18–546	Project 2 - Building an NFS Server	Assigned: 3/6/2002
Spring 2002	Electrical and Computer Engineering, CMU	Due: 3/27/2002

1 Introduction

In this project, you will build your own user-level NFS server. This project will expose you to the interface that a typical network file-server exports, and show you some of its drawbacks. In particular you will be building a NFS version 2 file server that runs as a stand-alone application. You will be provided with the remote procedure call (RPC) stubs, which will call into your code to perform the specific operations.

For this project you are expected to **work alone!**

2 Background NFS Version 2

The implementation of NFS version 2 began at Sun Microsystems in March 1984. Although NFSv2 is almost 20 years old, it is still one of the dominant network filesystem protocols in use. It has somewhat recently been superceded by NFSv3 which offers a number of improvements over the original protocol, and NFSv4 is currently in the standardization phase.

NFSv2 uses SUN RPC as its transport mechanism. A RPC is a message sent to a server with the expectation of a reply message (think of a procedure call being done remotely). To make a request the client needs to encode the arguments to the remote procedure call and transmit these arguments to the server. The client encodes the procedure's arguments according to the XDR specification. XDR specifies a common data format to be used between clients and servers. Once the server receives the RPC (message), it decodes the arguments, performs the procedure, encodes the reply arguments, and transmits them back to the client. In NFSv2, each reply argument contains a status field that must be set to indicate success or the type of error.

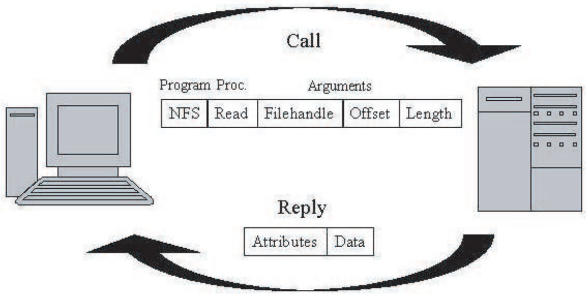

Figure 1: The call message identifies the program, the procedure within that program, and the parameters for the procedure. The reply conveys the procedure results back to the caller.

Figure 1 shows the typical communication path for a remote procedure call. The client marshals the arguments to the call, the server unmarshals them, performs the operation, marshals the reply, and finally transmits the reply back to the client.

2.1 NFSv2 Filesystem Model

The NFS protocol supports the common model of a hierarchically orgainzed filesystem comprised of directories, files, hard-links, and soft-links. NFS provides access to several different kinds of filesystem objects:

- *Files* are assumed to be uninterpreted byte sequences, which are opaque to the server. An NFS file may have multiple names or *Hard Links*, therefore it is necessary to maintain a link count for each file object.
- *Directories* are assumed to be lists of file names. Each directory can only have one name (hard links to directories are not allowed). The server is not responsible for resolving pathnames; this is done on the client.
- *Symbolic Links* are assumed to contain a text pathname that can be read and interpreted by the client. To the server, it contains an uninterpreted stream of bytes.
- *Special Files* appear to be supported by NFSv2, but in fact they are only place holders (usually holding device major and minor numbers) that refer to special files on the client.

2.2 NFSv2 Filehandles

NFSv2 uses *filehandles*, references to files or directories that are independent of the filename, to refer to files and directories in most procedures. An NFSv2 filehandle is a 32-byte structure containing an unique identifier for that filesystem object. The contents of a filehandle are opaque to the client. UNIX servers typically encode the inode number and filesystem identifier within the filehandle. Remember that a server must be able to translate a given file to a filehandle and vice-versa, because NFS uses stateless servers. NFSv2 clients expect filehandles to be persistent references to files (they will cache filehandles).

2.3 NFSv2 Procedures

Your main guide during this project is going to be the handout of Chapter 6: NFS Version 2 from the book "NFS Illustrated" by Brent Callaghan. This book goes into detail about the structures used, the status error codes, and descriptions of the procedure calls themselves.

The following is a list of procedures that you will be expected to implement and a brief description of their operation. For detailed information about their operation refer to the above mentioned handout.

- **NULL** – NULL Procedure, does nothing.

- **GETATTR** – Returns an objects attributes given its filehandle.
- **SETATTR** – Sets an objects attributes given its filehandle (watch out for some gotchas described in section 6.1 under sattr and in 6.2.3).
- **ROOT** – Unused, ignore.
- **LOOKUP** – Returns the filehandle for an object given a parent directory's filehandle and a file/dir name.
- **READLINK** – Returns the string stored in a file as the symlink name given the filehandle to the file.
- **READ** – Reads data from a file given a length, offset, and filehandle from which to read. Watch out for reads past the end of a file.
- **WRITECACHE** – Unused, ignore.
- **WRITE** – Writes data to a file given a length, offset, and filehandle. Watch out for writes in the middle of a file.
- **CREATE** – Creates a file given a filename, and a parent directory filehandle, returns a filehandle to that file.
- **REMOVE** – Removes a file given a parent directory filehandle and a filename.
- **RENAME** – Renames a file given a *to* directory filehandle, a *to* filename, a *from* directoy filehandle, and a *from* filename. The *to* and *from* directory filehandle may be the same.
- **LINK** – Create a hardlink to a file, given a *from* file filehandle, a *to* directory filehandle, and a *to* filename.
- **SYMLINK** – Creates a symbolic link, given a parent directory filehandle, a *from* filename, a path (opaque to the server), and attributes of the symbolic link.
- **MKDIR** – Creates a directory given a parent directory filehandle and a directory name, returns the created filehandle.
- **RMDIR** – Removes a directory given a parent directory filehandle and a directory name.
- **READDIR** – Returns a directory listing given a directory filehandle, a size of the results in bytes, and a cookie. A list of entries containing filenames, fileids (typically the inode number), and a cookie. The cookie is a reference to previous entry returned from a readdir. This is typically the offset of the directory entry.
- **STATFS** – Returns attributes of the filesystem that a given filehandle is located in.

2.4 Mount

Mount is a protocol that is used to mount or return the `root filehandle` of your filesystem. We have provided you with all the code, except for a single function, that handles the mount rpc. The function you must implement is located in `nfs_prot_svc.c` and is called *nfs_get_root*.

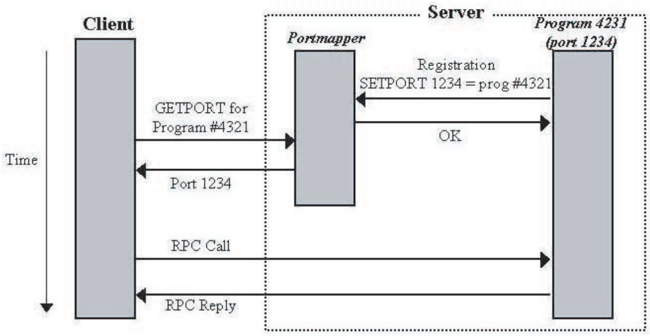

Figure 2: Clients use portmapper to discover the port on which an RPC service is listening. An RPC application registers its port with portmapper.

Figure 2 shows the flow of communication necessary for a client to locate the port on which the server is listening and to start communication to the server. Once the client knows the port number it can communicate directly with the server.

3 The Project

3.1 Overview

Your project is to write the code for an NFS server that will export a directory within the local file system as an NFS mount. The local directory that the NFS server is exporting becomes the `root` with respect to the NFS server and as such its filehandle should be returned on a mount call.

Since you are exporting a local directory, any NFS operations that you perform must be visible in the local directory. For example: if the NFS server receives a `CREATE` procedure call, the file must actually be created within the exported directory and must be visible from outside the NFS server.

The way in which you will be running your NFS server will be a little different from the way a regular NFS server is run. To run and mount your server you do not need to be root. A number of specific directories have been setup to be used as your export directory (the root of your server's filesystem) and as your mount directory (where you should mount your server) – more details are given in section 3.5. In typical operation, the mount command communicates with a process called `portmap` in order to locate the port on which the RPC service is running. In our environment, since we want to run multiple servers of the same type on a given machine, you will not be registering your server with portmap, and as such the port must be explicitly declared upon mounting the system. For your use, the mount command is specified in the file `/etc/fstab`, again directions are given in section 3.5 for mounting your server.

3.2 Source Code

We will provide you with a small number of files that implement the receiving, transmitting, and decoding of RPCs. You will only be required to modify **one** of these files: `nfs_prot_svc.c`.

We have also provided two stub files `mount.x` and `nfs_prot.x`. These files include the definitions for the `mount` and `NFS` RPCs. The make process runs `rpcgen` on the two `.x` files to generate a header file and two `.c` files for each of the `.x files`. The header files contain the structures you need to implement the procedure calls. The `.c` files contain the routines to encode/decode the arguments, and the routine to dispatch the procedure calls (you shouldn't have to interact with these files).

3.2.1 Provided Files

The full list of provided files is below. Become very familiar with them, especially the `nfs_prot.h` header file, which you will reference many times:

- `Makefile` – a Makefile to generate the stubs and compile the code.
- `nfs_main.c` – contains the `main` procedure that initializes the RPC library and is responsible for dispatching the RPCs to the generated RPC stubs.
- `mount.x` – Mount RPC proctocol definition. `rpcgen` generates three files:
  - `mount_dispatch.c` – contains a function for dispatching the mount procedure calls.
  - `mount_xdr.c` – contains the functions for encoding/decoding the mount RPC arguments.
  - `mount.h` – a header file containing the mount procedure argument structures, function prototypes, and mount constants.
- `nfs_prot.x` – NFS RPC protocol definition. `rpcgen` generates three files:
  - `nfs_prot_dispatch.c` – contains functions for dispatching the NFS procedure calls.
  - `nfs_prot_xdr.c` – contains functions for encoding/decoding the NFS RPC arguments.
  - `nfs_prot.h` – a header file containing the NFS procedure argument structures, function prototypes, and NFS constants.
- `nfs_prot_svc.c` – a template file that you will fill in with the code for the appropriate procedures.
- `mount_svc.c` – a file with the server definitions for the mount procedure calls.
- `nfs_fh_cache.cpp` – this file implements a filehandle to path cache. It is designed to make your life easier in resolving the path that a filehandle refers to (written in C++ for simplicity).
- `nfs_fh_cache.h` – 'C' header file prototyping filehandle cache functions.

3.2.2 Functions To Be Implemented

The following is a list of the functions that you must implement. A skeleton for each function exists in the file `nfs_prot_svc.c`.

- `nfsstat nfs_get_root (nfs_fh *fh)`
- `int nfsproc_getattr_2_svc(nfs_fh *argp, attrstat *result, struct svc_req *rqstp)`
- `int nfsproc_setattr_2_svc(sattrargs *argp, attrstat *result, struct svc_req *rqstp)`
- `int nfsproc_lookup_2_svc(diropargs *argp, diropres *result, struct svc_req *rqstp)`
- `int nfsproc_readlink_2_svc(nfs_fh *argp, readlinkres *result, struct svc_req *rqstp)`
- `int nfsproc_read_2_svc(readargs *argp, readres *result, struct svc_req *rqstp)`
- `int nfsproc_write_2_svc(writeargs *argp, attrstat *result, struct svc_req *rqstp)`
- `int nfsproc_create_2_svc(createargs *argp, diropres *result, struct svc_req *rqstp)`
- `int nfsproc_remove_2_svc(diropargs *argp, nfsstat *result, struct svc_req *rqstp)`
- `int nfsproc_rename_2_svc(renameargs *argp, nfsstat *result, struct svc_req *rqstp)`
- `int nfsproc_link_2_svc(linkargs *argp, nfsstat *result, struct svc_req *rqstp)`
- `int nfsproc_symlink_2_svc(symlinkargs *argp, nfsstat *result, struct svc_req *rqstp)`
- `int nfsproc_mkdir_2_svc(createargs *argp, diropres *result, struct svc_req *rqstp)`
- `int nfsproc_rmdir_2_svc(diropargs *argp, nfsstat *result, struct svc_req *rqstp)`
- `int nfsproc_readdir_2_svc(readdirargs *argp, readdirres *result, struct svc_req *rqstp)`
- `int nfsproc_statfs_2_svc(nfs_fh *argp, statfsres *result, struct svc_req *rqstp)`

3.2.3 Location Of Source Code

A copy of the source code will be located on each of the development machines in the directory `/home/gg2k/nfs_server.tgz`. Copy this file to your home directory, untar it and you should be ready to go...

```
% cp /home/gg2k/nfs_server.tgz .
% tar xvfz nfs_server.tgz
% cd proj2
% make    (Of course, you first need to fill in the functions...)
```

3.3 Examples and Hints

3.3.1 A CREATE Function Example

The following is an example, written in pseudo code, of what the create function might look like. Note that it performs operations on the local filesystem.

```
int
nfsproc_create_2_svc(createargs *argp, diropres *result, struct svc_req *rqstp)
{
  <Lookup argp->where.dir filehandle in filehandle cache to get pathname>

  <Change to directory specified by argp->where.dir filehandle>

  <Create local file called argp->where.name>

  <Set attributes of newly created file using attrs from argp->attributes>

  <Set the result filehandle for the newly created file in
   result->diropres_u.diropres.file and store it in the filehandle cache>

  <Set the attributes of the newly created file in
   result->diropres_u.diropres.attributes>

  <Set the result->status return code>

  return TRUE;
}
```

As you can see, a pointer to the arguments and a pointer to the result is passed into the create funtion. The third argument the `struct svc_req` is a RPC structure that contains authentication information (but for our purposes we can ignore it).

You can also see that you should familiarize yourself with the common filesystem calls such as: `open`, `stat`, `readdir`, `chmod`, `read`, `write`, `unlink`, `symlink`, `rename`, etc. as they will be used frequently.

3.3.2 Hints – Things To Watch Out For

- **Filehandle Caching**

  One of the challenges you will run into is that of converting a filehandle to a path name. To make things easier for you we have provided you with some functions that implement a *filehandle cache*. We highly recommend that you cache all the filehandles returned by your implementation of CREATE, MKDIR, and SYMLINK.

- **Allocating Memory**

It is important to note that any structures within the result structure which are pointers (pathnames, filenames, directory entries – in readdir, returned data – readlink and read) must first be allocated with `malloc` before they can be filled in. Don't worry about unallocating the space, the RPC library will do that for you. All of the functions that you implement that need explicit memory allocation will be commented as such.

- **Attributes**

  Make sure that the file type is set in the mode bits whenever a `SETATTR` is performed (not only in `SETATTR`, but also in `CREATE`, `MKDIR` and `SYMLINK`). It may be the case that the client does not set the type in the mode bits. If this is the case it may be necessary to perform a `GETATTR` to get the mode bits, so that they can be set correctly in the `SETATTR`.

- **READDIR**

  - `count` In the `READDIR` call you are given a `count` value specifying the max number of bytes that should be returned. Due to RPC overheads it may be necessary to divide this amount by 4 to be safe. So you would be returning a quarter of the amount specified. This should be fine as long as you set the last entry's `nextentry` field to NULL, and properly return end-of-file (EOF).
  - `cookie` The cookie value should be cast as an integer whenever it is accessed. It must also be in network format (use `htonl`) when returned, and conversley the cookie you receive will be in network format and should be converted to host format (use `ntohl`). Note, this is the only place that will require you to do any explicit data marshalling, the XDR routines will do the rest for you transparently. Again, an example has been added to the code.

- **UIDs and GIDs**

  In a real NFS server you would see a lot of traffic from a lot of different users, and the server would be running as root. Since, in the case of this project, the NFS server is running as a user – you – the operations it has to perform will also only come from a single user – you. This means that you don't have to worry about checking or changing permissions based on UIDs or GIDs.

3.4 Machines and Accounts

We have set aside a number of development machines: `nc55.pdl.cmu.edu`, `nc68.pdl.cmu.edu`, and `nc69.pdl.cmu.edu` for your use. These machines have been set up with your accounts. You will be receiving, or have received an email message telling you your login id, your password, and to which machine you have been assigned.

Since these are development machines they are behind a firewall and cannot be accessed from outside of the 128.2.134 subnet. So we have set up an intermediate machine, `blossom.pdl.cmu.edu` to which you must first login. Your login id to blossom will be the same as your login id to the development machines, but your password will be your ECE AFS password. If you did not have an ECE account previously, we will send you email with your initial password (change it!) and login id. Once you have logged into blossom, you can use ssh to log into the development machine to which you have been assigned.

For example, the start of your work sesssion may look something like:

```
[local] % ssh <login_id>@blossom.pdl.cmu.edu
[blossom] % ssh <login_id>@nc55.pdl.cmu.edu
[nc55] % ...
```

3.5 Running Your Server

For this project, you do not need root access to run or mount your nfs servers. A mount point for each user has been created under the directory /ux1/<login_id>_mnt. The filesystem that you export should be under /ux1/<login_id>_export. You probably want to ensure that your export directory is empty before you start.

To mount your nfs server, use the command:

```
% mount /ux1/<login_id>_mnt
```

This will cause mount to issue an RPC to your NFS server expecting the filehandle of the root to be returned.

To run your nfs server after you have compiled it, use the command:

```
% ./nfs_server -p <port #>
```

The port number that you must use will be assigned to you via email. You **must** use this port number in order to mount your server. If you should forget it, you can check the /etc/fstab file for the line corresponding to the mount point you have been assigned. For example, for the following entry the port number is 1234 and the user is dsf.

```
nc55:/  /ux1/dsf_mnt  nfs  user,noexec,soft,intr,nfsvers=2,port=1234,mountport=1234  0  0
```

3.5.1 Testing and Benchmarks

The best way to test your system is to start out with small operations, such as touching a file, or creating a directory, and them move on to larger more complicated ones such as readdir.

At a later stage we will make available a number of benchmarking or other test programs which you can run over your NFS server.

4 What To Turn In

- **Your Code** – Copy the final version of your code to a directory named final within your home directory on the development machine to which you have been assigned. Access to the development machines will be revoked at the end of the due day.

- **A Small Writeup** – You should turn into class, or you may leave the writeup in the `final` directory along with your code, but only if it is in **postscript** format.

  (a) A description of how you mapped files to filehandles.

  (b) The counts for each type of NFS operation called while running the benchmarks.

  (c) An explanation of which operation is called most frequently, and a description of two possible ways that you could cut down on the impact of the frequency of this operation. (This may include altering the protocol slightly)

  (d) A description of any features of the protocol that you thought were awkward and that you would change.