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)	Read or write page properties and page data
WritePage(fid,pageNum,lock,page properties)	
File Management	
GetFID()	Allocates a new file and returns its fid
ExpFID(fid)	If fid has no pages allocated, expunges (deletes) the file
DeallocatePage(fid,lock, pageNum)	Releases storage for page and removes page from page map
Status Query/Modification	
GetFIDStatus(fid)	Return file status values
SetFIDStatus(fid,mask,value)	Set client status values, ignore attempt to affect system values
ReadPageMap(fid,lock, pageMapNumber)	Return page map information to determine which pages are allocated
Lock(fid)	Return key, required in subsequent operations until file is unlocked
UnLock(fid)	Unlock file (set lock to zero)
Maintenance	
ReallocFID(fid)	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.
ResetLastFID(newFid)	
ReadRealPage(realAddress)	
GetVMap()	
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.

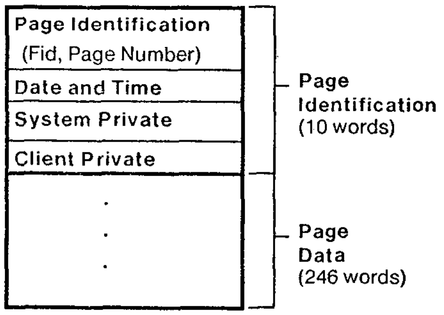

Figure 1. WFS Disk Page Format

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.

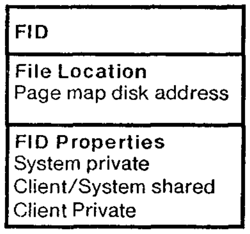

Figure 2. FID Directory Entry

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.

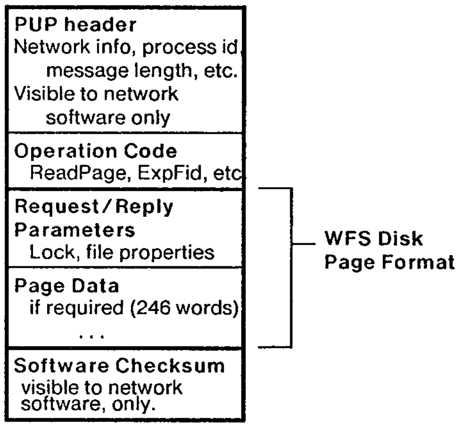

Figure 3. Request/Acknowledgment Packets

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.

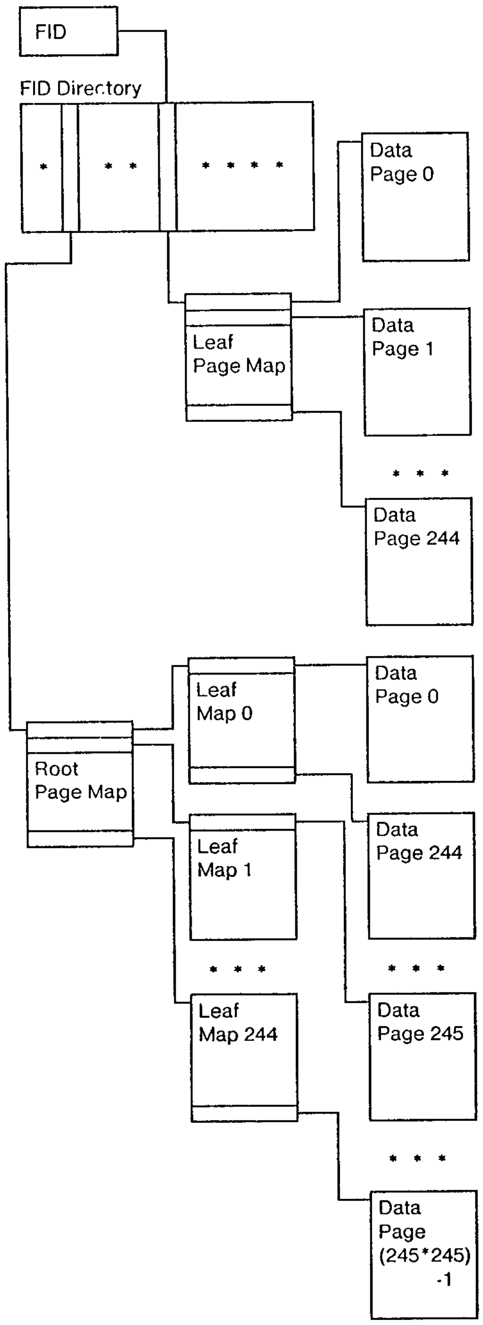

Figure 4. WFS File Structure. Small files use a single page map level, while larger files use a two level map. Empty data pages are not allocated on the disk.

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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.