The Object-Oriented Advantage in Prototyping a Remote File System

Michael W. Condry and Swee Boon Lim and Lup Yuen Lee*
Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

Remote file system caching tends to follow one of two schools of thought: whole file caching and block caching. A problem with either approach is that the most appropriate caching strategy for an application potentially will vary, even for files within a single application. We have prototyped a remote file system for the Choices object-oriented operating system that permits the caching strategy to be user selectable on a per file basis. Choices provides a convenient object-oriented toolkit for building file systems, which we employed, reusing code whenever possible. An application can suggest the most suitable approach based on analysis of its file's behavior; the file system then caches the file accordingly.

In our file system the client provides the driving force in the architecture. The server maintains a small cache, fulfills requests, and performs callbacks. The client supports both whole file caching and block caching. Only the client needs to be aware of the type of caching being used for a particular file. Different clients can provide different caching strategies at the same time but the data within a client it is kept consistent. The server and client cooperate to maintain the consistency of the file system via callbacks.

Because of the object-oriented architecture of the Choices file system we were able to prototype our system in approximately 6 weeks. There was a significant amount of code reuse.

1 Introduction

The architecture of remote file systems tends to be focused toward a community of users with the performance optimized for the "average" user. Caching of files in the file system is a typical example of such an architectural component that are global and associated with the file system. Two strategies predominate, whole file caching and block caching. Whole file caching, such as in Andrew[9, 4, 10] and AFS, keeps a copy of the entire file being accessed in local client disk space and returns it to the server (if necessary) when "finished". Block caching, used in NFS[7, 8] and Sprite[5, 3, 20] caches only a set of blocks for the file in both the server and possibly the client. A problem with either approach is that the most appropriate caching strategy for an application's file is not necessarily represented since the strategy is focused on the average file. The most effective approach potentially will vary, even for different files within a single application.

Our remote file system architecture allows the application to select the file caching strategy for every remote file it uses. When a remote file is opened the application indicates what it expects the most desirable form of caching method to be — based on an analysis of the application. The file system uses this method suggestion along with the state of the system to determine the most appropriate caching strategy that will be used by the file. The strategy recommended by the application is the first choice but the system may not be able to provide this choice or it may have to change. For example, whole file caching may be requested but in a diskless workstation environment only blocked based file caching can be supported. Similarly, with multiple requests of a different type the system must elect one, even if the method has to change during operation.

Our remote file system was implemented for the Choices object-oriented operating system[2]. The Choices object-oriented file system[12] enables us to reuse code from the file system and allows us to prototype our remote file system rapidly. This paper provides an overview of the current architecture of our

*Supported in part by a grant from AT&T Bell Laboratories and the National Center for Supercomputer Applications

remote file system for Choices, examines the current design of the Choices file system framework and provides some early analysis.

2 Choices facilities

Our remote file system is built upon the file system framework currently in Choices and an new interprocess communication (IPC) mechanism that we added.

2.1 Choices IPC

The remote file system is implemented with a simple reliable network-transparent IPC system provided by Choices[11]. The main objects in the system are Messages and Boxes, which are analogous to letters and mailboxes in the real world. In this IPC scheme, a Message is the fundamental unit of communication. A Message contains a finite length byte array, used to transmit data from one process to another. Messages are untyped – the IPC scheme does not enforce any particular encoding of the byte array enclosed in a Message. The size of a Message is limited by the packet size of the underlying transmission medium. In the present implementation of the IPC system based on Ethernet, Messages may be no longer than 1500 bytes.

A Box is a FIFO queue for holding incoming Messages. Messages may be retrieved from a Box only by processes which are in the same domain (i.e. virtual memory space) as the process which created the Box. The "proxy" of a Box may be exported by a process to another to allow the other processes to send Messages to the Box. Messages may be sent to a Box proxy but retrieval of Messages from a proxy is disallowed. Box proxies are exported via the reply-to field of Messages or the global nameserver. For example, in the client-server scenario of a distributed computer system, when the client sends a request Message to the server, the client may pass a Box proxy through the Message's reply-to field to the server. In the same scenario, a server may advertize its Box through the global nameserver so that clients may fetch the Box proxy from the nameserver and use the proxy to communicate with the server.

2.2 Prototyping file systems in Choices

The Choices object-oriented file system[12, 13] is structured as a framework of abstract base classes that represent generic file system entities (such as files, directories, disk partitions and disks). These abstract classes are specialized to implement specific file systems; the System V UNIX[18, 1], BSD UNIX[14], AIX and MS-DOS[15] file systems in Choices have been implemented this way. Our remote file system was built in the same way – we specialized abstract classes to represent remote file system entities, such as remote files and directories.

We have found numerous benefits in using the file system framework, which is essentially a toolkit for designing and building efficient implementations of file systems. The framework allows new file systems to be prototyped quickly due its object-oriented nature. For example, while implementing our remote file system, we maximized the reuse of existing code by adding new classes at the right depth in the class hierarchy, thereby reducing the amount of new code to be written and cutting down development time. Another benefit of using the object-oriented framework is that the new classes that we have introduced may be subclassed in future to prototype other types of remote file systems quickly.

The modularity enforced by the object-oriented framework helps to prevent the development of our file system from affecting the other existing file systems, while allowing multiple file systems to co-exist in a single computer. The framework also allows our remote file system to mount different file systems remotely (eg. a client that uses the BSD file system may mount a remote MS-DOS file system). Because all file systems share the same user interface within the framework, we were able to test our remote file system by running existing application programs without any modifications.

2.3 Extensions to the file system framework

There are two main classes of file system objects involved in the remote file system implementation: FileObjects and MemoryObjectDictionaries. FileObjects and MemoryObjectDictionaries in Choices are analogous to the UNIX concept of files and directories, respectively. FileObject and MemoryObjectDictionary are abstract classes whose concrete subclasses represent files and directories in specific file systems. For example, the Choices implementation of the BSD filesystem uses the classes BSDInode and BSDDirectory, which are subclasses of FileObject and MemoryObjectDictionary, respectively.

FileObjects generalize the notion of a file – a FileObject is a directly-accessible array of bytes in a persistent store. FileObjects support the read and write operations for accessing the data in the persis-

tent array. MemoryObjectDictionaries are containers that hold other MemoryObjectDictionaries or FileObjects. MemoryObjectDictionaries support the lookup operation, which searches for an object inside the MemoryObjectDictionary when given the name of the object. This operation is equivalent to a directory lookup.

For the remote file system, the FileObject and MemoryObjectDictionary classes were specialized to represent remote files and directories. RemoteFileObject is a subclass of FileObject that is instantiated on the client host only. A RemoteFileObject corresponds to the client's view of a remote file. Read and write operations on a RemoteFileObject are directed to a local cache first, and if the cache is missed, the operations are forwarded to the remote file server through a message-passing protocol based on the IPC scheme described earlier.

RemoteDictionary is a subclass of MemoryObjectDictionary that is instantiated on the client host only. A RemoteDictionary encapsulates the client's notion of a specific directory on a particular remote file server. In order to carry out the RemoteDictionary's lookup operation, the client and server hosts communicate with each other through the file service protocol.

3 File system architecture

Our file system performs client caching, server caching, and maintains consistency of files. File system consistency is achieved via a concurrency control protocol and a cache coherence protocol. The concurrency control protocol ensures that there are no conflicting accesses. The cache coherence protocol ensures that the client caches are correct.

3.1 Client caching

A unique feature of our file system is the availability of two client caching strategies, i.e. block caching and whole file caching. Our block caching strategy works like that of Sprite, where blocks of a file are cached in an in-memory cache. Our cache consists of a set of fixed size buffers. Whenever, there is a cache miss blocks are fetched from the server to satisfy the request. Our whole file caching strategy works like Andrew, a local copy is maintained on a local disk cache and requests are serviced by the in-memory cache if possible. If there is an in-memory cache miss, the request is serviced using the local copy.

The desired caching strategy provided by the file's user should be a hint. This is because the actual caching strategy may change as a result of other hints provided by another user of the same file. In our case, the file system may 'upgrade' block caching to whole file caching. Suppose process A opens file X requesting block caching, then process B opens file X requesting whole-file caching. As a result of B's request, file X will be copied from the server to the client. If process A accesses a block which results in an in-memory cache miss, there is no reason for the client to get this block from the server to satisfy the request, once there is a local copy of file X on the client. Even though process A has initially specified block caching as the desired strategy, the caching strategy has been upgraded to whole file caching as a result of process B's caching strategy hint. Conversely, if process A opens file X requesting whole file caching and process B opens file X requesting block caching, B's request can be satisfied with the local copy of X. Hence, process B's block caching hint is ignored and whole file caching is used. In our remote file system, whole file caching supersedes block caching. In other words, once a file has been opened for whole file caching, all previous and subsequent opens will be upgraded to whole file caching on that server.

Another feature of our design is that the client caching strategy is completely independent of the server. The client is solely responsible for the caching strategy. This scheme allows different caching strategies or a selected set of caching strategies to be implemented on different clients. For example, on diskless workstations, only block caching needs to be implemented. Server independent client caching also allows us to experiment with other client caching strategies without modifying the server. It also allows a single server to support workstations with diverse client caching strategies.

3.2 Concurrency control

Concurrency control ensures that there are no conflicting accesses which will compromise the integrity of the remote file system. We have chosen to implement a simple form of concurrency control since we do not expect much contention. Our primary concern is the integrity of the file system. Our current concurrency control scheme is to enforce the multiple-readers/single-writer (MRSW) policy. An access mode is associated with each open request. This access mode indicates whether the file will be modified. If the file will be modified, we consider it an open for writing, otherwise an open for reading. We enforce MRSW by blocking open requests which will result in conflicting accesses. These blocked opens will be un-

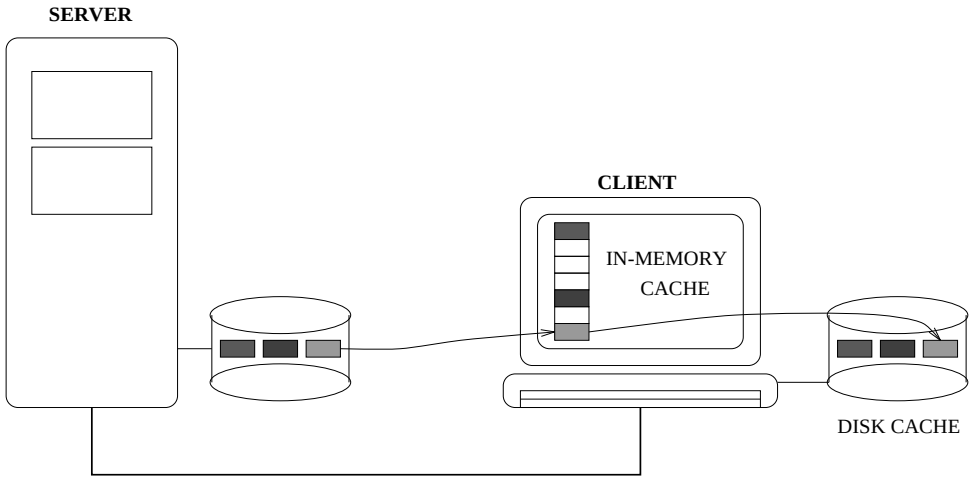


Figure 1: The caching facility

blocked only when the desired access can be granted. A blocked open is usually unblocked as a result of a close corresponding to the earlier open of the file. To ensure fairness, open requests are satisfied in FIFO order.

3.3 Cache coherence

We have elected to use a simple scheme for cache coherence. In our system the server is responsible for cache coherence. Only the server knows which clients has parts of a file cached. The server uses callbacks to inform the client to invalidate its cache and/or flush its cache back to the server. The client simply obeys these orders and complies as soon as possible. The client may not be able to comply with these callback orders immediately since the file may still be active at the time of the callback. For example, the client can only invalidate the cache after the file has been closed by all its users.

A client always flushes its dirty blocks back to the server. This may be a performance disadvantage since dirty blocks must be sent back to the server before they can be obtained by another client. However, this scheme does simplify the design of the file system and since we do not expect much contention, this scheme will suffice. Another advantage is that the server can keep track of changes to the file while the dirty blocks are being copied from one client to the next via the server.

3.4 Timeout mechanism

A file's client cache can be invalidated and/or flushed on the server's request via a callback if there are other conflicting requests for the file. However, this is not always true, and another mechanism is needed to invalidate and/or flush a file's client cache when they are no callbacks. We use a timeout mechanism to invalidate and/or flush stale files caches. A file's cache is stale when the file has not been used by any application on the client machine (i.e. inactive) for a fixed time period. We do not invalidate or flush a file's cache immediately when the file becomes inactive, since the file may be accessed again soon by another process on the same client. Therefore, we start a timeout timer once the file becomes inactive and when the timer expires, the client voluntarily invalidates and/or flushes the file's cache on the client and informs the server of these actions.

4 Implementation

In this section, we describe the mechanisms we used to implement the communication between the client and server, our concurrency control scheme, and our cache coherence protocol.

4.1 File service protocol

We have defined a communication protocol, based on Choices IPC, for use by remote file servers and and their clients. The protocol consists of several Remote Procedure Calls (RPC), where each client request requires a response. The following types of RPCs have been defined:

1. *Lookup* - checks whether a remote file exists and returns the file's status. The file may be created

on the server if it does not exist. This operation returns a handle for the remote file which will be used by the client or server to identify this file in subsequent RPCs or callbacks.

2. *Open* - open the file for access. An access mode is associated with each open. This access mode is used by the server to schedule accesses to control concurrency.

3. *Read* - read blocks from the file specified by a file handle.

4. *Write* - write blocks to the file specified by a file handle.

5. *Close* - used by the client to indicate to the server the status of a file's cache. This status contains the following flags: Invalidated, Flushed, HandleInvalidated.

6. *Keys* - used by the RemoteDictionary to obtain a listing of files in a particular directory on the server.

In addition to the above, a *Callback* message has been defined. The callback message specifies the file handle and a flag describing the action that the client should perform on the file's client cache. For example, the server would send the client a callback message to request that the client flush a particular file's cache back to the server and/or invalidate the cache.

4.2 Concurrency control

Our concurrency control scheme depends on the blocking of conflicting requests to maintain MRSW and FIFO fairness. The decision to block and unblock an open request can be made by both the server or the client. The client makes this decision when it can determine from its local state whether the open request will result in conflicts. Otherwise, the client consults the server, and the server makes the decision. We implement our concurrency control scheme by maintaining two queues on the server for each remote file, and a queue on each client for each remote file. The server maintains the active queue and the pending queue. The active queue contains the file handles of all active clients, that are defined as clients which currently have the file open. The pending queue is a queue of file handles which have requested to open the file but these opens requests must be blocked because of conflicting accesses. The client queue contains open requests which are active on the client, open requests which the client has decided to unblock in the near future, and callbacks received by the client.

The client can make a local decision when the new open request has the same access mode as the open requests on the client's queue and there are no callbacks on the queue. Callbacks on the client queue indicate conflicting open requests. Otherwise, the client cannot determine locally whether an open request should be blocked or unblocked; the client has to consult the server via a *Open* RPC request. The server will arbitrate the request and send a reply when the open request can be unblocked. The following events illustrate the operation of client and server queues:

1. Process A on client C1 opens file X for reading
2. Process B on client C1 opens file X for reading
3. Process C on client C2 opens file X for writing
4. Processes A and B close X
5. Process D on client C2 opens file X for writing
6. Process C closes X
7. Process E on client C1 opens file X for writing
8. Process D closes X
9. Process F on client C1 opens file X for reading

Stage 1: When process A opens X, the client queue for X on C1 will be empty, so C1 consults the server. This is required since C1 does not know whether X is currently used by another client. The server knows that X is not currently in use, so it places C1's handle for X on the active queue and sends a reply to C1. C1 places A's open request on the client queue for X and process A can proceed with its file access.

Stage 2: When process B opens X, C1 can determine locally that process B's open request need not be blocked and the server need not be consulted since the server already knows C1 has opened X for reading. Also, since C1 has not received any callbacks, meaning that there are no other conflicting open requests, C1 is safe to grant file X's read access to process B. B's open request is placed on the client queue.

Stage 3: When process C opens X on another client C2, C2's queue is empty. Hence, C2 consults the server. Since the server knows that C1 has X open for reading, it must block C2's request. The server places C2's file handle for X on the pending queue for X and sends a callback message to C1. When C1 receives the callback message, C1 records the callback in its queue. The callback cannot be serviced immediately because

process A and B are still using X. Consequently the callback is recorded in the queue for later servicing.

Stage 4: Eventually, A and B will close X and their open entries will be removed from the client queue. The callback entry is now at the head of the client queue. C1 services this callback by performing a *Close* RPC which indicates to the server that the C1 is done with X. After receiving the *Close* RPC request, the server removes C1's handle from the active queue. The server examines the pending list and determines that C2 is waiting to open X for writing, so C2's handle is placed on the active list and an *Open* RPC reply is sent to C2. Upon receipt of the reply, C2 places C's open request in X's client queue.

Stage 5: When D requests to open X for writing, C2 can determine locally that D's open request should be unblocked once C has closed X since there are no callbacks (i.e. other requests). D's open request is placed on the client's queue and D is blocked.

Stages 6, 7 and 8: When C closes X, the entry for C is removed from C2's queue for X. C2 notices that the entry for D is now at the head of the queue and subsequently unblocks D's open request. Similarly, when E opens X for writing, the request is placed on the server's pending list and the server sends a callback to C2. C2 records the callback on its queue and performs a *Close* RPC when D closes X. Upon receiving the *Close* RPC request from C2, the server removes C2 from the active list, removes C1 from the pending list, places C1 on the active list, and sends an *Open* RPC reply to C1.

Stage 9: When F opens X for reading, C1 must consult the server and the server will place C1's request on the pending list. Even though C1 can determine locally that F's open request can be unblocked as soon as E's has closed X, it must consult the server since the server has granted read/write access to C1 and not read-only access. This additional communication with the server is required to notify the server of the change in access mode such that the server can allow concurrent readers. Conversely, if E requests to open X for reading and F requests to open X for writing, the server must be consulted since C1 cannot determine locally if there are any other clients reading X.

The rules for scheduling a new open request on the server are as follows:

1. If the pending queue is not empty, or open for writing is requested, or open for reading is requested but the active queue contains an entry for write, then place the new request on the pending queue, else place the new request on the active queue.

2. If the new entry is placed on the active queue, send an *Open* RPC reply (i.e. the open request is not blocked).

3. If the new entry is at head of the pending queue, then send the appropriate callback messages to all clients in the active queue.

The rules for unblocking requests when a *Close* RPC request is received are as follows:

1. Remove the client's entry from the active queue.

2. If the active queue is not empty, then do nothing.

3. If the active queue is empty, then examine the pending queue. If the head of the pending queue is an open for reading request, then move the all consecutive open for reading requests from the the head of the pending queue to the active queue. If the head of the pending queue is an open for write request, then move that request to the active queue. Send replies to all clients in the active queue.

4. If the pending queue is not empty, examine the entry at the head of the pending queue and send appropriate callbacks to all clients in the active list.

4.3 Cache coherence

The cache coherence protocol relies on the callback mechanism. The server uses the active queues to determine the files which are cached by each client. For each file, the active queue for that file contains all the clients which are using the file, i.e. have the file cached. The server does not know how the file is cached, it only knows that the client has cached the file. The server pending list together with the active list determines the appropriate callbacks to sent. Each callback message from the server contains a file handle to identify the file of interest to the client and two flags indicating the action that the client should perform on its cache. The flags are:

1. *Invalidate* - whether the client should invalidate its cache.

2. *Flush* - whether the client should flush the dirty blocks back to the server.

These flags may be combined to request both a flush and invalidation of the client cache for a particular file. In the above example, the server will send an invalidate callback to C1 when process C requests to open X for writing. Since C1 has X open for reading, there is no need to flush dirty blocks. When process E requests to open X for writing, the server sends a flush and invalidate callback to C2. When process F opens X for reading, the server will send a flush callback to C1. There is no need to invalidate C1's cache since C1 may open it for reading soon, but the cache should be flushed back to the server so that another client may open the file for reading.

The general rules for issuing callbacks on the server based on the active and pending queues are as follows:

1. If a reader is active and a writer is pending, send invalidate callback(s).
2. If a writer is active and a reader is pending, send flush callback.
3. If a writer is active and another writer is pending, send invalidate and flush callbacks.

The callback mechanism is closely tied to the concurrency control scheme. A callback is always issued as soon as possible, i.e. as soon as a conflict is detected. The callback is placed on the client's queue so that the client can service the callback as soon as the file is closed by all users on the client. The callback indicates to the client that there are conflicting accesses and no further local decisions should be made on the client. Since the caching strategy is server independent, the client is responsible for carrying out the appropriate actions specified by the callback. For example, if the file is block cached, only the in-memory cache needs to be searched to invalidate and/or flush the blocks belonging to the file. If the file is whole file cached, then the local copy will be copied back to the server, if modified, then the local cache will be invalidated.

5 Client design

Figure 2 illustrates the relationship between the components of the client side of the remote file system. The RemoteFileSystemCache maintains the in-memory cache and the local disk cache. If a block of a remote file is required, a find operation is issued to the RemoteFileSystemCache. The RemoteFileSystemCache cache will search the in-memory cache for the block. If there is an in-memory cache miss, a buffer for the block is allocated. Depending on a flag, the find operation may copy the block from the server or from the local copy into the buffer, or it may fill the buffer with zeros if the block is supposed to be overwritten later. The find operation also locks the buffer until the buffer is released, after which the buffer may be marked to indicate that the buffer has been modified (i.e. dirty). Eventually, the RemoteFileSystemCache will run out of in-memory buffers, and a Least-Recently-Used buffer replacement strategy will used. If the block to be replaced is not dirty, it is invalidated, otherwise it is flushed to the local copy or to the file server, depending on whether the file is whole cached or block cached.

A RemoteFileEntry is like a UNIX Inode for remote files. It is contains information about a remote file, such as the name of the file, the server Box for communicating with the server, the file handle issued by the server, the caching strategy, the location of the local copy if there is one, and the client queue of the file. The RemoteFileTable simply maintains a table of RemoteFileEntries and enables the callbackDaemon to locate the client queue for a particular file. The RemoteFileTable is similar to the in-memory UNIX Inode table, except it is used only by remote files.

As mentioned earlier, the remote file system is integrated into the Choices object-oriented file system framework using the RemoteFileObject and RemoteDictionary classes. RemoteFileObject methods such as open, close, read, and write have been redefined to support remote operations. For example, the open method participates in concurrency control and maintains the client queue for the file. The close method monitors the client queue and may start a timeout timer if no one else is using the file. The read and write methods obtain blocks/buffers from the RemoteFileSystemCache, which may generate RPC requests to the server to obtain blocks which are not cached by the client. Similarly, we redefined the methods of MemoryObjectDictionary to support the listing and lookup of remote directories.

In addition to the above objects, there are two daemons. The callbackDaemon receives callbacks from the file system servers and records every callback into the client queue of the appropriate file. The timerDaemon is used to invalidate and/or flush client file caches. A file's client cache is invalidated and/or flushed when the timeout timer expires (i.e. the file is stale) or when there is a callback entry on the file's client queue. In the former case, the timer is started when all users have closed the file. In the latter case, when the last user has closed the file, the close operation forces the

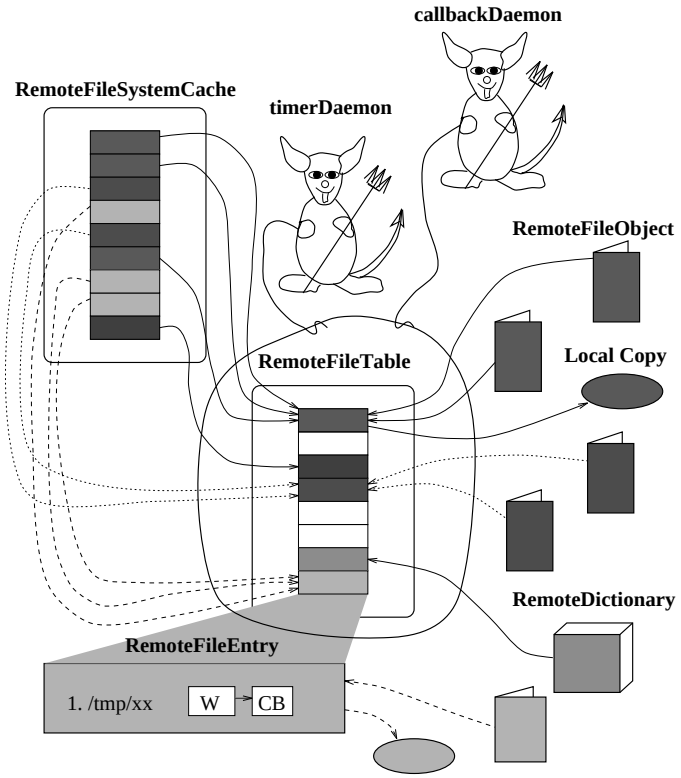


Figure 2: Client components

timer to expire, signalling the timerDaemon to flush and invalidate the file as soon as possible.

6 Server design

The remote file server consists of a single process (called the File Server Controller) that interprets incoming request Messages and performs file operations accordingly. The server process accesses the following components:

1. Server Box: This Box stores request Messages sent by clients. The server advertizes its Server Box through the global nameserver.

2. Data Boxes: When a client wishes to write some data to a file, it sends a *Write* request Message to the Server Box accompanied by the data to be written, stored within the same Message. If the data does not fit into a single Message, the left-over data is sent in multiple Messages to the Data Box. The server maintains a Data Box for each active file handle and informs the client about the Data Box through the reply of the *Lookup* request.

3. File Access Scheduler: The File Access Scheduler encapsulates the concurrency control and cache coherence policies in the file server, and controls the opening of files and the issuing of callbacks. *Open* and *Close* requests are passed by the server process to the File Access Scheduler and replies to *Open* requests are held back until the Scheduler has determined that the file is safe to be used by the client, with regard to concurrency control and cache coherence. The Scheduler maintains the active and pending file access queues described earlier.

4. Prefetch Cache: The server maintains a fixed-size cache for each active file handle. This cache is optimized for the sequential reading of data from files. All *Read* requests go through the cache first, and in the case of a cache miss, the server fills the entire cache by reading in consecutive blocks from the file. *Read* requests that access the last block stored in the cache will also cause the cache to be refilled with subsequent file blocks.

Figure 3 illustrates the flow of Messages and data among the file server components. In this diagram, springs represent processes, while slotted boxes and

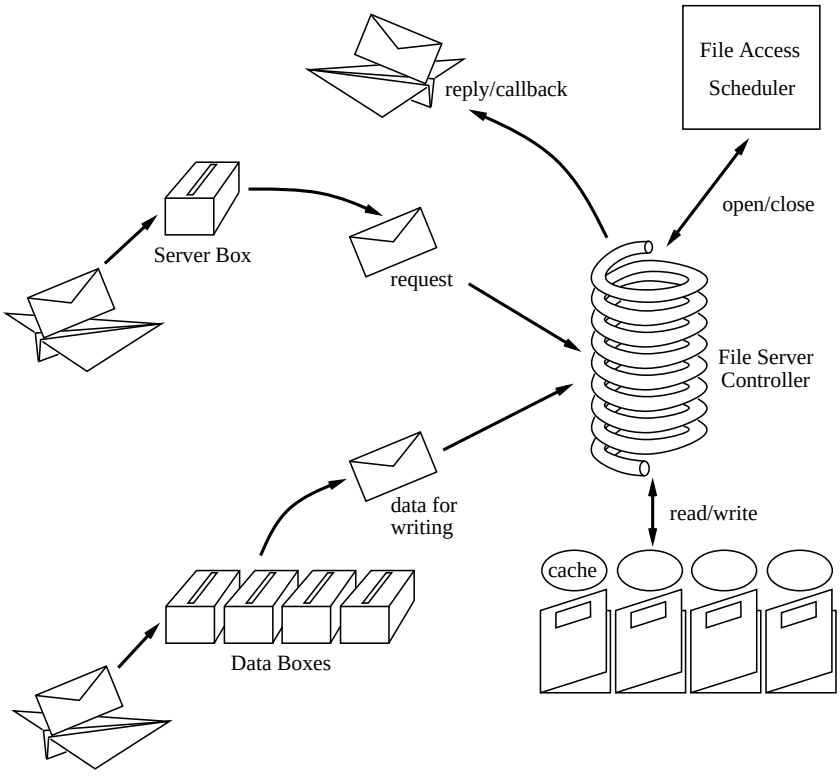


Figure 3: Server components

envelopes represent Boxes and Messages in the Choices IPC scheme. Paper planes denote the remote delivery of Messages through the Ethernet.

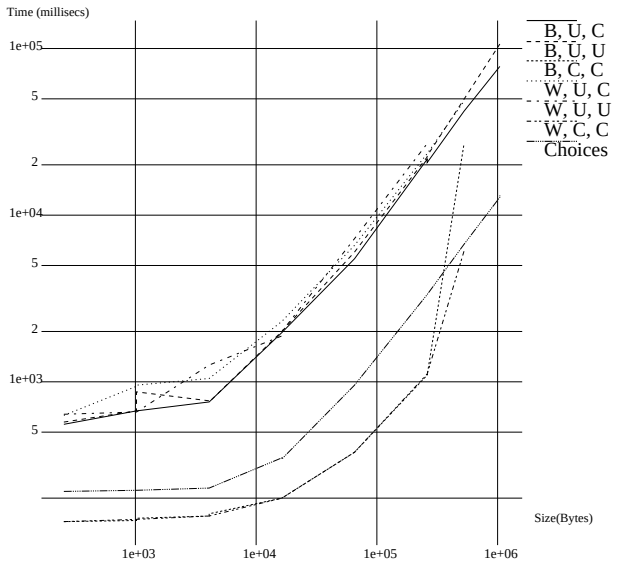
7 Analysis

At the time of this writing we have only a few performance results. Figure 4 exhibits the performance due to caching. The graph plots the time taken to read whole files of varying lengths. The file sizes range from 256 bytes to 1 megabyte, while the times range from 145 milliseconds to 106 seconds.

The performance data we have obtained indicate that our server caching scheme degrades the performance of remote file access under certain conditions. This suggests that server caching should be a client or application tunable parameter in the remote file system in order to achieve maximum performance. The caching strategy employed by the remote file server is currently restricted to the prefetching of a fixed number of consecutive blocks from the file, but to suit different types of file access patterns we should provide other server caching strategies. For example, sequential file access may benefit from the prefetching strategy currently implemented, but this strategy may cause severe performance degradation when used for demand-paged program execution, which has been observed to produce wildly random file access patterns. Ideally, an application should be able to supply a hint to the file server to select a particular prefetching strategy that will work best with the application's file access characteristics.

We would like to implement an new access mode, to represent write-over access, in addition to the read and modify access modes already provided by our remote file system. The new access mode would allow further optimization of our cache coherence scheme. For example, the flush operation could be eliminated if an open write-over (eg. open and truncate) request followed an open modify or write-over request. At the moment, we are still evaluating the effectiveness of the our current cache coherence scheme. As mentioned earlier, we do not expect much contention and hence the additional complexity may not be justified.

We experienced many benefits from the object-oriented method of constructing our remote file system. The object-oriented approach in prototyping file systems has helped in producing our file system in a short period of time: six weeks by a three-person team



B, U, C:	Block cache, cold cache, server caching enabled
B, U, U:	Block cache, cold cache, server caching disabled
B, C, C:	Block cache, warm cache, server caching enabled
W, U, C:	Whole file cache, cold cache, server caching enabled
W, U, U:	Whole file cache, cold cache, server caching disabled
W, C, C:	Whole file cache, warm cache, server caching enabled
Choices:	Accessing file on local disk

Figure 4: File access time

(of which two people were unfamiliar with the file system framework prior to the project), including time for writing and running test suites and benchmarks. The remote file system was built with SPARC Choices[16] and at present runs on a cluster of six Sun SPARC IPC workstations (with 8 megabytes of RAM each) connected by Ethernet. Due to the object-oriented nature of Choices, all file system code we have written will port easily to other versions of Choices (e.g. Choices for the multiprocessor Encore Multimax). We discovered that the object-oriented file system framework is indeed flexible enough to accomodate many types of file systems, especially our remote file system, which is different in many respects from the other file systems that are currently in Choices.

8 Summary

We rapidly prototyped an object-oriented remote file system for Choices. A focus of our remote file system was to address the problem of remote file system caching in an object-oriented manner, encapuslating the caching with the file object rather than with the file system. This allowed us to offer both block and whole file caching and permit the most appropriate one be provided to the application. Our file system is built on the Choices operating system, an existing object-oriented operating system.

Our remote file system follows a client-server architecture. The client provides the driving force in the design of this system. The server maintains a small cache, fulfills requests, and performs callbacks. The client supports both whole file caching and block caching. Only the client needs to be aware of the type of caching being used for a particular file. The server and client cooperate to maintain the consistency of the file system via callbacks.

Using the object-oriented capabilities of Choices, we reused much of the file system code leading us to an operational prototype in approximately 6 weeks. This exhibits a productive use of object-oriented designs.

References

[1] M.J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[2] R. Campbell, G. Johnston, and V. Russo. Choices (Class Hierarchial Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, July 1987.

[3] J. K. Ousterhout et al. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proc. of 10th Symp. on Operating System Principles*, pages 15–24, Dec 1985.

[4] J.H. Howard et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb 1988.

[5] Mary G. Baker et al. Measurements of a distributed file system. In *Proc. of 13th Symp. on Operating System Principles*, pages 198–212, Oct 1991.

[6] David K. Gifford, Roger M. Needham, and Michail D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, March 1988.

[7] Sun Networking Group. NFS: Network File System protocol specification. *Sun Microsystems RFC 1057*, Mar 1989.

[8] Chet Juszczak. Improving the performance and correctness of an NFS server. In *Proc. 1989 Usenix Winter Conference*, pages 53–63, Dec 1989.

[9] M.L. Kazar. Synchronization and caching issues in the Andrew File System. *USENIX Winter Conference*, 1988.

[10] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *SIGOPS 91*, 1991.

[11] Lup Yuen Lee. PC-Choices Object-Oriented Operating System. Master's thesis, University of Illinois, August 1992.

[12] Peter W. Madany. An Object-oriented Framework for File Systems. Technical Report UIUCDCS-R-92-1751, University of Illinois, June 1992.

[13] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[14] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[15] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC.* Microsoft Press, 10700 Northrup Way, Box 97200, Bellevue, Washington 98009, 1985.

[16] David K. Raila. The Choices Object-Oriented Operating System on the SPARC Architecture. Master's thesis, University of Illinois, May 1992.

[17] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh. RFS Architectural Overview. In *Proceedings of the Summer 1986 USENIX Conference*, pages 248–259, Atlanta, Georgia, 1986.

[18] K. Thompson. Unix Implementation. *Bell System Technical Journal*, 57(6):1931–1946, July 1978.

[19] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. *ACM Operating Systems Review*, 17(5):49–69, October 1983.

[20] M.N. Nelson B.B. Welch and J.K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, Feb 1988.