SUBJECT: DC CASE STUDY

TOPIC: ANDREW FILE SYSTEM (AFS)

MEMBERS:

218A1098 – Abdul Riyaz Ansari

218A1102 – Jagadish Nadar

219A1099 – Vedant Birla

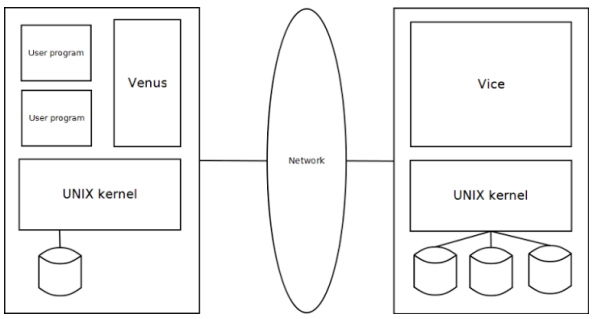
219A1100 – Artha Pillai

219A1101 – Damian Pillai

INTRODUCTION

The Andrew File System was introduced at Carnegie-Mellon University in the 1980's. The main goal of this project was simple: scale. Specifically, how can one design a distributed file system such that a server can support as many clients as possible? Interestingly, there are numerous aspects of design and implementation that affect scalability. Most important is the design of the protocol between clients and servers. In Network File System, for example, the protocol forces clients to check with the server periodically to determine if cached contents have changed; because each check uses server resources (including CPU and network bandwidth), frequent checks like this will limit the number of clients a server can respond to and thus limit scalability. AFS also differs from NFS in that from the beginning, reasonable user visible behaviour was a first-class concern. In NFS, cache consistency is hard to describe because it depends directly on low-level implementation details, including client-side cache timeout intervals. In AFS, cache consistency is simple and readily understood: when the file is opened, a client will generally receive the latest consistent copy from the server.

AFS ARCHITECTURE



Vice: The Andrew File System provides a homogeneous, location-transparent file namespace to all client workstations by utilizing a group of trustworthy servers known as Vice. The Berkeley Software Distribution of the Unix operating system is used on both clients and servers. Each workstation's operating system intercepts file system calls and redirects them to a user-level process on that workstation.

Venus: This mechanism, known as Venus, caches files from Vice and returns updated versions of those files to the servers from which they originated. Only when a file is opened or closed does Venus communicate with Vice; individual bytes of a file are read and written directly on the cached copy, skipping Venus

This file system architecture was largely inspired by the need for scalability. To increase the number of clients a server can service, Venus performs as much work as possible rather than Vice. Vice only keeps the functionalities that are necessary for the file system's integrity, availability, and security. The servers are set up as a loose confederacy with little connectivity between them.

AFS VERSION 1

The first version (which we will call AFSv1, but actually the original system was called the ITC distributed file system had some of the basic design in place, but didn't scale as desired, which led to a re-design and the final protocol (which we will call AFSv2, or just AFS). One of the basic tenets of all versions of AFS is whole-file caching on the local disk of the client machine that is accessing a file. When you open() a file, the entire file (if it exists) is fetched from the server and stored in a file on your local disk. Subsequent application read() and write() operations are redirected to the local file system where the file is stored; thus, these operations require no network communication and are fast. Finally, upon close(), the file (if it has been modified) is flushed back to the server.

When a client application first calls open(), the AFS client-side code (which the AFS designers call Venus) would send a Fetch protocol message to the server. The Fetch protocol message would pass the entire pathname of the desired file to the file server (the group of which they called Vice), which would then traverse the pathname, find the desired file, and ship the entire file back to the client. The client-side code would then cache the file on the local disk of the client (by writing it to local disk). As subsequent read() and write() system calls are strictly local in AFS; they are just redirected to the local copy of the file. Because the read() and write() calls act just like calls to a local file system, once a block is accessed, it also may be cached in client memory.

Thus, AFS also uses client memory to cache copies of blocks that it has in its local disk. Finally, when finished, the AFS client checks if the file has been modified (i.e., that it has been opened for writing); if so, it flushes the new version back to the server with a Store protocol message, sending the entire file and pathname to the server for permanent storage. The next time the file is accessed, AFSv1 does so much more efficiently. Specifically, the client-side code first contacts the server (using the TestAuth protocol message) in order to determine whether the file has changed. If not, the client would use the locally-cached copy, thus improving performance by avoiding a network transfer.

PROBLEMS WITH VERSION 1 AND IMPROVEMENT IN PROTOCOL

A few key problems with this first version of AFS motivated the designers to rethink their file system. To study the problems in detail, the designers of AFS spent a great deal of time measuring their existing prototype to find what was wrong.

In their study, the authors found two main problems with AFSv1:

• Path-traversal costs are too high-

When performing a Fetch or Store protocol request, the client passes the entire pathname to the server. The server, in order to access the file, must perform a full pathname traversal, all the way down the path until finally the desired file is located. With many clients accessing the server at once, the designers of AFS found that the server was spending much of its CPU time simply walking down directory paths.

• The client issues too many TestAuth protocol messages-

AFSv1 generated a large amount of traffic to check whether a local file (or its stat information) was valid with the TestAuth protocol message. Thus, servers spent much of their time telling clients whether it was OK to used their cached copies of a file. Most of the time, the answer was that the file had not changed.

The two problems above limited the scalability of AFS; the server CPU became the bottleneck of the system, and each server could only service 20 clients without becoming overloaded. Servers were receiving too many TestAuth messages, and when they received Fetch or Store messages, were spending too much time traversing the directory hierarchy.

Thus, the AFS designers were faced with a problem:

How should one redesign the protocol to minimize the number of server interactions, i.e., how could they reduce the number of TestAuth messages? Further, how could they design the protocol to make these server interactions efficient? By attacking both of these issues, a new protocol would result in a much more scalable version AFS.

AFS VERSION 2

AFSv2 introduced the notion of a call-back to reduce the number of client/server interactions. A call-back is simply a promise from the server to the client that the server will inform the client when a file that the client is caching has been modified. By adding this state to the system, the client no longer needs to contact the server to find out if a cached file is still valid. Rather, it assumes that the file is valid until the server tells it otherwise; notice the analogy to polling versus interrupts. AFSv2 also introduced the notion of a file identifier instead of pathnames to specify which file a client was interested in. An FID in AFS consists of a volume identifier, a file identifier, and a “uniquifier” (to enable reuse of the volume and file IDs when a file is deleted). Thus, instead of sending whole pathnames to the server and letting the server walk the pathname to find the desired file, the client would walk the pathname, one piece at a time, caching the results and thus hopefully reducing the load on the server. The key difference, however, from NFS, is that with each fetch of a directory or file, the AFS client would establish a call-back with the server, thus ensuring that the server would notify the client of a change in its cached state. Thus, in the common case where a file is cached at the client, AFS behaves nearly identically to a local disk-based file system. If one accesses a file more than once, the second access should be just as fast as accessing a file locally.

CACHE CONSISTENCY

Because of call-backs and whole-file caching, the cache consistency provided by AFS is easy to describe and understand. There are two important cases to consider: consistency between processes on different machines, and consistency between processes on the same machine. Between different machines, AFS makes updates visible at the server and invalidates cached copies at the exact same time, which is when the updated file is closed. A client opens a file, and then writes to it (perhaps repeatedly). When it is finally closed, the new file is flushed to the server (and thus visible). At this point, the server then "breaks" call-backs for any clients with cached copies; the break is accomplished by contacting each client and informing it that the call-back it has on the file is no longer valid. This step ensures that clients will no longer read stale copies of the file; subsequent opens on those clients will require a re-fetch of the new version of the file from the server (and will also serve to re-establish a call-back on the new version of the file). AFS makes an exception to this simple model between processes on the same machine. In this case, writes to a file are immediately visible to other local processes (i.e., a process does not have to wait until a file is closed to see its latest updates). This makes using a single machine behave exactly as you would expect, as this behaviour is based upon typical UNIX semantics. Only when switching to a different machine would you be able to detect the more general AFS consistency mechanism.

SCALE AND PERFORMANCE OF AFS V2

With the new protocol in place, AFSv2 was measured and found to be much more scalable that the original version. Indeed, each server could support about 50 clients (instead of just 20). A further benefit was that client-side performance often came quite close to local performance, because in the common case, all file accesses were local; file reads usually went to the local disk cache (and potentially, local memory). Only when a client created a new file or wrote to an existing one was there need to send a Store message to the server and thus update the file with new contents.

OTHER IMPROVEMENTS

The designers of AFS took the opportunity when building their system to add a number of features that made the system easier to use and manage. For example, AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines. AFS also takes security seriously, and incorporates mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired. AFS also includes facilities for flexible user-managed access control. Thus, when using AFS, a user has a great deal of control over who exactly can access which files. Finally, as mentioned before, AFS adds tools to enable simpler management of servers for the administrators of the system. In thinking about system management, AFS was light years ahead of the field.

FEATURES AND BENEFITS

File backups — AFS data files are backed up nightly. Backups are kept on site for six months.

File security — AFS data files are protected by the Kerberos authentication system.

Physical Security — AFS data files are stored on servers located in the UCSC Data Centre.

Reliability and availability — The AFS servers and storage are maintained on redundant hardware.

Authentication — AFS uses Kerberos for authentication. Kerberos accounts are automatically provisioned for all UCSC students, faculty and staff.

SUMMARY

AFS shows us how distributed file systems can be built quite differently than what we saw with NFS. The protocol design of AFS is particularly important; by minimizing server interactions (through whole-file caching and call-backs), each server can support many clients and thus reduce the number of servers needed to manage a particular site. Many other features, including the single namespace, security, and access-control lists, make AFS quite nice to use. The consistency model provided by AFS is simple to understand and reason about, and does not lead to the occasional weird behaviour as one sometimes observes in NFS.