WFS: A Simple Shared File System for a Distributed Environment

by Daniel Swinehart, Gene McDaniel, and David Boggs
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA

Abstract:

WFS is a shared file server available to a large network community. WFS responds to a carefully limited repertoire of commands that client programs transmit over the network. The system does not utilize connections, but instead behaves like a remote disk and reacts to page-level requests. The design emphasizes reliance upon client programs to implement the traditional facilities (stream IO, a directory system, etc.) of a file system. The use of atomic commands and connectionless protocols nearly eliminates the need for WFS to maintain transitory state information from request to request. Various uses of the system are discussed and extensions are proposed to provide security and protection without violating the design principles.

1. Introduction

Existing file systems implement different levels of service for their clients, and correspondingly leave different amounts of work for the clients to do. Traditionally, file systems have evolved to provide more and more functionality from simple file access to complicated arrangements which provide sharing, security, and distributed data storage.

This paper describes WFS, a file system that provides a concise set of file operations for use in a distributed computing environment. Designed by the authors in 1975, and built by one of us (Boggs) in under two months, WFS has successfully supported a number of interactive applications.

The filing needs of *Woodstock*, an early office system prototype, dictated the functional and performance criteria of WFS. Woodstock provided facilities for creating, filing, and retrieving simple office documents, and a rudimentary facility for exchanging these documents as electronic messages.

Woodstock's hardware environment was a network of minicomputers, each providing specialized functions (terminal control, editing, filing, message services, etc.) in support of the overall application. WFS was designed as the shared filing component, storing Woodstock documents on high-capacity disks attached to one of these processors.

During development, Woodstock used small local disks on each editing processor. The software that supported the editing application had to provide facilities for transforming access to physical disk pages into higher-level functions. These included character and word I/O, file positioning, and functions for opening and closing files. The application also implemented its own hierarchical document directory structure.

WFS was designed after the rest of the system was operational. Consequently, it was easy to define its functional specification, since Woodstock already provided the higher-level functions. The local file access was to be replaced by network access to a shared file system running on another machine. A file system based upon page-level access to randomly addressable files would be adequate, and a small amount of file sharing needed by the application could be accommodated by a simple locking mechanism at the file level. A two month limit on implementation time, combined with a conviction that a very simple file system organization could achieve

the same purposes as existing more complex designs, led to the system described here.

2. System Description

2.1 The Client's File System Model

In this paper, a *server* is a program that supplies a well-defined service over a computer network to *client* programs, which use the service to implement some application. A client program may or may not be operating in direct response to the actions of a human *user.*

WFS is a server that provides its clients with a collection of files. It is currently implemented on a dedicated Xerox Alto research minicomputer [Thacker *et al*] augmented by one or more disk drives, each with a transfer rate of around 7 megabits per second and a capacity of from 80 to 300 megabytes. A WFS file contains up to 60,516 data pages, each 246 16-bit words long. Clients may write pages in any order, and WFS waits to allocate space for a page until it is first written. A file is denoted by a 32-bit unsigned integer, its *file identifier (FID).* WFS allocates FIDs for new files, on request, from a single name space. There is no additional naming or directory structure within the system. For this reason, and because of the carefully limited repertoire of operations, an application programmer might well choose to view each FID as a handle on a "virtual disk", interfaced through a moderately intelligent controller.

2.2 WFS Operations

The complete set of WFS operations is shown in Table 1. Each operation involves an exchange of network packets using the protocol described in the next section. The operations partition into four groups, used for:

- Reading and writing pages of files
- Allocating and deallocating FIDs and pages of files
- Obtaining and modifying file properties
- Performing system maintenance activities

The most commonly executed operations are those used for reading and writing a selected file page, given its FID and page number. A number of *page properties* are returned along with each page that is read (see below), and client modifications to some page properties may be specified during each write operation.

The second group of operations allows one to create a file (with no assigned pages) and obtain its FID, to expunge a FID (illegal if any pages remain), and to deallocate the storage for a page. In addition, there is an operation that allows a client to create a file with an explicitly specified FID value. WFS reserves a range of FID values for this purpose when it creates a new file system.

Operation	Description
Page Transfer	
ReadPage(fid,pageNum) WritePage(fid,pageNum,lock,page properties)	Read or write page properties and page data
File Management	
GetFID() ExpFID(fid) DeallocatePage(fid,lock, pageNum)	Allocates a new file and returns its fid If fid has no pages allocated, expunges (deletes) the file Releases storage for page and removes page from page map
Status Query/Modification	
GetFIDStatus(fid) SetFIDStatus(fid,mask,value) ReadPageMap(fid,lock, pageMapNumber) Lock(fid) UnLock(fid)	Return file status values Set client status values, ignore attempt to affect system values Return page map information to determine which pages are allocated Return key, required in subsequent operations until file is unlocked Unlock file (set lock to zero)
Maintenance	
ReallocFID(fid) ResetLastFID(newFid) ReadRealPage(realAddress) GetVMap()	These operations allow examination of the system at the disk logical and physical page level. In addition, the FID allocation routines can be used to restore the file system using backup information.
WFSPing()	WFS merely acknowledges this operation. It allows one to check the basic communications path.

Table 1. WFS Operations

The third group allows the client to find out what file pages are allocated, and to examine a FID's current *file properties*. One of the operations allows the client to modify those file properties that are under its control.

The fourth group provides maintenance facilities. Utility client programs use them to copy WFS files to a backup store, restore selected files, rebuild WFS volumes from backup, and repair client-level file structures.

2.3 Properties

WFS associates with each data page a set of *page properties*, some of which are of interest to the client (see Figure 1). WFS reads and writes the page properties along with the data. The first few fields provide a safety check since they duplicate the FID and page number, and the system checks them on each page access. They may also be used by low-level crash recovery routines to reconstruct damaged file structures. The *client* fields are assigned and interpreted by the client. The client may ask WFS to compare a page's client properties against the ones supplied in a command, and to abort the command if they fail to match. This allows the system to validate client assertions about the page in question.

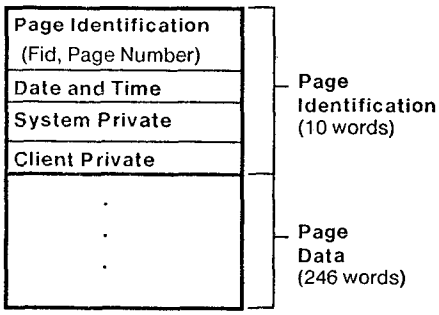

Figure 1. WFS Disk Page Format

Similarly, each FID has a set of *file properties* (see Figure 2). The system uses some of this space to record the status of the file directory entry (free, allocated, deleted, expunged). The client cannot change these. Other properties are cooperatively maintained by the system and its clients. Whenever a file is dirtied, WFS sets the file's *dirty* bit. A client that desires higher reliability may backup dirty files and then clear this bit. Finally, some space is reserved for client-private uses; WFS does not touch these properties.

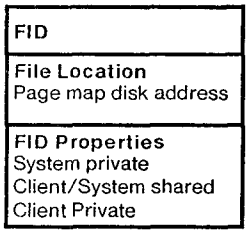

Figure 2. FID Directory Entry

2.4 File locks

A client may lock a file, preventing access by anyone without the proper key. The lock operation returns a key that must be supplied with all subsequent operations on the file, until either the client ussues an unlock operation or the lock breaks. WFS will break a file's lock if no operation has been performed on the file for a minute or so. A system restart breaks all locks. A key of zero fits an unlocked file. A client can detect a broken lock because the non-zero key will not fit the lock on an unlocked file.

key	*lock*	*access*	*file state*
0	0	allowed	unlocked
0	X	denied	locked
X	X	allowed	locked
X	Y	denied	locked
X	0	denied	unlocked

These locking operations provide primitives that are adequate to implement completely safe sharing mechanisms (see section 4.2.)

2.5 Communications Protocol

Within the Xerox research community, the foundation for process-to-process communication is an internetwork packet (or *datagram*), as opposed to a stream (or *virtual circuit*) [Boggs *et al*]. However, many of the applications that use the Xerox internetwork choose to hide the packet boundaries and to assure reliable transmission by means of a stream facility constructed from the packet protocols. A stream is an example of a connection-based protocol: a substantial amount of state must be correctly maintained at both ends for the duration of the connection.

The WFS protocol, on the other hand, is based on the direct transmission of internetwork packets, and does not rely on the reliable delivery of every packet. WFS provides an example of a connectionless protocol: the

server maintains no state between packets, and the client maintains very little—often none.

To perform a WFS operation, a client constructs a *request* packet containing the operation code and any necessary parameters, and sends it to the selected WFS host (see Figure 3). WFS processes commands in the order in which they arrive and then returns a *response* packet to the sender. The response contains the requested data or a failure code. The server is entirely passive: it never initiates activity, but only responds to requests.

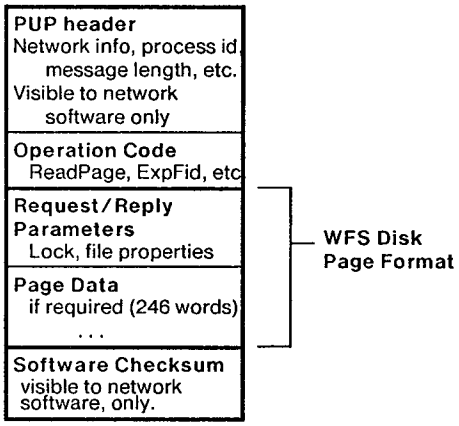

Figure 3. Request/Acknowledgment Packets

Since the reliable delivery of request packets and their responses is not guaranteed, the client must take the appropriate steps to assure robust performance. It usually suffices to retransmit a request if a reasonable interval has elapsed without receiving its response. The operations are designed so that any write action will have the same effect if it is repeated. In addition, it must not be possible for packets to be delayed for so long that write and read operations can occur out of order without detection. This behavior is not difficult to arrange in our environment, but would have to be dealt with if the methods were generalized.

2.6 File System Implementation

WFS is written in BCPL [Richards], supported by a simple custom-tailored operating system and communications package.

For each file, WFS maintains a *page map* that translates client page numbers into physical disk addresses and identifies unallocated pages. Depending on the current length of the file, the page map is either one or two levels deep (see Figure 4).

The FID directory is a hash table implemented as a contiguous, fixed-size file at a known disk address. Entries in the directory associate FIDs with their corresponding file properties and top-level page map locations.

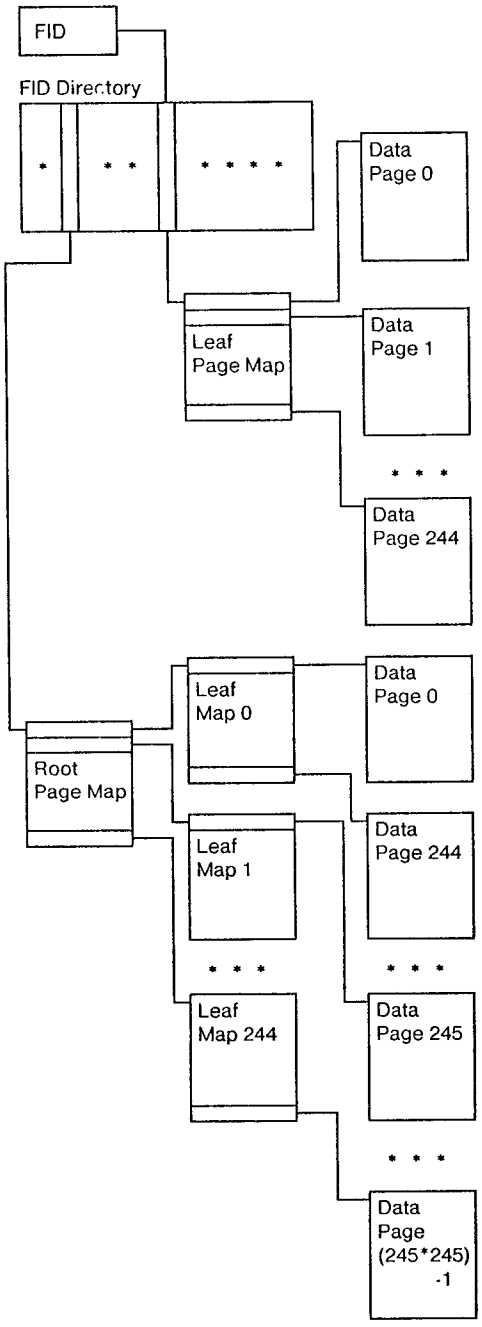

Figure 4. WFS File Structure. Small files use a single page map level, while larger files use a two level map. Empty data pages are not allocated on the disk.

A single process interprets client operations in the WFS server. This process sequentially extracts request packets from the network input queue, checks them for validity, and dispatches to the indicated operation. When the operation completes, the process returns a response packet to the requesting client. By using this simple, sequential scheme, lockup behavior is impossible, and starvation (unfair treatment of a particular client) is very unlikely.

During a write operation, WFS reads the specified data page (and in some cases auxiliary pages) before writing it, in order to validate its FID, page number, and other page properties. If a discrepancy is found, the operation is rejected (see section 2.5.) The system writes the data into its assigned disk page immediately, before returning the acknowledgment packet.

Although a WFS application will occasionally make closely spaced references to the same data page, such references are not frequent enough to warrant special treatment. However, multiple references to auxiliary disk pages (page maps, directories, and allocation bit tables) predominate. For this reason, WFS uses a substantial percentage of main memory as a write-through cache of recently referenced disk pages. Discarding the least recently referenced page whenever cache space is needed favors retention of the auxiliary pages, while accommodating the infrequent case of closely spaced accesses to the same data page.

Since pages to be changed are always written immediately, the cache is entirely redundant and is maintained for efficiency only; any page of it, or all of it, can be discarded for any reason (including a system crash) without affecting the integrity of the file system.

2.7 Performance

WFS has never been used in an environment subject to a high volume of concurrent accesses by a large number of hosts. However, we did measure its performance under a heavy load generated by one to three hosts running the *Woodstock* application. Table 2 provides the performance figures obtained from these tests (see [McDaniel] regarding the network-based instrumentation tool). The table compares both reading and writing times of WFS with times obtained by performing the same activities using the local disk. The WFS times include the cost of the client's service routines that provide packet composition, transmission and response interpretation activities as well as the actual WFS software and disk access times. In each case, one or more Woodstock users manually produced a very high request rate. While the table doesn't detail this observation, we found that the network transmission times through the high-bandwidth Ethernet local network [Metcalfe-Boggs] were negligible. Measurements of subsequent server/client configurations have produced comparable results.

Write operations yielded poorer results than read operations in the tests because WFS reads data pages to validate them before writing new contents (see section 2.6).

In the single-user (lightly loaded) case, WFS improved Woodstock's average input response time over the local disk's time for several reasons: WFS's disks were faster than Woodstock's local disks, requested pages were sometimes still in the WFS main memory cache, and the amount of arm motion on the local disk was reduced because it no longer had to seek between a code swap-area and the user data area.

In general, performance has been adequate for a number of nontrivial applications. Notice that the measurements exhibit nearly linear degradation with increasing load. A system implementing more sophisticated scheduling methods could improve this performance.

Read Page	AVG	MIN	MAX
Using Local Disk	60	30	90
WFS with one user	48	20	260
with two users	76	20	330
with three users	100	20	330

All times in milliseconds

Write Page	AVG	MIN	MAX
Using Local Disk	47	10	110
WFS with one user	73	30	260
with two users	109	30	350
with three users	150	40	420

Table 2. WFS Performance Observations. In multiple-user experiments, system users manually produced extremely demanding loads. Maximum load for the same number of users could be somewhat greater.

3. Design Philosophy

The principle theme of the WFS design is that client programs must provide the higher-level abstractions usually associated with file systems, while WFS implements a simple, low-level abstraction with relatively few operations and with high reliability. Low-level, reliable file service in WFS stems from its passive, *atomic* operations which are characterized by the following properties:

- Each operation may access at most one data page, and no more than a few auxiliary disk pages.
- Each operation runs to completion before WFS acknowledges it. A write operation is not complete until the data is on the disk. Between operations, WFS retains no state information that can not be regenerated from the contents of the disk.
- Command and protocol boundaries are the same—each command and response comprises a single internet packet.
- Clients access the server through connectionless protocols—each packet proceeds independently over the network.

The receipt of a command acknowledgment is an assurance that the overall integrity of the file system is correct at the "virtual disk" level. This means that a subsequent crash recovery or other reinitialization in either the client or the server will be invisible except for a possible time delay. Although this approach places additional burdens on the client and ultimately limits the efficiency of deletion and copy operations, it simplifies the protocol design by limiting operations and responses to single packets. It also improves the ease with which a reasonable and fair response to client requests can be guaranteed. We believe this property was crucial to meeting our time constraints for implementing Woodstock.

The connectionless protocol frees WFS from the requirements of maintaining communication state information during client interactions, and reduces the work clients must do to communicate with WFS. Since we have found that the size and computing overhead of high-level communication code often exceeds that needed to provde the higher-level abstractions, this reduction becomes more important when client programs are implemented on personal computers which may not be particularly powerful.

If the client receives an acknowledgment for a write request, then the write operation has clearly occurred. The write algorithms are also constructed to reduce the possibility that the state of the file system can become inconsistent at the file and page level. Therefore, our atomic property provides a high probability, but not an absolute guarantee, that an unacknowledged write request has been performed either in its entirety or not at all. The WFS system and protocol have no facilities for assuring that higher-level transactions involving changes to multiple data pages have this property, although a client-based algorithm can achieve this goal [Paxton].

4. Functional Capabilities and Implications

This section examines the extent to which the WFS design can support generally useful file system activities. We first look at uses that do not involve the sharing of files, then extend the discussion to shared applications. Finally, we consider the comparative cost to the client of using WFS instead of a more functionally rich system.

4.1 Single User Applications

We contend that, for uses that do not involve sharing, WFS is functionally sufficient, since a more traditional system (e.g., character-level I/O and directory functions) can be built using the "virtual disk" provided by the page access operations. A single implementation of these facilities might well satisfy the needs of a number of applications. Our application was Woodstock; other applications are described elsewhere [Paxton], [Shoch-Weyer].

Clients must provide their own naming and file directory structures. If an application creates a file and forgets the FID returned by WFS, the file is lost, although client programs can be written to scan the FID directory and find it again. The Woodstock application implements a directory by keeping FIDs "hidden" in text files where document names are referenced. Since the FID is a sufficient handle to access the file, Woodstock can easily and efficiently find a file regardless of the context of its reference. Other applications have made quite different arrangements, all of which are of no concern to WFS.

We have found that it is straightforward to rewrite device drivers using network communications rather than driving the disk directly. Since WFS makes no assumptions about the structure of application files except that they are a sequence of pages, specific file structures are conventions enforced only by the application. For example, the conversion of Woodstock to WFS instead of a local disk required no file structure modifications.

As indicated in section 2.5, some network configurations can lead to the arrival of so-called *delayed duplicate* packets, which can cause write and read operations to occur out of order. The rather primitive communication protocols in WFS would need to be augmented for the system be be usable in an environment where this behavior was possible. One approach would be to retain sufficient mutual state information between client and server hosts (i.e., a simple connection) that packets arriving out of order could be detected and discarded. The packet sequence numbers used to detect delayed or fraudulent packets would be allowed to repeat only over extremely long intervals (months or years.) See [Lampson-Sturgis] for an example of this approach.

4.2 Shared Applications

In examining WFS's ability to support shared access to files, it is useful to consider the following three categories of file system state:

> Long-term information endures throughout a file's lifetime or longer. Examples are the data files themselves, the system allocation tables, and the FID directory.
>
> Medium-term information is retained across atomic operations. The timeout lock that enables the sharing of data is the only medium-term state WFS keeps, whereas traditional file servers also maintain medium-term information associated with communication connections, open files, and the like.
>
> Short-term information is the state that must be kept during the execution of an atomic operation. In WFS, though there may be large amounts of such information, all that state may be discarded after an operation completes without sacrificing the integrity of the file system.

Clearly, the maintenance of medium-term information is necessary for any reasonable set of file system facilities. We believe that the client can maintain all such information, except for that required to enable the locking of data when shared access is possible. The goal is an overall improvement in the size and cleanliness of the total system.

WFS's medium-term lock information must also be augmented by client activities to obtain file sharing with behavior that can be guaranteed. While Woodstock's approach to sharing is quite primitive, Paxton discusses the design of a file system that uses WFS as its base and that provides reliable shared access to user files [Paxton]. Clark describes time limit locks in a shared resource system. In his system, IO device routines implemented on top of a virtual memory facility must implement reliable service, in the presence of memory locks which will break after their time limits expire [Clark].. The DFS [Israel *et al*] system uses time limit locks as part of its approach to sharing, although DFS itself handles lock timeouts.

4.3 Cost Considerations

Implementing the higher abstractions on client machines costs them code space and execution time, although much of this expense is recovered because the interface to the server is simpler. Correspondingly, WFS saves code space which it may use for disk buffers, and saves execution time which it may provide to more users.

Our insistence upon the atomic operations property has led to some objectionable inefficiencies. An obvious example is the requirement that clients deallocate files, one page at a time, in order to delete them. Another drawback is that there is no provision for high-speed access to consecutive pages. In section 5.2 we suggest some simple extensions to handle these kinds of operations.

5. Possible Extensions

5.1 Privacy and Security

Any host that can communicate with WFS has full access to all operations on all WFS files. Thus, security *cannot* be guaranteed, and privacy can be guaranteed only if the application encrypts everything. In this area alone WFS is not adequate to meet the functional needs of a generally useful file server (see [Birrell-Needham] for a discussion about the attributes of a universal file server).

For our experimental applications, the absence of server-enforced security was reasonable, because security and privacy were supplied by application programs. Again, we were willing to impose more responsibility on the client, in return for the flexibility to experiment with different user-level protection schemes, or to defer protection issues altogether.

Methods for communications privacy and for access control would have to be added to WFS to achieve acceptable security in a more hostile environment. By

applying recent work in both these areas, this could be accomplished without affecting the simplicity or robustness of the current design.

Communications privacy (see [Kent] for a general discussion) can be supplied by a number of encryption approaches, and can be compatible with the atomic, connectionless design of WFS. The methods developed in [Needham-Schroeder] and [Rivest *et al*] are particularly relevant to this application.

Flexible use of a file server causes more problems than an encryption system can handle easily, but they are problems that a capability-based access mechanism can solve [Birrell-Needham]. One reasonable approach for WFS would adapt a method, described in [Needham], for adding capability access to a conventional file server that has login authentication. To perform an operation, a client would now have to present an unforgeable capability for a file instead of the file's FID. The file system would create and return such a capability in response to a file creation request from an authenticated user. This initial capability would allow the possessor arbitrary access to the file. Additional operations would allow the client to request different capabilities for the same file, with restricted access rights (e.g., read-only). Such capabilities could be passed safely to other users. Clients would use these capability facilities to produce applications exhibiting the desired user-level protection.

WFS would implement these capabilities as records encrypted with a private key. The records would include the FID and the file access rights associated with the capability. The capability generated at file creation time would grant full rights to the creator. This approach would allow WFS to locate the relevant FID, check access, etc., by merely decoding the incoming capability, without the need for additional information. The required user authentication could be handled by supplying an operation that would return a "user identification capability" when presented with a user name and correct password.

In this section we have discussed minor extensions to WFS that would increase the privacy and security of its transactions without sacrificing the partitioning of client and server responsibilities. However, to build into the server the additional transaction-based interface that Paxton produced in the client machine [Paxton] would require a fundamental redesign. Systems that provide capability or transaction-based facilities at the server level are reported in [Needham-Birrell], [Israel et al], and [Birrell-Needham].

5.2 Changes for Efficiency

The performance of WFS is ultimately limited by one of its strengths: the independence of each page-level request. When it is known that an application will require the successive use of a substantial number of contiguous file pages, much better performance would be possible if this knowledge could be used to optimize their transfer to and from the disk. One way to do this would involve extending the command set to include an explicit statement that a range of pages will be needed, counting on the server's page caching methods to transfer them efficiently into its main memory in advance of their use. Another method would not involve any new commands, but would require the elaboration of the command interpreter to allow the processing of more than one incoming operation at a time. Information about sequential disk access could be passed on to the disk-management level, where the same efficient transfer scheduling decisions could be made.

Although the network software and hardware delays are smaller than disk access time, they are not negligible. The latter method above, allowing multiple outstanding requests, could also result in an average increase in network throughput.

If the basic page transfer performance were improved, one major source of inefficiency would remain: the absence of operations for deleting entire files, copying their contents, etc. These operations were omitted in order to guarantee the client response times and file integrity properties discussed at length above. It would be straightforward to spawn a process within WFS to submit successive page-level requests (at the same priority as client requests) until the task was complete. System integrity at the virtual disk level would not be impaired, although a server crash could prevent the file-level task from completing (see [Lampson-Sturgis] for a more robust approach to the system crash problem). The server could acknowledge the operation either on receipt of the request or on final termination; both approaches are problematical, since they violate the atomic property in one way or another. An alternative would be for the client to retain the burden of sequencing these activities, but to speed them up using one of the bulk-transfer methods proposed above.

While none of the methods discussed in this section have been tried, we are confident that their application would result in a shared page-level file system with very impressive overall performance.

6. WFS Applications

In addition to the Woodstock system, now defunct, whose requirements drove the development of WFS, a number of applications have been built that continue to use WFS for their files. Two of them are described in separate articles (see [Paxton] and [Shoch-Weyer].)

A final example of an application with a set of higher-level characteristics different from Woodstock is an implementation of an experimental telephone directory data base. This application uses entirely different naming structures and access methods than Woodstock does, but can coexist with other WFS-based applications.

The telephone directory application runs on personal computers in the Xerox internetwork, providing access to approximately 40,000 entries. Each entry associates a name with a telephone number and other public information. All the entries are stored within a single WFS file with a fixed, known FID. The user supplies a key, and the application responds with one or more entries whose names match the key (the key is an initial substring). A typical single-entry query can be completed in approximately one-half second, reading an average of three WFS data pages.

For this simple application, a data base method using B-Trees [McCreight] was an obvious candidate. An available B-Tree package (which runs in the client machine) and WFS made an ideal combination: the former implements a particular high-level data structure, given operations that can read and write numbered data pages of any fixed size; the latter implements just these operations without in any way interpreting the contents of the pages.

7. Conclusion

We have demonstrated empirically that a very simple central file server, teamed with appropriate file system elaborations in the client host, can meet or exceed many of the capabilities of more comprehensive central facilities at acceptable cost to the client. Clients benefit from the flexibility and file system robustness resulting from this approach. Extensions to meet more stringent performance requirements and to provide adequate security seem possible without major modification to the design. Although this approach has been quite successful, it remains to be seen which of the possible partitionings of server-client functions will prove to be the most powerful and convenient.

References

[Birrell-Needham]
A. Birrell and R. Needham, A Universal File Server, to appear in *Communications of the ACM.*

[Boggs et al]
D. Boggs, J. Shoch, E. Taft, and R. Metcalfe, Pup: An Internetwork Architecture, to appear in *IEEE Transactions on Communication.*

[Clark]
D. Clark, *An Input/Output Architecture for Virtual Memory Computer Systems,* MIT MAC TR-117, January 1974.

[Israel et al]
J. Israel, J. Mitchell, and H. Sturgis, Separating Data from Function in a Distributed File System, *Proc. Second International Symposium on Operating Systems,* IRIA, Rocquencourt, France, October 1978; to appear in D. Lanciaux, ed., *Operating Systems,* North Holland.

[Kent]
S. Kent, *Encryption-based Protection for Interactive User-Computer Communication,* MIT MAC TR-162, May 1976.

[Lampson-Sturgis]
B. Lampson and H. Sturgis, Crash Recovery in a Distributed Data Storage System, to appear in *Communications of the ACM.*

[McCreight]
E. McCreight, Pagination of B*-Trees with Variable-Length Records, *Communications of the ACM,* 20(9):670-674, September 1977.

[McDaniel]
G. McDaniel, METRIC: A Kernel Instrumentation System for Distributed Environments, *Operating Systems Review* 11(5):93-99, November 1977.

[Metcalfe-Boggs]
R. Metcalfe and D. Boggs, ETHERNET: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM,* 19(7):395-404, July 1976.

[Needham]
R. Needham, Adding Capability Access to Conventional File Servers, *Operating Systems Review,* 13(1):3-4, January 1979.

[Needham-Birrell]
R. Needham and A. Birrell, The CAP Filing System, *Operating Systems Review* 11(5):11-16, November 1977.

[Needham-Schroeder]
R. Needham and M. Schroeder, Using Encryption for Authentication in Large Networks of Computers, *Communications of the ACM,* 21(12):993-999, December 1978.

[Paxton]
W. Paxton, Client-Based Transactions to Maintain Data Integrity, *Proceedings of the Seventh Symposium on Operating System Principles,* 1979.

[Richards]
M. Richards, BCPL: A Tool for Compiler Writing and System Programming, *AFIPS Conference Proceedings* (SJCC) 35:557-566, 1969.

[Rivest et al]
R. Rivest, A. Shamir, and L. Adelman, A Method for Obtaining Digital Signatures and Public-key Cryptosystems, *Communications of the ACM,* 21(2):120-126, February 1978.

[Shoch-Weyer]
J. Shoch and S. Weyer, Page Level Access to a Network File Server from Smalltalk, to appear.

[Thacker *et al*]
C. Thacker, E. McCreight, B. Lampson, R. Sproull, and D. Boggs, Alto: A Personal Computer, *Computer Structures: Readings and Examples* (Siewiorek, Bell, and Newell, eds.), 1979, to appear.