6.824 - Fall 2002

6.824 Lab 4: Semantic File System

Introduction

The Semantic File System (proposed by Gifford in [1]) provides access to a file system by fulfilling queries for file attributes rather than organizing files in a hierarchy of directories. A typical use of the Semantic File System as originally proposed might look as follows:

```
% ls /sfs/owner:/smith
bio.txt paper.tex prop.tex
```

In this example the user has requested all documents owned by user smith. This is done by listing the *virtual* directory `owner:/smith`. The system creates this directory "on the fly" and arranges to populate it with the files which match the search criterion (i.e. are owned by smith). In this lab you will implement a scaled-down version of the semantic file system using an SFS user level server similar to the toolkit described in this paper [2]. In this document "SFS" will refer to the self-certifying file system; we will refer to the semantic file system by its full name. The final product of this lab will be a user-level file server that enables semantic file system-like access to a local directory. A working daemon might function as follows:

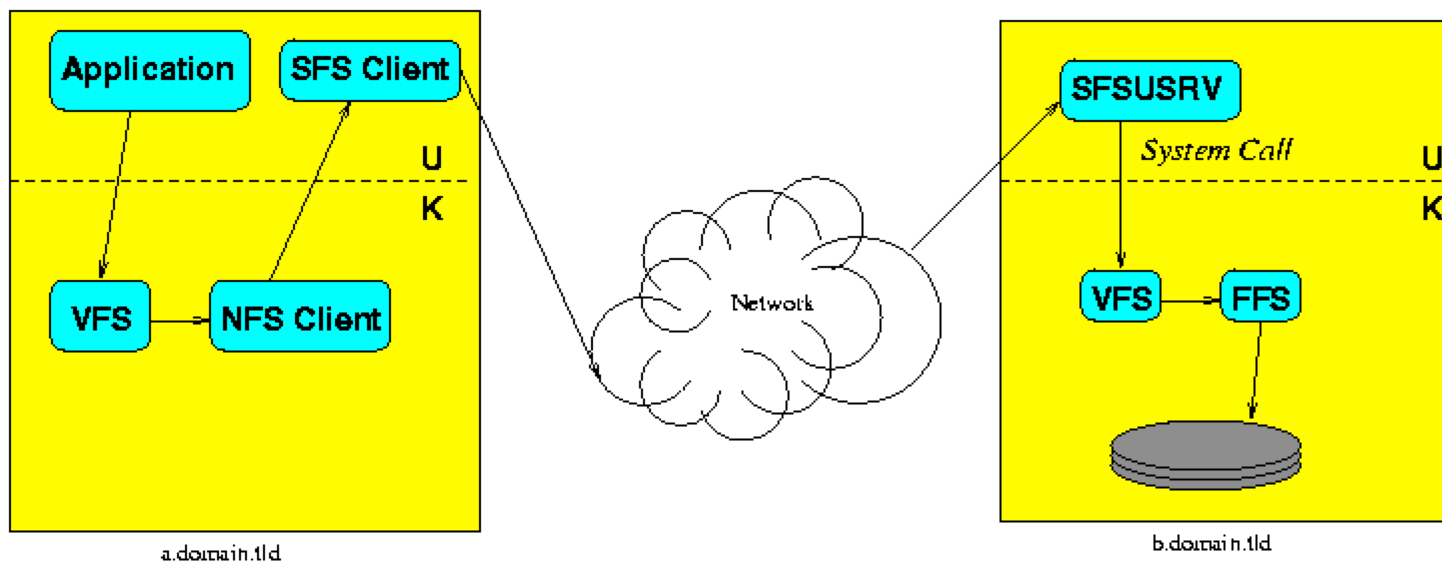
```
pain% ./sfsusrv -s -p 0 -f ./sfsusrv_config
sfsusrv: version 0.6, pid 30555
sfsusrv: No sfssd detected, running in standalone mode.
sfsusrv: Now exporting directory: /tmp/export-jsr
sfsusrv: serving /sfs/4391@pain.lcs.mit.edu:48ucs56f4q5atfm4hqfwkmx6kwqu6497


pain% ls /tmp/export-jsr
README          foobar.c

suffering% cd /sfs/4391@pain.lcs.mit.edu:48ucs56f4q5atfm4hqfwkmx6kwqu6497
suffering% ls
README          foobar.c
suffering% cd :extension=c
suffering% ls
foobar.c
```

```
suffering% cat foobar.c
...
```

In this example, I have listed the files stored in `/tmp/export-jsr` (a directory on pain's local disk) with a `.c` extension by accessing a specially named semantic directory (`:extension=.c`) under the SFS mount point for that file system on a remote machine. Note that the ls is done from suffering: because of a risk of deadlock, it is not possible to access files on the local machine via SFS. Also note that you can only access the /sfs/.... directory from suffering, and hence you need to use another computer, such as pain, to run sfsusrv.

The SFSUSRV file server

You will implement your semantic file system by modifying an existing user-space SFS server (sfsusrv). sfsusrv reads SFS RPCs from a network connection, and executes them using ordinary UNIX system calls such as open() and read(). The following shows how sfsusrv fits into the rest of the NFS/SFS system:



Your job will be to extend sfsusrv to answer semantic file system queries as well as ordinary file accesses.

Building the sfsusrv software

To get started with the software, you should unpack the existing sfsusrv from `/home/6824/labs/sfsusrv.tar.gz`.. This project uses a static makefile so configuration is not necessary. However, this makefile will not function correctly on machines other than the class machines.

```
% cd
% tar xzvf /home/6824/labs/sfsusrv.tar.gz
% cd sfsusrv
% gmake
```

Running sfsusrv

Before you run sfsusrv, you need to generate a public/private key pair for use by the server. Run the following command:

```
% sfskey gen -KP sfs_host_key
 Creating new key for sfs_host_key.
      Key Name: jsr@pain.lcs.mit.edu Press return
```

Note: seeing the following warning messages is normal when using sfskey on pain:

```
/var/sfs/sockets/agent.sock: No such file or directory
sfskey: sfscd not running, limiting sources of entropy
```

You'll want to run sfsusrv with these arguments:

- **-s** Run in stand-alone mode, without the support of a 'master' SFS server. You should always use this option when working on this lab.
- **-p *port*** Listen on the specified port for connections. If port is zero, sfsusrv will bind to an unused port, and tell you what port it got. You probably want to use the zero option to avoid port conflicts with other students.
- **-f *config-file*** Read additional options from the specified configuration file. The important options in the config file are the directory to export and the location of the server's public key (which you created using sfskey above). Here's an example:

  ```
  export /tmp/export-jsr
  keyfile sfs_host_key
  ```

To export a directory from pain using sfsusrv, do the following:

```
pain% mkdir /tmp/export-$USER
pain% echo hello > /tmp/export-$USER/test.file
pain% echo export /tmp/export-$USER > cfg
```

```
pain% echo keyfile sfs_host_key >> cfg
pain% ./sfsusrv -s -p 0 -f ./cfg
sfsusrv: version 0.6, pid 30555
sfsusrv: No sfssd detected, running in standalone mode.
sfsusrv: Now exporting directory: /tmp/export-jsr
sfsusrv: serving /sfs/4391@pain.lcs.mit.edu:48ucs56f4q5atfm4hqfwkmx6kwqu6497
```

The last line that sfsusrv prints is the path name under which your exported directory (/tmp/export-$USER on pain) will appear on SFS client machines. Your path will be different from the one printed above. The 4391 is the port number that sfsusrv is listening to. You can log into suffering and look at your exported files (but change the `/sfs/...` pathname to the one that your sfsusrv printed):

```
suffering%  ls /sfs/4391@pain.lcs.mit.edu:48ucs56f4q5atfm4hqfwkmx6kwqu6497/
test.file
```

You can learn more about how SFS works, and about why `/sfs/...` pathnames look the way they do, by reading the "SFS paper".

Tracing RPCs

Once you've gotten sfsusrv running, you can use it to trace NFS RPCs. Kill your existing sfsusrv with control-C, and start a new one with the ASRV_TRACE environment variable set to 10:

```
pain% env ASRV_TRACE=10 ./sfsusrv -s -p 0 -f ./cfg
```

Now, on suffering, browse the exported file system. You'll have to use the `/sfs/...` pathname printed out by the current instance of sfsusrv, since the port number will change each time. As you browse, the server on pain will print out a complete trace of all NFS requests it receives. (Large structures may be truncated; if this is ever a problem, try higher values than 10.)

Now capture the output in a file:

```
pain% env ASRV_TRACE=10 ./sfsusrv -s -p 0 -f ./cfg |& tee nfs.trace
```

If you're using a Bourne-like shell, try this instead:

```
pain% env ASRV_TRACE=10 ./sfsusrv -s -p 0 -f ./cfg 2>&1 | tee nfs.trace
```

After setting up sfsusrv to trace NFS traffic, run the following commands (substitutiong the correct self-certifying pathname):

```
suffering% cd /sfs/...
suffering% rm junk
rm: junk: No such file or directory
suffering% echo hello > junk
suffering% cat junk
hello
suffering% cat junk
hello
```

Now stop sfsusrv, and look at the RPCs in the `nfs.trace` file. You can see a summary of the RPCs using `grep ' serve ' nfs.trace`, though in order to see arguments and return values you'll have to look at the whole file. Answering the following questions will help you understand how NFS and sfsusrv interact.

- Which RPCs correspond to the creation of junk?
- Which to the first cat?
- Which to the second cat?
- Explain any differences between the RPCs caused by the two cat commands.

A Semantic File System

Now that you understand how the sfsusrv software works you can modify it to process semantic file system queries. Your sfsusrv should make it look as if there are directories corresponding to queries, but should calculate their contents on the fly, since those directories don't actually exist on the server machine.

You are not required to implement the semantic file system as described in Gifford's paper. You only have to serve queries of the following form:

```
:extension=ext
```

When you see a reference to a file whose name looks like this, you should produce a virtual directory containing all files in the original directory whose names end with `ext`. Note that because your program is an NFS server, unlike the implementation described in Gifford's paper, you do not need to create symbolic links to the "real" files.

All of the files that are eligible to satisfy queries will be located in the current working directory; you are not responsible for indexing an entire file system. For example,

```
ls /sfs/.../foo-dir/:extension=.c/
```

should list all of the files that end in .c in directory foo-dir.

Getting Started

If you aren't extremely comfortable with the NFS specification, refer to the RFC. sfsusrv implements NFS version 3. Another great resource is NFS Illustrated by Brent Callaghan (published by Addison-Wesley).

You should use sfsusrv as a starting point to implementing your daemon. You'll "override" the way sfsusrv handles a subset of the NFS3 RPC calls to provide for the semantic queries. You'll need to modify the handling of at least the following RPCs:

- **NFSPROC3_LOOKUP** -- You'll need to modify lookup handle requests to lookup a virtual directory and to lookup files in virtual directories.
- **NFSPROC3_READDIR** -- This RPC should be modified to return a list of matching files in response to a query. You should be able to simply modify the existing readdir to filter out the non-matching names.
- **NFSPROC3_CREATE, NFSPROC3_REMOVE** -- These RPCs (unlike, for instance, READ and WRITE) take as arguments a directory file handle and a filename instead of a file handle. As a result they will need to be handled specially.
- **NFSPROC3_GETATTR / NFSPROC3_ACCESS** -- GETATTR, ACCESS and possibly other RPCs will have to be modified to handle virtual directories and their files.

Each RPC is implemented in a method named "nfs3_RPCNAME" in the file client.C. client.C contains code which handles NFS calls. filesrv.C contains some utility functions and a cache of file descriptors associated with recently accessed files.

You may also find that you need to keep some additional state about the status of a query. The fh_entry structure defined in filesrv.h is a good place to add this additional state.

Your server is free to access the local file system via standard POSIX system calls (read, write, readdir, etc). Because your program is doing disk I/O you may not access it via SFS locally because of the danger of deadlock.

It is important to remember when designing this experiment that NFS filehandles are opaque data structures. You are free to form the file handles you return in any way you see fit (as long as they are 64 bytes or shorter).

Requirements

Your daemon should

- support the following semantic query type:

:extension=*ext*

- support the listing of virtual directories
- support read, write, rename, create and remove operations on files in virtual directories
- transparently pass all non-semantic file operations to the remote SFS server.

Testing

A test script has been provided: it is located in `/home/6824/labs/sfsusrv`. The script takes a single argument: the self-certifying pathname to your server. Since the tester accesses files via SFS it must be run on a machine other than the one your server is running on. For example, if you ran your server on pain, you might run the following on suffering:

```
suffering% /home/6824/labs/sfsusrv/sfs-test.pl /sfs/2668@pain.lcs.mit.edu:5drekhdhsvme4s5rnkzi8imi8dpwygmd/
Setting up test files in /sfs/2668@pain.lcs.mit.edu:5drekhdhsvme4s5rnkzi8imi8dpwygmd
Testing basic sfsusrv behavior...
    Testing ls..passed
    Testing write/read...passed
Testing semantic behavior...
    Testing ls...passed
    Testing write/read...passed
    Testing create...passed
    Testing remove...passed
    Testing rename...passed
Testing semantic behavior in a subdirectory...
    Testing ls...passed
    Testing write/read...passed
    Testing create...passed
    Testing remove...passed
    Testing rename...passed
```

The tests are in three phases:

- **basic sfsusrv behavior** this phase just checks to see if sfsusrv still functions. An unmodified server should pass this test.
- **semantic behavior** runs the listed operations in a semantic directory at the root level (e.g. /sfs/1234@pain:.../:extension=a/).
- **semantic behavior in a subdirectory** as above, but in a subdirectory (e.g. /sfs/1234@pain:.../dir/:extension=a/).

Feel free to browse the source of the tester to gain a fuller understanding of what it is testing.

Submitting your lab

To submit your lab place a tarball in `~/handin/sfs-handin.tar.gz` that contains source files and a makefile. Do not include object files or binaries please. To make the tarball run the following commands:

```
% cd
% cd sfsusrv
% gmake clean
% cd
% tar czvf ~/handin/sfs-handin.tar.gz sfsusrv
```

The lab is due before class on Thursday, October 3.

Filehandle FAQ

Q: What is an NFS filehandle?

A: An NFSv3 filehandle is a 64-byte opaque data structure used to identify a 'file'. The fact that the filehandle is "opaque" means that the server generates the filehandle with whatever internal structure it wishes and the client never interprets the structure of the filehandle. The client may compare the filehandle to others or return it to the server, but it has no need to understand how it was constructed.

Q: What can/should I put in an NFS filehandle?

A: Anything you like, as long as you follow a few guidelines:

- file handles must be less than 64 bytes in length

- distinct files/directories should have distinct file handles. You might ask what a 'file' means in the context of an assignment that asks you to create the appearance of virtual directories. To play it safe, make sure that any two files with different inodes have different filehandles and that any two queries that are for a different extension or reference a different directory have different file handles. This rule means that, for instance, you shouldn't assign a query directory the same filehandle as the file system directory it references (i.e. /sfs/.../dir/:extension=foo/ should not have the same filehandle as /sfs/.../dir/). This also means you shouldn't, for instance, return zero for the filehandle of every file.

- filehandles are consistent. The same file (subject to the definition above) should always have the same filehandle. Don't make filehandles unique by incrementing a counter or repeatedly using a random number generator.

Q: Ok, but how do I actually get my filehandle into the NFS reply?

A: The nfs_fh3 structure defines the representation of a filehandle. It supports two important operations (for this descriptions assume we have declared nfs3_fh *fh):

- fh->data.setsize (n) -- allocate n bytes of space to hold this filehandle.
- fh->data.base () -- return a pointer to the allocated data. You can then use memcpy, for instance, to copy data onto this base pointer.

Q: why do I see

```
/var/sfs/sockets/agent.sock: No such file or directory
sfskey: sfscd not running, limiting sources of entropy
```

when I use sfskey on pain?

A: That is because we are not using SFS (ie the /sfs directory doesn't work) on pain. Do not be alarmed, sfskey will still work well enough for our purposes.

References

[1] *Semantic File Systems* David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. 13th ACM Symposium on Operating Systems Principles, October 1991

[2] *A Toolkit for User Level File Systems*, David Mazeries, Usenix 2001.