *Initializer)*

[53]

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

[59]

SE_Init()

[60]

*SE_Bind1(***in** *RPC2_Handle ConnHandle,* **in** *RPC2_CountedBS *ClientIdent)*

[61]

*SE_Bind2(***in** *RPC2_Handle ConnHandle)*

[62]

*SE_Unbind(***in** *RPC2_Handle ConnHandle)*

[63]

*SE_NewConnection(***in** *RPC2_Handle ConnHandle,*
in *RPC2_CountedBS *ClientIdent)*

[64]

*SE_MakeRPC1(***in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc,*
inout *RPC2_PacketBuffer **RequestPtr)*

[65]

*SE_MakeRPC2(***in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc,*
inout *RPC2_PacketBuffer *Reply)*

[66]

*SE_GetRequest(***in** *RPC2_Handle ConnHandle,*
inout *RPC2_PacketBuffer *Request)*

[67]

*SE_InitSideEffect(***in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc)*

[68]

*SE_CheckSideEffect(***in** *RPC2_Handle ConnHandle,* **inout** *SE_Descriptor *SDesc,*
in *long Flags)*

[69]

*SE_SendResponse(***in** *RPC2_Handle ConnHandle,*
in *RPC2_PacketBuffer **ReplyPtr)*

[70]

*SE_PrintSEDescriptor(***in** *SE_Descriptor *SDesc,* **in** *FILE *outFile)*

[71]

*SE_SetDefaults(XXX_Initializer *SInit)*

[72]

*SE_Activate(***in** *XXX_Initializer *SInit)*

[104]

*MRPC_MakeMulti(***in** *long ServerOp,* **in** *ARG ArgTypes[],* **in** *long HowMany,*
in *RPC2_Handle CIDList[],* **in** *long (*HandleResult)(),* **in** *struct timeval *Timeout,*
<Variable Length Argument List>)

[106]

*MRPC_UnpackMulti(***in** *long HowMany,* **in** *RPC_Handle ConnHandleList,*
in out *ARG_INFO *ArgInfo,* **in** *RPC_PacketBuffer *Response,* **in** *long rpcval,*
in *long thishost)*

[107]

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