CH 50

CH 50 – Andrew File System

- The main focus of AFS was to introduce **scale** into a distributed file system such that a server can support as many clients as possible. Also, AFS is very different from NFS, such that in NFS cache consistency is hard to describe since it depends directly on low level implementation details including client side cache timeout intervals. In AFS cache consistency is simple and readily understood: when a file is opened a client will *generally* get the latest consistent copy from the server.
- **AFS Version 1**: In this version, when an *Open()* call is made to the server, with the entire filename being sent to the server, the server sends the entire file to the client, which is then ***written*** to the local disk of the client. Any subsequent *Read()* or *Write()* functions are redirected to the **local disks** thereby increasing the efficiency and avoiding the network transfer of files and the content. Once the operation is completed *AFS Client* would check if the file was modified, and if so it flushes the new version back to the server with the path and a ***Store*** *protocol message*. Next time the file is accessed by this client, it would use the ***TestAuth*** *protocol message* in order to determine whether the file has been updated, since the last time the client accessed it. If so it would fetch and store the file in the client local disk, if not the client would directly access the cached version of the file, thereby avoiding the network transfer.
- **Problems with version 1**: **(1)** – **Path traversals were costly**: When performing the Fetch and Store protocol request, the entire path traversal sent by each of the client made the server spend too much of its CPU time simply walking down directory paths. **(2)** – **Too many TestAuth messages**: Like in NFS with GETATTR, AFSv1 generated a large amount of traffic only to check whether a local file was valid or has been modified, Thus, this resulted in there servers spending too much of their time on telling the clients whether it was *OK* to use the local cached copy of the file. **(3) Load was not balanced across Servers**: Solved with Volumes. **(4)** The server used a single distinct process per client thus inducing context switching and other overheads. All these problems resulted in limiting the scalability of the system.
- **AFS Version 2**: A notion of **callback** is introduced. In this the client assumes that a file is valid until the server tells it otherwise. This is t o reduce the TestAuth message protocol crowding on the server side. Also, AFSv2 introduced the notion of file identifier which includes – volume identifier, file identifier and a *uniquifier* which enabled the reuse of the volume and file IDs when a file is deleted. Thus instead of sending the entire path for a file the client would walk the pathname one piece at a time caching the results and reducing the load on the server. The key difference from NFS is that with each fetch of a path the AFS client would establish a callback with the server
- **Cache Consistency**: Because of callbacks discussed in the previous point, the notion of cache consistency is simplified. There are two important cases to consider – Consistency between processes on different machines, and consistency between processes on the same machine. **On Different Machines** –When a client gets a copy of the file, it would update the visibility and invalidate any cached copies, once the update (if any) is completed to the file, then the connection with server is broken for callbacks. When the client wants to restart working on the file it would initiate a new callback connection. This step ensures that that client will no longer read stale copies of the file and subsequent opens on those clients will require a re-fetch of the file. Thereby making a consistent copy of the latest version. **On Same Machine** – in this case writes to a file are immediately made visible to other local processes. **Also there is a notion of last writer wins**. Meaning that if there are multiple clients

editing the same file, the client which closes the file last would be the one whose changes would be saved to the AFS permanent storage.

- Performance of AFSv2: **First**, in many cases, the performance of each system is roughly equivalent. **Second**, an interesting difference arises during a large-file sequential re-read (Workload 6). Because AFS has a large local disk cache, it will access the file from there when the file is accessed again. NFS, in contrast, only can cache blocks in client memory; as a result, if a large file (i.e., a file bigger than local memory) is re-read, the NFS client will have to re-fetch the entire file from the remote server. **Third**, we note that sequential writes (of new files) should perform similarly on both systems (Workloads 8, 9). AFS, in this case, will write the file to the local cached copy; when the file is closed, the AFS client will force the writes to the server, as per the protocol. NFS will buffer writes in client memory, perhaps forcing some blocks to the server due to client-side memory pressure, but definitely writing them to the server when the file is closed, to preserve NFS flush-on-close consistency. However, realize that it is writing to a local file system; those writes are first committed to the page cache, and only later (in the background) to disk, and thus AFS reaps the benefits of the client-side OS memory caching infrastructure to improve performance. **Fourth**, we note that AFS performs worse on a sequential file over- write. Overwrite can be a particularly bad case for AFS, because the client first fetches the old file in its entirety, only to subsequently over- write it. NFS, in contrast, will simply overwrite blocks and thus avoid the initial (useless) read. **Finally**, workloads that access a small subset of data within large files perform much better on NFS than AFS. In these cases, the AFS protocol fetches the entire file when the file is opened; unfortunately, only a small read or write is performed. Even worse, if the file is modified, the entire file is written back to the server, doubling the performance impact. NFS, as a block-based protocol, performs I/O that is proportional to the size of the read or write.
- **Other Improvements**: **(1)** AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines. NFS, in contrast, allows each client to mount NFS servers in any way that they please, and thus only by convention would files be named similarly across clients. **(2)** AFS also takes security seriously, and incorporates mechanisms to au- thenticate users and ensure that a set of files could be kept private if a user so desired. NFS, in contrast, had quite primitive support for security for many years. **(3)** 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. NFS, like most UNIX file systems, has much less support for this type of sharing. **Finally**, as mentioned before, AFS adds tools to enable simpler man- agement of servers for the administrators of the system. In thinking about system management, AFS was light years ahead of the field.