# An Architecture for Opportunistic Network File System Services


Dutch T Meyer
Department of Computer Science
University of British Columbia
Vancouver, British Columbia
dmeyer@cs.ubc.ca

Michael DiBernardo
Department of Computer Science
University of British Columbia
Vancouver, British Columbia
mddibern@cs.ubc.ca



## ABSTRACT

Disk drives are widely cited as a critical bottleneck in systems performance. This is particularly true of notebook computers, which are significantly constrained in their power consumption and their limited size. Bound by these physical constraints we can expect disk drives to become an increasingly problematic component of home and business class computers. In contrast, high speed wireless and gigabit wired networking hardware is an increasingly common component of these devices. From a cost/performance standpoint, the rapid pace of development in computer networking represents an opportunity for fast reliable networking to supplant some traditionally local file system services and operations. In this work, we show how network connectivity, even when sporadic, can lend itself to optimistic services providing significant performance benefits. We detail a simple kernel-mode event forking mechanism as part of an architecture for providing these services. We further describe our experience in using this architecture to create a efficient opportunistic network block cache client and server for Windows based clients. Finally, we argue that this opportunistic model is a powerful way to create efficient, transparent, and fault tolerant file system services.


## 1. INTRODUCTION

The slow relative performance of disk drives is a major design factor in operating system development, requiring careful design and tuning of file system caches and paging subsystems. While the maximum raw throughput of disk drives can be quite high, performance can drop to 1MB/sec under seek workloads. [7] Also, there are many routine maintenance operations performed on file systems that are expensive, both computationally and in terms of IO, such as defragmentation and virus scanning. The cost is particularly high in notebook computers, which have much tighter performance and power constraints than desktop systems. These maintenance operations are currently performed on a per-system basis, even though much of the content that is stored across these systems is similar. Network file systems can effectively address some of these problems, but can also carry a significant cost in performance and complexity, as well as in reliability and concurrency semantics.

Notebook computers, which are increasingly popular, now come standard with a powerful array of networking technologies. Current features include Gigabit Ethernet and 802.11g; we anticipate that the state of wired and wireless networking will continue to advance rapidly. Furthermore, volatile and non-volatile memory are increasing in performance, and decreasing in cost at a much faster rate than disk drives. While these factors do not signify the imminent obsolescence of disk drives, they do provide a significant opportunity for providing opportunistic support for common operations over a network, as we will show.

### 1.1 Motivating Example

Before we begin describing the technical structure of our architecture, we should solidify the scope and nature of the model with a motivating example. Here we will describe the use of a class of network-based file system services and show how they are opportunistic, reliable, and efficient. We imagine this class of services being of primary benefit to users of notebook computers running a popular operating system.

Consider a notebook user on the way to work. They may ride public transit and use the time to perform common business operations on their computer. In this setting the computer acts and responds as it always would. Once our user arrives at work they dock their laptop and continue the project they were working on, while also accessing the local network and World Wide Web. In this mode of operation, the system continues to function with no visible difference, except in significantly increased performance.

Transparently, the user's virus scanner does not need to activate locally, because all blocks and files accessed are checked against a known correct files and blocks on a centralized server. Simultaneously, most read-only file operations need not activate the disk, because they can be serviced from a memory mapped instance of the user's operating system, which is again provided over the network. One can imagine many more examples, of opportunistic reliable services, some of which are mentioned in Section 6.1.

Our proposal is for a simple and effective architectural solution to create opportunistic network-based services that operate through a transparent client interface. Our client mechanism allows file system events to be forked, creating parallelizable external and internal requests. By forking file system events, we allow file system operations to be serviceable across a network interface, and also enable the centralization of shared common services (e.g. virus scanning). Service providing components are not difficult to implement and benefit all users scalably.

Our current implementation leverages the file system minifilter and device class block driver mechanisms of the Windows NT/XP driver model, though we do not consider it platform specific. In our presentation, we assume that our system is being deployed in a trusted computing environment. As such, we do not discuss the security implications of our approach at this time.

# 2. WINDOWS IO SYSTEM STRUCTURE

Because the Windows operating system is an uncommon implementation platform for academic operating systems research, we first present several architectural aspects of the OS that are of import to our approach.

## 2.1 Windows IO Manager Overview

The Windows IO manager is designed with extensibility in mind. The cache is centralized, but can still integrate with third-party file systems, drivers, and services that are added to the operating system[9].

The file system drivers are arranged in a "stack" (Figure 1). File system requests flow down this stack towards the disks. Once they are serviced, completion routines can further process and manipulate system state by being registered as callbacks. The generic File Object is used by all drivers, at every level, thus providing consistency of access to file properties and operations. While the stack shown in Figure 1 would seem to display direct communication with registered ancestor and successor elements of the stack, in reality these communication channels are mitigated by the IO manager. Events are also generalized, and are represented as Interrupt Request Packets (IRPs). These few generic data structures allow third parties to create modular and extensible drivers and file system extensions[9].

A device may handle an IRP in a variety of ways, at any level in the driver stack. For instance, If a driver is capable of fulfilling a file system request, it can mark the request as "completed" and pass the completed IRP back up the driver stack, without having to propagate it further downwards. Requests that are identified as being unsuccessful can be handled similarly. IRPs may also be marked as "pending", in which case they are blocked and deferred for later processing. This allows kernel worker threads to handle events that are too time consuming to block other pending events, or that require an interrupt that has been masked in the active thread context [9].

We also make use of the TDI (Transport Driver Interface), which allows drivers at any level to access the network redirectors in a protocol-neutral way. This permits drivers to access network resources, which is a critical component of our system architecture.

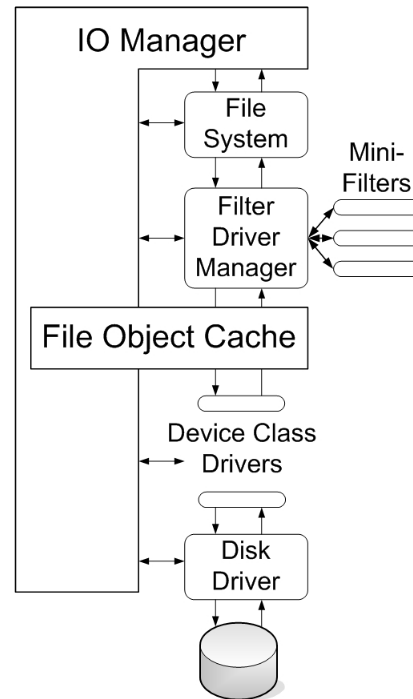


**Figure 1: Architectural overview of the Windows IO Manager.**

## 2.2 File System Filter Drivers

File system filter drivers allow developers to customize the behavior of an installed file system. For example, this is used by virus scanning programs to verify the security of programs loaded from disk. Filter Drivers run in kernel space on behalf of a filter manager, and may be loaded and unloaded dynamically. These drivers operate at the file block level, and each is loaded at a predefined elevation, establishing its relation to all other drivers in the stack.

## 2.3 Device Class Drivers

Device class drivers may be loaded at any level of the driver stack. We are primarily concerned with low level drivers that act on block level requests, which is the primary distinction that we will draw between device class drivers and filter drivers. Also, Device class drivers must be loaded at boot time and cannot be unloaded dynamically.

# 3. OPPORTUNISTIC FILE SERVICES

## 3.1 Overview

The basic structures used in our network services architecture consist of a client-side event forking (or redirection) mechanism and a centralized network service provider. Using a disk or file level driver, client file system requests are trapped in the file system path before they reach the physical disk driver. At this point, the request can either be sent to disk, or forwarded to an external interface. Similar techniques have been used in a variety of other systems[6]. By transferring these requests over a fast reliable network,

we provide opportunities for performance benefits over the local file system path. Simple centralized file servers can be implemented which act on these file requests. We believe that whenever file system requests are made that are time consuming and redundant (in the context of the entire network), that the alternate path provided by a network interface provides a compelling alternative.

It should be noted that external and local paths can be pursued concurrently. We are are still developing heuristics that will allow us to identify when each of these options should be selected, but reasoning based on common workloads leads us to the following assumptions:

- Read access to system files, dlls, and executables are very likely to contain similar content across all machines. These are excellent candidates for remote execution.
- Laptops and other power-constrained devices should prefer to avoid spinning the disk whenever possible, and so a blocking external call is preferable. Since the latency on a modern and closely coupled network dwarfs that of a disk seek, the performance impact of a quick query to a network host is acceptable.

We now turn our attention to a specific implementation of our Network Block Cache Client. This system is designed to demonstrate the effectiveness and utility of our architecture. In it, a central server provides access to a large cache of disk blocks. We envision this being most applicable in a network containing many windows laptops. Since these devices share many common pages, there is significant opportunity for exploiting redundant data, in the spirit of [2]. In addition, the readily available (but slow) disk path means that we can use the service opportunistically. When it is beneficial, we will use it, but sudden loss of connectivity will in the worst case force us to use the physical disk with a negligible overhead.

## 3.2 Network Block Cache Client

For the Network Block Cache example, we elected to perform synchronous requests, first considering the network, then going to disk when the network or requested block is unavailable. We take this action whenever we receive a read request on a file we consider "interesting"[1]. If at any time our network infrastructure delays or is unavailable to refer our request to the physical disk path. Figure 2 illustrates the program flow through the block level driver.

Our client operates as a disk block level implementation, even though it was a more difficult, because of the opportunity to hash disk contents during reads. While this is not utilized in our current prototype, we envision leveraging this capability as a hashing function for a fuller implementation of the network block cache. The client driver consists of 3,400 source lines of code, including an excessively rich set of networking functions. Much of this is boilerplate code to initialize and load the block level driver.

[1]We have several potential notions of what constitutes an interesting read, for the purposes of this paper we will refrain from defining any particular criteria for an interesting file.

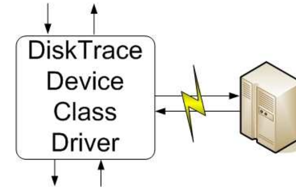


**Figure 2: Architectural overview of our Network Block Cache Client Driver.**

## 3.3 Network Block Cache Server

The current implementation of the cache server is quite simple. At startup, the server reads an index file that maps disk offsets to file regions, and subsequently reads the file data and associates it with the appropriate disk offsets. This index file is generated by an off-line tool discussed in section 4. Thus, the startup time of the server is potentially quite slow if the size of the payload being cached is large.

The client initiates a new connection per block-level request, and thus a potentially large number of requests may need to be handled per unit of time. The server maintains a thread pool, and dispatches individual requests to the pool to be handled by the worker threads. Using the thread pool removes the overhead of creating a destroying a thread to handle each request; however, because a connection is initiated for each request, the proportion of time spent performing TCP handshakes is potentially quite large. Going forward, removing this per-connection functionality will be a major key to improving performance.

Several external libraries are used to provide the server functionality. Requests are handled by performing a binary search on disk offset, using the STL implementation of binary search. Threading is provided by the Boost C++ distribution [1]. The thread pool implementation is an unofficial extension to Boost by Philipp Henkel [5].

# 4. SUPPLEMENTARY TOOLS

In Section 3.3, we discussed how the network cache server uses an index file to associate file data with offsets from the client's disk. It was our intention to produce a utility that would generate this index using the Windows Defragmentation API; however, time constraints prevented us from doing so. We envision this tool walking the client directory tree and identifying candidate files for caching on the network. The layout of these files on disk would then be examined using the Windows DeviceIoctl system call, and the relevant disk offset ranges would be written to the index file for interpretation by the server.

This approach requires that the user and the operating system do not perform any operations that alter the layout of files on disk. Thus, utilities such as 'defrag' would have to be disabled. We are not sure if the Windows operating system performs any "built-in" maintenance operations that rearrange file contents on disk; if this is the case, then the index would have to be generated regularly to ensure consistency. This is obviously less than ideal, and we are thus considering other methods for generating and maintaining this index.

## 5. PERFORMANCE EVALUATION

We performed two experiments to test the performance of our architecture versus the performance of the local disk drive. As our results are preliminary, we felt it was important to be conservative in our testing. The client machine on which we ran the tests was a dual core 2.4Ghz Processor, with 1GB of ram, and a Western Digital SATA Hard Drive with 8 MB Cache running at 7200 RPM. We expect the relative performance of out network centric services to be better on slower machines; as this is a more powerful machine that most current laptops it is a conservative choice for testing. In contrast, the server was run on an Apple iBook 1.2 GHz G4 with 768MB of RAM. The test machines were connected to a local-area network with a 10Mbs connection. Recall that our current implementation establishes a full TCP setup and tear down with every file system event. The combined effect of these arrangements were clearly disadvantageous for demonstrating the utility of our approach, because the environments that we are targeting would be likely have higher quality networking and server hardware, and slower client hardware.

The first experiment we conducted was a simple one: We copied a large file from one location to another, and monitored the time required to copy all of the file fragments using both the local disk drive and the network cache. We used the TraceView utility provided in the Microsoft IFS Driver Development Kit to monitor how long it took to service each of the multiple requests required to complete the entire file copy: the results of this test can be seen in Figure 3. While the network approach outperforms the disk at first, the disk overtakes the network once about 530K of data has been transferred. With more realistic experimentation we expect that this number would increase significantly; however, it is to be expected that the disk will eventually outperform network in terms of throughput. Still, even at this early stage of development, the most recent Windows based file system trace we are aware of suggests that this could encompass nearly 80% of file read-only file system requests[10].

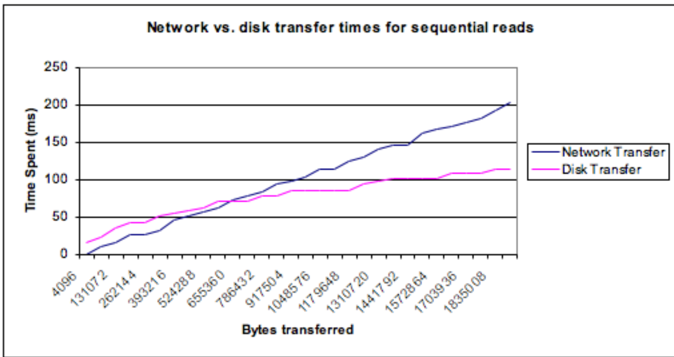


**Figure 3: Transfer times with respect to transfer size for network and disk transfers. Disk transfers overtake network transfers in terms of performance at roughly 530K.**

The second experiment involved issuing a series of 4096K read requests to files of varying sizes, with uniformly random seeks being made in between each read. These random 4k seeks can comprise 50% of file system benchmarks[7]. The increasing file size can be seen as a measure of increasing randomness in disk accesses. For each trial, 3 series of 1000 such reads were issued, and the average time required to complete all 1000 reads was measured. In between each series, we issued a large number of sequential reads on a small collection of different files so as to clear the cache before the next run. This test was implemented as a Perl script, using Perl's unbuffered 'sysread' function. The results of this experiment can be seen in Figure 4.

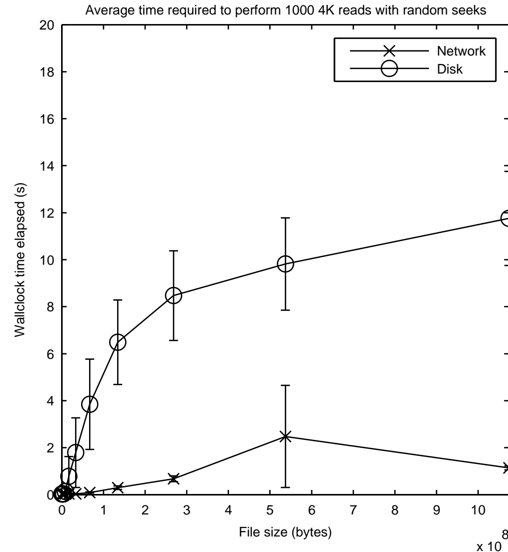


**Figure 4: Transfer times with respect to file size for network and disk transfers. Read requests were for 4K of data. Before each request, a uniformly-distributed random seek was made to some point in the file.**

One can see that our approach significantly outperforms the disk in this scenario, and that the performance gap increases with the file size. This is because random seeks on a small file are much more likely to exhibit locality of requests than random seeks on a much larger file, and so the cache becomes less and less effective as the file size grows larger. Seeks have essentially no effect on our approach, however, because the retrieval time is independent of the ordering of the virtual locations of the file fragments. Also notice that the network approach has much more consistent performance in the presence of many random seeks: In general, the standard deviation of the request time for network requests is much smaller than that for disk requests.

One might be surprised to see that the network cache performance still slowed with increasing file size, expecting our response time to be completely independent of this variable. We believe that the slight increase is also a result of decreased local cache effectiveness, which is naturally faster than network access. However, the magnitude of the effect is much less severe with our approach because of the slow relative performance of disk seeks.

## 6. SUMMARY

We have detailed a simple and effective architectural design to allow file system requests to be processed and monitored

outside of the traditional file system path. We believe that this architecture will enable a new class of opportunistic file system services. We have demonstrated the potential for this architecture with an initial prototype and evaluated its effectiveness showing a significant improvement in important file system operations. Furthermore our opportunistic approach means that we are never fully reliant on network connectivity. When necessary, we can fall back to disk based services; however, whenever possible, we can transparently provide fault tolerant network based services.

## 6.1 Future Work

We intend to further show the effectiveness of our architecture by completing several prototype implementations. Currently we are evaluating the potential for centralized virus scanning, disk defragmentation, bad sector reclamation, and write through caching. We also imagine diabolically pinning pages on our network server in order to provide automatic patching of client programs.

Our initial tests identify instances in which our approach can mitigate some of the performance problems typically associated with certain secondary storage access patterns. An extensive review of such patterns for Windows is provided in [10] and [3], and we are actively considering utilizing the results of these findings in future work.

To date, we have focused our efforts on read-only data paths; however, we believe that our approach could be extended to provide fault-tolerant write operations.

There are many improvements that could be made to the server to increase its efficiency and its utility. In addition to adjusting the communication protocol to remove the existing connection overhead, we have also considered indexing data via hashing function, so that identical blocks of data do not have to be maintained separately. This would reduce both the memory requirements and the time required to services requests by reducing the total size of the index.

## 6.2 Related Work

In the context of our work, Microsoft's new ReadyBoost[8] and ReadyDrive[8] technologies are relevant, as are the associated Hybrid Disks technologies they support and emulate. While these could be seen to limit the applicability of our work in that they improve the relative performance of disk drives, they are complementary in nature and do not encompass the fuller potential for opportunistic network services (i.e. virus scanning, degragmentation). Insofar as our architecture has been applied as a network cache, it holds some similarity to GMS[4] and related distributed memory systems. Obviously our system is far more limited in scope, and correspondingly benefits from simple design and error handling. Single Instance Storage (SIS)[2] is similar in some ways to our shared cache, but targets a different space. Finally, Galen Hunt and other have done extensive work utilizing the windows device driver framework [6], in various creative and useful ways.

# 7. REFERENCES


[1] The boost c++ libraries. `http://www.boost.org/`.

[2] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, 2000.

[3] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. *SIGMETRICS Perform. Eval. Rev.*, 27(1):59–70, 1999.

[4] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 201–212, New York, NY, USA, 1995. ACM Press.

[5] P. Henkel. Boost threadpool implementation. `http://threadpool.sourceforge.net/`.

[6] G. C. Hunt. Creating user-mode device drivers with a proxy. In *Proceedings of the 1st USENIX Windows NT Workshop*, pages 55–59, 1997.

[7] Microsoft. Readydrive - an interview with ruston panabaker. `http://download.microsoft.com/download/4/d/9/4d9eb837-9065-4b4d-864e-58%5a6cebd0a2/ReadyDrive.wmv`.

[8] Microsoft. Windows readydrive and hybrid hard disk drives. `http://www.microsoft.com/whdc/device/storage/hybrid.mspx`, May 2006.

[9] D. A. Solomon and M. E. Russinovich. *Inside Windows 2000*. Microsoft Programming. Microsoft Press, One Microsoft Way, Redmond, Washington 98052-6399, 3 edition, 2000.

[10] W. Vogels. File system usage in windows nt 4.0. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 93–109, New York, NY, USA, 1999. ACM Press.