

srfs kernel module

Nir Tzachar

September 29, 2003

1 Introduction

1.1 A distributed file system

A distributed file system should appear to the user as a traditional file system. The user can create files, delete them, open and read files, write data etc. . . all without need to concern herself with the underlining implementation. Under the hood, the distributed file system usually resides on several servers, which provide load balancing and fault tolerance. There two extremes of distributed file systems schemes:

1. Total Replication - each server contains a replica of all the data. (for example - afs, coda ..)
2. Total Distribution - each file resides on a single server, and the collection of all files on all servers comprise the file system.

scheme 1. provides more fault tolerance and load balancing, but demand much space. scheme 2. is conservative regarding space, but is not tolerant to faults.

Usually, distributed file systems choose the middle way, combining the two to achieve the best of all worlds. (i.e; most peer-to-peer file systems like kazza, napster etc.)

There are many problems when implementing such a file system, amongst them are data synchronization, file locking, and coherence.

1.2 Self stabilization

A self stabilizing system is a system that can automatically recover following the occurrence of (transient) faults. The idea is to design a system which can be started in an arbitrary state and still converge to a desired behavior.

1.3 Object Store

An object store raises the level of abstraction presented by a storage control unit (a hard disk, for example.) from an array of 512 byte blocks to a collection of objects. The object store provides “fine-grain”, object-level security, improved scalability by localizing space management, and improved management.

One might ask how this relates to (our) file system? well, for many years, file system designers had to deal mainly with data storage management at the software level (usually in the file system implementation) which is cumbersome, slow and adds a noticeable overhead to system performance - hence you get many different file systems, all try to manage the “array of 512 byte blocks” in a better way (such file systems are: ext2, ext3, FAT, NTFS, reiserfs4 etc..). An object store “makes life easier” for the file system designer, which can now concentrate on the higher levels of the file system such as hierarchical layout, distribution, journaling and others.

Needless to say, such storage devices are the bleeding edge of technology, and are not yet widely available. Later, I will describe how we simulated such a device, on top of an existing file system.

1.4 The Linux VFS layer

The Virtual File System (otherwise known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs. It also provides an abstraction within the kernel which allows different filesystem implementations to co-exist. [7]

1.5 Kernel modules

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality [9]

2 Project’s definition

Srfs is a Self stabilizing, replicating file system (by ”replicating” we mean that the system’s data is replicated over all servers). This file system design can take several servers (containing any data, synchronized or not), and combine them to a single file system which WILL be coherent eventually. This part of the file system is implemented as a Java program.

My goal was to create a transparent interface between the user and sdfs. I achieved this goal by building a file system for the linux operating system, implemented as a kernel module and interfacing the VFS.

3 The implementation

3.1 overview

Modern operating system distinguish between user space and kernel space. User space is the environment provided by the kernel for process execution. This includes (amongst other things) its address space - which is maintained by the kernel - I/O operations, etc ... Kernel space is the environment in which the kernel executes, which is maintained by the kernel itself. No use of external libraries is possible (i.e; the kernel has to implement its own encryption algorithms). User space processes, wishing services from the kernel, do so by using the system calls interface (eventually, all interactions between user space processes and the kernel are achieved by means of interrupts and traps).

Sdfs, as a whole system, composes of 2 main parts: a kernel module and a user space daemon. The kernel module is, essentially, a file system implementation (using the object storage paradigm) which provides hooks to allow a user space daemon to interact with it and influence its behavior. This file system is running in kernel space, and is actually a part of the kernel itself. The user space daemon, which was implemented by Ronen Kat, is responsible for the global aspects of the system (such as self stabilization, replication etc...).

Since these two parts operate in different “worlds” (kernel/user space), a way of communication between them must be introduced. The most appropriate in this case is via a character device, which serves as a bi-directional channel between user and kernel space.

3.2 The object store

As mentioned before, no object store device is available (yet) as a commodity, so I had to simulate one. I used the Stackable file system idea to this end: take a regular linux file system (ext2, ext3, nfs ..), and use it to provide low level disk management. Use the lower file system’s notion of files to create objects, denoted by their object id (which is the file name). To clarify things, a typical object store will look like this:

the lower file system (which is hidden from everyone, even the kernel itself)

```
--\  
--|-- 0  
--|-- 1  
--|-- 22  
--|-- 33  
--|-- 44
```

the upper file system

```
--\  
--|-- file1 (corresponds to object id 22)  
--|-- directory1 (corresponds to object id 33)  
--|  |  
--|  |--file2 (corresponds to object id 44)
```

(object id 0 stores the superblock and inode information.).

3.2.1 interfacing with the vfs

A basic file system in the linux world consists of inodes, dentries and files.

- *inode* – our old and dear friend, the inode. The inode is the basic element of every filesystem in the unix world. Its purpose is to hold information about data location and attributes, such as creation time, ownership, etc. . .
- *dentry* – Dentries are used to speed filename lookups. Each filename element (in /usr , / is the first element, usr is the second) has a dentry associated with it, and filename resolution is performed according the the dentry cache. Furthermore, each dentry points to its relevant inode.
- *file* – a file is

A filesystem, wanting to register itself in some linux kernel, need to supply the vfs a presentation of these objects, and the relevant functions to operate on them. This is how the kernel achieves a kind of OO design (and interface implementation).

So, my object store had to implement these data types, which I did as follows:

- *inode* – Each inode on the object store has a pointer to a lower object, which is actually an opened file on the lower filesystem. All meta data is handled by the object store.
- *dentry* – Here, nothing special has been done, except from meeting the vfs requirements.
- *file* – Same as above.

This layout helps me achieve the desired behavior, ie. when a user writes data to a file in the object store, the data gets propagated to the appropriate object on the lower filesystem (via the `file->dentry->inode->object` pointers).

3.2.2 the “hooks” interface

All major functions of the object store contain hooks, which allow a user level program to influence its behavior. These hooks are in the form of downcalls, from the kernel to the user program, requesting or announcing events. The relevant kernel function which issued the downcall, will sleep (by this, i mean the user process who performed the system call will sleep) and wait for an answer. Once an answer is received, the function acts accordingly.

3.3 the character device

A character (char) device is one that can be accessed as a stream of bytes (like a file). The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. [1]

Srfs’s char device is actually a whole subsystem, which consists of several parts:

- *the data stream* - The actual data stream was constructed, to provide the communication infrastructure.
- *parser* - A parser kernel thread is created, to handle communications from user space and create the appropriate “job” objects (which can be replies or requests).
- *worker* - A worker kernel thread is also created, whose job is to execute the requests from the daemon (ie. requests for object names, creation of files etc. . .). Requests are in the form of “job” objects, placed in his queue by the parser thread.
- *replier* - A replier kernel thread is also created, whose job is to handle replies from the daemon. Replies are in the form of “job” objects, placed in his queue by the parser thread.

It may seem a little complicated, but 2 simple examples will help clarify things. Lets look at the following execution paths, first a request from the caching daemon.

Lets say a signature decision must be made on file X. On each server, the local caching daemon will issue a request, via the char device, for info on file X. The parser thread will parse the request, and place an appropriate “job” object on the worker thread’s queue. The worker thread, in its turn, will retrieve the information, and send it back to the caching daemon. Now, lets look at the opposite direction: A user tried to open file Y for writing. The user program issued the `sys_open` system call, which in turn, activated our file open method.

In the file open method of srfs, there is a hook for the caching daemon, so a global lock on the file can be achieved. Now, an “open file” request will be sent to the caching daemon, and the user process will sleep till an answer is received (or a time out has occurred). The caching daemon will decide on an answer (consulting with its peers...), and send it. The parser thread will recognize the response, and place it in the replier queue, which in turn will wake up the sleeping user process and deliver the reply to it.

A operation manual

A.1 system requirements

- kernel 2.4.18 and upwards (but not 2.5, 2.6 yet)
- devfs support
- gcc version 2.95 and upwards
- gnu make
- a victim file system (for the object store)

A.2 compilation

The distribution includes , in addition to the file system module, several tool to create the basic objects of the object store. The first step, is to edit the Makefile, and point the “linux” variable to your kernel tree. You can then make the module, and the tools.

A.3 operation

1. Insert the module with insmod. The module takes two parameters:
 - debug_level - an int, ranges from 0(lowest) to 9(highest).
 - communicate - 0 to disable communications, 1 to enable.
2. create a file system to be used as the object store (will be referenced as the \$object_store_dir).
3. create the srfs mount point (will be referenced as the \$srfs_dir).
4. create the object store file system - use “mksrfs \$object_store_dir”.
5. mount srfs - “mount -t srfs -o \$object_store_dir none \$srfs_dir”.
6. have fun.

References

- [1] Alessandro Rubini , Jonathan Corbet Linux Device Drivers, 2nd Edition June 2001
- [2] Daniel P. Bovet, Marco Cesati Understanding the Linux Kernel, 2nd Edition Daniel P. Bovet, Marco Cesati 2nd Edition December 2002
- [3] E. Zadok. FiST: A File System Component Compiler. PhD thesis, published as Technical Report CUCS-033-97 (Ph.D. Thesis Proposal). Computer Science Department, Columbia University, 27 April 1997.
- [4] Michael Beck, Harald Bohme, Mirko Dziadzka, Robert Magnus, et al. Linux Kernel Programming, 3rd edition, Addison-Wesley, August 2002
- [5] Moshe Bar, Linux File Systems, McGraw-Hill, New Jersey, 2002
- [6] Ohad Rodeh and Avi Teperman, "ZFS - A scalable distributed file system using object disks". 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03), April 2003, pp. 207-218.
- [7] Richard Gooch, Overview of the Virtual File System kernel documentation, Documentation/filesystems/vfs.txt
- [8] Shlomi Dolev and Ronen I. Kat, "Self stabilizing distributed le system", 21st IEEE Symposium on Reliable Distributed Systems, Workshop on Self-Repairing and Self-Configurable Distributed Systems, October 2002, pp. 384-379.
- [9] The Linux Kernel Module Programming Guide, <http://www.faqs.org/docs/kernel/>