USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

Proceedings of the FREENIX Track:
1999 USENIX Annual Technical Conference

Monterey, California, USA, June 6–11, 1999

Porting the Coda File System to Windows

Peter J. Braam
Carnegie Mellon University

Michael J. Callahan
The Roda Group, Inc.

M. Satyanarayanan and Marc Schnieder
Carnegie Mellon University

© 1999 by The USENIX Association
All Rights Reserved

Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Porting the Coda File System to Windows

Peter J. Braam
Carnegie Mellon University
Michael J. Callahan
The Roda Group, Inc.
M. Satyanarayanan
Carnegie Mellon University
Marc Schnieder
Carnegie Mellon University

Abstract

We first describe how the Coda distributed filesystem was ported to Windows 95 and 98. Coda consists of user level cache managers and servers and kernel level code for filesystem support. Severe reentrancy difficulties in the Win32 environment on this platform were overcome by extending the DJGPP DOS C compiler package with kernel level support for sockets and more flexible memory management. With this support library and kernel modules for Windows 9x filesystems in place, the Coda file system client could be ported with very little patching and will likely soon run as well on Windows 9x as on Linux. We ported Coda file servers to Windows NT. For fileservers the Cygwin32 kit was used. We will not report here on the port of the Coda client to Windows NT, which is in an early stage. In both cases cross compilation from a Linux environment was most helpful to get a good development environment.

1. Introduction

The purpose of this paper is to convey our progress on porting a sophisticated distributed file system running on the Unix platform to Windows NT and Windows 9x. Coda [1], [10] boasts many valuable features such as read/write server replication, a persistent client cache, a good security model, access control lists, disconnected and low bandwith operation for laptops, ability to continue operation in the presence of network and server failures and well-defined consistency semantics. Using Coda as a vehicle to study the feasibility of porting complex Unix systems to Windows should be very interesting.

The Coda project began in 1987 with the goal of building a distributed file system that had the location transparency, scalability and security characteristics of AFS [11] but offered substantially greater resilience in the face of failures of servers or network connections. As the project evolved, it became apparent that Coda's mechanisms for high availability provided an excellent base for exploring the new field of mobile computing. Coda pioneered the concept of disconnected operation and was the first distributed file system to provide this capability [13]. Coda has been the vehicle for many other original contributions including read/write server replication [1], log-based directory resolution [14], application-specific conflict resolution [15], exploitation of weak connectivity [12], isolation-only transactions, and translucent cache management [17].

Coda has been in daily use for several years now on a variety of hardware platforms running the Mach 2.6, Linux, NetBSD and FreeBSD operating systems, and most of its features are working satisfactorily. Therefore we have started to focus on further improvement of performance and ports of Coda to other platforms. Ports to NetBSD, FreeBSD and Linux have confirmed that the underlying code relies almost solely on the BSD Unix API -- as intended -- and avoids Mach specific features to the maximum possible extent. During these ports various subsystems were recognized to depend too closely BSD specific file system data structures, and we replaced these with platform independent code, anticipating that this would greatly ease further ports in particular those to Windows. Our optimism about porting the user level code to Windows NT or Windows 95 using a POSIX implementation turned out to be justified. Our code compiled with very few patches. We will de-

scribe how this led to Coda servers running on Windows 9x and Windows NT.
However, a filesystem also depends on kernel code, and it turned out to be much less trivial to get that working on Windows 9x. We will describe these difficulties in detail. We will not report here on the kernel module for the Coda client on Windows NT.

This paper is organized by expanding briefly on the features and implementation of Coda. We then summarize some of the differences between the Win32 API and the Unix API which affect Coda. Finally we describe how we overcame serious difficulties to get clients running on Windows 9x. We finish by describing the porting effort for servers to Windows NT, and by summarizing the lessons we learned.

2. What is Coda?

Coda is the collective name for the programs and kernel modules which make up the Coda file servers and clients. Coda is implemented as a collection of substantial user level programs together with a small kernel module on the client which provides the necessary Coda file system interface to the operating system. The user level programs comprise Vice, the server, and Venus, the client cache manager.

The file server Vice is implemented entirely as a user-level program servicing network requests from a variety of clients. (For performance reasons, Vice can use a few custom system calls to access files by inode, but this is a detail.)

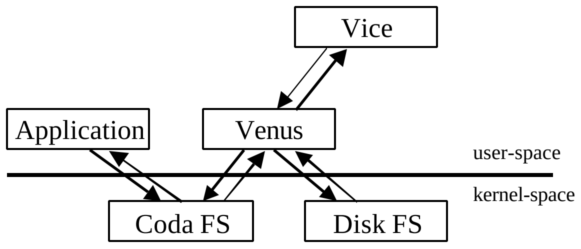

Figure 1

On the client the kernel module does some caching of names and attributes but is mostly there to re-direct system calls to Venus. The kernel module is called the *Minicache*. The user level programs make sophisticated use of the standard Unix API and they are multithreaded using a user level co-routine thread package. The filesystem metadata on clients and servers are mapped into the Vice/Venus address space and are manipulated using a lightweight transaction package called *rvm* [16]. The experience of the Coda project has been that placing complicated code in user-level processes offers tremendous development advantages without incurring unacceptable performance compromises [2]. This arrangement is shown in figure 1.

To the greatest extent possible, we wanted to replicate this structure -- and reuse code -- in the Windows ports. We faced several challenges in doing so. First, the user-level programs themselves represent a significant porting effort. Second, we must provide a kernel module to provide the Win32 file system services for Coda volumes. The second task has several components. The kernel module must translate Win32 requests to requests that Venus can service. Since the Venus interface was designed to do this for BSD Unix filesystems, and not for Win32 filesystems, some plumbing between the two filesystem models was needed. More fundamentally, given that the design uses a user level cache manager, we had to provide an environment for the user-level code that reproduces the concurrency properties of Unix upon which the Coda implementation model is based. This is discussed in more detail below. This appeared to be a highly difficult task for Windows 9x. It did not represent a fundamental problem for the Windows NT port.

For the purpose of this paper we will mostly concentrate on the client side of Coda. The two main components of a Coda client are *Venus*, the client cache manager, and the *Minicache*, which is the kernel filesystem code. When a Coda filesystem is used on a client, a program running on the client will make system calls which are directed to the filesystem in the kernel. The virtual filesystem passes this on to the Minicache. This does some preprocessing for the call and then passes the request on to Venus, which is running as a user-level process on the client machine. Venus contacts file servers or retrieves data from the persistent cache on the client machine and responds to the Minicache, after which the system call returns to the caller. In order to avoid many unnecessary context switches between the user program and the cache manager, the Minicache kernel code caches a small amount of naming information inside the kernel [2]. This enables the Minicache to complete most requests without involving Venus.

When a file is first opened, Venus resolves the name, and transfers it from a server holding the file to the client cache directory. Together with the file, the server delivers a *callback promise* to notify the client when the file has been updated by another client. This enables the client to do a subsequent opening of the file without contacting the servers, provided that callback for the file has not been broken. Another key feature is that read and write requests are directly redirected to the so called *container file* in the cache directory on the client machine, and are not processed by the Venus. This makes read and write calls almost as efficient as they are for locally stored files. Local modifications are propagated back to the server at close time, when the Minicache contacts Venus again.

For the purpose of this paper it is important to dwell a little longer on the basic mode of operation. The requests which are passed from the kernel to Venus and vice versa are transferred through a very simple character device (especially written for Coda), we will call this the *Coda device*. The main event loop inside Venus boils down to a *select loop* on socket file descriptors for the connections with servers and the file descriptor of the Coda device. When data arrives from the Minicache, or from the servers, the select call returns to allow Venus to take appropriate action. The system call interface described above hinges on the implementation of select for the Coda device as follows. When the kernel is dispatching its system call request to Venus, on behalf of the calling process, it adds this calling process to a wait queue associated with the device. Then it wakes up Venus which proceeds to process a read system call on the Coda device, after which Venus can process the request from the kernel. When Venus is ready to deliver the result, it issues a write call to the Coda device. The write system call handling for this device finds out for which process the reply is meant, and removes this process from the wait queue. The process making the system call can now be scheduled and proceed.

3. Strategy

Venus contains the bulk of the code needed to get a client running, and it is quite a complicated program. The task of debugging the Minicache kernel code in the presence of such a large module is not attractive, particularly since Venus' operation depends on a correctly implemented Minicache and vice versa. The Coda team faced this before with the port of Coda to NetBSD and decided to develop *Potemkin Venus.* Potemkin is a program that appears to be a genuine Venus, connected to servers and responding to requests coming from the Minicache through the Coda device. When using Potemkin a directory tree of files residing on the "client" itself can be mounted as a Coda filesystem. Potemkin is a simple program, does no fancy caching and is merely a tool to test the implementation of the Minicache kernel code for the Coda filesystem. It also utilizes the same select loop as Venus does.

Clearly the first element of porting Coda to Windows is to port and adapt Potemkin to Windows and to create a Minicache.

4. Windows 9x Problems

Although in many ways NT represents a more attractive and appropriate target for Coda, we decided to look at Windows 95 support first. Its Win32 implementation appeared to have roughly the capabilities we need (TCP/IP sockets, memory-mapped files, threads), and, unlike NT, its installable filesystem interface is documented (in the Windows 95 Device Driver Kit).

To explain the results of our first Windows 95 experiment, it helps to provide a brief description of its architecture. The Windows 95 kernel is called the Virtual Machine Manager (VMM). All Windows applications run in a single virtual machine called the System VM. Each Win32 process has its own memory context and may have multiple threads which are managed by the VMM. Other virtual machines may be created to support DOS applications; these VMs contain only a single address space and a single thread. Programs running in DOS VMs (so-called "DOS boxes") have access to the standard DOS interrupt API and a protected-mode extension called DPMI (as well as virtualizations of typical PC hardware), but not the Win32 API. Win32 is implemented by libraries that run in user-space in the System VM and use many undocumented VMM interfaces. Fortunately, the Windows 95 DDK does specify the internal interface between the VMM component which provides file-oriented I/O to the virtual machines (called the IFSMgr) and installed filesystems. To put it in Unix terms: the (analogues of) user-level libc and kernel-internal vfs interfaces are documented, while the actual kernel system calls are largely undocumented.

Working from the DDK information, we began to implement a Potemkin-like system for Windows 95. The kernel-mode component consisted of an installable network filesystem which would enqueue file requests for a custom Win32 Potemkin venus-like process. (At this point, we made no effort to reuse Unix code, but wrote directly to the Win32 API.) The requests and responses were transferred using the Win32 DeviceIoControl API call, rather than a character device, and threads making requests synchronized with the Potemkin process by waiting on VMM semaphores.

Initially, this approach seemed to work. It was possible to start a DOS window, mount the filesystem, and manipulate files on it. However, the system would sometimes freeze if the user tried to manipulate the Potemkin drive using Windows applications. Unfortunately, this reflected a fundamental problem: the implementations of many Win32 API functions rely on 16-bit libraries in the System VM which are non-reentrant. Thread access to these libraries is serialized, system-wide, by a mutex called the Win16Mutex. The Microsoft Press book *Inside Windows 95* [3] is emphatic that the lowest-level Win32 API calls (implemented by a library called Kernel32) do not try to acquire this mutex, but unfortunately this is not so: even the very simple file I/O calls which our Win32 Potemkin made to service requests would sometimes try to acquire Win16Mutex. If the process making the request of Potemkin had already acquired it, the two processes would deadlock. Unfortunately, since the Windows user interface itself relies very heavily on 16-bit code protected by the Win16Mutex, the whole system would appear dead at this point. *Unauthorized Windows 95* by Andrew Schulman [4] and *Windows 95 System Programming Secrets* by Matt Pietrek [5] explore some of the ways that the Win16Mutex is actually used in Windows 95.

5. On DOS and Coda for Windows 9x

We rapidly concluded that the limitations of the Win32 API implementation in Windows 9x ruled it out as a host environment for Venus. Two possibilities at least remained. A somewhat unrealistic option would have been to implement all of Coda in the installable filesystem VMM component. A more attractive alternative was to implement the cache manager not as a Win32 process but as a DOS program. While Win32 applications often contend for the single Win16Mutex, the Windows 9x design actually gives DOS programs significantly better treatment: the actual VMM core is more reasonably flexible OS kernel which, like a Unix kernel, permits multiple VMs to have simultaneous outstanding system calls. This approach sidesteps the Win16Mutex problem and frees us from worrying about further non-reentrancy in the Windows 95 Win32 implementation. Furthermore, the application environment in a DOS box need not be as hostile as one might imagine: D.J. Delorie's DJGPP port of gcc provides a 32bit Unix-like libc, and Phar Lap sells a Win32-subset implemention that runs in DOS boxes.

We decided to explore the possibility of hosting Venus in a DOS box, using DJGPP's compiler and libc. The first step was to identify what APIs Venus needed that were not already provided. They were:

- TCP/IP networking accessed using the standard BSD sockets API
- a *select()* call that could wait on both networking sockets and the Minicache simultaneously, and
- a limitted form of *mmap()*. Specifically, Venus needs to be able to allocate virtual memory at fixed virtual addresses in its memory space. On Unix systems, this is accomplished using an anonymous *mmap()* call.

The TCP/IP implementation in Windows 9x runs in kernel mode as part of the VMM. The published Win32 networking API is implemented by a user-level library that uses undocumented VMM calls. There is no built-in support for TCP/IP access from a DOS box. However, there is a sketchily-documented internal VMM interface to the networking stack. The approach we adopted was to use this internal interface to implement a sockets-like API which could be exposed to DOS boxes.

A second, related issue was support for the *select()* call that drives the Venus event loop. This *select()* call needs to wait on multiple UDP and TCP sockets and simultaneously on the queue of requests from the Minicache. To support this, our kernel sockets module exposes an interface that the Minicache uses to participate in *select()* calls.

Finally, as mentioned above, Venus uses a package called *rvm* that provides a transactional, memory-resident database of filesystem metadata. This package relies on being able to read the database contents into the same region of virtual memory every time Venus runs. Unfortunately, DPMI, the interface that allows 32-bit applications running in DOS boxes to allocate memory, does not permit the application to specify the virtual address where the newly-allocated memory block will land. (To be precise: there is a version of DPMI that does specify a way to do this, but the Windows 95 implementation does not implement the feature.) As a result, we wrote a kernel module that implemented a separate memory allocation system for 32-bit DOS applications and which does permit the application to choose which virtual pages to allocate.

Once these pieces were in place, our port became surprisingly straightforward. To manage our source code effectively we resisted the temptation to go native. Instead, we cross-compile from Linux workstations. For debugging, we use gdb's remote debugging feature: a specially modified debugging stub runs Venus under debugger on Windows 95, communicating over a TCP socket (using our socket implementation) with a gdb process running on Linux. The result is a highly effective work environment for Coda development: we can compile and debug the Windows 95 Venus without leaving xemacs!

It is perhaps worth mentioning two other aspects of our approach which significantly speeded the porting process. First, we implemented a work-around for Windows 95's lack of support of dynamically loaded filesystem drivers. We wrote a small shim filesystem driver that would load and register itself at boot time as required by the Windows 95 architecture. All actual filesystem requests it would divert to a dynamically-loaded Minicache module that we could update multiple times without rebooting. Second, during development we modified Venus to use an intermediate relay program to converse with the Minicache. That is, instead of having Venus read requests from the Minicache and send replies directly back, we had Venus read and reply to Minicache requests using a UDP socket. A separate relay program would transfer requests and replies between the Minicache and the UDP socket that Venus used. This had a couple significant advantages: a) the relay program provided an excellent debugging log of interactions between Venus and the Minicache and helped identify quickly, when a bug occurred, which component was at fault; b) by using a modified relay program, it was possible to test Venus with synthetic requests.

6. Coda Client on Windows 9x: Results

The experiment of porting the Coda client on Windows 95 has been gratifyingly successful. Although it is not stable enough to use in production, it *is* possible to install Microsoft Office into a Coda filesystem and then use the resulting installation to do real work—including using Office while disconnected from the Coda server cluster. In fact, it has even worked to install Office into a Coda filesystem while in disconnected mode and then to allow the reintegration process to update the servers with all hundreds-of-megabytes of Office code. While it is too early to report precise figures, Coda performance when in connected mode with a cluster of Linux Coda servers is within a factor of two or so of the built-in SMB client when run against Samba on similar Linux servers.

Despite the enormous differences between Unix and Windows environments Coda, with the exception of the kernel modules, builds from a single source archive, with very little conditional compilation. The overall arrangements of source are shown in figure 2.

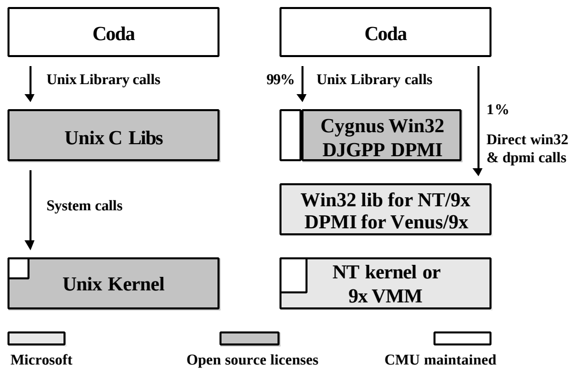

Figure 2

7. Coda servers on Windows 95 and NT

The Coda servers are much easier to port. They do not rely on the upcall mechanism and do not need any kernel support for their operation. Here we chose to use the Cygnus Cygwin32 library [18], which implements a Unix C library on the Win32 environment. This port proceeded smoothly and led to working servers within a few months. During the port of Coda to Linux those subsystems which appeared to rely on BSD specific data structures defined by the host environment were replaced with data structures private to the Coda package, that could exist on all platforms. This strategy proved very valuable, for porting both servers and clients, since the cache manager and file server compiled with very few patches under DJGPP and Cygwin.

One notable difficulty is that Windows NT executes asynchronous procedure calls which do not appear to mix well with Coda's user level thread library. While a work around could be found by modifying Cygnus' select call implementation, we expect that a better solution will come forward when we implement Coda threads on NT Fibers.

Initial impression were that the performance will need tuning and that some more native features are desirable, particularly on Windows NT.

8. Summary and lessons learned

Using a variety of freely available software packages, a very complex porting problem could be tackled. A severe lack of clear documentation on Microsoft's part, made us go wrong in several ways before finding a solution to our problems. The kernel environment for Windows 95, while difficult and hostile as a development environment, allowed the implementation of socket, special memory allocation and filesystem support in a way that circumvented the user level Win32/Win16 mutex problems. Porting our servers was comparatively straightforward using the Cygwin32 library from Cygnus.

Looking back at our work, we vividly remember that initially and particularly after the first set backs, we were very doubtful if Coda could run on Windows. It turned out that with some creativity it did become possible to achieve our goals and obtain acceptable quality. Kernel related software for Windows is difficult to write, but the POSIX environments offered by DJGPP and Cygwin32 are truly remarkable, and should allow may other ports to proceed smoothly.

Acknowledgements

The development of Coda over the last decade has been made possible through the generosity and sustained support of many funding agencies and corporations. The funding agencies include the National Science Foundation (NSF), under contract CCR-8657907, and the United States Air Force (USAF) and Defense Advanced Research Projects Agency (DARPA), under contract numbers F33615-90-C-1465, F19628-93-C-0193 and F19628-96-C-0061. Corporate sponsors include the Intel Corporation, and Novell Inc. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the USAF, DARPA, Intel, Novell, CMU, or the U.S. Government.

We wish to thank members of the Coda team for discussion.

References

1. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E Okasaki, E.H. Siegel, D.C. Steere, Coda: a highly available file system for a distributed workstation environment, *IEEE Transactions on Computers,* Vol. 39, No. 4, April 1990.
2. D.C. Steere, J.J. Kistler, M. Satyanarayanan, Efficient User-Level File Caching on the Sun Vnode Interface, *Proceedings of the 1990 Summer Usenix Conference*, June 1990, Anaheim, CA.
3. Adrian King, *Inside Windows 95,* Microsoft Press, 1993
4. Andrew Schulman, *Unauthorized Windows 95,* IDG Books, 1994
5. Matt Pietrek, *Windows 95 System Programming Secrets,* IDG Books, 1995
6. http://www.microsoft.com/hwdev/ntifskit/default.htm
7. http://www.cygnus.com/misc/gnu-win32
8. D.G. Korn, Porting Unix to Windows NT, Usenix 1997 Annual Technical Conference
9. http://www.sysinternals.com/ntfilmon.htm
10. http://www.coda.cs.cmu.edu/index.html
11. Howard, J.H., An Overview of the Andrew File System, Proceedings of the USENIX Winter Technical Conference Feb. 1988, Dallas, TX.
12. Mummert, L.B., Ebling, M.R., Satyanarayanan, Exploiting Weak Connectivity for Mobile File Access , M. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995, Copper Mountain Resort, CO.
13. Kistler, J.J., Satyanarayanan, M., Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems Feb. 1992, Vol. 10, No. 1, pp. 3-25.
14. Kumar, P., Satyanarayanan, M., Log-Based Directory Resolution in the Coda File System. Proceedings of the Second International Conference on Parallel and Distributed Information Systems Jan. 1993, San Diego, CA, pp. 202-213.
15. Kumar, P., Satyanarayanan, M., Supporting Application-Specific Resolution in an Optimistically Replicated File System. Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems Oct. 1993, Napa, CA, pp. 66-70.
16. Satyanarayanan, M., Mashburn, H.H., Kumar, P., Steere, D.C., Kistler, J.J., Lightweight Recoverable Virtual Memory. ACM Transactions on Computer Systems Feb. 1994, Vol. 12, No. 1, pp. 33-57 Corrigendum: May 1994, Vol. 12, No. 2, pp. 165-172.
17. Satyanarayanan, M., Ebling Maria E., Translucent Cache Management for Mobile Computing, to appear in proceedings of the workshop on Ubiquitous Computing, Atlanta GA, 1997.
18. Geoffrey J. Noer, Cygwin32: A Free Win32 Porting Layer for UNIX Applications, 2nd USENIX Windows NT Symposium, 1998 August 3-4, 1998 Seattle, Washington, USA