Διακομιστες κρυφης αποθηκευσης με διατηρηση τοπικοτητας δεδομενων για κατανεμημενα συστηματα αρχειων

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από τη Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τη

Λαμπρινή Κώνστα

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Φεβρουάριος 2009

Dedication

To my lovely family...

Acknowledgements

At this point, i would like to mention all those people that strongly supported me during the design and implementation of this work.

I wish to thank my supervisor, Prof. Stergios Anastasiadis, for his significant guidance and precious advice throughout this research. I am mostly grateful to my family that contiguously encouraged and supported me. I would also like to thank the members of the Systems Research Group *(SRG)* at the University of Ioannina, for the perfect collaboration. Especially, Andromachi Hatzieleftheriou and Giorgos Margaritis who were always willing to discuss different issues that helped me improve several checkpoints of my thesis.

Finally, it should be noted that the work presented in this thesis was supported in part by the project Interreg IIIA Greece-Italy 2000-2006 Grant No I2101005.

Table of Contents

List of Figures

List of Tables

Abstract

Lamprini K. Konsta, MSc, Computer Science Department, University of Ioannina, Greece. February, 2009. Hades: Locality-aware Proxy Caching for Distributed File Systems.

Thesis Supervisor: Stergios V. Anastasiadis.

Recent trends in business and research collaboration encourage secure data sharing over wide area networks with the minimal intervention from the end user. Alhough traditional file transfer mechanisms have been used for secure data sharing over the last decades, they face the main disadvantage of getting the user to explicitly initiate the whole transfer mechanism, which bears significant replica bookkeeping overhead to the user. As an alternative, caching proxies have been lately introduced to reduce WAN latency by caching data closer to the client.

In this thesis, we propose alternative storage management issues in caching proxy servers for distributed file systems, based on Andrew File System. We organize the requested data at the disks of the proxy server using locality-aware approaches. Additionally, we introduce improvements in the mapping mechanism from remote to local data and consider cost-aware replacement methods. Thus, we succeed to improve existing performance of retrieving files from proxy's disk cache, especially in the case of concurrent file accesses. In a prototype implementation that we developed, we experimentally compare alternative distributed file systems as components of the proxy servers. Through extensive measurements, we demonstrate throughput improvements at the proxy server up to 80% in comparison to the disk-based cache of Andrew File System.

Εκτεταμενη Περιληψη

Λαμπρινή Κώνστα του Κωνσταντίνου και της Γιαννούλας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος, 2009. Διακομιστές κρυφής αποθήκευσης με διατήρηση τοπικότητας δεδομένων για κατανεμημένα συστήματα αρχείων.

Επιβλέπων: Στέργιος Αναστασιάδης.

Οι σύγχρονες ανάγκες επιχειρηματικής και ερευνητικής συνεργασίας ενθαρρύνουν την αυτοματοποιημένη και ασφαλή κοινοχρησία δεδομένων πάνω από δίκτυα ευρείας περιοχής με ελάχιστη εμπλοκή των τελικών χρηστών. Η απαίτηση από τον τελικό χρήστη να αντιγράφει τα δεδομένα κοντά στους υπολογιστικούς πόρους με την εκτέλεση εντολών μεταφοράς αρχείων δημιουργεί σημαντική επιβάρυνση και καθυστερήσεις. Συνεπώς, θα ήταν προτιμότερη η ύπαρξη ενός διακομιστή κρυφής αποθήκευσης που θα αντιγράφει αυτόματα τα απομακρυσμένα δεδομένα και θα αποκρύπτει τις καθυστερήσεις μεταφοράς κατά τις επαναληπτικές χρήσεις των δεδομένων. Η δημιουργία διακομιστών κρυφής αποθήκευσης για κατανεμημένα συστήματα αρχείων ελκύει αρκετά το ενδιαφέρον της ερευνητικής κοινότητας τα τελευταία χρόνια, κυρίως ως προς την κατεύθυνση της διασύνδεσης υπαρχόντων συστημάτων αρχείων με τοπικά συστήματα αρχείων με τελικό σκοπό την αποδοτικότερη αποθήκευση απομακρυσμένων δεδομένων.

Τα περισσότερα συστήματα αρχείων σχεδιάστηκαν αρχικά για να εξυπηρετήσουν τις αποθηκευτικές ανάγκες χρηστών σε τοπικά συστήματα αρχείων. Έτσι, η κρυφή αποθήκευση από τη μεριά του χρήστη περιοριζόταν μόνο στην κύρια μνήμη, καθιστώντας μη αναγκαία την κρυφή αποθήκευση στο σκληρό του δίσκο. Μια χαρακτηριστική εξαίρεση αποτελεί το Andrew File System, ένα κατανεμημένο σύστημα αρχείων που δίνει τη δυνατότητα της προσωρινής αποθήκευσης δεδομένων στο τοπικό σύστημα του πελάτη, για μεγαλύτερη διαθεσιμότητα σε κατανεμημένα συστήματα αρχείων.

Το Andrew File System έχει χρησιμοποιηθεί επιτυχώς τις τελευταίες δύο δεκαετίες

κυρίως σε συστήματα γενικού σκοπού. Τα δεδομένα μεταφέρονται από τους απομακρυσμένους εξυπηρετητές ανα τμήματα σταθερού μεγέθους και αποθηκεύονται στην κρυφή μνήμη στο δίσκο του πελάτη. Κάθε απομακρυσμένο τμήμα δεδομένων αποθηκεύεται σε ένα μόνο τοπικό αρχείο. Ωστόσο, σε σύγχρονα επιστημονικά περιβάλλοντα είναι πολύ συχνή η ύπαρξη πολλαπλών αιτήσεων για πολύ μικρά ή αρκετά μεγάλα απομακρυσμένα αρχεία. Τότε, η υπάρχουσα προσέγγιση του Andrew File System μπορεί να μην είναι αρκετά αποδοτική, κυρίως ως προς τη διαχείριση δεδομένων και μεταδεδομένων.

Στην παρούσα εργασία, διερευνούμε θέματα αποθήκευσης δεδομένων σε ενδιάμεσους διακομιστές για κατανεμημένα συστήματα αρχείων. Αντιγράφουμε αυτομάτως τα δεδομένα που ζητούνται από τον αρχικό διακομιστή αρχείων στους δίσκους του ενδιάμεσου διακομιστή κρυφής αποθήκευσης. Εκεί οργανώνουμε τα δεδομένα με τεχνικές διατήρησης της τοπικότητας. Επιπλέον, εισάγουμε βελτιώσεις στο μηχανισμό απεικόνισης των απομακρυσμένων δεδομένων τοπικά και λαμβάνουμε υπόψη θέματα αντικατάστασης των δεδομένων. Σε μια πρωτότυπη υλοποίηση που αναπτύξαμε, εξετάζουμε τις παραπάνω βελτιώσεις με εναλλακτικά συστήματα αρχείων ως μέρη του ενδιάμεσου διακομιστή. Με εκτεταμένες μετρήσεις που πραγματοποιήσαμε διαπιστώνουμε βελτίωση στη ρυθμαπόδοση του ενδιάμεσου διακομιστή μέχρι 80% σε σύγκριση με το Andrew File System.

Chapter 1

Introduction

1.1 Thesis scope

Recent trends in business and research collaboration encourage secure data sharing over wide-area networks, aiming at achieving the best possible performance with the minimal intervention from the end user. Traditional file transfer mechanisms such as FTP have long be used for secure data and file transferring. However, such mechanisms face the main disadvantage of getting the user to explicitly initiate the whole transfer mechanism, which bears significant replica bookkeeping overhead to the user and only makes data usable after an entire file has been fully replicated locally. Thus, caching proxies have been lately introduced as an alternative approach, to reduce WAN latency by caching data closer to the client. Such proxies automatically replicate datasets and hide transfer delays during the repetitive use of data. Nache is a representative example of a caching proxy server for NFSv4, designed to retain a consistent cache of remote file servers in a distributed environment in order to improve file accesses performance by redirecting requests that were initially intended for the file server to the intermediate caching proxy [4]. In fact, the design of caching proxies for distributed filesystems is mainly attracting

research interest in the direction of getting existing filesystems interoperable with local file systems for persistent caching purposes. Therefore, we explore basic storage management issues in caching proxy servers for distributed file systems. Our main goal is to point out the need for efficient storage management in caching proxies so that we make performance of accessing cached data from proxies comparable to or better than direct accesses from local file system.

Traditional distributed filesystems were originally designed for serving the storage needs of users within the same organization at a single geographical site. The assumed use of a local-area network limited client-side caching to main memory and made unnecessary the corresponding disk-based caching. However, wide area networks may introduce latencies that may be orders of magnitudes greater than direct disk accesses. Such long latencies encourage the design of a client-side disk cache for effective storage management. Andrew File System and its descendants make a notable exception as they provide the capability to temporarily store data at the local file system of the client machine for improved scalability and availability in distributed environments [14].

Although Andrew can achieve effective storage management in distributed environments, it makes the assumption that the client machines from individual users are powerful enough to relieve centralized servers from computations. Nevertheless, this is not the case when building caching proxies for data sharing among large numbers of clients within an organization. Therefore, it is essential to build an effective caching proxy that would reduce server's load while it is not based on the above assumption.

In our proposed architecture, we investigate alternative locality-aware storage management methods to improve Andrew's efficiency and performance. Although Andrew has been widely successful for over two decades in general file system use, it does not offer a proxy caching service as it limits caching to the local file system. Initially, it creates a large number of individual local files at the client and subsequently uses each of them to store an individual chunk requested from the server. Remote data is replicated in chunks of a configurable fixed size. However, in modern scientific and business environments it is common to have numerous small files or enormously large ones. Then, the existing AFS approach of having a separate local file per chunk might not be the best possible in terms of data access or metadata management efficiency.

The most widely used method in recent published literature is to map each remote file

to a local file in the caching proxy [4,16]. It offers a consistent view of remote data as it appears at the remote server. On the other hand, web caching proxies can store multiple remote files per local cache file. They manage local data in a way that serves their design objectives and have already been broadly used for over a decade in content distribution networks. Originally, copies of web pages requested by users were replicated on proxy servers close to the web browsers over traditional local file systems. However, related experimentation in published literature demonstrated several performance deficiencies related to metadata management of multiple small files, frequent creation and deletion of files, excessive disk head movement from poor clustering of jointly used data or access overheads from multiple small writes. Subsequently, customized file systems emerged that complementarily addressed the above issues through special internal architectures and new access interfaces. On the contrary, we claim that apart from offering a consistent view of the data as they appear at the origin server, the caching proxy should be free to manage its local data in whatever way serves its design objectives better. Therefore, we propose innovative storage management methods, combining existing approaches to manage either multiple remote small or large files. Furthermore, we consider data replacement issues to enhance existing performance.

In the present thesis, we propose alternative storage management issues in proxy servers for distributed file systems, based on Andrew File System. We organize the requested data at the disks of the proxy server using locality-aware approaches. Additionally, we introduce improvements in the mapping mechanism from remote to local data and consider cost-aware replacement methods. Thus, we succeed to improve existing performance of retrieving files from proxy's disk cache, especially in the case of concurrent file accesses. In a prototype implementation that we developed, we experimentally compare alternative distributed file systems as components of the proxy servers. Through extensive measurements, we demonstrate throughput improvements at the proxy server up to 80% in comparison to the disk-based cache of a commonly used distributed file system.

1.2 Thesis outline

The remainder of this thesis is organized as follows:

In chapter 2, a majority of caching facilities for distributed file systems is presented. We review previous related search in the area of caching systems for distributed filesystems. Initially, client-server architectures where caching is done primarily at the client's disk cache are examined. Then, we present caching proxies that relieve centralized file-servers and improve performance by caching data closer to the client. The storage allocation methods that each system uses are examined. Furthermore, some pre-existing modifications made to the AFS Cache Manager are displayed. Finally, we depict how web caching proxies manage their storage space to improve performance.

In chapter 3, an overview of Andrew File System is presented. The storage management that AFS uses is described as well as the basic AFS structures that we modified in Hades implementation.

In chapter 4, we present the basic design issues that emerge in proxy servers and led to our prototype implementation. We detcet the design inefficiencies of existing caching systems, including Andrew File System. Furthemore, we define the design goals of our study along with our architectural decisions, emphasizing on storage space management and file replacement. Finally, we present an overview of the proposed architecture.

In chapter 5, we introduce the design and implementation of the Hades proxy server. Hades is a locality-aware caching proxy for distributed file systems that was implemented as a combination of a modified OpenAFS client and a regular user-level NFS server. Then, the modifications made to the OpenAFS client are thoroughly examined, emphasizing on the storage management and replacement methods.

In chapter 6, we evaluate our implemetation across a microbenchmark and an actual application. We make extensive experimental evaluation on the parallel retrieval of remote files and the reuse of cached data across multiple clients. According to our allocation algorithm, data are clustered in proxy cache in such a way that leads to throughput improvements at the proxy server up to 80% in comparison to the disk cache that OpenAFS uses.

Finally, in chapter 7 we outline our conclusions and future work.

Chapter 2

Related Research

In this section, we describe approaches that have been previously proposed in order to achieve high performance in distributing file systems when we need to access data available from remote file servers. Furthermore, we review previous research related with disk-based caching as well as caching proxies that lie between clients and fileservers. Next, we examine storage allocation methods that have already been proposed to effectively manage data that are cached in the local disk caches. Finally, we present recent research related to data and metadata management in web caching proxies.

2.1 Caching in network filesystems

Caching has been a well-accepted solution for effective file storage during the last decades. Most popular distributed filesystems use a cache to gain performance improvement. Especially for network filesystems in a distributed environment, client-side performance heavily depends on the number of RPC calls that are made to the servers. It is worth mentioning

that in network-based filesystems, the latencies that are introduced by the network are orders of magnitudes greater than direct disk access. Thus, in order to improve performance one should try to minimize these latencies, using a client-side cache or one that would simply cache data closer to the user, so as to reduce the need to go to the network. The basic idea is the following: when a client needs to operate on a file of a remote file server, it should make an RPC call to the server only the first time he accesses the file and then hold a valid copy of it in the appropriate cache, so as to make future operations on the file aim at its copy in the cache. As a result, network traffic and server load are reduced. Furthermore, disconnected operation is better supported because even though a server loses connection with the network, the client may still have copies of the server's files in its cache. Generally, for both network and non-network based filesystems, caches on a faster medium improve performance as they reduce the amount of traffic to the slower medium.

2.1.1 Caching facilities for distributed filesystems

A variety of caching facilities for filesystems has been introduced lately. Most distributed filesystems rely on a client-server architecture where caching is done primarily at the client. Network filesystems like Andrew File System, DFS, Network File system (NFS) and Coda, support client-side caching. NFS enforces only weak-cache consistency. However, in some later NFS versions, like NFSv4, the Sprite cache consistency protocols were used to improve cache consistency by using server callbacks. Although NFS is primarily used in LANs, Andrew is better used for file-sharing in WANs. It supports client-side file caching and cache consistency through callbacks. What is more, there exist some kernel facilities, like FS-Cache, that can be used by network filesystems to take advantage of persistent local storage to cache data and metadata. FS-Cache uses CacheFS as its major caching source for storing and retrieving data.

Except from client-side caching, there have been introduced caching proxies who lie between local clients and remote fileservers. In wide area networks, a caching proxy that would cache data closer to the clients is preferrable as it reduces the need to acquire data from the remote fileserver. Such proxies intend to enable a consistent cache of the fileservers' files so as to generally improve client performance by bringing the data closer

to the client. Nache is a representative example of such systems.

Apart from caching proxies, efficient research has been done in the area of web caching proxies. Individual cache files or general-purpose file systems are used to store one or more URLs that are required by web clients and fetched from several web servers.

2.2 Storage allocation in related caching systems

A large number of disk-cache storage management methods have been introduced in recent published literature. In general, the overall time to access cached data from a caching proxy may vary according to the way data are placed in the proxy's disk cache. Early systems copied entire files from the file server to the client. This approach, originally used in Andrew, was problematic because it incurred high transfer latency and large resource requirements at the client. In later approaches, the designers adopted partial caching approaches. One possible solution manages the remote files in fixed-size chunks that it copies and stores onto corresponding individual local files at the client. That is the current allocation technique of Andrew File System.

However, in more recent prototypes the system dynamically replicates the directory and file naming structure from the origin server to the cache. It also transfers the file contents on demand in pages of configurable size. Locally, the system uses a typical file system or a raw disk partition to temporarily store the data of the cache. FS-Cache is a representative example that was recently introduced by Howells to be used by a network file system to cache data on local disks [5].

2.2.1 FS-Cache

FS-Cache is a kernel facility that can be used by network file systems to achieve effective data caching. It improves client performance and reduces network traffic as it avoids accessing the network to acquire remote server files. It gives the client the opportunity to cache locally files that it fetches from remote file servers. It is primarily designed for use with network file systems, such as AFS, NFS and CIFS. It can support different types of cache that have different trade-offs while it puts little overhead to the client file system. There are two types of cache: CacheFS and CacheFiles. They are used for

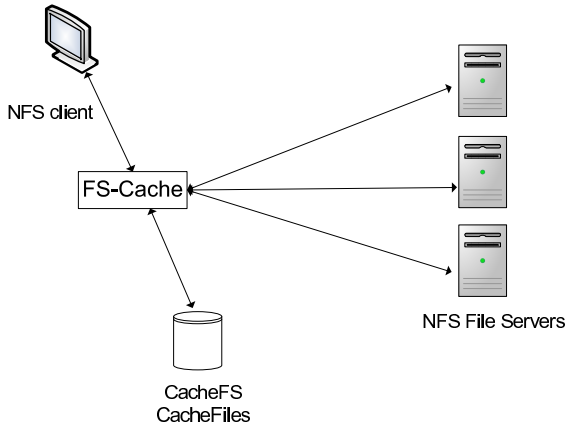


Figure 2.1: FS-Cache architecture.

storing and retrieving data. FS-Cache forwards the requests that are issued from the network filesystems to the available data caches (figure 2.1). Both caches can be added or removed at any time. CacheFS uses a block device as a cache. The block device can be mounted using the mount system call to make the cache available. If the cache is not needed any longer, it can be deactivated using the umount system call. CacheFiles defers from CacheFS in that it does not use a block device as its cache but a directory in an already mounted filesystem. CacheFiles is usually used when CacheFS cannot be used, probably because we are not able to acquire a block device. It uses the VFS/VM filesystem interfaces to get another filesystem (such as Ext3) to do the necessary I/O on its behalf.

Client filesystems can use FS-Cache to obtain caching services. FS-Cache is a thin layer in the kernel that directs the above requests for caching services to the available caches. Client file systems need not know the type of the attached caches as they get in contact with them through FS-Cache. If two different file systems issue requests for the same file, FS-Cache will face every request individually. The cache will finally have two different copies of the same file, which is known as cache aliasing. It is possible that the system will not have a cache at one time, or a remote file may be larger than the cache size limit. Thus, FS-Cache tries to ensure that a remote file will be available for use before it downloads and stores it in the cache. When a network file system requests a file, FS-Cache serves data out of the cache in pages. To access files in the cache, it is necessary to use sequences of keys, where keys are arbitrary sequences of binary data. To search for a file, one must examine the successive keys that correspond to indexes which

may lead to the required file.

A major characteristic of FS-Cache is that it can support disconnected operation. If the network comes unavailable, the network file system will be able to continue accessing the files through the available caches. When the network becomes available again, it can synchronize them with the server in case they were modified while we were working offline. To achieve disconnected operation, FS-Cache provides three facilities: reservations, pinning and auxiliary data. Reservations let the network file system reserve a chunk of the cache for a file, so that the file can be loaded or expanded up to the specified limit. Pinning guarantees that files would be available in the cache even when working offline. When a file is pinned in the cache, it is sure that it would not be removed from it so as to free space for other files. Auxiliary data permits the network file system to keep track of a certain amount of writeback control information in the cache.

2.2.2 Nache

Previous evaluations of systems using the FS-Cache facility showed some performance limitations due to double buffering across the local file system and the client of the network file system. Therefore, Gulati et al. implemented the Nache caching proxy for the NFSv4 [4]. The proxy uses an NFSv4 client to access the remote server, an NFSv4 server to reexport the client to the local users, and CacheFS to cache files in persistent storage (figure 2.2). However, in our proposed architecture we consider AFS as an alternative basis for building a proxy server. We modified the AFS proxy server to improve read performance of files stored at the origin server.

Nache is a caching proxy for NFSv4 designed to retain a consistent cache of remote file servers in a distributed environment. It can be shared between multiple local NFS clients who wish to access files of a remote NFS file server. The objective is to cache data closer to the clients so as to improve file accesses performance by redirecting requests that were initially intended for the file server to the intermediate cache proxy. The main idea is the following: should a client issue a request for a file located in a remote file server, it must first search if this file exists in its own cache and if not try to go to the network to acquire it. To avoid directing the request to the server which may cause the server to overload, the client directs the request to the intermediate cache proxy. If the required file exists in

the cache proxy, the client fetches the data from it and stores them in its local cache to satisfy future requests for this file. If the requested file does not exist in the cache proxy, the proxy forwards the request to the NFS server.

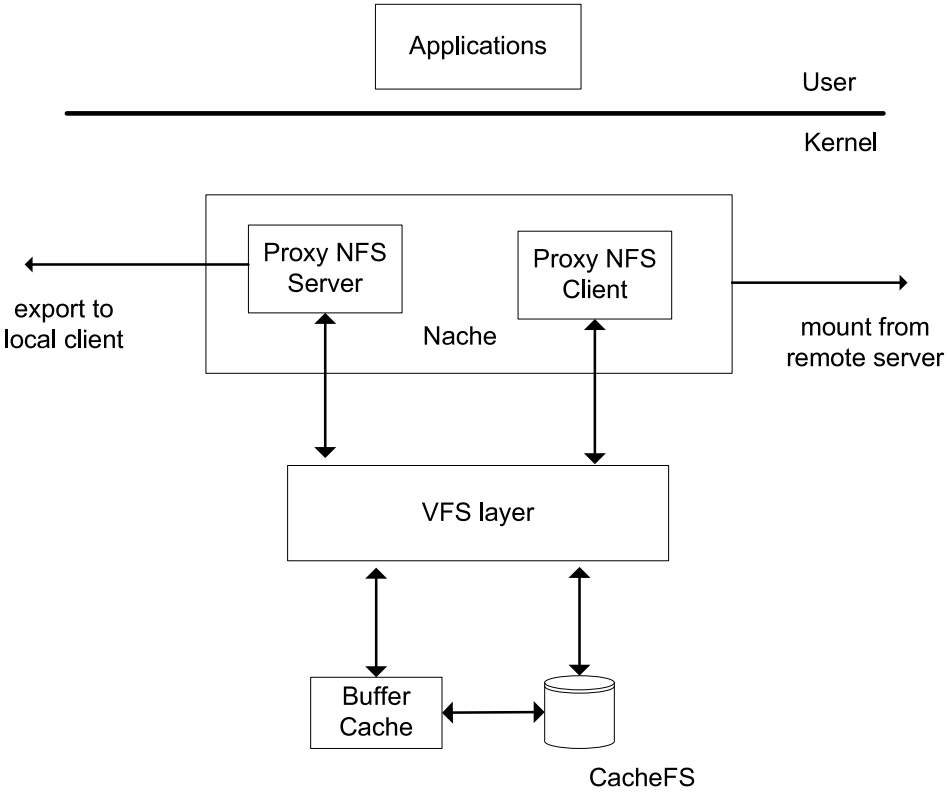


Figure 2.2: Nache architecture block diagram.

Thus, the cache proxy operates both as a server for the NFS clients and as a client for the remote NFS server. Its goal is to cache the remote data closer to the clients so as to reduce network traffic towards the server and improve the network latencies that clients face when they need to access remote files. The cache proxy server lies between the clients and the servers and is located much closer to the clients to reduce frequent WAN accesses.

The Proxy NFS client is responsible for the communication with the remote NFS server so as to mount the server's files. The proxy may mount either the root directory or one of its sub-directories. Then, the Proxy NFS server exports these files to the local clients. Those two components communicate via the VFS layer. Nache uses as its major cache the system's buffer cache but CacheFS is also used to add persistence to the cache. CacheFS is a caching facility available for use with NFS. It can be used to enhance the performance of a distributed file system such as NFS and uses a mounting protocol which presupposes

that the cache must manually be attached to each NFS mount after the mount has been made. CacheFS was originally designed for AFS but it can also be used by any other distributed file system. Its main feature is that it can cache any back file system on the front (local) file system. In Nache, CacheFS can be considered as an extension of the buffer cache that is located on the disk, not in the memory. CacheFS does not maintain the directory structure of the source filesystem. Instead, files are located in cache in the form of a database to make file search easier. A partition in a block device can be used for caching and a local mount point can be specified for it to let any file system (like NFS) mount it and discover the available cached files. The data size that can be stored in CacheFS can vary from few file pages to whole large files.

To be more specific, there exist some special client and server kernel modules that communicate through an unmodified VFS layer. When a client needs to access a remote file, it sends an RPC request for it to the cache proxy. The proxy has two types of modules: the server-side and the client-side module. The request is received from the proxy's server-side module that forwards it to the proxy's client-side module through a VFS interface. If the file does not exist in the cache, the client-side module forwards it to the remote server and stores the response to the client-side buffer cache. As a result, successive requests for the same file, or part of it, can be satisfied from the nearby proxy instead of the remote file server.

2.2.3 Recent caching systems

In more recent prototypes, the system dynamically replicates the directory and file naming structure from the origin server to the cache. It also transfers the file contents on demand in pages of configurable size [5,16]. Locally, the system uses a typical file system or a raw disk partition to temporarily store the data of the cache. Sivathanu and Zadok proposed the xCachefs framework that allows to persistently cache the data from any slow file system to a faster file system [16]. xCachefs offers a performance enhancement of 64% and 20% for normal read and read-write workload respectively over NFS. They use a directory structure at the cache as exact copy of the source file system, while we organize the cached data at the proxy in ways that improve storage locality.

Matthews et al considered the dynamic reorganization of the stored data in order to

improve the read performance of the log-structured file system [8]. In a different work, Vongsathorn and Carson proposed a disk subsystem that adaptively corrects the disparity between expected and actual access pattern by reorganizing the disk data [18]. Instead, in Hades we organize the remote data when first cached at the proxy server disks by file id and requesting user.

2.2.4 Previous modifications made to the AFS Cache Manager

Stolarchuk uses several hints in order to improve the speed of the common case of the AFS Cache Manager [17]. The AFS Cache Manager fetches files from the AFS file server, and caches them into a local file system. Given this model, users expect reads of locally cached files to perform at local file system rates. However, read performance of the AFS cached files is half the read performance of the local file system. Stolarchuk examines the reasons for the large performance difference, and displays the modifications made to AFS so that reads of locally cached files perform within 10% of the performance of the local file system. After reducing the overheads of cache consistency checks and file-to-chunk mapping, access of the AFS cache becomes comparable to that of the local file system. Additionally, we consider storage locality as an alternative direction to improve performance.

2.3 Web Caching Proxies

A somewhat similar storage allocation problem showed up in web proxy servers. World-wide web proxies are widely used to allow web clients access web pages that several web servers offer, even behind firewalls. The objective is to improve user latency and reduce server load as well as network traffic, especially in wide-area networks. Web proxies have adopted the idea of data caching to generally improve performance. They lie between web clients and web servers. When a web client issues a request for a web object, the intermediate web proxy tries to satisfy it on behalf of the web server. If the requested object relies in its cache, the web proxy forwards it to the client. Otherwise, it fetches the object from the appropriate server, stores it in its cache and finally forwards it to the initial client.

Although most web browsers cache data in their local disk or main memory, web caching proxies are considered to be more efficient as they improve cache hit rates. They use large caches to satisfy huge amounts of requests from a variety of web clients. In case a cache fills up, an efficient cache replacement algorithm is used to free enough space for the newly arrived requests. To further improve hit rate some caches employ prefetching methods based on the assumption that if a web page is requested, several related pages are likely to be requested in the near future.

In web caching proxies, storage locality concerns can be handled by grouping files and metadata into clusters stored on consecutive blocks of disk. The clustering is based on the temporal locality of the access requests. Additionally, the web proxy server can treat large files specially and transfer them directly to the disk bypassing the memory cache [15]. In order to reduce management overheads for small files, the system may group the files by size and store them in a buddy organization. Thus, it eliminates file space fragmentation and reduces considerably the overhead for file creations and deletions. Aggregation of the written data in memory and subsequent appending to disk can provide additional write throughput improvement [7]. We now analytically present the recent approaches that were introduced with regard to the storage allocation problems in web caching proxies.

2.3.1 Hummingbird file system

Caching web proxies usually use general-purpose file systems to store web objects. Many widely used web proxies, like Apache and Squid, use the standard Unix file system (UFS) for data caching. However, several other filesystems have been designed for the same purpose.

Hummingbird is a light-weight filesystem library that web proxies can use to effectively store web objects they receive from web file servers [15]. It is made to run on top of a raw disk partition. It manages a large memory cache and has two major characteristics:

1. it separates object naming and storage locality through direct application-provided hints

2. its clients are compiled with a linked library interface for memory sharing

Comparison with UFS

It has been proved that UFS has a number of features that could limit file system performance. On the contrary, Hummingbird is able to simplify these features in order to improve the overall performance. To be more specific, UFS uses a hierarchical name space which means that files are separated across directories. A pathname translation may require a long time interval, especially because a linear search is executed to locate a file in its directory contents. This is not needed by a web proxy that wishes to have a flat name space and the ability to specify storage locality. As opposed to UFS, Hummingbird uses a flat name space for its files.

Furthermore, UFS keeps file meta-data on disks, in separate i-nodes. Synchronous disk-writes are used to update file-metadata and to preserve consistency. A web proxy does not need to execute such synchronous disk-writes. It can replace them with asynchronous writes to improve performance. Hummingbird keeps most metadata in memory and, if needed, uses asynchronous writes to update the on disk file metadata. It also stores cached files in fixed-sized 8KB blocks.

In order to minimize the disk head positioning time, Hummingbird attempts to store file blocks contiguously. To further improve performance, UFS tries to prefetch blocks for a file that is sequentially accessed. For small files, UFS attempts to minimize access time. However, if a large number of files are sequentially requested, large disk delays may be observed due to the reference stream locality not corresponding with the on-disk layout. To overcome this problem, UFS lets the user place files into directories and attempts to store the files that a directory contains in contiguous disk blocks called cylinder groups. The main problem is that users have the responsibility to construct a hierarchy with directory locality that matches future usage patterns. In contrast, Hummingbird uses locality hints generated by the proxy to store collocated files together. *Clusters* are the unit of disk access in contrary to disk blocks in UFS. They usually contain files and some file metadata. Files are grouped into clusters, typically 32 or 64KB, according to existing locality hints, so as to collocate files together. Least recently used files are examined. If the least-recently-used file has a list of collocated files, then these files are added to the cluster if they are in main memory. When we read a cluster from the disk, all collocated files that are contained in the cluster are read.

Furthermore, in traditional file systems like UFS, the standard filesystem interface copies data from kernel VM into the application's address space. It also caches file blocks in its own buffer cache. However, many web proxies have their own application-level VM caches to achieve more effective cache management. If a file is recently accessed from the disk, we may end up with two in-memory copies of this file: one in the web proxy application-level cache and another in the filesystem's buffer cache. To eliminate the problem of multiple buffering, we need a single unified cache. Hummingbird can solve the above problem, as it is a filesystem implemented by a library that accesses the raw disk partition, so as to avoid caching file blocks in the filesystem's buffer cache.

Data and metadata management

Hummingbird stores two types of objects in main memory, files and clusters. It can then move clusters from main memory cache to disk in order to free main memory space. Data stored in the disk can be categorized into four regions:

1. clusters with the real data and metadata
2. mappings of files to clusters
3. the hot cluster log used to cache frequently used clusters
4. the delete log used to store small records describing intentional deletes

Two daemons are used to perform the maintenance activities in Hummingbird: one to reclaim main memory space by writing files into clusters and another one to reclaim disk space by deleting unused clusters. Hummingbird keeps three types of metadata: filesystem metadata, file metadata and cluster metadata. It uses two LRU lists to determine which files or clusters to move from main memory to disk, in order to free main memory space. The one list is used for files which have not yet been packed into clusters and the other is used for clusters that are in memory. To retain file metadata Hummingbird uses a hash table that stores pointers to the file information. If a file is not contained in any cluster, Hummingbird must keep the filename (usually its URL), file-size and a list of files that should be collocated with it, as the file's metadata. When the file is added to a cluster, it keeps the cluster ID and the file reference count for that file. Similarly, a cluster table

is used to maintain information about each cluster in the disk, like a list of files in the cluster and the last time-accesed.

2.3.2 The BUDDY storage management method

Recent work has shown that disk I/O overhead is becoming an important bottleneck in the performance of web proxies. Especially, it has been found that the most important source of overhead is associated with storing each file in a separate file. Many web proxies fetch URLs from web servers and store their contents in separate files in their cache. When the cache fills up, a cache replacement algorithm is used to delete files in order to store the new files. This means that a file creation in the cache is followed by a file deletion. If we consider that the median size of a cached file is 3Kbytes and file system operations, such as file creation or file deletion, may take up to 50 milliseconds, we conclude that the rate at which a web server can store data to disk is 60 Kbytes/sec, which is much lower than the data transfer rates that the current disks can sustain.

To alleviate the above file management ovehead Markatos et al. proposed a storage management method called BUDDY that stores several URLS per file [7]. They tried to improve overall performance by altering the way that URL contents are stored on the web proxies' cache. BUDDY is a file management algorithm that stores remote files of the same size in the same cached file. All files that are smaller than one block are stored in one cache file, files with size in between one and two blocks are stored in another separate file, until a predefined number of blocks is reached. If a file's size is larger than this upper bound, a separate cache file is used to store the contents of it. When a new file-write request is issued, BUDDY first identifies the appropriate file in the cache to store it, according to its size. It then seeks for the first free slot in that file to store the contents of the remote file there. When a file-delete request is issued, BUDDY first seeks for the cache file that stores the file's contents, identifies the appropriate slot in it and marks it as free, so as to make it reusable for a future file-write request. When a file-read request is issued, BUDDY finds the slot in the appropriate file and reads its contents. Thus, BUDDY manages to:

1. Reduce file management overhead as remote files do not need individual cache files to store their contents. One only needs to know the cache file and the slot in it that

the remote file's contents are stored. Hence, only metadata for cache files need to be managed rather than metadata for the remote files' contents.

2. Eliminate file space fragmentation by placing same-sized remote files in one cache file. BUDDY forces each remote file to occupy consecutive bytes within a single cache file.

2.3.3 File space management algorithms

The next largest source of overhead, after the storage of each remote file to a single cache file, is the cost associated with file write operations. This source of overhead is due to disk latencies incurred by writing data scattered all over the disk. Although it reduces file management overhead, BUDDY does not improve write throughput to a considerable extent. Markatos et al. proposed two file space management algorithms to reduce disk seek overhead and perform write operations at maximum speed.

The STREAM file space management algorithm

The STREAM algorithm was inspired from log-structured file systems and intends to improve write performance. The main idea is to store all remote files in a single file contiguously, if possible, in slots of 512 bytes long. STREAM tries to reduce disk seek and rotational overhead by writing to the disk in a log-structured mode. It tries to make all writes in contiguous blocks. If the disk is full, write operations continue from the beginning of the disk.

However, it was observed that STREAM did not write to disk at maximum throughput. This happened because when a process is writing few blocks of a page in a page that is not in the main memory cache, both disk-read and disk-write occurs. The operating system must first read the page from the disk, make all updates in the main memory and then write the page to the disk. To reduce this overhead, a packetized version of STREAM is used.

The STREAM-PACKETIZER file space management algorithm

STREAM-PACKETIZER uses a packetized buffer that is one page long and can reach an upper boundary. When a file write request is issued, it is not sent to the filesystem for

writing to the disk but it is stored in the packetizer contiguously with the previous file requests. File write requests are forwarded to the filsesystem only when the packetizer fills up or a request that is not contiguous with the previous arrives.

2.3.4 Methods to reduce the disk head seek time

Once write operations proceed at maximum speed with the use of STREAM-based algorithms, read operations represent the next single largest source of overhead. In fact, this overhead is due to head movements when read requests are issued. Markatos et al. proposed two methods that reduce disk seek overhead associated with read operations [7].

The LAZY-READ method

If a file read request is issued between file write requests, the head has to move from the point it finished writing data to the point it starts reading data, complete the read request and move back to the previous point. To reduce this extra head movement overhead, a lazy-read approach was introduced, that is much like STREAM-PACKETIZER. In LAZY-READ method, a file read request is not initially forwarded to the appropriate data in the disk. It is first stored into an intermediate buffer, but not yet satisfied. When the buffer fills up, the read requests it contains are forwarded to the file system, each request to the appropriate file.

The LAZY-READ-LOC method

In the LAZY-READ method, if the locality of the stream is taken into account, file read overhead can be further reduced. When a user requests an HTML page, he will probably request all the embedded images as well. Successive requests from a web client may not necessarily arrive contiguously at the web proxy server because it can support a large number of clients issuing concurrent requests. Therefore, files that correspond to contiguous requests from a single client may be stored in the magnetic disk hundreds of Kbytes away from each other. However, storing requests arriving from a single web server to a single locality buffer would improve performance to an extent. LAZY-READ-LOC is an algorithm that uses several locality buffers to put together requests from a single web server. Is main intention is to preserve locality of the URL stream. The idea is

to store successive requests from a web client to nearby disk locations so as to re-access them quickly in case the user issues a request for them in the future. If requests from a single web server are stored in a single locality buffer, then URLs from the same web server requested within a short time interval will probably be written in contiguous file locations. When a proxy fetches a URL from a web server, it tries to locate the buffer that stores requests from this server and saves the data in this buffer. If such a buffer does not exist, an existing buffer is chosen to write its data to the disk and it is then reused to store the contents of the requested URL.

2.4 Summary

In distributed environments, a large amount of files is usually shared between multiple cilents and fileservers. Especially in wide area networks, latencies introduced by the network may be much greater than direct disk accesses. Thus, the use of caching systems that would cache data closer to the client is essential. A summary of the major caching systems that we studied and were previously described is presented in table 2.1. Hades is the caching proxy server that we implemented and propose in the present thesis. It must be noted that SSMWP stands for Secondary Storage Management for Web Proxies and refers to the data and metadata management methods that were proposed by Markatos et al.

Each of the above caching systems uses its own storage allocation method to effectively place data in the disk cache. In recent prototypes, the directory and file naming structures can dynamically be replicated from the origin server to the cache. Data can be fetched and stored in cache in fixed-size cache files, as part of the local filesystem or even at a single raw partition. The basic goal of each system is to manage data in cache in such a way that the overall performance of accessing remote data is significanlty enhanced. Similar storage allocation problems showed up in the area of web caching proxies, where data and metadata can be grouped into clusters stored on consecutive blocks of disk to improve existing performance.

In this thesis, we consider Andrew File System as an alternative basis for building a proxy server for distributed filesystems. We first investigate previous modifications made

Table 2.1: Major caching systems and their basic characteristics

	FS-Cache	Nache	Hummingbird	SSMWP	Hades
Used by network filesystems	√	√			√
Supports multiple clients		√	√	√	√
Supports disconnected operation	√				
Caching proxy		√	√	√	√
Layer in the kernel	√				
Double buffering	√	√		√	√
Uses a flat namespace			√		
Clusters data and metadata			√	√	√
Uses a raw disk partition as its disk cache			√		
Multiple remote files per local cache file				√	√
Writes to the disk in a log-structured mode				√	
Read or write requests kept in buffers				√	
Uses locality buffers				√	
Stores a file in contiguous blocks					√
Stores parts of the same file in nearby locations					√
Clusters data based on file or user id					√

to the AFS Cache Manager that improve access of the AFS cache, by reducing cache consistency checks and file-to-chunk mapping. However, we consider storage locality as an alternative direction to improve performance. At the same time, we acquire metadata management and file replacement methods that support the design of an efficient caching proxy server in wide area networks.

Chapter 3

Andrew File System

In this chapter, we present the Andrew File System (AFS), a location-transparent distributed filesystem that can support growth up to thousands of workstations while providing users, application programs and system administrators with the amenities of a shared file system. We first examine Andrew's advantages over contentional filesystems, emphasizing on its scalable architecture. Then, we define the client-server model that Andrew uses as well as the set of modifications to the client machines' kernel that enable communications with the server processes running on server machines. Furthemore, we analyze the way remote AFS files are distributed in order to form a uniform namespace. Next, we define the basic on-disk and memory-based data structures of the OpenAFS system that we modified in our prototype implementation. Finally, we examine how AFS manages its storage space and give a detailed description of the steps that are followed in order to store a remote AFS file on the client's disk cache.

3.1 AFS: A Distributed File system

Andrew File System is a distributed file system that enables co-operating hosts (clients and servers) to efficiently share file system resources across both local area and wide area networks. It is similar to Sun Microsystems Network File System (NFS). AFS is capable of scaling to thousands of users. It enables users to share and access all of the files stored in a network of computers as easily as they access the files stored on their local machines. The file system is called distributed for this exact reason: files can reside on many different machines, but are available to users on every machine.

AFS is based on a distributed file system originally developed at the Information Technology Center at Carnegie-Mellon University in 1984. The idea was to provide a campus-wide file system for home directories which would run effectively using a limited bandwidth campus backbone network. In 1989, the Transarc company was formed to evolve the Andrew File System into a commercial product. Transarc renamed the product from Andrew File System to AFS. In 1990, the Open Software Foundation (OSF) chose AFS from Transarc as the Distributed File System (DFS) component of its Distributed Computing Environment (DCE).

AFS joins together the file systems of multiple file server machines, making it as easy to access files stored on a remote file server machine as files stored on the local disk. A distributed file system, like AFS, has two main advantages over a conventional centralized file system:

- Increased availability: A copy of a popular file, such as the binary for an application program, can be stored on many file server machines. An outage on a single machine or even multiple machines does not necessarily make the file unavailable. Instead, user requests for the program are routed to accessible machines. With a centralized file system, the loss of the central file storage machine effectively shuts down the entire system.

- Increased efficiency: In a distributed file system, the work load is distributed over many smaller file server machines that tend to be more fully utilized than the larger and usually more expensive file storage machine of a centralized file system.

AFS hides its distributed nature, so working with AFS files seems like working with files

stored on the user's local machine, except that we can access many more files. What is more, because AFS relies on the power of users' client machines for computation, increasing the number of AFS users does not slow AFS performance appreciably, making it a very efficient computing environment.

3.1.1 Scalable Architecture

There are three important problems in making a distributed file system scalable. If a single server handles a large number of clients, we get both server congestion and network overload. Inadequate client-side caching causes excessive network traffic. Finally, if the server performs the bulk of the processing of all operations, it will become overloaded sooner. A scalable system, like AFS, must address all these issues correctly.

In a standard AFS configuration, clients provide computational power, access to the files in AFS and other "general purpose" tools to the users seated at their consoles. There are generally many more client workstations than file server machines. AFS file server machines run a number of server processes, so called because each provides a distinct specialized service: one handles file requests, another tracks file location, a third manages security, and so on.

AFS controls network congestion and server overload by segmenting the network into a number of independent clusters. Unlike NFS and RFS, AFS uses dedicated servers. Each cluster contains a number of clients plus a server that holds the files of interest to those clients, such as the user directories of the owners of the client workstations. Each machine is a server, a client or both in rare situations. The above configuration provides fastest access to files residing on the server, on the same network segment. Users can access files on any other server, but the performance will be slower. The network can be dynamically reconfigured to balance loads on servers and network segments.

AFS uses aggressive caching of files, coupled with a stateful protocol, to minimize network traffic. Clients cache recently accessed files on their local disks. The original AFS implemetation cached only entire files, which was not practical in case we needed to access a part of a very large remote file. AFS 3.0 divides the files into chunks of a default size (usually 256KB), and caches individual chunks separately. The AFS servers participate actively in client cache management, by notifying clients whenever the cached

data becomes invalid. Finally, AFS reduces server load by moving the burden of name lookups from the server to the clients. Clients cache entire directories and parse the filenames themselves.

3.1.2 Client-server model

AFS uses a client-server model. In general, a server is a machine, or a process running on a machine, that provides specialized services to other machines. A client is a machine or process that makes use of a server's specialized service during the course of its own work, which is often of a more general nature than the server's. Some machines act as both clients and servers. In most cases, users work on a client machine, accessing files stored on a file server machine. AFS divides the machines on a network into two basic classes, file server machines and client machines, and assigns different tasks and responsibilities to each.

File server machines store the files in the distributed file system, and a server process running on the file server machine delivers and receives files. AFS file server machines run a number of server processes. Each process has a special function, such as maintaining databases important to AFS administration, managing security or handling the disk space where a set of files resides. This modular design enables each server process to specialize in one area, and thus perform more efficiently. Not all AFS server machines must run all of the server processes. Some processes run on only a few machines because the demand for their services is low. Other processes run on only one machine in order to act as a synchronization site.

The other class of machines are the client machines, which generally work directly for users, providing computational power and other general purpose tools. Clients also provide users with access to the files stored on the file server machines. Clients do not run any special processes, but do use a modified kernel that enables them to communicate with the AFS server processes running on the file server machines and to cache files. This collection of kernel modifications is referred to as the *Cache Manager*.

3.1.3 The Cache Manager

The Cache Manager resides on client machines rather than on file server machines. It is not technically a stand-alone process, but rather a set of extensions or modifications in the client machine's kernel that enable communication with the server processes running on server machines. Its main duty is to translate file requests, made by application programs on client machine, into remote procedure calls to the File Server. The Cache Manager first finds out which File Server currently houses the requested file. When the Cache Manager receives the requested file, it caches it before passing data on to the application program (figure 3.1).

The Cache Manager also tracks the state of files in its cache compared to the version at the File Server by storing the callbacks sent by the File Server. When the File Server breaks a callback, indicating that a file changed, the Cache Manager requests a copy of the new version before providing more data to application programs.

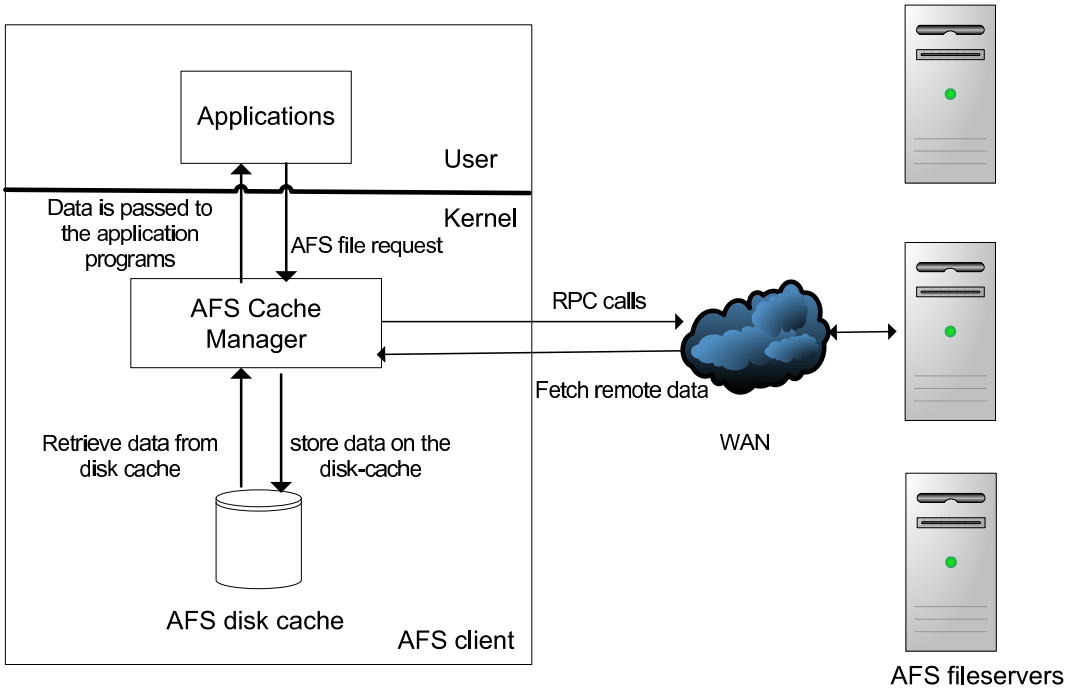


Figure 3.1: The AFS Cache Manager.

More specifically, the Cache Manager helps us to access all files that remote AFS fileservers export when working on an AFS client machine. It plays fundamental role at the client kernel, because it improves data transfer efficiency through local memory and on-disk structures. When we access a file, the Cache Manager on our client machine

requests the file from the appropriate file server machine and caches a copy of it on our client machine's local disk. Application programs on our client machine use the local, cached copy of the file. This improves performance because it is much faster to use a local file than to send requests for file data across the network to the file server machine.

Because application programs use the cached copy of a file, any changes we make are not necessarily stored permanently to the central version stored on the file server machine until the file closes. At that point, the Cache Manager writes our changes back to the file server machine, where they replace the corresponding parts of the existing file. If a file server machine becomes inaccessible, we can continue working with the local, cached copy of a file fetched from that machine, but we cannot save our changes permanently until the server machine is again accessible.

3.2 Basic definitions

We now identify cells and volumes in a distributed environmnent that supports multiple clients and fileservers. Files that AFS fileservers offer are kept on volumes while client and server machines belong to cells. Files that are grouped into volumes can be distributed accross many machines and yet provide a single, uniform namespace that is independent of the storage location.

3.2.1 Cells

A *cell* is a grouping of client machines and server machines defined to belong to the same organization. An *AFS site* is a grouping of one or more related cells. Each cell's administrators determine how client machines are configured and how much storage space is available to each user. The organization corresponding to a cell can be a company, a university department, or any defined group of users. For example, the cells of the Systems Research Group at University of Ioannina form a single site.

By convention, the subdirectories of the /afs directory are cellular filespaces, each of which contains subdirectories and files that belong to a single cell. For example, directories and files relevant to the Systems Research Group of University of Ioannina cell are stored in the subdirectory /afs/srg.cs.uoi.gr While each cell organizes and maintains its own

filespace, it can also connect with the filespace of other AFS cells. The result is a huge filespace that enables file sharing within and across cells. The cell to which a user's client machine belongs is called his local cell. All other cells in the AFS filespace are termed foreign cells.

3.2.2 Volumes

A *volume* is a unit of disk space that functions like a container for a set of related files, keeping them all together on one partition. AFS groups files into volumes, making it possible to distribute files across many machines and yet maintain a uniform namespace. For instance, a volume may contain all files belonging to a single user. Volumes can vary in size, but are smaller than a partition.

Volumes are important to system administrators and users for several reasons. Their small size makes them easy to move from one partition to another, or even between machines. The system administrator can maintain maximum efficiency by moving volumes to keep the load balanced evenly. In addition, volumes correspond to directories in the filespace. Most cells store the contents of each user home directory in a separate volume. Thus the complete contents of the directory move together when the volume moves, making it easy for AFS to keep track of where a file is at a certain time. Volume moves are recorded automatically, so users do not have to keep track of file locations. Volumes also allow files that are much larger than a single disk. Read-only volumes can be replicated on several servers to increase availability and performance. Finally, each volume can be individually backed up and restored.

Each volume has a unique volume identifier. Additionally, each file is identified by a file identifier *fid*, which consists of a volume ID, a vnode number and a vnode uniquifier. Historically, AFS uses the term vnode to mean a Vice inode. Hence, the vnode number is an index into the inode list of the volume. The uniquifier is a generation number, incremented each time the vnode is reused.

3.2.3 Uniform Namespace

AFS provides a single uniform namespace that is independent of the storage location (figure 3.2). Although the AFS cell that a client machine belongs to is administratively

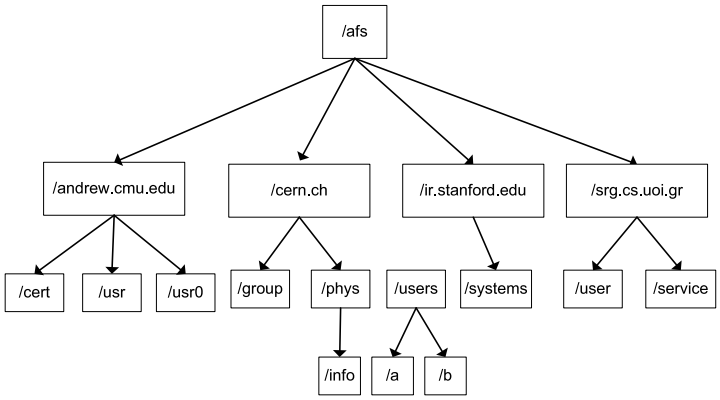


Figure 3.2: The uniform namespace of Andrew File System.

independent, we probably want to organize the local collection of files in a way that clients from other cells access the information stored on it. AFS enables cells to combine their local filespaces into a global filespace, and does so in such a way that file access is transparent. Users only need to know the pathname of the file, which looks the same in every cell. Thus every user at every machine sees the collection of files in the same way, meaning that AFS provides a unifore namespace.

Each client workstation must have a local disk that contains a few local files, plus a directory on which it mounts the shared file hierarchy. Conventionally, each workstation mounts the shared tree on the same directory (usually /afs). The local files include the system files essential for minimal operation, plus some files the user may want to keep local for reasons of security or performance. Hence each client sees the same shared name space, plus its own, unique, local files. The local disk also acts as a cache for recently accessed shared files.

3.3 Major structures

We now present the basic OpenAFS structures along with their most significant fields. In our prototype implementation, we modified the majority of these structures in order to improve overall performance of cached data.

3.3.1 On-disk structures

OpenAFS uses /var/cache/openafs as its default caching directory. CacheItems, CellItems and VolumeItems are included in this directory. Depending on the total cache size, a number of Di subdirectories contain Vi files of default size which are used to store whole or pieces of remote AFS files. The above files are stored in the disk of the OpenAFS client and are updated periodically.

CacheItems file

Initially, the Cache Manager creates the binary-format file called CacheItems. This file is used to store an index of all the cache files. It contains the fid, offset, and size of each file in the cache, together with some additional information, which enables the Cache Manager to determine which cache file contains the AFS data that is requested by an application. It is structured as an array, with one entry (about 40 bytes) and stored on the disk. Every entry in the CacheItems file is a special structure which identifies a specific cache file (figure 3.3). A remote AFS file can be saved in one or more files in the disk cache, depending on the file size. When a file in the disk cache is chosen for storing a part of an AFS file, an entry is added for it in the CacheItems file recording the fid of the remote AFS file, the relative chunk number and the inode of the cache file. If for example a remote AFS file is stored in five files in the disk cache, the struct fcache entry for each cache file will contain the same fid (as they refer to the same file), but different chunk number(0 to 4).

```
struct fcache {
    struct VenusFid fid;        /* Fid for this file */
    afs_int32 modTime;          /* last time this entry was modified */
    afs_hyper_t versionNo;      /* Associated data version number */
    afs_int32 chunk;            /* Relative chunk number */
    afs_inode_t inode;          /* Unix inode for this chunk */
    afs_int32 chunkBytes;       /* Num bytes in this chunk */
    char states;                /* Has this chunk been modified? */
};
```

Figure 3.3: The fcache structure

Vn files

AFS organizes its storage space into multiple chunk files. The *chunk file* is the data transfer and store unit. A large number of fixed-side cache files (called Vn files) are initially created in the cache, which is located in the client's local filesystem. Each of them is subsequently used to satisfy requests that are issued from user programs in the client workspace. Vn files are the actual data store units in the OpenAFS client's disk cache. A Vn file can store a chunk of cached AFS data on a client machine that is using a disk cache. When the Cache Manager initializes itself, it verifies that the local disk cache directory houses a number of Vn files equal to the largest of the following: 100, one and a half times the result of dividing the cache size by the chunk size (cachesize/chunksize * 1.5) or the result of dividing the cache size by 10 MB (10,240). Vn files are stored in Di subdirectories of the /var/cache/openafs directory.

Remote AFS files are also separated into chunks. Each remote data chunk is stored in one unique local cache file. If a request for a large remote AFS file is issued, this file will be stored in multiple local cache files. In future read requests for this remote file, all the local cache files that keep the remote data chunks have to be discovered. We must open, read and close each of these cache files in order to acquire the desired data.

Cacheinfo file

Another file is used to define configuration parameters for the Cache Manager and is located in /etc/openafs/. When OpenAFS client is initialized, the cacheinfo file is created and initialized. The file contains a single line of ASCII text. Its format is *mount*:*cache*:*size* where *mount* is the local disk directory at which the Cache Manager mounts the AFS namespace, *cache* is the local disk directory to use as a cache and *size* is the cache size as a number of 1-kilobyte blocks. Larger caches generally yield better performance, but a disk cache must not exceed 95% of the space available on the cache partition. Cache Manager usually mounts the AFS filespace at /afs.

The default cache directory is /var/cache/openafs. However, we can modify Cacheinfo file to set another local directory as the cache directory. Ext2 and ext3 can be used as cache partitions unlike Reiserfs, XFS and tmpfs that cannot be used. When an openafs-client is initialized and started for the first time, cache directory is created. Every time

openafs-client restarts, Cache Manager only checks to see if some cache files are missing and so have to be created or if some files must be deleted. The default cachesize is 50000 1KB-blocks.

VolumeItems file

The VolumeItems file records the mapping between volume name and mount point for each volume that the Cache Manager has accessed since its initialization on a client machine using a disk cache. The Cache Manager uses the mappings to respond correctly to queries about the current working directory, which can come from the operating system or commands such as the UNIX pwd command. As it initializes, the Cache Manager creates the binary-format VolumeItems file in the local disk cache directory, and it must always remain there.

```
struct dcache {
    struct afs_q lruq;          /* Free queue for in-memory images */
    struct afs_q dirty;         /* Queue of dirty entries that need written */
    afs_rwlock_t lock;          /* Protects validPos, some f */
    afs_rwlock_t tlock;         /* Atomizes updates to refCount */
    afs_rwlock_t mflock;        /* Atomizes accesses/updates to mflags */
    afs_size_t validPos;        /* number of valid bytes during fetch */
    afs_int32 index;            /* The index in the CacheInfo file */
    short refCount;             /* Associated reference count. */
    char dflags;                /* Data flags */
    char mflags;                /* Meta flags */
    struct fcache f;            /* disk image */
};
```

Figure 3.4: The dcache structure

3.3.2 In-memory structures

Dcache entries

To make cache file indexing more efficient, a portion of Cachefiles entries is kept in memory, in some special structures. Rather than keeping all of the Cacheinfo data in memory or keep searching it on disk, the Cache Manager keeps a subset of this data in memory, in

```
struct vcache {
    struct vnode *v;
    struct afs_q vlruq;          /* lru q next and prev */
    struct vcache *hnext;        /* Hash next */
    struct afs_q vhashq;         /* Hashed per-volume list */
    struct VenusFid fid;
    struct mstat{
        afs_size_t Length;
        afs_hyper_t DataVersion;
        afs_uint32 Date;
        afs_uint32 Owner;
        afs_uint32 Group;
        afs_uint16 Mode;
        afs_uint16 LinkCount;
    } m;
struct dcache *dchint;
};
```

Figure 3.5: The vcache structure

dcache entries. The dCacheSize is the number of these entries that are kept in memory. The default dCacheSize is currently half the number of cache files, but not less than 300 and not more than 2000.

Valid cache files are associated with dcache entries. A struct dcache is associated with an on-disk struct fcache, as it is shown in Figure 3.4. The dCacheSize setting should approximate the size of the workstation's working set of chunks. If the chunk size is large, this is close to the number of files whose contents (not metadata) are in the working set. If the chunk size is very small, then it's probably some multiple of that number.

Vcache entries

Another in-memory structure is used to store metadata about files in AFS (figure 3.5). Any time we need to get information about a file that is not in the vcache, we must make an RPC to the remote fileserver. So, we don't want the vcache to be too small, since that would result in lots of extra RPC's and considerable performance loss. The ideal vcache size approximates the size of the workstation's working set of AFS files, including files

```
struct volume {
    /* One structure per volume, describing where the volume is located
     * and where its mount points are. */
    struct volume *next;        /* Next volume in hash list. */
    afs_int32 cell;             /* the cell in which the volume resides */
    afs_int32 volume;           /* This volume's ID number. */
    char *name;                 /* This volume's name, or 0 if unknown */
    struct server *serverHost[MAXHOSTS];     /* servers serving this volume */

    struct VenusFid dotdot;     /* dir to access as .. */
    struct VenusFid mtpoint;    /* The mount point for this volume. */
    afs_int32 rootVnode, rootUnique;    /* Volume's root fid */
};
```

Figure 3.6: The volume structure

for which we only care about metadata. Vcache entries cache information obtained via fileserver RPC's and can be considered as specializations of struct vnodes. A struct vcache keeps a reference to a struct dcache, for the first chunk that stores AFS file's information. We can then find subsequent chunks storing the AFS file using specific hash tables.

Volume entries

The volume cache stores cached information about volumes, including name-to-ID mappings, which volumes have RO clones, and where they are located (figure 3.6). The size of the volume cache should approximate the size of the workstation's working set of volumes. Entries in this cache are updated periodically every 2 hours.

Data cache hash tables

Data cache hash tables are used to locate the local cache file that keeps a data chunk of a remote AFS file. When an OpenAFS client initializes itself, two hash tables are allocated in memory. The *afs_dchashTbl* contains the indexes of the local cache files that keep remote data chunks, while the *afs_dcnextTbl* table is used to find the cache file index incase of collisions. When a request for a remote file offset is issued, the chunk number that the remote offset belongs to is calculated, as it is shown in figure 3.7. The remote

fid and chunk are hashed to acquire an integer (e.g. i) that indicates the appropriate position in afs_dchashTbl. It is then checked whether the table's index (e.g. j) in position i is the index of the cache file that keeps the searched data. If not, afs_dcnextTbl must be checked. Therefore, it is checked whether the index in position j of afs_dcnextTbl (e.g. k) is the appropriate cache file index. If not, the index in position k of afs_dcnextTbl is now checked an so on. If we cannot locate the appropriate cache file index after searching in this hash chain, the file has not been stored locally in cache, so we must find a new free cache file to map and store the remote data.

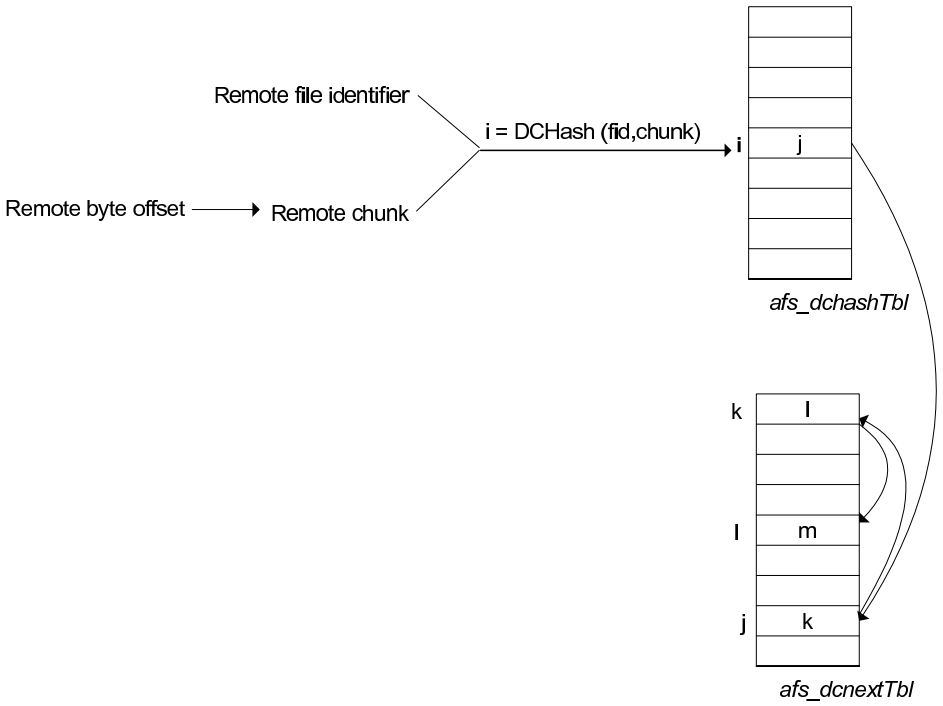


Figure 3.7: Hash tables used to locate data at the client's disk cache.

3.4 Storage management in AFS

In this section, we are going to describe how AFS manages its storage space. In following chapters, the modifications made to the existing storage management methods will be examined. These modifications led to a general performance improvement by reducing storage space fragmentation.

When an OpenAFS client initializes itself, it creates a disk cache as part of the local

ext2/3 filesystem. Each cache file has maximum size equal to the chunk used for the transfers from the server (typically 256KB). The total number of cache files is configurable and depends on the size of the disk cache. Cache files appear as regular files with names Vi, where the index i takes values between 0 and a maximum configurable value.

As it has already ben mentioned in the previous section, for each local cache file, there exists a special fcache structure on the CacheItems file. These structures associate each Vi file with a chunk of a remote file. The fields of fcache store metadata, such as the identifier of the remote file, the offset, the chunk size, and the inode of the local file. An array of fcache structures is stored persistently on disk. For improved indexing efficiency, a subset of the fcache structures is also maintained in memory as a collection of dcache structures. Periodic updates keep the fcache contents consistent with their memory counterparts. In fact, fcache structures are the on-disk images of the corresponding dcache entries. Each dcache entry describes a Unix file on the local disk that is serving as a cached copy of all or part of a remote AFS file, as it is shown in figure 3.8.

3.4.1 Circular queues

Dcache entries live in three circular queues:

1. freeDSlot
2. freeDCList
3. DLRU

and move between them, depending on their current state. Struct dcache entries are initially kept in a circular queue called *freeDSlot* used for free dcache entries. A struct dcache entry is in the freeDSlot queue when not associated with a cache slot (local cache file). Otherwise, it is in the *DLRU* queue. Cache entries in the DLRU queue are either associated with remote AFS files, or they are in the *freeDCList* queue and are not associated with any remote file. Entries are moved from DLRU to freeDCList when the corresponding cache files need to be replaced (e.g. when cache fills up to 95%), and from freeDCList to DLRU when we need to acquire a dcache entry to store some new data in cache.

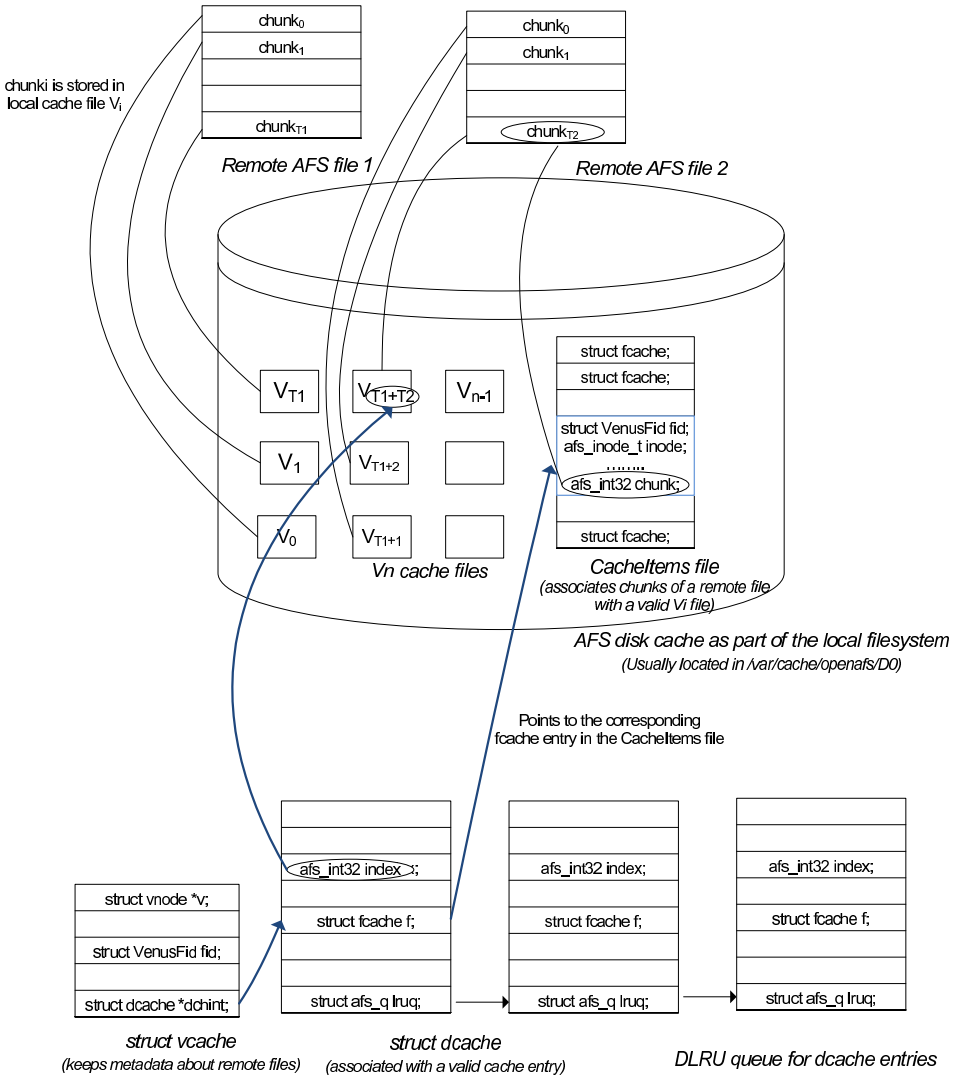


Figure 3.8: Major structures of Andrew File System and their correlation.

When cache files are initially created, afs_InitCacheFile function is called to initialize them. Given a file name and inode, it sets up that file to be an active member in the AFS cache. In order to map disk cache files with their in-memory dcache entries, afs_GetDSlot function is called, that takes the first entry from freeDSlot queue and adds it in front of the DLRU queue. Dcache entries in DLRU queue represent active cache files that are used to store remote data. Whenever a dcache entry is referenced, it is moved in front of the DLRU queue so as to retain most recently used chunks in front of the queue. Furthermore, this dcache entry must be put in the freeDCList as it is not yet used to store any remote data.

3.4.2 Mapping from a remote to a local file offset

Whenever we need to access a remote AFS file, we must first examine whether this file is already cached. There exists an afs_GetDCache function that maps a remote byte offset to the relative offset in the local cache file that keeps this remote data. This function take as a parameter the pointer to the vcache entry for the remote file and the byte position in the file desired, and returns the offset within the chunk where the resident byte is located. We first compute the chunk number of the chunk containing the given byte. To examine whether this byte is already cached, a thorough search in afs_dchashTbl and afs_dcnextTbl is done. These two hash tables contain cache file indexes. If we know the cache file index, we can find the corresponding dcache entry using afs_GetDSlot function. We must then examine two fields of the corresponding dcache structure, fid and chunk. If these fields have the same value with the file identifier in the vcache entry as well as the already computed chunk number, we conclude that the given byte id already stored in cache. We can reach cache file through dcache entry and the corresponding on-disk image fcache entry (struct dcache has one struct fcache field).

3.4.3 Allocating a new local cache file to store remote data

If none of the indexes found in hash tables indicates a cache file that keeps the searched remote data, we conclude that the remote chunk is not cached yet. Thus, we must find one or more free cache files to store the file's data. The queues that were previously described are used for this purpose. We must search for available and free dcache entries, which means that these entries must not be associated with any cache file or any remote chunk of data. If freeDCList is not empty, we choose its first dcache entry to store remote data. The next step is to fill in the newly-allocated dcache record with the correct information (such as fid or chunk) so as to describe the remote data that is stored in it. We must then add the right information to the two hash chains, in order to be able to locate file in future requests.

To sum up, each request to the offset of a remote file can be mapped quickly to the corresponding offset of a local file at the client through the afs_dchashTbl hash table. The hash function translates the identifier and the offset of the remote file to a hash table position. A separate auxiliary hash table chains the additional entries required in the case

of collisions. If the requested chunk is not available locally, a new index entry is allocated along with a free local file to store the data transferred from the remote server.

Chapter 4

Architectural Definitions

In this chapter, we examine the basic design goals of our study and present a high-level description of our proposed architecture. Initially, we introduce the major issues that determine the creation of a successful and effective caching system. Then, we detect the design inefficiencies of existing caching systems, including Andrew File System. We then define the architectural decisions that were taken and led to our prototype implementation, emphasizing on storage space management and file replacement. Finally, we examine how our efficient scheme improves performance of multiple concurrent file accesses in distributed environments.

4.1 Design issues

Caching has long been used to reduce the operation cost of distributed filesystems over wide-area networks. Some of the basic reasons why caching proxies are chosen to support secure data sharing over multiple clients in a wide-area network are described below:

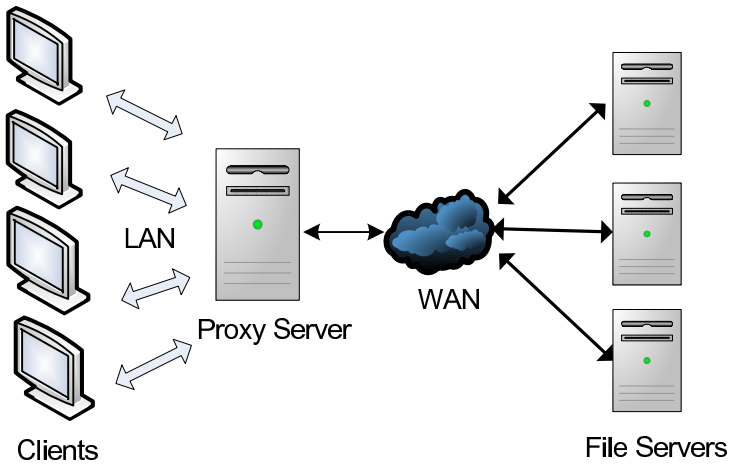


Figure 4.1: The basic architecture of a proxy server in a distributed file system.

- Remote fileservers are not able to satisfy a large amount of concurrent requests from a single client as easily as the intermediate proxy servers can.
- Multiple clients share physical storage resources that leads to a total reduction of the network bandwidth requirements.
- In general, the total cost of accessing data from a remote server may be orders of magnitude greater than the corresponding cost of accessing them from an intermediate proxy server in a local area network, including the cost of leasing network bandwidth with comparable transfer capacity.

In the present study, we introduce a caching proxy for distributed filesystems that wishes to sustain secure data sharing in WANs, while it satisfies the above criteria. However, even though caching proxies generally improve performance of accessing cached data, intermediate layers in the path from the origin server to the client may introduce performance bottlenecks and reduced parallelism in the data transfers. Consequently, it is quite probable that the perceived throughput will be lessened while the total time for accessing cached data may be increased. Hence, a variety of resource management issues had to be thoroughly examined while designing the new caching proxy server. The basic design issues that were initially investigated and led to our prototype implementation are described in the following sections

4.1.1 Storage allocation

In network filesystems, retrieving data exactly from a remote fileserver may cause latencies that are much greater than the corresponding time to fetch data from the local file system or memory cache. Furthemore, when developing cost-effective filesystems it is preferable to keep cached data on hard disks rather than the main memory of the proxy server. Thus, to improve retrieval time of cached data, one should explore innovative storage management methods in order to effectively allocate the disk storage space of the cache.

A variety of different approaches to the storage allocation problem were recently introduced by the research community. The dominant strategy is to map each remote file to a single local cache file. However, early approaches copied entire files from the remote file server to the client, increasing transfer latency as well as the overall resource requirements at the client. Later approaches attempted to overcome these restrictions by copying and storing parts of remote files in cache, placing each file part into a single local cache file. In more recent prototypes, the system dynamically replicates the directory and file naming structure from the origin server to the cache, while transferring the file contents on demand in pages of configurable size. What is more, on web caching proxy servers files and metadata are grouped by size and stored into clusters on consecutive blocks of disk, to reduce latencies and improve performance.

However, we claim that the proxy server should not be restricted to offer a consistent view of the data as they appear at the remote server. A caching proxy for distributed file systems should be free to manage its local data in ways that serve its design objectives better. Thus, the existing storage management methods should be enhanced in order to achieve

- better mapping of remote data to the local cache file, by organizing metadata in a more effective way
- successful clustering of remote data to the local disk of the proxy, in order to improve the existing performance of accessing cached data.

Experimental measurements to AFS

In order to examine Andrew's caching efficiency, we made a variety of experiments with its open-source variant (OpenAFS v. 1.4.5) over Linux kernel v. 2.6.18. We measure

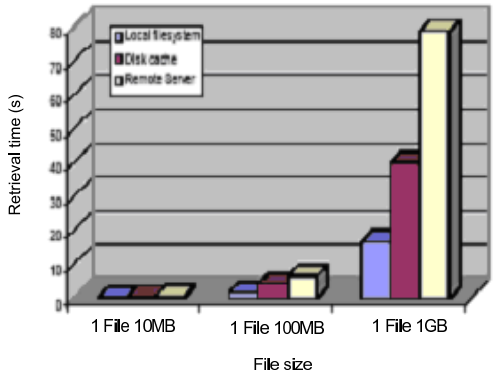


Figure 4.2: Time to retrieve one large file directly from the local file system in comparison to accessing it through OpenAFS from the local disk or the remote server.

the retrieval time from the disk cache through OpenAFS as well as the corresponding retrieval time from the local filesystem or the remote fileserver. As we see in Figures 4.2 and 4.3, the retrieval time from the disk cache through OpenAFS is is about 2-3 times greater for large files and 1.3-1.8 times greater for numerous small files in comparison to the retrieval time from the local file system. On the other hand, fetching files from the remote server (another node on the same gigabit Ethernet switch in our experiments) costs about 150-200% the retrieval time from the OpenAFS disk cache for large files and 250-550% for numerous small files. Thus, we concluded that it is essential to explore alternative methods for the mapping of the remote data to local files in disk cache to

- reduce storage space and metadata management overhead
- enhance the existing performance of retrieving cached data

However, to examine whether Andrew can effectively support multiple users, we measured the total time to satisfy concurrent requests issued by different clients. For example, we issue concurrent requests for five different remote files of 1GB size. We observed that these concurrent reads of files stored at the remote server, require time that may be orders of magnitude greater than the corresponding time to read the same files sequentially. More specifically, it takes about 395 seconds to concurrently fetch the remote files and about 366 seconds to read them in parallel from the disk cache. Additionally, the time to sequentially read from disk cache only one of these files is about 70 seconds, while the corresponding time to read the same file (in case it was sequentially fetched and stored in cache) is about 40 seconds. We investigated where each portion of time is wasted and

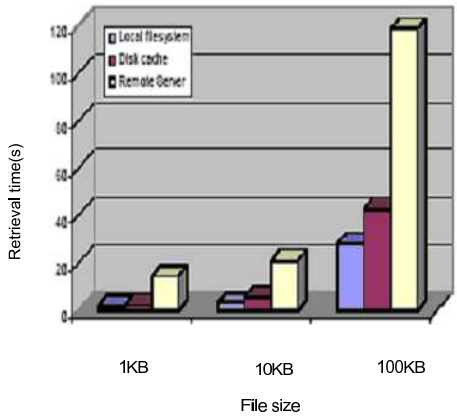


Figure 4.3: Time to retrieve numerous small files directly from the local file system in comparison to accessing them through OpenAFS from the local disk or the remote server.

concluded that increased latency occurs due to disk space fragmentation that is caused by multiple concurrent reads. That was the basic motivation for our proposed method that changes the way cached data are placed on proxy's hard disk.

With the existing allocation algorithm of AFS, files that are concurrently fetched from remote servers may scatter over the client's disk cache. In order to avoid file fragmentation, we proposed a method that groups together parts of the same remote file in consecutive blocks at the client's disk. Consequently, we achieved to reduce file retrieval times in all cases. Specifically, in the above file retrieval example we reduced fetch time from the remote server from 395 seconds to 249 seconds and read time from the disk cache from 366 to 253 seconds while the time to sequentially read one file was reduced from 70 seconds to 40 seconds. Thus, even though the files were concurrently stored in cache, the time to read each of them from the disk cache remains the same as they had been sequentially fetched and stored in cache.

In fact, these are the basic reasons why we modified the storage method that Andrew File System uses. However, we aim not only at improving the Andrew's caching efficiency, but we also planned to offer a new caching service that would support multiple concurrent clients in a distributed environment.

4.1.2 Data Replacement

A variety of data replacement methods have been lately introduced in the area of caching systems. They intend to remove the most appropriate files from a full cache, in order to satisfy the incoming requests. We consider data replacement a major issue for thorough investigation in proxy servers of distributed file systems. Recent page replacement policies in local storage hierarchies are able to simultaneously take into consideration multiple access features such frequency and recency in order to maximize the hit ratio across different workloads. In specific, the Adaptive Replacement Cache (ARC) has been designed to automatically keep a balance between recency and frequency in an online and selftuning manner [9]. In addition, a very interesting deterministic online algorithm for replacing files of specific size and retrieval cost in a limited-size cache has been proposed [19]. However, in the area of web proxies, the most successful policies combine recency with file size, popularity or fetching latency.

Hence, the selection of an effective data replacement algorithm to manage data when the proxy disk cache fills up is essential. In our proposed replacement algorithm we do not only take into account two major factors, recency and fetching latency, but we also try to maintain locality in the data that are already cached on the proxy's hard disk.

4.2 Design goals

With our proposed architecture we initially intended to improve the performance of accessing data from the proxy's disk cache. We wished to enhance Andrew's storage management and replacement methods in order to support concurrent requests from multiple clients in distributed environments. As we have already mentioned in previous sections, the majority of the published literature on proxy caching architectures typically refers to web environments with predominantly read-only workloads of limited reliability and consistency demands. In addittion, web proxies support access granularity of entire files and have limited security constraints due to the public nature of the transferred data. Furthermore, the disk-based caching system that is most commonly used for file systems has mainly been developed to run directly on personal workstations and is not optimized to support concurrent requests from large numbers of users arriving from different client

machines. Thus, the design of a new proxy that would satisfy all the above criteria is essential. To be more specific, an efficient caching proxy server for distributed file systems must achieve the below basic goals:

- Adopt an innovative way of managing requested data on the proxy cache to improve the spatial storage locality.
- Create clusters of files in the proxy cache that have common characteristics, in order to improve file access performance by reducing
  - storage space fragmentation
  - metadata management overhead
- Relieve clients that issue a burst of concurrent requests in a wide area network, as the proxy cache is preferred to a disk-based caching system to satisfy them.
- Use existing standard protocols to allow reuse of data available on the proxy server across different clients.
- Replace files that have not been used recently with consideration of their fetching latency from the origin server.

4.3 Proposed architecture

In this section we present a high-level description of our caching proxy server. First, the new allocation algorithm is depicted which modifies the way remote data are cached on proxy's hard disk as well as the corresponding mapping between remote and cached data. Then, we introduce the replacement algorithm that is used to effectively manage data in the proxy cache when it fills up. Our basic goal is to replace files that are not frequently used while the time to fetch them again from the remote server is relatively low. At the same time, we try to maintain locality in the cached data to avoid fragmentation. Consequently, we prefer to replace some space from the last cache file that has already been used for replacement.

Below we present the block diagram of the proposed architecture that expands Andrew File System over two directions: storage management and file replacement.

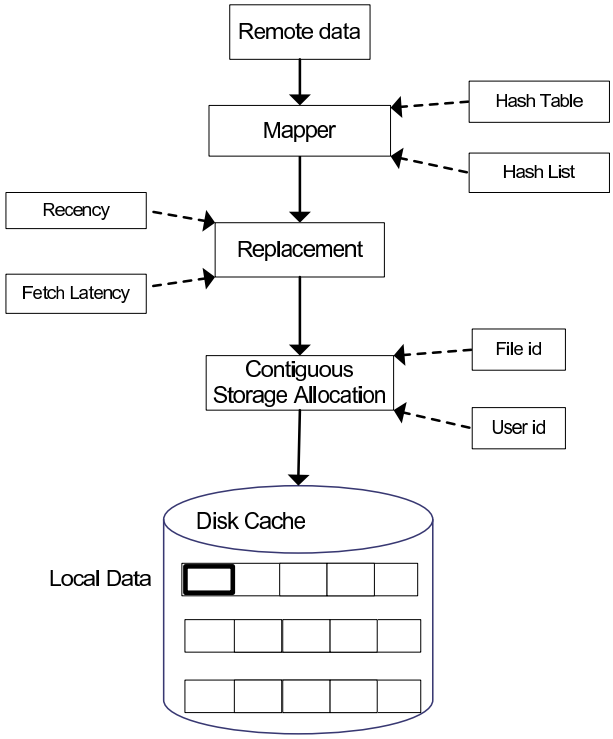


Figure 4.4: Proposed architecture

4.3.1 Storage management

To satisfy our basic goals, we must adopt a new locality-aware storage management method that changes the way remote data are stored in the proxy cache. As latencies occur due to fragmentation in the proxy cache, we modified the way remote file parts are stored on the proxy's hard disk. The Andrew File System stores each remote data part as a separate file on the disk cache. Therefore, when multiple clients need to access different remote files, the data parts of each file may be scattered over the proxy' s disk cache. Consequently, the overall time to read each file from the disk cache may be much greater than the corresponding time to read the same file, in case the initial requests were issued sequentially.

Thus, the chunks of a single remote file must be grouped together in nearby locations at the proxy cache. In our system, we investigate the placement of incoming data into consecutive locations grouped by the identifier of the remote file to which the data corresponds or the requesting user. More specifically, we organize the chunks as contiguous segments of a large file in the proxy server, called *proxy file*. The chunks of the same remote file that we fetch with a single request from the origin server are stored consec-

utively at the same local file. In addition, we try to keep in nearby locations parts of the same file that are not requested concurrently. We also store on the same local file the data chunks fetched subsequently either from the same remote file, or from different remote files by the same user. The proxy file has size that is only limited by the local file system.

Due to the spatial locality that we enforce at the proxy, we anticipate that file access time will be improved as we manage to place parts of the same file in contiguous locations on the hard disk. Furthermore, we manage to reduce file management overhead as local cache files can keep data from different remote files whereas separate parts of the same remote file can be stored on contiguous locations inside a local cache file. If we only know the local cache file and the coresponding slot where the remote file's contents are stored, we can retrieve the requested data. Hence, we do not need to manage metadata for remote files' contens. In contrast, only metadata for local cache files need to be managed. Additionally, data repetitively used by a user can be retrieved from the proxy cache with low access overhead. However, the actual performance seen by the end user also depends on the behavior of the other users concurrently utilizing the same proxy server.

Apart from improving spatial storage locality and the corresponding file access performance, our system also achieves our third goal because it no longer assumes that client machines are powerful enough to release centralized servers from computations. Once a file is fetched and stored in the proxy cache, future requests that are issued by client machines for this file will be satisfied from the intermediate proxy. Consequently, the general load at the servers will be significantly reduced.

4.3.2 File Replacement

The deployment of file system proxies is mainly motivated by the need to access data across wide-area networks. As a result, different files requested from the proxy incur fetch latencies that vary according to the actual location of the origin server. In our replacement policy, we keep track of the amount of time needed to fetch each chunk to the proxy. We aim to preserve locality and avoid fragmentation during replacement. Thus, we treat as a single unit, called *chunk run*, the group of chunks that are stored consecutively on the proxy and belong to the same remote file.

For each run, we keep track of the average fetch latency across its chunks. We categorize the chunk runs as *local* or *remote* depending on whether their average latency is lower from or exceeds a configurable threshold. At the next replacement operation, we pick as victim the chunk run that is earliest in the LRU list and has the lowest average latency. As a result, we first favor the local runs for replacement. If our search for a local run fails in a pass along the LRU list, then we pick for replacement the remote run that has been least recently used.

4.4 Summary

Caching has long been used to achieve effective file storage in distributed environments. Especially in wide area networks, intermediate caching proxies that cache data closer to the clients are frequently used. They intend to reduce the total cost of accessing remote data while they manage to significantly reduce the load at the remote fileservers. However, to design an effective caching proxy server a variety of resource management issues have to be thoroughly examined such as storage space management and file replacement. Data must be located in the proxy's disk cache in such a way that the performance of accessing cached data is relatively high. Additionally, metadata must be organized in an effective way to achieve better mapping from remote to local data. Furthemore, data must be replaced in such ways that reduce the overall access cost.

AFS is a distributed file system that has been successfully used for over two decades in general file systems. It is better used in wide area networks where latencies introduced by the metwork encourage the design of a client-side disk cache for effective storage management. In an effort to understand Andrew's caching efficiency, we experimented with its open-source variant. We concluded that concurrent file reads may cause storage space fragmentation due to the existing file management methods.

In the present thesis, we propose a new caching proxy server for distributed filesystems, based on Andrew File System. We present a new file management method that modifies the way remote data are kept in AFS client's cache, in order to satisfy concurrent requests and improve overall performance. However, AFS does not offer a caching proxy server as it limits caching to the local filesystem. In the design we propose, we aim not only

at improving Andrew's caching efficiency but we plan to offer a new caching service to support multiple cncurrent clients in a distributed file system. Finally, we propose innovative file replacement methods that enhance the existing replacement methods of Andrew File System. Thus, we manage to preserve locality and avoid fragmentation based on the following idea: we replace files that have not been used for a long time and can be accessed quickly in future requests.

Chapter 5

Implementation of Hades

In this chapter, we present the basic modifications made to the OpenAFS client in our implementation of the Hades proxy server. The new locality-aware allocation algorithm is introduced, that groups together either remote data with the same file identifier or files requested by the same user in a distributed environment. As a result, the performance of accessing data from the proxy's disk cache is significantly improved, especially in cases of multiple concurrent requests from different users. Finally, we display the extensions made to the existing replacement algorithm that wish to improve performance while preserving locality and avoiding fragmentation in the proxy cache.

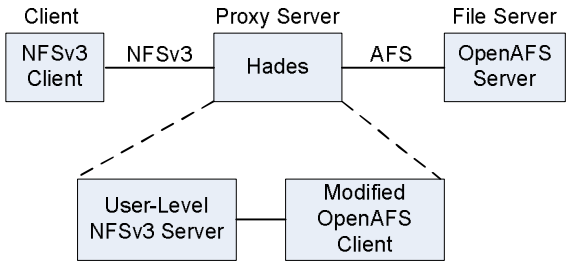


Figure 5.1: The Hades system combines a modified OpenAFS client with a user-level NFS server

5.1 Hades proxy server

We implemented the *Hades* proxy server as a combination of a modified OpenAFS client and a regular user-level NFS server as shown in Figure 5.1. Regular OpenAFS file servers export multiple files that can be accessed from different concurrent OpenAFS clients. Hence, we modified appropriately OpenAFS client to allow Hades access such remote files. Hades can initially access remote files through a modified OpenAFS client, and re-export the accessible remote files through a regular NFS server. Consequently, Hades acts as a client and a server at the same time. Finally, client machines use a normal NFS client to avoid accessing remote files from the remote file server. Users issue requests that are better and more efficiently satisfied from the intermediate Hades proxy server.

One of our main design goals was to explore alternative methods for the mapping of the remote data to local cache files, in order to improve the existing file retrieval performance. Thus, we proposed and implemented a method that modifies the existing mapping techniques as well as the way that remote files are stored in the local disk of the OpenAFS client. We keep remote files in consecutive offsets in each local file at the proxy cache. What is more, we retain different parts of the same remote file in nearby disk locations. Our intention is to reduce storage space fragmentation as well as the corresponding disk access overhead. In addition, we reduce metadata management overhead as for each remote file we have to manage only one and not multiple local cache files.

To achieve our design goals we expanded the OpenAFS client along the following three directions:

1. We preallocate multiple large local files and do our own space management for each of them.

2. We expand the mapping structure of each local file to store multiple chunks that belong to different remote files.

3. We keep low the average access cost by replacing locally cached chunks according to their access recency and fetching latency.

Below, we explain in more detail our implementation along each of the above directions.

5.2 Cache Files

A number of preallocated fixed-size cache files are initially created in the proxy' s cache. The maximum size of each local file is only limited by the settings of the local file system at the proxy as well as the needs of the clients' applications. A small number of large cache files is usually needed. We consider as parameters: the number and the size of the local cache files as well as the number of the different remote AFS files that can be stored in each local cache file along with their maximum size. We use a separate bitmap to manage the storage space of each file and we call *cacheblock* the respective unit of storage allocation. Cacheblock is a parameter of our system that can be configured according to our needs. The default cacheblock size is 4KB. The size of chunks that we transfer between the proxy and the origin server is typically a multiple of the cacheblock size. When a client issues a request for a part or an entire AFS file

1. a connection with the remote fileserver will be established

2. the data will be fetched from the remote server to the proxy's kernel memory in parts of default chunksize (usually 256KB)

3. a cache file will be chosen

4. the data will be stored in the local cache file in parts of usually 4KB size

To choose the right local cache file for the remote data, we search for the requested number of consecutive cacheblocks starting from the local file that was used more recently.

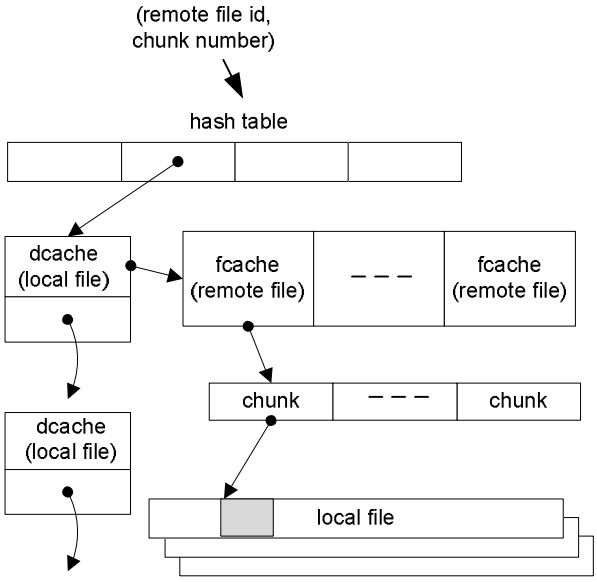


Figure 5.2: The main structure of the modified OpenAFS client in Hades.

5.3 Bitmap List

A list containing special bitmap structures relative with the local cache files is retained in the proxy's kernel memory. These structures are inserted in the bitmap list when initially created. Each such bitmap consists of k bits, representing the k blocks of each local cache file, where $k = Filesize\ /\ cacheblock$. Thus, for each local cache file we can find which blocks are free to store remote data. We try to store remote files in consecutive cache file offsets, as applications request them, to maintain locality and avoid space fragmentation. To achieve this, we must search for consecutive space in one local cache file to store the requested data as a whole. We must note that bitmap access is atomic. Two different processes are not able to search concurrently for free space in the same local cache file. For an empty cache, the data initially requested will be stored in the first cache file at the bitmap list that has enough free space to store the entire request. Processes can only access atomically the bitmap list to reserve bits. We then map remote data to the corresponding local cache file and store the fetched chunks at it.

5.4 Mapping

To achieve one of our basic goals and retain parts of the same file in nearby disk locations, we had to expand the vcache structure. Thus, for each remote file cached at the proxy, we added pointers to the bitmap and respective dcache structures of the local cache files that keep parts of it. In more details, *bmap* is the corresponding bitmap pointer used to search and allocate free blocks in order to store the requested remote data, while *lchunk* is the corresponding dcache pointer. If this is the first time that a request for this file is issued, dcache and bitmap pointers are NULL. Each time remote file parts are fetched and stored in cache, the above pointers are updated to point at the right local cache file. Additionally, we added other fields that include the local file offset where the request is stored, the initial chunk number and the total size of the requested data. To be more precise, *StartOffs* is the local cache file offset where the remote data is stored. After searching in the bitmap list, we compute StartOffs byte by multiplying the first free bit (first bit with value 0) in the bitmap with the cacheblock parameter. Every remote AFS file is divided into one or multiple data chunks, each of whom has a unique chunk number between 0 and a maximum value depending of the file's size. Similarly, *InitChunk* is the number of the first chunk of the requested data, used to compute the initial local file offset where each remote chunk will be stored. For example, if we issue a request for a remote file of size 1MB, the remote data will be fetched from the remote server in four chunks of 256KB. The initial chunk number is 0 and the initial offset of each chunk is computed by the macro AFS_COMPUTEINITOFFSET as:
InitialOffs=AFS_COMPUTEINITOFFSET(StartOffs,(chunk - InitChunk));
where #define AFS_COMPUTEINITOFFSET(A,B) (A + (B afs_OtherCSize)),*
afs_OtherCSize is equal to chunksize, as this is defined by the system.
Finally, we added the field *size* to represent the initial size of each file request. This is essential as we must know beforehand the total number of bits we have to allocate from the bitmap to store the requested data.

The in-memory dcache structures as well as their corresponding on-disk images, are usually used to map successfully remote file chunks to local cache files. Hence, we expanded these structures to support the Hades implementation because it is different from the OpenAFS implementation in the way remote data is mapped and stored in local cache

files. Each local cache file may be used to keep different parts of remote files in contrary to the OpenAFS implementation where each local cache file keeps only a unique chunk of a remote AFS file. Consequently, we expanded the dcache structure of each local file to maintain an array of pointers to fcache structures of remote files. The basic field we added to the dcache structures is *struct fcache *Rfiles*, to associate a local cache file with all the remote files that store chunks there. This is a departure from the original OpenAFS implementation, where each local file could only store one chunk of a single remote file and only needed one fcache pointer. Further more, we expanded the corresponding fcache structures to associate each on-disk image with the chunks of remote files that are stored in it. In each fcache structure that belongs to the above array, we maintain an array of chunk descriptors called *chunkT*, to locate all the chunks of remote files that have been stored in the respective local cache file. In such descriptors we keep information about the corresponding chunk of remote data, including the chunk number, the chunk's total size as well as the chunk's starting and ending offset at the local cache file.

5.5 Allocation

5.5.1 Data clustering based on the remote file's identifier

As we have already pointed out, our main intention is to maintain the requested data in consecutive byte offsets. Assuming *temporal locality*, we argue that if a remote data part is requested once, it is possible enough that a future request for a consecutive part of the same file will be issued. Thus, we aim at keeping parts of the same remote file in nearby locations in the proxy's disk cache, in order to minimize the disk access/fragmentation overhead that may occur when a future read request for this remote file will be issued. If we receive a request for part of a remote file, we must first identify if this data is already cached. We can use the dcache and fcache structures along with the hash tables to map the identifier and the chunk of the remote file to the local file and the corresponding offset where it is stored.

If our search fails, the requested part is not locally cached and we have to reserve the needed number of chunks at the first local file (starting from the last used) that has

enough consecutive space available. The allocation is done according to our allocation algorithm that was previously described. We must first check whether another file's part is already cached. If so, we examine whether there is enough free space to store remote data in this last local file so as to keep these two data parts in nearby locations. If this local cache file does not have enough consecutive free space, we keep searching in the subsequent local cache file. If none of the local cache files satisfies our criteria, we must free some space in the disk cache according to our replacement algorithm. On the other hand, if this is the first time we receive a request for a remote file, we search for enough free space starting from the local cache file that was last used. If we have not freed any space in the disk cache, we choose the first local cache file in the bitmap list to store the new data. Otherwise, we select the last file where an existing data part was replaced by a new file request. In both cases, we search for a local cache file that has free space to store the remote data consecutively and in total.

When we have finally chosen the local cache file to store the remote chunks, we must properly update the corresponding bitmap structure along with the dcache and fcache structures to keep track of the remote chunks that will be cached. We must then compute the starting and ending offset of each chunk and update the appropriate fields in the array of chunk descriptors in the corresponding fcache structure. Then, the remote data chunks will be fetched and stored locally in cacheblocks. It must be pointed out that writes to the same local file are implemented atomically.

5.5.2 Data clustering based on the user's identifier

In order to improve read performance of cached data requested by the same user, we introduced a new criterion in our allocation algorithm. We keep in mind the id of the user making the request in addition to the remote file's id, in order to keep in the proxy cache clustered the data of different files requested by the same user. We must notice that in the original OpenAFS implementation, a special structure is kept for each active user in a linked list maintaining all users issuing requests for remote files. When we receive a request for a remote file, we must firstly identify the corresponding user. We then check if this user has already issued another remote file request. Thus, we added to the above structure a pointer *bmap* to the bitmap of the local file where the user cached data more

recently. During a request, if the user attempts to cache data for the first time, the bitmap pointer is null and we only cluster data based on the identifier of the remote file. Otherwise, we examine two cases

1. It is the first time that a part of this remote file is requested. In such cases, we take into account the user's id criterion and search for free space in the local file where the user last cached data. If we don't find sufficiently large space to fit the request there, we continue the search in the subsequent local file.

2. Data of this file are already cached. In such caces, we search for free space according to the first file id criterion, as we aim at keeping parts of the same file in nearby locations at the proxy's disk cache.

In both cases, when the local cache file to keep remote data is finally selected, we update the corresponding fields of the structure kept for the user and the vcache structure, to satisfy future requests issued by the same user.

5.6 Hashing

To map a remote byte offset to the corresponding local cache file offset, we must follow the following steps:

1. Find the chunk number of the remote AFS file for the given byte offset.

2. Compute the relative offset of the given byte offset into this chunk.

3. Discover the local cache file where this chunk is stored. In this case, mapping is implemented through search in the appropriate structures.

4. Find the offset within the local cache file where the resident byte is located.

5.6.1 Hash lists

Our implementation differs from the original OpenAFS implementation in the way the search is implemented in step 3. In the mechanism that we use to map remote chunks to local file offsets, we use a variation of the hash table used in the original OpenAFS.

We should note that in our implementation the space of each local file is partitioned across the different chunks of multiple remote files. This was not the case in the original implementation of OpenAFS, where each local file could only store a unique remote chunk. Therefore, our system hashes different chunks to the same entry of the afs_dchashTbl hash table. In order to address the need to map different chunks to the same local file, we remove the auxiliary afs_dcnextTbl table that was previously implementing an open addressing scheme for collisions. Instead, we implement hashing with chaining after attaching a linked list to each entry of the hash table, as it is shown in figure 5.3.

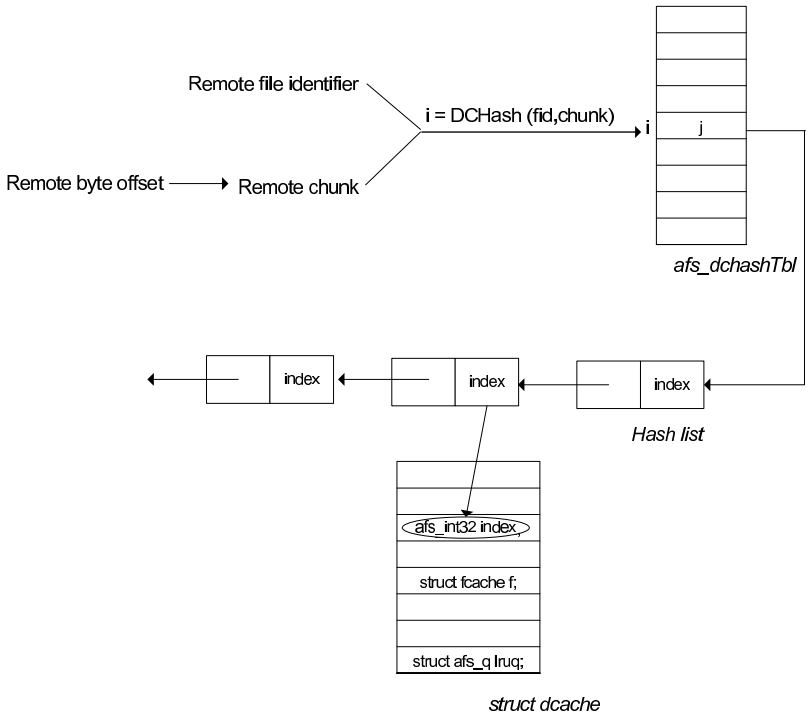


Figure 5.3: Using hash tables and hash lists to locate remote data at the proxy cache of Hades.

We introduced hash lists as in our implementation multiple remote files can be stored in one local cache file in contrary to the original openAFS implementation. If we maintained hash tables, it could be possible to face the following problem: for two different remote files that are cached in the same local cache file with index i, the corresponding index would appear multiple times in the offset i of the hash table or in the corresponding hash chains. Consequently, the search in the hash tables might not be accomplished successfully. However, using hash lists instead of hash chains means that every local cache file index i would appear only once in each linked list, even if there exist two different

remote files that have been cached in this file. Hence, the total size of each linked list is reduced as every index i would only appear once in each local cache file for all remote files that are cached in it.

5.6.2 Searching in hash lists

Each element of the original afs_dchashTbl is a pointer to a linked list. For example, the pointer in the offset i of afs_dchashTbl structure points to a linked list containing structures with a common characteristic: they maintain indexes of the local cache files that keep parts of remote AFS files for whom the fid and chunk hashing will return the number i. In more details, when a new remote file request arrives we must first allocate the appropriate local cache file to store the remote data. We must then hash the identifier of the remote file and the requested chunk number to a hash table entry so as to examine whether the index of this local cache file exists in the appropriate hash list. We then search through the attached linked list for a dcache structure that contains pointer to the requested remote file. The fcache structure that corresponds to the remote file should also contain pointer to the requested chunk number. If the search succeeds, we found the chunk locally cached and so we move it up in the beginning of the linked list to be able to find it easier in future requests. Otherwise, we add a new node to the linked list of the hash table and make it point to the right dcache structure after the chunk is transferred from the remote server.

5.7 Replacement

The existing replacement method of OpenAFS only considered the files' recency when selecting which files to replace from the proxy cache. One LRU algorithm was initially used to find the files that have been least recently used. Then, some of these files were selected as victims for replacement.

Instead, we take into consideration another parameter called *fetch latency*, that we add to each list node in the LRU list. There, we store the amount of time needed to fetch each chunk from the remote server to Hades. We consider a chunk as local if its fetch latency is lower than a preconfigured threshold. During replacement, we prefer as victims

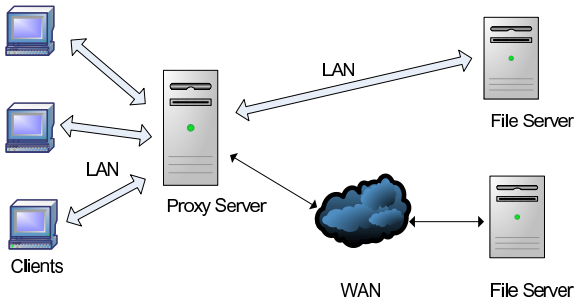


Figure 5.4: We prefer as victims for replacement the least recently used local files rather than the remote ones.

the local LRU chunks rather than the remote ones (figure 5.4). We should notice that our basic criterion still remains the file's recency. However, fetch latency identifies which files can be fetched in a relatively short time. The main idea of our replacement algorithm is the following: we want to replace files that have not only been used for a long time but can be accessed again quickly, in case users issue future requests for them.

It must be pointed out that our replacement method should preserve locality and avoid fragmentation. Consequently, we group together chunks that are stored consecutively in the disk cache and are parts of the same file. We use the term *chunk run* for these chunks. Accordingly, in allocation algorithm of Hades, the last chunk run that was used to replace files is preffered as the first place in proxy cache to search for free space. To maintain locality and avoid fragmentation, we try to replace consecutive space from a single chunk run before we move to another. We treat chunk runs as a single unit, so we assign a fetch latency and recency value to each of them. For the fetch latency we compute the average of the corresponding values that are assigned to each chunk of the unit. The chunk run's recency is determined according to the least recently used chunk of the run. Thus, in case we need to replace some files in the proxy cache, we choose the chunk run that is the least recently used and has the lowest average latency. A remote chunk run will not be chosen as a victim, unless none of the local files satisfies our criteria. If we pass the LRU list and do not find a local chunk run with a desirable fetch latency, then we replace the remote chunk run that has been least recently used.

5.8 A File Retrieval Example

Let's assume that a local application in an openafs client machine issues an open and a subsequent read request for a remote AFS file. The file open system call is usually intercepted and diverted to the kernel. The linux kernel identifies that this isn't a local file but a remote AFS file. The file open system call is then intercepted by a small 'hook' installed in the workstations's kernel. The user's program is then suspended and the intercepted request is diverted to a special program, implemented as a user-level process on the workstation, the Cache Manager which will implement the open and read system calls. The Cache Manager must first check if we have the permissions to open the remote AFS file. It then associates this file with a structure in kernel-memory that is kept for every remote AFS file that is requested by the users, the struct vcache. In case the file needs to be opened in O_TRUNC mode, the current time is kept in one of the fields of the above in-memory structure, so as to retain file consistency. Whenever a file needs to be opened, the kernel must associate it with a special structure kept in kernel's memory, the emphstruct file. A struct file is a kernel structure that never appears in user programs. The file structure represents an open file. (It is not specific to device drivers; every open file in the system has an associated struct file in kernel space.) It is created by the kernel on open and is passed to any function that operates on the file, until the last close. After all instances of the file are closed, the kernel releases the data structure. As a result, the remote AFS file is being opened and associated with an appropriate struct vache and struct file, used to handle the operations on this file.

If the remote AFS file has been successfully opened, the next step is to check whether the remote AFS file is already in memory (in the page cache) so as to read it from there. If not, the Cache Manager must find out if it is already stored in the cache. The hash table and the corresponding hash lists are used for this purpose in order to identify the local file that stores the requested chunk.

However, if the chunk is not locally available, the Cache Manager picks a local file and opens it for access. We favor the last local chunk run that was used for replacement, if it exists and has enough space, or the local file where parts of this file were cached lately. It then updates accordingly the dcache and fcache structures to keep track of the remote data.

Since it has opened the cache file, the Cache Manager must fetch the data from the remote AFS fileserver and store them in the open cache file. The file data is being fetched from the remote fileserver using RPC system calls. The server sends RX packets of size equal to chunksize. We must note that if a remote AFS file needs more than a file to store its data, we must make different RPC calls to fetch the data. Each cache file is filled independently, as we fetch data by making a separate RPC call to the fileserver. The requested data is transferred from the remote server to the local kernel buffers. The Cache Manager then reads the data from the kernel space per 4096 bytes and writes them to the appropriate cache file. Subsequently data is copied to the user-level address space of the application and the local cache file. It must be noticed that the Cache Manager tries to read the remote data per 4096 bytes and repeats the same process for every individual page. The Cache Manager then truncates the cache file to keep its correct size and closes it, freeing the struct file that was previously allocated for the cache file. If subsequent requests are issued for the same file or parts of it, we take advantage of locally cached file contents, and also metadata related to volumes, remote files and local chunks. Thus, the requested data is accessed from the proxy cache in a much faster manner.

In future read requests for the same file, data is read in pages of 4096 bytes. For each page, we repeat the same process. The remote offset is mapped into a local cache file offset, the local file is opened, data is read from the appropriate offset, copied to the user space and the local cache file is finally closed.

Chapter 6

Experimental Evaluation

In the present section, we first describe the experimentation environment that we used to develop and evaluate the Hades prototype. Then, we make an extensive experimental evaluation on the parallel retrieval of remote files and the reuse of cached data across multiple clients.

6.1 Environment

In our experiments, we used rack-mounted x86 servers with one quad-core processor 2.33GHz, 2GB RAM and gigabit ethernet nic. Every server has two SATA disks each of 250GB, 7.5KRPM and 16MB buffer. We modified the kernel module of the open-source variant of Andrew File System (OpenAFS 1.4.5) over the Debian distribution of Linux kernel version 2.6.18. We use Kerberos version 5 and version 2.2 of user-level NFS server. Unless otherwise specified, we used the default chunk size of 256KB and cacheblock size 4KB, respectively, for transfer and storage of data at the proxy cache.

6.2 Retrieval of Cached Data

Our first set of experiments uses a microbenchmark that we run directly at the proxy server. Our purpose is to evaluate the comparative advantage of Hades with respect to OpenAFS, when we read files stored at the origin server. We measure the latency to read each file block and the corresponding transfer throughput at the proxy server. We consider three file access modes that differ in the concurrency of the transfers and the involvement of the origin server during their service. We refer with Par and Seq to the parallel and sequential transfers, respectively, and we use Cd and Wm for the cold and warm proxy disk cache. Below we describe our three access modes:

Par/Cd. The files are requested with the proxy cache empty. The proxy server first prepares the mapping from the requested files to the local files, then it transfers the files from the origin server to the local page cache in chunk units, and finally it copies the files to the user-level memory of the proxy server in blocks of 4KB.

Par/Wm. The files are requested concurrently after the proxy disk cache has been warmed up. We enforce local disk accesses by flushing the memory page cache before starting the experiment.

Seq/Wm. Depending on the file size, the previous two cases initiated multiple threads across one or two users to request concurrently multiple files. In this mode, we only have one user making a sequence of file accesses from the warm disk cache of the proxy server. In our experiments we transfer files of four different sizes:

100KB. We have either two users reading in parallel two separate sequences of 1000 files each, or one user reading a sequence of 1000 files.

1MB. We have either two users reading in parallel two separate sequences of 500 files each, or one user reading sequentially 500 files.

100MB. In the first two modes we have the transfer of five files in parallel, while in the third mode we only read one file sequentially.

1GB. We transfer in parallel five files for the first two modes, and do a sequential read of one file for the third one.

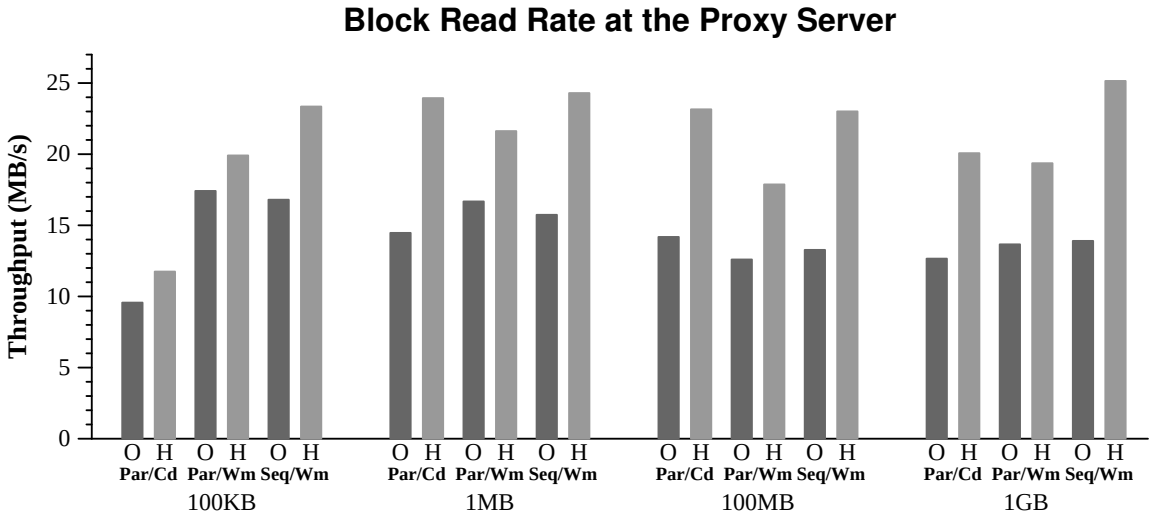


Figure 6.1: We measure the file access throughput at the proxy server across different sizes of transferred files. Consistently, Hades achieves a substantial throughput improvement with respect to OpenAFS that gets up to 80%. See text for explanation of the Par/Seq and Wm/Cd abbreviations.

In Figure 6.1, we measure the average throughput during the sequential and parallel file transfers across the different file sizes. Each experiment is run five times, in each mode and for each file size. We then calculate the average value of those runs in each case. The different runs of each case give similar values that converge to the same average value. It is remarkable that Hades improves the measured throughput across all cases. The lowest throughput that we measure is 9.56 MB/s, when the OpenAFS client reads in parallel two sequences of 100KB files from a cold proxy cache. The corresponding Hades throughput is 23% higher at 11.74 MB/s. The highest throughput of OpenAFS is 17.42 MB/s for 1000 files of 100KB read from a warm proxy cache, while the highest throughput of Hades is 25.15 MB/s for a single file of 1GB read from a warm proxy cache.

We attribute the improvement of Hades to different reasons across the cases that we examine. In Figure 6.2 we can see the breakdown of the block read latency. The read time of each block is spent across (i) mapping the requested block to the offset of the local cache file, (ii) fetching from the origin server and storage to the local cache, (iii) copying from the local cache to the user-level memory. In the category of fetching, we include the

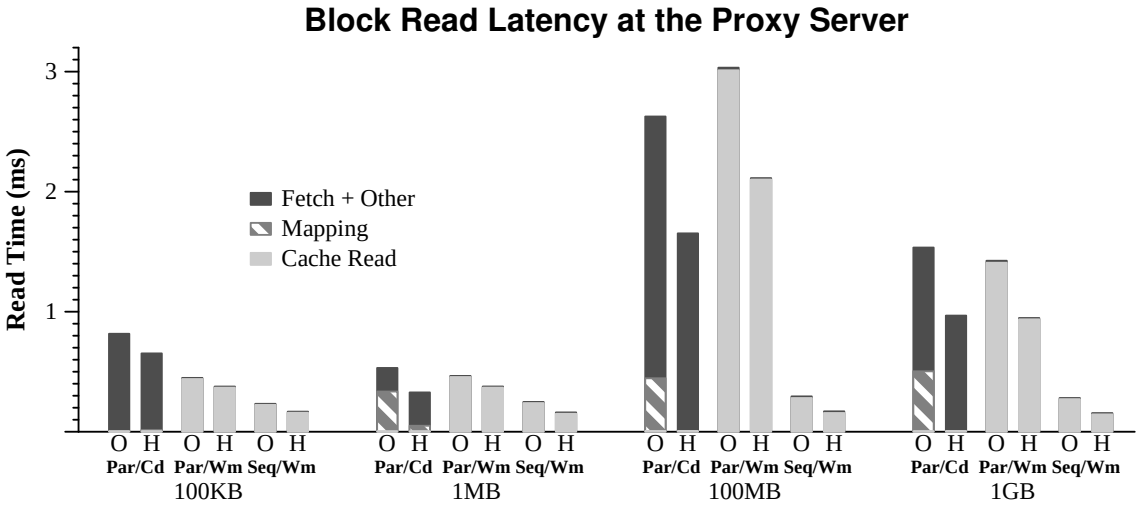


Figure 6.2: At the proxy server, we measure the time to read multiple files in parallel from the origin server (Remote), and in parallel (Parallel) or sequentially (Single) from the proxy disk cache. The latency to transfer each file block to the proxy server is broken down into fetching from the origin server, mapping to the local file, reading of the local file. In comparison to OpenAFS, Hades reduces substantially the block access latency up to 59%.

rest of unaccountable transfer delays.

We note that the initial read from the cold proxy cache incurs substantial mapping overhead in OpenAFS. This is the cost to insert into the hash structure the information to find the cached remote blocks next time we look for them. Hades avoids this overhead by storing together in an array of chunk descriptors the mapping of all the chunks that correspond to the same remote file. We simplified additionally the hashing structure by attaching a linked list to each entry of the hash table. The length of the lists is short, since we only use a limited number of large local files. Other optimizations that we did include adding a hint for the local file of each remote file, and moving to the front of the list a found local file.

When the files are accessed from a warm proxy cache (Wm), the component fetch+other is negligible. Also, the mapping overhead is insignificant after the mapping structure has been created during the warming up. Thus, the dominant component in accessing the warm cache of the proxy is to get the data from the local disk. The reduction of the block read latency during the parallel transfers (Par/Wm) of Hades can be attributed to the

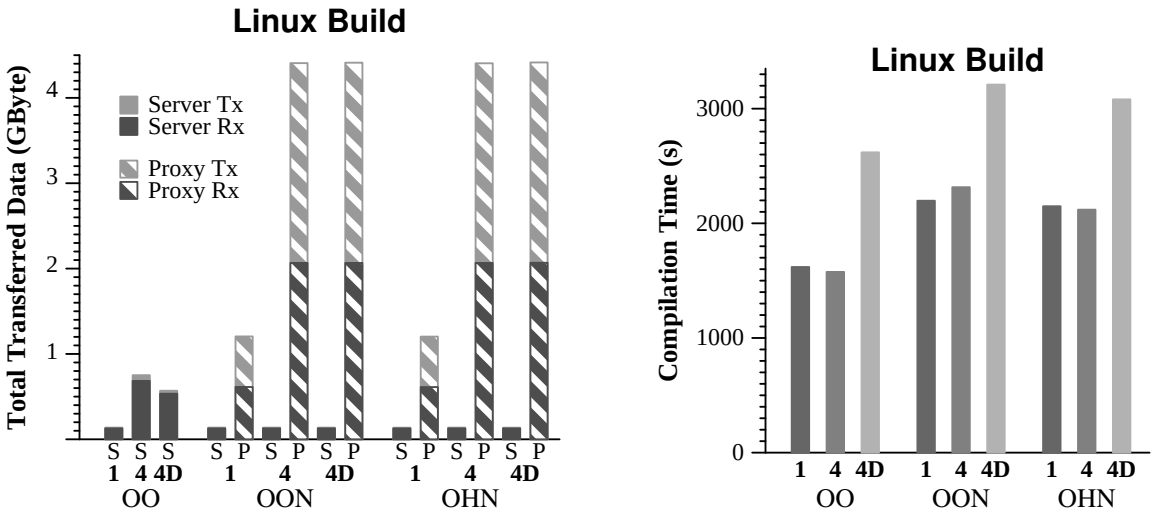
spatial locality in the storage of the cached data. In particular, we store to the same local file the chunks of either the same remote file or different remote files retrieved from the same user. Instead, OpenAFS distributes the retrieved chunks across an equal number of separate files in the proxy server. The same reason leads to the reduced block read time of Hades in comparison to OpenAFS when reading one or multiple files sequentially by one user (Seq/Wm). In comparison to the sequential transfer, parallel transfers share the available disk bandwidth and expand correspondingly the page read latency. For example, the bar height of the Seq/Wm measurement is approximately half or one fifth of the Par/Wm measurement depending on whether we have two or five parallel transfers.

6.3 Software Compilation

As an application to examine the general benefits of proxy caching, we use the building of linux kernel version 2.6.18. We assume that the source code is made commonly available from an OpenAFS volume (i) directly to OpenAFS clients (OO), (ii) to NFS clients through a proxy server running unmodifed OpenAFS client and user-level NFS server (OON), (iii) to NFS clients through the Hades prototype (OHN). We know in advance that the relative benefits of Hades in comparison to the original OpenAFS client are mostly evident when we retrieve large files from a warm cache. Instead, the present experiment we retrieve small files of a few kilobytes from a cold cache. Nevertheless, the software build is a baseline benchmark typically used in such types of experimentations [4].

In Figure 6.3(a) we measure the number of received and transmitted bytes at the origin (S) and the proxy (P) server, when we have one client (1), four clients (4) and four clients with the proxy at a distance from the origin of 50ms round-trip time (4D). Obviously, when we increase the number of clients from one to four, there is corresponding increase in the throughput of the origin server at the OO configuration. Instead, the intervention of the proxy server keeps constant the consumed bandwidth at the origin server, as we see in cases OON and OHN.

On the other hand, even with one client talking to the proxy, the NFS system consumes an excess of four time more network bandwidth than what OpenAFS requires for the same connection. Admittedly, the NFSv3 protocol that we use has been previously



(a) (b)

Figure 6.3: We build the Linux kernel on one (1) client, four (4) clients, and four clients with the origin 50ms away (4D). O refers to OpenAFS, N to NFS and H to Hades. The proxy cache is cold before each experiment that uses it. (a) We measure the total number of received and transmitted bytes in the origin (S) and the proxy (P) server. (b) With cold proxy cache, the intervention of the proxy server increases the compilation time. For retrieved files of only a few kilobytes each, Hades only achieves a modest reduction from 2315 to 2119 s (8.5%) in comparison to the original OpenAFS.

described as too chatty [4,13]. In fact, the version 4 of NFS makes more efficient the communication between the client and the server for example through delegations and compound statements. However, for a simple scenario, where multiple clients only read data from a common server without any modifications, the cost of NFS seems excessively high. Therefore, our consideration of OpenAFS as an alternative protocol for building proxy servers demonstrates a lot of potential. In Figure 6.3(b), we compare the compilation time across the different systems configurations and numbers of clients. As we see, the direct connection between the OpenAFS client and server leads to the shortest compilation time. The benefit of Hades with respect to the unmodified OpenAFS is only limited to 8.5%. This behavior is justified from the small file sizes that typically dominate source codes.

6.4 Summary

In summary, we notice that our decision to cluster at the proxy server the cached data requested from the same remote file or by the same user ends up to a substantially improved read performance from the warm cache. Additionally, we improve the read performance from the cold cache by making more efficient the mechanism of mapping remote chunks to local file offsets.

Overall, we conclude that proxy servers can reduce the required network bandwidth from the origin server, but they may introduce access delays during the first access of the requested data from a cold cache. Furthermore, OpenAFS requires significantly less bandwidth when compared to NFS, even though the latter is considered defacto choice for proxy server in the latest related research.

Chapter 7

Conclusions - Future Work

7.1 Conclusions

In the present thesis, we examined the design of Hades, a locality-aware proxy server for distributed filesystems, based on Andrew File System. We proposed a new storage allocation algorithm that alters the way remote data are kept in the disk cache of the proxy server. Furthermore, we presented a new file replacement method that keeps low the average cost while it preserves locality by replacing locally cached chunks according to their access recency and fetching latency.

Hades proxy server improves the efficiency of storage and metadata management in a distributed file system, by storing on nearby locations of the same local cache file parts of a unique remote file or files requested by the same user. The performance and related cost across different file sizes and numbers of clients was experimentally evaluated. We observed that Hades improves throuhput accross all different cases of concurrent file reads. Accordingly, Hades reduces the block read latency across parallel transfers or parallel retrievals from the proxy cache. In fact, it achieves a substantial throughput improvement with respect to OpenAFS that gets up to 80%. Furthermore, it reduces the

block access latency up to 59%. Finally, we concluded that Hades proxy server can reduce the required network bandwidth from the origin server.

7.2 Future Work

In the future, we plan to study more grouping criteria such as the identifier of the origin server that keeps the requested data. Furthermore, we wish to experimentally evaluate additional applications, such as accessing biomedical data from a warm cache rather than a cold one. Finally, we intend to investigate alternative data replacement methods using our prototype.

Bibliography

[1] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM Sigmetrics*, pages 34–43, June 2000.

[2] A. J. Borr. Secureshare: Safe unix/windows file sharing through multiprotocol locking. In *USENIX Windows NT Symposium*, pages 13–23, Seattle, WA, Aug. 1998.

[3] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, pages 193–206, Monterey, CA, 1997.

[4] A. Gulati, M. Naik, and R. Tewari. Nache: Design and implementation of a caching proxy for nfsv4. In *USENIX Conference on File and Storage Technologies*, pages 199–214, San Jose, CA, 2007.

[5] D. Howells. Fs-cache: A network filesystem caching facility. In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2006.

[6] S. Jin and A. Bestavros. Popularity-aware greedydual-size web proxy caching algorithms. In *IEEE International Conference on Distributed Computing Systems*, pages 254–261, Taipei, Taiwan, 2000.

[7] E. P. Markatos, M. G. H. Katevenis, D. Pnevmatikatos, and M. Flouris. Secondary storage management for web proxies. In *USENIX Symposium on Internet Technologies and Systems*, pages 93–114, Boulder, CO, 1999.

[8] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *ACM Symposium on Operating Systems Principles*, pages 238–251, Saint Malo, France, 1997.

[9] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies*, pages 115–130, San Francisco, CA, 2003.

[10] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems. In *USENIX Winter Technical Conference*, pages 305–313, San Francisco, CA, 1992.

[11] E. Otoo and A. Shoshani. Accurate modeling of cache replacement policies in a data grid. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 10–19, San Diego, CA, Apr. 2003.

[12] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. Nfs version 3 design and implementation. In *USENIX Summer Technical Conference*, pages 137–152, Boston, MA, June 1994.

[13] B. Pawlowski, S. Shepler, C. Beame, B. Callagham, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *SANE Conference*, Maastricht, Netherlands, May 2000.

[14] M. Satanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–21, May 1990.

[15] E. Shriver, E. Gabber, L. Huang, and C. A. Stein. Storage management for web proxies. In *USENIX Annual Technical Conference*, pages 203–216, Berkeley, CA, 2002.

[16] G. Sivanathu and E. Zadok. A versatile persistent caching framework for file system. Technical Report FSL-05-05, Department of Computer Science, SUNY Stony Brook, Stony Brook, NY, 2005.

[17] M. T. Stolarchuk. Faster afs. Technical Report TR 92-3, CITI, University of Michigan, Ann Arbor, MI, 1992.

[18] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software-Practice and Experience*, 20(3):225–242, 1990.

[19] N. E. Young. On-line file caching. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, San Francisco, CA, 1998.

Author's Publications

Lamprini Konsta, Stergios V. Anastasiadis, Hades: Locality-aware Proxy Caching for Distributed File Systems, Technical Report DCS2009-1, Department of Computer Science, University of Ioannina, January 2009.

Lamprini Konsta, Stergios V. Anastasiadis, Hades-Managing Storage in Caching Proxies for Distributed Filesystems, EuroSys, Glasgow, Scotland, UK, April 2008 (poster).

Short Vita

Lamprini Konsta was born in Preveza, Greece in 1983. She was admitted at the Computer Science Department of the University of Ioannina in 2001. She received her BSc degree in Computer Science in 2005 and she is currently a postgraduate student at the same department. She is a member of the Systems Research Group of the University of Ioannina since 2007. Her main research interests lie in the field of caching and storage systems.