RFS: A Mobile-Transparent File System for the Rover Toolkit

Anthony D. Joseph, George M. Candea, and M. Frans Kaashoek
M.I.T. Laboratory for Computer Science, Cambridge, MA 02139, U.S.A.
{adj, candea, kaashoek}@lcs.mit.edu

The mobile computing environment presents application designers with a unique set of communication and data integrity constraints that are absent in traditional distributed computing settings. These constraints make it difficult to use existing, mobile-transparent applications and complicates the development of new mobile-aware applications – applications that take advantage of information about the mobile environment to offer users better interactive performance and better utilization of network bandwidth. The difficulty arises mainly because an *incremental* port is usually not possible, due to the need to completely shift all of the application's data storage from one model (a file system) to a model that is better suited for disconnected operation (an object repository or database). To make the porting process easier for application designers, we are adding a file system proxy, the *Rover File System* (RFS), to the Rover toolkit's object-based model [1, 2]. This will allow the Rover toolkit to support both a file model and an object model for mobile applications.

Adding RFS to the toolkit raises a number of issues relating to file caching, prefetching, and conflict detection and resolution. Many of these issues have already been addressed to some extent by the Coda file system [3], however, we are also addressing the issues associated with integrating a file system model with an object-based model. Specifically, how to map file system objects (inodes, directories, and files) onto Rover objects, cluster objects for prefetching, use Rover's conflict detection and resolution model, and partially replicate files.

RFS, as a part of the Rover toolkit, offers application designers a simple alternative: those portions of an application's state that are difficult to port, that change infrequently, or exhibit update conflicts that are easy to detect and resolve can rely on RFS for storage. On the other hand, application state that is sensitive to bandwidth and latency, or where conflicts may be complicated to detect or handle, can use the full functionality of Rover's object management support as well as conflict resolution and detection. We expect this to significantly ease the process of adding mobile-awareness to applications.

RFS consists of two components: a user-level installable file system located on the client and a remote file system proxy running on a Rover server. RFS uses Rover's reliable, queued communication mechanism with automatic message compression and batching. The RFS client is composed of a small kernel module and the RFS process, which runs as a Rover client application. RFS makes use of two caches: a shared memory/disk cache, managed by the Rover Access Manager, and an in-memory cache managed directly by RFS. The latter cache is used to minimize the amount of communication between the RFS client and the Access Manager.

Our prototype implementation supports disconnected operation by caching remote inodes, files, and directories; however, it is currently limited to read-only access. We decided to start with a simple read-only implementation so we could immediately start analyzing performance and experimenting with various prefetching and caching strategies.

We have some initial performance numbers for RFS versus the Network File System version 2.3 (NFS). Our experimental environment consisted of an IBM ThinkPad 560 (120-MHz Pentium, 40 MB RAM, Linux 2.0.30) client and a 120-MHz Intel Pentium server with 32 MB RAM running Linux 2.0.28. The machines were connected via a 128 Kbit/s ISDN link. Performance measurements were made with a remote file system containing 32 MB in 1289 files, 170 directories, and 14 symbolic links. Both the Rover client and server were configured to use asynchronous flushing of queued communication log buffers. Each experiment was first performed with all caches (RFS client, Rover Access Manager, NFS client, and disk buffer) empty. The same experiment was then immediately repeated to measure hot cache performance.

This work was supported in part by the Defense Advanced Research Projects Agency under contract DABT63-95-C-005, by an NSF National Young Investigator Award, and by grants from IBM, Intel, and AT&T.

Cache State	File System	Seq Dir	Seq Dir Rand File	Sequential File ASCII	Sequential File Binary
Cold	RFS	69	230	8	73
	NFS	57	430	200	200
Hot	RFS	3	21	1	1
	NFS	57	93	0.3	0.3

Table 1: Read Latencies for Rover File System (RFS) and Network File System (NFS). Times are in seconds.

Using the UNIX `find` command, we measured sequential directory read, sequential directory read & random file read, and ASCII & binary sequential file read performance. The results (see Table 1) show that, overall, RFS outperforms NFS over slower networks. With an empty cache, RFS has a higher latency for sequential directory reads, but offers up to 25x better performance than NFS in the other categories. With a hot cache, relative performance of sequential directory reads is improved (19x better than NFS), but relative performance of sequential file reads is less than that of NFS. The positive results can be explained by Rover's automatic data compression and the caching policies employed by Rover and RFS. Both techniques are designed for the high-latency, low-bandwidth communication found in mobile environments. The negative results are due to the overheads associated with the user-level RFS and Rover implementations and with the data object transport protocol over TCP. In contrast, the NFS client resides entirely in the kernel and uses UDP. These results are the consequences of our design decisions that favored ease of development and portability over performance. RFS was written in just a few person-weeks.

We plan to extend the file system to provide full support for both read and write, as well as prefetching, hoarding, and disconnected operation with lease-based concurrency control and server callbacks. We expect to address many of the issues raised by the Coda file system and others, while further tuning RFS performance.

The Rover File System will provide the final component of a complete development environment. Using RFS, application writers will be able to gradually and incrementally port existing mobile-transparent applications to a mobile-aware environment. We believe this will be an important incentive for migrating to a mobile computing environment.

References

1. A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, 1995.
2. A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.
3. L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, 1995.