Why Deployed SUNs Should Have Disks.

M. Satyanarayanan
Mike West

10 July 1984

1. The deployment this fall should be a *dress-rehearsal* of the large-scale deployment scheduled for 1985 and 1986. Except in scale, the deployment this fall should be as close as possible to the final deployment.

2. The target IBM workstation will have disks. Therefore, our fall 1984 deployment should match this.

3. Unless pressing need is experienced in the current deployment, there will be NO motivation for effort in tasks such as:

   a. disk maintenance,

   b. mechanics of software distribution,

   c. and development of software for disconnected operation.

   Necessity is the mother of invention.

4. It is claimed that deploying diskless SUNs allows recycling disk servers as file servers. What is to become of the diskless SUNs in that case? If each SUN had its local disk, we would have 75 stand-alone usable systems to be recycled. Otherwise we would have 75 worthless pieces of hardware.

5. In addition to the above primary reasons, there are a host of secondary reasons not to go diskless:

   a. VICE design has always assumed workstations with disks. During the design phase, diskless operation was explicitly ignored because it was argued that the cost of disks was declining to a point where disklessness was not desirable. *We should revisit the design if our ground rules have been changed.*

   b. Since disklessness implies non-local paging, we will NEVER be able to disassociate the performance penalties caused by non-local file access from the penalties of non-local paging.

   c. The figures cited indicate that disky SUNs will cost about $2000 more than diskless ones. For 75 SUNs, this amounts to $150K. We believe this is a trivial price to pay for the ability to more closely simulate the final deployment.

   d. SUN will be discontinuing disk server support from December 1984. They will then be switching to a file server. Are we really interested in deploying a file server cascade? (Remember this is similar to PC Server!). If we persist with disk servers, who will maintain the software?

The problem is exacerbated by the need to boot workstations off disk servers. How will the initial authentication be done? Will workstation ROMs contain code for authentication handshakes? Where will passwords be stored? Concern has been expressed over the presence of passwords in the clear in core dumps; their presence in ROMs can hardly cause less anxiety. Note that we CANNOT afford to leave out authentication in the boot sequence. Otherwise the system is vulnerable to Trojan horse attacks by malicious workstations masquerading as disk servers. In fact the entire area of network booting is open to Trojan horsemanship.

Security in a diskless environment is clearly a fascinating research topic. It may ultimately be shown that the abovementioned fears are groundless, and that diskless operation is indeed compatible with security. However, such a statement cannot be made about the present SUN disk server environment.

In view of these observations I recommend that we take one of the following positions:

1. *Make it a policy decision that we are not concerned with security.*
   This will avoid raising false hopes, and will enable us to gain efficiency in other areas, since we will no longer have to do encryption anywhere.

2. *Avoid diskless operation in our Fall deployment.*
   If we insist on it as a long term goal, we should launch an effort to understand and build a secure diskless environment.

- Each request consists of an ***Operation*** and a set of ***Parameters***.

- A reply consists of a ***Return Code*** and a set of result parameters.

- A possible side-effect of a request is the transmission of a large object, generically referred to as a ***Bulk-Unit***. For example, when dealing a file server, an entire file may retrieved by an appropriate request. Mechanisms used to transfer bulk-units are referred to as ***Bulk-Transfer*** mechanisms. Each client-server connection deals with one kind of bulk-unit and an associated bulk-transfer mechanism.

- RPCs are synchronous with respect to the client and server. Bulk-transfers occur between the relinquishing of control by the server and the resumption of execution by the client.

- An RPC data type called a ***Bulk-Descriptor*** acts as a placeholder for a bulk-unit in a parameter list. It contains the information needed to effect a bulk-transfer of the appropriate object.

- A server detects server-client communication failure synchronously on RPC_GetRequest and RPC_SendResponse calls. There is no way for the server to detect situations where the communication link is healthy but the client process is sick (e.g., in an infinite loop).

- A client also detects server-client communication failure synchronously, on a RPC_MakeRPC call. To permit detection of sick servers, a timeout mechanism is provided on RPC_MakeRPC. If a client desires further reassurance, it may periodically generate dummy RPC requests on each of its connections. By convention, all servers recognize and respond immediately to the opcode ***Ping***. No automatic pinging is done by the RPC stubs.

A ***Security Level*** is associated with each connection between a client and a server:

- Currently three levels of security are supported:
  - neither authenticated nor secure
  - authenticated but not secure
  - both authenticated and secure.

- ***Authenticated*** in this context means that the client and server start out as mutually suspicious parties and exchange credentials during the establishment of a connection. A secret encryption key, known a priori only to the server and client is used in authentication handshakes.

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

- The RPC package makes no assumptions about the format of client identities or about

the mapping between clients, servers and shared secret keys. A server-supplied ***Callback Procedure*** is invoked during the authentication sequence to validate client identities and obtain keys.

1.1. An Example of a Trivial Client

```
main( )
  {
  int c1, c2, c3;                          /*to hold connection ids*/
  /* random initial processing */
  RPC_ClientInit( /* appropriate arguments */);

  c1 = RPC_Bind( /* arguments for Service 1 */);
  c2 = RPC_Bind( /* arguments for Service 2 */);
  c3 = RPC_Bind( /* arguments for Service 3 */);

  while (WorkExists)
    {
    /* random processing */

    RPC_MakeRPC(c1, /* appropriate arguments */);

    /* other processing; calls via connections c2 and c3 as needed */
    }

  RPC_UnBind(c1);
  RPC_UnBind(c2);
  RPC_UnBind(c3);
  }
```

1.2. An Example of a Trivial Forking Server

```
main( )
  {
  int IamAWorker;

  /* random initial processing */
  RPC_ServerInit( /* service name */, RPC_FORKINGSERVER );

  while (TRUE)
    {
    /* random processing */

    RPC_Accept(IamAWorker,/* other arguments */);
    if (IamAWorker = = 0) break;
    /* else I am the listener */

    /* other processing; then parent goes back and listens for more */
    }

  /* Only a worker would get here */
  while (TRUE)
    {
    /* random processing */
```

```
    RPC_GetRequest(/* appropriate arguments */);
    /* process this request and fill in bulk descriptor */
    RPC_SendReply(/* appropriate arguments */);

    if (/* this was a Disconnect request */) break;
    }

  RPC_EndWork( );
  }
```

1.3. An Example of a Trivial Nonforking Server

```
main( )
  {
  /* random initial processing */
  RPC_ServerInit( /* service name */, RPC_NONFORKINGSERVER );

  while (TRUE)
    {
    /* I am a listener */

    RPC_Accept(IamAWorker,/* other arguments */);

    while (TRUE)
      {
      /* I am now a worker */

      /* random processing */

      RPC_GetRequest(/* appropriate arguments */);
      /* process this request and fill in bulk descriptor */
      RPC_SendReply(/* appropriate arguments */);

      if (/* this was a Disconnect request */) break;
      }
    RPC_EndWork( );
    }
  }
```

Editorial Note:

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 ***Request*** and ***Response 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.

For convenience, the following sections describe RPC runtime data types and data structures using C definitions. C syntax is being used here as a means of specification, in conjunction with the comments.

These definitions are found in a the C header file "/usr/local/rpc/include/rpc.h". Those header files are the authoritative source of these definitions, and will be more up-to-date than this manual.

1.4. Constants

```
        #define RPC_VERSION "$Header: rpcglobs.mss,v 1.3 84/04/13 15:25:01 satya Exp $"
/*
The above string is in theory a random magic string.  In practice it is the header inserted by the RCS system to uniquely identify
this revision level.  It is used in RPC initialization and bind calls to ensure that the client runtime system, server runtime system,
and the header files on both sides are all mutually consistent
*/

        #define BUFFPREFIX    16              /*Size of buffer prefix used by stub.*/
/*
The following is the minimum sized buffer to hold both requests and responses:
*/
        #define MINBUFFSIZE (BUFFPREFIX + \
            (sizeof(struct RPC_ReqHdr) < sizeof(struct RPC_RespHdr)\
            ? sizeof(struct RPC_RespHdr) : sizeof(struct RPC_ReqHdr) ))

        #define RPC_FORKINGSERVER    341    /* random on purpose */
        #define RPC_NONFORKINGSERVER  1123   /* random on purpose */


/*
The following constants are used to indicate the security-level of RPC connections.  They are likely to be extended in future.
*/
        #define RPC_OPENKIMONO 938          /*Neither authenticated nor encrypted*/
```

```
#define RPC_ONLYAUTHENTICATE 12        /*Authenticated but not encrypted*/
#define RPC_SECURE 33291               /*Authenticated and encrypted*/

#define RPC_KEYSIZE 8                  /*Size in bytes of the encryption keys used in RPC*/
```

/*
RPC procedure return codes:
These may also occur in the RPC_ReturnCode and RPC_BulkReturnCode fields 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.)
*/

```
#define RPC_SUCCESS  0
#define RPC_FAIL     -1
#define RPC_NOCONNECTION -2
#define RPC_TIMEOUT  - 3
#define RPC_BULKSUCCESS 0
#define RPC_BULKFAILURE -4
#define RPC_BULKTRAGEDY -5
#define RPC_NOBINDING -6
#define RPC_DUPLICATESERVER -7
#define RPC_ALREADYLISTENER -8
#define RPC_NOTLISTENER -9
#define RPC_NOTWORKER -10
#define RPC_ALREADYWORKER -11
#define RPC_NOTCLIENT -12
#define RPC_TOOLONG -13
#define RPC_WRONGVERSION -14
#define RPC_NOTAUTHENTICATED -15
#define RPC_CLOSECONNECTION -16
```

/*
I will think up more
*/

/*
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. This applies to both Subsys and Opcode fields. fields in the RPC_ReqHdr.
*/

```
#define PING -1                        /*All servers should return RPC_SUCCESS upon seeing this
                                        request. Used for end-to-end pinging by clients.*/
```

/*
Bulk-transfer protocol identification.
*/

```
#define EXPBULKPROTO1  23              /*random on purpose*/
```

/*
RPC server options:
These are used as bit settings in the ServerOptions field of RPC_ServerInit.)
*/

```
#define RPC_REVIVESERVER      0x1      /*This process is being restarted as a listening server. Reset
                                        internal data structures, and force Unix to reuse port numbers.
                                        The latter function is important when a server process is killed
                                        and a new one started in its place.*/
```

/*
Debugging aids:
The global external variables RPC-ServerDebugLevel and RPC-ClientDebugLevel control the level of debugging output produced on stdout in the server and client respectively. A value of 0 turns off the output altogether; values of 1, 10, and 100

are currently meaningful. The default values of these variables is 0.

The global external variables RPC - ServerPerror and RPC - ClientPerror control the printing of Unix error messages on stdout in the server and client respectively. A value of 1 turns on the printing, while 0 turns it off. The default value for these variables is 1.
*/

1.5. Types

```
typedef
  int RPC_Integer;                    /*32-bit, 2's complement representation. On other machines, an
                                      explicit conversion may be needed.*/

typedef
  char RPC_Byte;                      /*A single 8-bit byte.*/


typedef
  char RPC_ByteSeq[1];                /*Should really be char [*]*/
/*
A contiguous sequence of bytes. This is merely a placeholder in the definitions below. At runtime, you are expected to know
where this item begins and how long this sequence is. Use array notation, or a pointer to step through this. Don't expect
sizeof( ) to work correctly on anything containing an RPC_ByteSeq element.
*/


typedef
  RPC_ByteSeq RPC_String;             /*no nulls except last byte*/
/*
A null-terminated sequence of characters. Identical to the C language string definition.
*/


typedef
  struct
    {
    RPC_Integer SeqLen;               /*length of SeqBody*/
    RPC_ByteSeq SeqBody;              /*no restrictions on contents*/
    }
  RPC_CountedBS;
/*
A means of transmitting binary data.
*/

typedef
  struct
    {
    RPC_Integer MaxSeqLen;            /*max size of buffer represented by SeqBody*/
    RPC_Integer SeqLen;               /*number of interesting bytes in SeqBody*/
    RPC_ByteSeq SeqBody;              /*No restrictions on contents*/
    }
  RPC_BoundedBS;
/*
RPC_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
  RPC_Byte RPC_EncryptionKey[RPC_KEYSIZE];
/*
Keys used for encryption are fixed length byte sequences
*/
```

1.6. Data Structures

```
/*
Fields filled in by stubs are identified explicitly
*/

struct RPC_ReqHdr                    /*Fixed-length; format and length determined by ProtoVersion.
                                     */
  {
  RPC_Integer ProtoVersion;          /*which protocol version to use. Filled by client*/
  RPC_Integer BodyLength;            /*of the portion after the header. Filled by client.*/
  RPC_Integer Tag;                   /*unique identifier for this message on this connection; always
                                     has an odd value; filled by client stub */
  RPC_Integer Subsys;                /*Subsystem name. Filled by client. Value should be greater than
                                     0.*/
  RPC_Integer Opcode;                /*Operation meaningful to this subsystem. Filled by client. Value
                                     should be greater than 0.*/
  RPC_Integer NoReturnValue;         /*If zero, a reply from the server is expected. If nonzero, no
                                     reply is expected.*/
  };

struct RPC_ReqBlock                  /*This is what actually gets sent over the wire */
  {
  struct RPC_ReqHdr Header;          /*Length field contains length of next element */
  RPC_ByteSeq Body;                  /*bytes corresponding to the parameters*/
  };

struct RPC_ReqBuffer                 /*Allocate this as buffer for requests and use in RPC calls*/
  {
  RPC_Integer BufferPrefix[BUFFPREFIX/sizeof(RPC_Integer)];
                                     /*for stub use only*/
  struct RPC_ReqBlock ActualRequest;
  };

struct RPC_RespHdr                   /*Fixed-length; format and length determined by ProtoVersion
                                     */
  {
  RPC_Integer ProtoVersion;          /*which protocol version to use. Filled by server.*/
  RPC_Integer BodyLength;            /*of the portion after the header. Filled by server.*/
  RPC_Integer Tag;                   /*unique identifier for this message on this connection; always
                                     an even value: filled by server stub.*/
  RPC_Integer InReplyTo;             /*Tag value of request; filled by server stub */
  RPC_Integer ReturnCode;            /*standard or operation-dependent error code. Filled by server.
                                     */
  RPC_Integer BulkReturnCode;        /*indicates what happened to bulk transfer. Filled by server
                                     stub.*/
  };

struct RPC_RespBlock                 /*This is what actually gets sent over the wire */
  {
  struct RPC_RespHdr Header;         /*BodyLength field contains length of next element.*/
```

```
        RPC_ByteSeq Body;                    /*bytes corresponding to the non-bulk results.*/
        };

      struct RPC_RespBuffer                  /*Allocate this as buffer for Responses and use in RPC replies*/
        {
        RPC_Integer BufferPrefix[BUFFPREFIX/sizeof(RPC_Integer)];
                                             /*for stub use only*/
        struct RPC_RespBlock ActualResponse;
        };
```

```
/*
Historical note:
The definition of RPC_BulkDescriptor that used to be here is now superfluous.  It looks like a RPC_BoundedBS and is viewed
as such.
*/
```

2. Bulk-Transfer Protocols

Each bulk-transfer protocol has its own bulk-descriptor format. For purposes of transmission, these descriptors are merely viewed as data of type RPC_CountedBS.

2.1. Protocol EXPBULKPROTO1

At the present time, only one bulk-transfer mechanism is being supported. It is symbolically referred to as EXPBULKPROTO1 and provides whole file transfer between client and server. The bulk-descriptor associated with this protocol is defined below. The corresponding C header file is "/usr/local/rpc/include/ftp.h".

2.1.1. Constants

```
/*
Global constants which are used by the bulk-transfer mechanism
*/

/*
The following op codes filled in by the server and passed to its stub.  The same codes are sent back to the client on the
bulk-transfer channel to prepare it for bulk-transfer
*/
        #define FTP_SEND   1                 /*Transfer file from server to client.*/
        #define FTP_RECEIVE 2                /*Transfer file from client to server.*/
```

2.1.2. Format of Bulk Descriptor

```
/*
Bulk-descriptor definition.
*/

/*
Right now we are only concerned with a bulk-transfer mechanism for dealing with one file transfer.  Bulk-descriptors for
dealing with multiple file transfers may be developed later.
```

```
*/

/*
The client process fills in some fields of the descriptor and calls its RPC stub. The server process fills in other fields of these
descriptors, creates a response block and a pointer to the bulk-descriptor and calls its RPC stub. The bulk-transfer is ALWAYS
initiated by the stub on the server side. On successful completion of the bulk transfer, it sets the RPC_BulkReturnCode field in
the response block to RPC_BULKSUCCESS. If the bulk transfer fails, it is set to RPC_BULKFAIL. The response block is then
sent to the client, and the server stub returns control to the server. On receipt of the response block, the client stub returns
control to its caller.
*/

/*
What follows is the format of the SeqBody component of an RPC_CountedBS. The SeqLen component will be set to the
current size of the struct FTP_Descriptor) variable.
*/

struct FTP_Descriptor
  {
  RPC_Integer Operation;          /*FTP_SEND or FTP_RECEIVE; filled in by Server.*/
  RPC_Integer Length;             /*Number of bytes to be transmitted; filled in by Server for
                                  FTP_SEND and by Client for FTP_RECEIVE: usually equal to
                                  length of file. A value of -1 on FTP_SEND means the entire file*/
  RPC_Integer Protection;         /*on the side where the file is to be created. On FTP_RECEIVE,
                                  client fills in this field; the server may choose to inherit this
                                  protection or to override it before calling RPC_SendResponse().
                                  On FTP_SEND client fills in this field with the desired protection,
                                  or setsit to 0; in the latter case the protection on the server is
                                  inherited.See chmod(2) in Unix manual for details*/
  RPC_String ClientFileName;      /*File name on client side: filled in by Client */
  RPC_String ServerFileName;      /*File name on Server side; filled in by Server */
  };
```

3. Client-related RPC Calls

RPC_ClientInit

Initialize RPC stub to be a client

Call:

*int RPC_ClientInit(IN int MaxSockets, IN char *VersionId, IN int ClientOptions)*

Parameters:

MaxSockets	Maximum number of sockets that may be used by the RPC stub. A value of -1 indicates that the client does not care how many are used.
VersionId	Set this to the constant RPC_VERSION. The current value of this string constant must be identical to the value at the time the client runtime system was compiled.
ClientOptions	Currently there are no valid client options; this parameter is here for future use.

Completion Codes:

RPC_SUCCESS	All went well
RPC_FAIL	Unable to initialize client.
RPC_WRONGVERSION	The header file and client runtime system versions do not match.

Initializes the RPC stub in this process. Since connections are NOT inherited, every process must make this call, and perform bindings before RPC requests can be made. The MaxSockets parameter is advisory information for the stub. If you get a wrong version indication, obtain a consistent version of the header files and the RPC runtime library and recompile your code.

RPC_Bind

Create a new connection

Call:

*int RPC_Bind(IN int SecurityLevel, IN char *Subsysname, IN char *Hostname,*
*IN int Bulkproto, OUT int *cid,*
*IN RPC_BoundedBS *ClientIdent,*
IN RPC_EncryptionKey SharedSecret)

Parameters:

SecurityLevel	One of the constants RPC_OPENKIMONO, RPC_ONLYAUTHENTICATE, or RPC_SECURE
Subsysname	The name of the subsystem whose services are desired
Hostname	The network host name where the subsystem is located
Bulkproto	The bulk-transfer protocol to be used on this connection
cid	A small integer returned by the call, identifying this connection
ClientIdent	Adequate information for the server to uniquely identify this client and to obtain SharedKey. Not interpreted by the RPC stubs. Only the callback procedure on the server side need understand the format of ClientIdent. May be NULL if SecurityLevel is RPC_OPENKIMONO
SharedSecret	An encryption key known by the callback procedure on the server side to be uniquely associated with ClientIdent. Used by the RPC stubs in the authentication handshakes. May be NULL if SecurityLevel is RPC_OPENKIMONO.

Completion Codes:

RPC_SUCCESS	All went well
RPC_NOTCLIENT	You are not properly initialized.
RPC_NOBINDING	The specified host or subsystem could not be contacted
RPC_WRONGVERSION	The client and server runtime systems are incompatible.
RPC_NOTAUTHENTICATED	A SecurityLevel other than RPC_OPENKIMONO was specified, and the server

did not accept your credentials.

RPC_FAIL Some other mishap occurred. May also occur sometimes in lieu of RPC_NOTAUTHENTICATED.

Creates a new connection and binds to a remote server on a remote host. At the end of this call, a worker process has been forked to deal with this client. On a nonforking server, the server enters worker state.

A client/server version number check is performed to ensure that the runtime systems are compatible. You are advised to do a similar higher-level check, to ensure that the client and server application code are also compatible.

The SecurityLevel parameter determines the degree to which you can trust this connection. If RPC_OPENKIMONO is specified, the connection is not authenticated and no encryption is done on future requests and responses. If RPC_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 RPC_SECURE is specified, the connection is authenticated and all future transmissions on it are encrypted using a session key generated during the authentication handshake.

RPC_MakeRPC

Make a remote procedure call

Call:

*int RPC_MakeRPC(IN int cid, IN struct RPC_ReqBuffer *Request,*
*OUT struct RPC_RespBuffer **Reply,*
*IN struct timeval *BreathOfLife)*

Parameters:

cid	identifies the connection on which the call is to be made
Request	A properly formatted request buffer.
Reply	Value ignored on entry. On return, it will point to a response buffer holding the response from the server. Do not count on this buffer remaining around after the next RPC – MakeRPC() call.
BreathOfLife	Maximum time to wait for remote site to respond to any communication. Used internally to timeout blocking operations. A NULL pointer indicates infinite patience. Struct timeval specifies time in seconds and microseconds; see gettimeofday(2) in the Unix manual for further details.

Completion Codes:

RPC_SUCCESS	All went well
RPC_NOTCLIENT	You are not properly initialized.
RPC_NOCONNECTION	Bogus connection specified
RPC_TIMEOUT	A response was not received soon enough.
RPC_TOOLONG	The server tried to transmit a response with a BodyLength field that was too large to deal with. Future requests on this connection will get RPC_FAIL.
RPC_FAIL	Other assorted calamities (such as a broken connection)

The workhorse routine, used to make remote calls after establishing a connection. In strict Unix style, the call is sequential and blocks until complete. All bulk transfers are finished before the call completes. The listed completion codes are from the local RPC stub. Check the RPC_ReturnCode and RPC_BulkReturnCode fields of the reply to see what the remote site thought of your request. If

the remote site takes longer than BreathOfLife to reply to any individual blocking operation, the connection is deemed broken and future requests on this connection will be met with a response of RPC_FAIL.

The timeout mechanism also provides a way for the client to perform end-to-end pinging if it so desires. By convention, an opcode of ***Ping*** is recognized by all servers and is responded to immediately by all of them with a return code of RPC_SUCCESS. Timer-driven pinging of the server by the client is easily implemented with these facilities.

Note that BreathOfLife specifies patience for individual actions. To some extent, therefore, the exact effect of a particular timeout value is implementation-dependent. If an overall time limit for the entire RPC is desired, the client should start an alarm clock before calling RPC_MakeRPC. If the alarm clock runs out, this connection should be abandoned.

If the NoReturnValue field of the request is nonzero, it is assumed that the server will not attempt to send a response. Consequently this call will return without attempting to read a server reply. In that case Reply will be NULL. Beware: if an errant server does send a response to such a request, you are in deep trouble; future RPC requests on this connection will behave strangely

Encryption, if any, is done in place and will cause the request buffer to be clobbered.

RPC_Unbind

Terminate a connection

Call:

int RPC_Unbind(IN int cid)

Parameters:

cid	identifies the connection to be terminated

Completion Codes:

RPC_SUCCESS	All went well
RPC_NOTCLIENT	You are not initialized properly.
RPC_NOCONNECTION	The cid is bogus
RPC_FAIL	Other assorted calamities

Removes the binding associated with the specified connection. Normally the server-level disconnection should be done by an RPC just prior to this call.

4. Server-related RPC Calls

RPC_ServerInit

Declare onself a listening service process

Call:

*int RPC_ServerInit(IN char *MySubsysName, IN int ServerType,*
*IN char *VersionId, IN int ServerOptions)*

Parameters:

MySubsysName	Well-known subsystem name. Uniquely identifies this process on this machine.
ServerType	Possible values are RPC_FORKINGSERVER and RPC_NONFORKINGSERVER
VersionId	Set this to the constant RPC_VERSION. The current value of this constant must be equal to the value at the time the server runtime system was compiled.
ServerOptions	A bit mask of options. RPC - REVIVESERVER is the only meaningful option to a server at present.

Completion Codes:

RPC_SUCCESS	All went well
RPC_DUPLICATESERVER	You have a twin. You may wish to retry with the RPC - REVIVESERVER option.
RPC_ALREADYLISTENER	You have already said you are a listening server
RPC_WRONGVERSION	The header file and server runtime system version numbers do not match.
RPC_FAIL	Something else went wrong.

Makes this listening server process known to the rest of the world. Sets up name server tables so that when a client performs an RPC_Bind( ) operation specifying this subsystem-host pair, the underlying socket mechanism will know where to go. A process can be a listening server for at most

one subsystem pair. If you get a wrong version indication, obtain a consistent version of the header files and the RPC runtime library and recompile your code. If this server was recently killed, Unix may not allow you to start another server with the same service name for a certain period of time. To alleviate this problem, use the RPC – REVIVESERVER option.

RPC_Accept

Listen for a bind request from a client

Call:

*int RPC_Accept(OUT int *WhoAmI, OUT int *BulkProto, IN int (*GetKeys)(), OUT int *SecurityLevel, OUT RPC_BoundedBS **ClientIdent)*

Parameters:

WhoAmI — Value on return indicates whether I am a listening or working server. The working server receives a value of 0, while the listening server receives the process id value of the newly-forked working server. Remember that the listening server itself becomes the working server when an RPC_Accept is done in a nonforking server.

BulkProto — In the working server, the value returned is the name of the bulk transfer protocol to be used. In the listening server a value of NULL is returned.

GetKeys — Pointer to a callback procedure with the following formal declaration:

```
int GetKeys(IN ClientIdent, OUT IdentKey, OUT SessionKey)

  RPC_BoundedBS *ClientIdent;

  RPC_EncryptionKey IdentKey;

  RPC_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. May be NULL if no secure bindings to this server are to be accepted.

SecurityLevel — On return, this will contain RPC_OPENKIMONO, RPC_ONLYAUTHENTICATE, or RPC_SECURE.

ClientIdent — On return, if SecurityLevel is other than RPC_OPENKIMONO, this will contain the identity of the client. This is identical to the information passed in the corresponding RPC_Bind call on the client side, and to the callback procedure GetKeys().

Completion Codes:

RPC_SUCCESS — All went well .

RPC_NOTLISTENER
You did not declare yourself to be a listening server process

RPC_NOTAUTHENTICATED
Someone tried to do an authenticated RPC_Bind to me, but failed. ClientIdent contains the identity of the alleged client. Take suitable action and reissue RPC_Accept.

RPC_FAIL Something else went wrong

This process must have made an RPC_ServerInit call previously. The call blocks until someone, somewhere does an RPC_Bind( ) to this listening server. If a forking server, the server RPC stub forks a new server process, which will serve the RPC request on that connection. In a nonforking server, the server enters the working state.

It is verified that the client and server RPC runtime systems are compatible.

The security level of this connection is specified in the corresponding RPC_Bind call on the client side. If RPC_OPENKIMONO is specified, no authentication is done. Otherwise authentication is done and the identity of the accepted client is returned in ClientIdent.

The GetKeys() callback procedure is used by the RPC stub when creating authenticated connections. Unsuccessful RPC_Bind()s by clients are reported to the server; this may be of use in dealing with malicious clients.

RPC_GetRequest

Wait for next request from my client

Call:

*int RPC_GetRequest(IN struct timeval *BreathOfLife,*
*OUT struct RPC_RequestBuffer **Request)*

Parameters:

BreathOfLife A timeout interval to be used in all blocking system calls. If NULL, infinite patience is assumed. This is not a highly accurate mechanism, but it does detect inactive clients. Note that the underlying sockets also use keepalives, so this parameter is needed only if you wish to detect the case where the application program at the remote site is inactive. Note that this is per-blocking system call, not for this entire RPC call.

Request Value ignored on entry. On return, it will point to a requestbuffer holding the response from the client. Do not count on this buffer remaining around after the next RPC – GetRequest() call.

Completion Codes:

RPC_SUCCESS I have a request for you

RPC_NOTWORKER
You are not a working server process.

RPC_TOOLONG The client tried to transmit an enormous request. Future RPC_GetRequest() calls will get RPC_FAIL.

RPC_TIMEOUT Specified time interval expired.

RPC_CLOSECONNECTION
The remote site deliberately closed this connection

This call may be issued only by a worker server process. The call blocks until a request is available or until the specified timeout period has elapsed. Obtaining a RPC_CLOSECONNECTION return code to this call is usual way a server learns of the disappearance of a client.

RPC_SendResponse

Respond to a request from my client

Call:

*int RPC_SendResponse(IN struct RPC_ResponseBuffer *Reply,*
*IN FTP_Descriptor *BDesc, IN struct timeval *BreathOfLife)*

Parameters:

Reply	A filled-in buffer containing the reply to be sent to the client on completion of bulk transfer.
BDesc	A bulk descriptor, or NULL. If non-NULL, the bulk transfer defined by the descriptor will be carried out before the reply is sent. We may extend this to multiple bulk descriptors later.
BreathOfLife	Timeout interval for blocking operations. Note that this is per-blocking system call, not for this entire RPC call.

Completion Codes:

RPC_SUCCESS	I sent your response, and tried to perform the bulk transfer, if any.
RPC_NOTWORKER	You are not a working server process.
RPC_TIMEOUT	The specified timeout period was exceeded.
RPC_FAIL	Some irrecoverable failure happened.

This call may be issued only by a worker server process. If BulkDesc is NULL, the Reply is sent back to the client and the call terminates. Otherwise the bulk transfer specified is carried out first, and then Reply is sent to the client. In that case, the RPC_BulkReturnCode field will be filled in by the bulk transfer stub. If it cares, the server should examine this field on completion of the call.

Encryption, if any, is done in place and will clobber both parameters. The timeout mechanism is not particularly accurate.

RPC_EndWork

Terminate worker server

Call:

int RPC_EndWork( )

Parameters:

None

Completion Codes:

RPC_NOTWORKER

You are not a working server process.

This call is typically issued after a Disconnect request is received by the working server. In the initial implementation it will merely result in process destruction. It is present in case the cost of forking workers becomes unacceptable; in that case something smart can be done with these semi-dead processes waiting for resurrection

In the case of a nonforking server, this call returns the server from a working to a listening state.

RPC_EndListen

Terminate a listening server

Call:

int RPC_EndListen( )

Parameters:

None

Completion Codes:

RPC_NOTLISTENER

You are not a listening server process.

This call is present mainly for symmetry. If a listening server chickens out, and decides it cannot handle any more binds to it, it issues this call. RPC_Binds to this subsystem-hostname pair will no longer be routed to this process. However process destruction will not occur until all the forked worker processes have terminated.

Appendix I
Summary of RPC-related Calls

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

[12]

*RPC_ClientInit(IN int MaxSockets, IN char *VersionId, IN int ClientOptions)*

[13]

*RPC_Bind(IN int SecurityLevel, IN char *Subsysname, IN char *Hostname, IN int Bulkproto, OUT int *cid, IN RPC_BoundedBS *ClientIdent, IN RPC_EncryptionKey SharedSecret)*

[15]

*RPC_MakeRPC(IN int cid, IN struct RPC_ReqBuffer *Request, OUT struct RPC_RespBuffer **Reply, IN struct timeval *BreathOfLife)*

[17]

RPC_Unbind(IN int cid)

[18]

*RPC_ServerInit(IN char *MySubsysName, IN int ServerType, IN char *VersionId, IN int ServerOptions)*

[20]

*RPC_Accept(OUT int *WhoAmI, OUT int *BulkProto, IN int (*GetKeys)(), OUT int *SecurityLevel, OUT RPC_BoundedBS **ClientIdent)*

[22]

*RPC_GetRequest(IN struct timeval *BreathOfLife, OUT struct RPC_RequestBuffer **Request)*

[23]

*RPC_SendResponse(IN struct RPC_ResponseBuffer *Reply, IN FTP_Descriptor *BDesc, IN struct timeval *BreathOfLife)*

[24]

RPC_EndWork( )

[25]

RPC_EndListen( )

Appendix II
Usage Notes for the ITC SUN Systems

The directory "/usr/local/rpc" on all the machines contains the C header files and runtime routines for using RPC. Note that "/usr" on a diskless machine is a symbolic link to "/pub/usr" on its disk server.

In your client and server source programs include the files "rpc.h" and "ftp.h".

Compile thus:

```
cc -I/usr/local/rpc/include  client.c /usr/local/rpc/lib/librpc.a -o client.out
cc -I/usr/local/rpc/include  server.c /usr/local/rpc/lib/librpc.a -o server.out
```

The following external variables may be set for debugging on the client side:

```
extern int RPC_ClientDebugLevel; /* Default 0; higher values ==> verbose output */
extern int RPC_ClientHash; /* Default 0; set to 1 to see a '#' after each
                                  block of bulk-transfer */
```

The variables RPC – ServerDebugLevel and RPC – ServerHash perform a similar function on the server side.

References