AFS notes

1. Notes from reviews:
    a. How are concurrent accesses handled?
    b. Moving volume requires creating a copy
2. What is the key goal of AFS?
    a. Scalability: more clients
    b. Question: Why?
        i. Popular services, such as sharing data, tend to get more popular
        ii. Would like incremental growth – add new client, add a server to a pool, rather than stepwise growth
    c. Implications:
        i. Security becomes important (Note to holly: addressed in other papers; they do solve the problem in a good way)
        ii. Heterogeneous hardware/software
        iii. Hard to maintain same semantics as for a single site

3. Goals
    a. Network file system
    b. Scale – how big?
        i. Large number of clients
        ii. Client performance not as important
        iii. Central store for shared data, not diskless workstations
    c. Consistency
        i. Some model you can program against
    d. Reliability
        i. Need to handle client & server failures
    e. Naming
        i. Want global name space, not per-machine name space
            1. compare to NFS, CIFS
            2. Gain: transparency if file moved
4. **AFS** version 1:
    a. Process per client – like RPC, Pilot
    b. Name lookups on server
    c. Cache validation with callback on access
    d. Result:
        i. Low scalability: performance got a lot worse (on clients) when # of clients goes up
        ii. QUESTION: what was bottleneck?
            1. Server disk? Seek time ? disk BW?
            2. **Server CPU?**
            3. Network?
            4. Client CPU/Disk?

e. Evaluation performance: Andrew Benchmark
    i. Used by many others
    ii. QUESTION: What does it represent?
        1. A: nothing.
        2. Has a mix of workloads, can see how they respond
    iii. Pieces:
        1. Make dir – create directory tree: stresses metadata
        2. Copy – copy in files – stresses file writes / creates
        3. Scan Dir (like ls –R) – stresses metadata reads
        4. ReadAll – find . | wc – stresses whole file reads
        5. Make – may be CPU bound, does lots of reads + fewer writes
    iv. QUESTION: What is missing?
        1. All pieces do whole-file reads / writes
        2. Missing productivity applications, scientific applications
    v. QUESTION: they use a different platform for prototype and final version. is this relevant?
        1. A: the prototype evaluation is to show where bottlenecks are
        2. A: evaluation of final one shows what bottlenecks remain, compare against other systems

5. **AFS** v2
    a. CONTEXT: designed for systems with local disks
    b. QUESTION: What is the goal?
        i. Local-file Latency?
        ii. Local-file Throughput?
        iii. Server throughput?
        iv. Server latency?
    c. Transparent to clients – match Unix naming / Whole file caching
        i. QUESTION: Why not partial files?
        ii. Usage study shows most files accessed in entirety
        iii. Simplifies protocol / consistency
        iv. Read / write handled completely locally
        v. QUESTION: What workload is this optimized for
    d. Local disk cache
        i. QUESTION: Why? Increases latency (local disk access)
        ii. Reduces load on server by having a larger client cache
        iii. Stat information in memory (to avoid hitting disk, plus may be hard to write to disk)
    e. Relaxed but well-defined consistency semantics
        i. Get latest value on open
        ii. Changes visible on close

1. Write-through to the server (minimizes server inconsistency, but increases load compared to write-back when evicted)

iii. Read/write purely local – get local unix semantics
1. programs not location-transparent

iv. Metadata is global synchronous

v. QUESTION: different from Unix. Is it a problem? When?
1. Unix semantics
a. Any change to a file or file system visible to **next** operation (e.g. read returns data just written)
b. The last-close semantic is the semantic that requires that an open file remain available to any process which has the file open regardless of any changes in file or process characteristics which may take place after the file is opened. It is called the last-close semantic because the best known consequence of the last-close semantic is that when a file is deleted, the file is not removed until the file is closed by the last process which has it open.

vi.

f. Global name space: /afs
i. Names are same on all clients
ii. Can move volumes between servers, nothing changes

g.

6. Performance improvements
a. Call backs
i. Server notifies client if file changes
ii. QUESTION: RPC paper said use datagrams, not connections, maintain no state on server for scalability and crash recovery. What is the difference?
iii. QUESTION: Why?
1. Reduces load on server- no client polling

iv. How bad is it for the server?
1. QUESTION: how much state does the server manage?
a. Call back per file cached
2. QUESTION: What can the server do to reduce this?
a. Limit # of callbacks

v. QUESTION: How does this impact performance?
1. Closing a file can be slow because other cachers must be notified synchronously (before changing the data).

vi. What happens on failure?
1. After client failure, clients re-establish all callbacks
2. After server failure, …
3. QUESTION: do clients re-establish callbacks?

vii. QUESTION: are there other alternatives to callbacks
    1. Leases: callbacks that timeout automatically and don't have to be dropped. Less scalable – requires more frequent polling

b. Name resolution
    i. Id-based names
        1. Servers never do path lookups
            a. PRINCIPLE: make clients do the work – use the CPU cycles
    ii. 2-level name space
        1. Volumes + Files + uniquifier
            a. Use separate vnode # instead of inode# - not expose internal information or format of inode#s
        2. WHY?
            a. Efficiency: search n+m instead of n*m locations
        3. Why Uniquifier?
            a. Can reuse table slots – makes lots of things easy if you don't have to check for duplicates
            b. HOW IMPLEMENT? Machine ID + counter?
    iii. location independent names
        1. Can move volumes between servers
    iv. Clients cache volume → server mapping
        1. Volume location is a **hint**
            a. Piece of information that can improve performance if correct, but has no semantically negative consequences if incorrect

c. Threaded single-process server
    i. Thread per request, not per client
    ii. Allows overlapped network I/O
    iii. QUESTION: How do you set the number? What determines the number you need?
d. New open-by-ID file system call
    i. Can open by inode number
    ii. QUESTION: is it required that you have an ID for opening files?
        1. e.g. Windows makes this hard – doesn't have inode numbers
e. Summary: opening a file
    i. Walk path recursively
        1. If directory in cache with callback, go on
        2. If in cache w/o callback, check
        3. if not in cache, fetch + get callback
    ii. Open file
        1. If in cache with callback, use

2. W/o callback – very callback
3. not in cache – fetch w/ callback

f. Scalability results:
   i. QUESTION: do they show improved scalability?
      1. Definition from Cisco: Capacity of a network to keep pace with changes and growth.
   ii. QUESTION: are their results consistent?
      1. ANSWER: they do give standard deviations from multiple runs
   iii. Helped a lot
   iv. High level points:
      1. Shift work to clients
      2. Call-backs instead of polling
      3. Threads instead of processes
g. QUESTION: What workload is this optimized for?
h. QUESTION: what changes would you make for large file / random access workloads?
i. QUESTION: What else would have to change?
j. QUESTION: What happens when you open / read / write a file?
   i. Lookup each component directory of path name
   ii. Check cache first – if have in cache, and have callback, use
      1. Else ask server to update callback or fetch from server
   iii. Read/write: do locally
   iv. Close: copy changes up to server
   v. QUESTION: what about temp files?
      1. Don't put them on AFS – use local disks
k. New semantics:
   i. Get latest version on close
   ii. Write everything locally
   iii. Copy all back on close
   iv. QUESTION: how compare to Unix semantics?
   v. QUESTION: how would you detect the difference?

7. Manageability Improvements
   a. Volumes
      i. Contain a bunch of directories, but small enough to finit many on disk
         1. Allows management at a granularity higher than files, but smaller than disks
         2.
      ii. Unit of partitioning
         1. Use recursive copy-on-write to move data
      iii. Unit of replication
      iv. Unit of backup
         1. QUESTION: How do you get a consistent backup?
         2. Use copy-on-write to CLONE

v. Unit of applying quotas
vi. Logically separate from FS name space and underlying disk partitions
    1. Table of mount points indicates name space of volumes
vii. Volume mapping (what server has a volume) is SOFT STATE
    1. Can try to use it, but if stale, will learn real value
    2. PURELY OPTIMIZATION, LOW COST

8. AFS techniques
9. AFS scaling techniques
    a. Location transparency
        i. How?
            1. Name doesn't specify the machine containing the data
            2. Client machines aren't responsible for remembering where data is
        ii. Why?
            1. Can repartition data between servers
            2. Can move data to a new server without accessing all clients
        iii. Impact?
            1. Client names not always the most useful
            2. Challenges in merging two organizations – need a truly global namespace
    b. Client caching
        i. How?
            1. Cache lots of data on disk
            2. Cache whole files
            3. Non-coherent caching of filename->server mappings: hints
        ii. Why?
            1. Reduces load of serving bytes from server
            2. Reduces cache coherence – just check on open/close, not later
            3. Most reads/writes go to local disk
            4. Non-coherent names act as hints; can use, but detect failure and recover
        iii. Impact?
            1. Latency may be worse, as client disk is slower than server memory
            2. Latency to read large files may be bad
            3. Can't access files bigger than local disk
            4. Partial file caching without locking exposes problems with demand paging; subsequent pages may not be available
    c. Notification

i. How?
    1. Clients ask for callbacks when a file changes
ii. Why?
    1. Removes need to poll for changes to a file, e.g. check if changed on open
iii. Impact?
    1. Server must record state about client caches
    2. Complicates failure recovery; must re-establish callbacks

d. Partition data / aggregate data
    i. How?
        1. Group directories into volumes, smaller than a disk
    ii. Why?
        1. Small enough to be reasonable moved for load balancing
        2. Admins can think about volumes, not files or directories
    iii. Impact?
        1. May be difficult to support large files, directories with large amounts of data (exceeds volume size limits)

e. Replication
    i. How?
        1. Maintain multiple copies of read-only data
    ii. Why?
        1. Spreads load across more than one server
    iii. Impact?
        1. Must be concerned about how to update read-only data (it does happen occasionally)
        2.

f. Relaxed semantics
    i. How?
        1. Consistency is maintained as open-close consistency, not read-write consistency
    ii. Why?
        1. Allows a single consistency check – when opening a file
        2. More accesses go to local disk
    iii. Impact?
        1. Programs doing block-based sharing (e.g. databases) not supported
        2. Clients that can't cache whole files (e.g. small devices) need a separate protocol

g. Functional Specialization
    i. How?
        1. File servers are dedicated machines with more memory and disk

ii. Why?
    1. Removes need to fairly share between interactive tasks and file server
    2. Can optimize hardware/software on server; e.g. different scheduling decisions
    3. System can depend on more resources; not as worried about efficiency in low-resource environment
    4. Can assume file servers trusted, managed by centralized administration
iii. Impact?
    1. Not application to all environments – e.g. peer sharing

h. Move work to client
    i. How?
        1. Clients do name->id parsing
    ii. Why?
        1. Clients have extra cycles; they are waiting for a response anyway
        2. Client cycles scale with the number of cycles

i. Exploit workload properties
    i. How?
        1. Treat read-only files differently
        2. Tmp directories local
    ii. Why?
        1. Can avoid cache coherence
        2. Can replicate

j. Batching
    i. How?
        1. Do as many operations at once as
        2. Grant/revoke multiple callbacks at once
        3. Transfer more data at once
    ii. Why?
        1. Amortize startup costs

k. Minimize system-wide knowledge and change:
    i. How?
        1. Hints for volume->server mapping
    ii. Why?
        1. clients don't need full knowledge of all servers; file location at server level rarely changes
        2. Server can redirect them to correct location if mapping is old

10. Limits:

a. Can't support disk-less clients well
b. Can't handle large files well – need to copy in entirety
c. Latency to first byte for uncached files is high

11. Comparison to NFS
    a. Implemented on RPC / XDR for data format conversion
        i. QUESTION: Why? Is it necessary? Easier to port / heterogeneous
        ii. Can use RPC-level security solutions
    b. Stateless protocol
        i. QUESTION: Why? Easy recovery from failure
        ii. No information retained across RPC invocations
        iii. Easy crash recovery – just restart server, client resends RPC request until server comes back
        iv. Server doesn't need to detect client failure
        v. Problem: what if client retries a non-idempotent operation
            1. E.g. remove?
    c. Servers don't do name operations
        i. Clients work with File Handles – like AFS FID,
    d. Naming
        i. Servers can export any directory (like Windows sharing)
            1. Only exports a single file system – doesn't cross mount points
        ii. Clients mount anywhere in name space
        iii. Each client can mount files in a different place
        iv. QUESTION: What are benefits / drawbacks?
        v. QUESTION: How handle cycles?
            1. A: NFS servers won't serve files across mount points
            2. Clients must mount next file system below in the FS hierarchy
            3.
    e. NFS file semantics
        i. Clients cache data for 30 seconds
        ii. Clients can used cached data for 30 seconds without checking with server
        iii. Servers must write data to disk before returning (no delayed writing)
            1. QUESTION: What are performance implications?
        iv. Attribute cache for file attributes – kept for 3 seconds
            1. Used to see if attributes have changed
            2. Discarded after 60 seconds
    f. Caching
        i. Servers cache blocks
        ii. Clients cache blocks, metadata
            1. Attribute cache has metadata, checked on open, discarded after 60 seconds
            2. Data cache flushed after 30 seconds
        iii. ssues:
            1. Unix files are removed until last handle closed

        a. On stateless server, causes file to be deleted while still in use
        b. Solution in NFS: rename file, remove on local close
    2. Permissions may change on open files
        a. Unix allows access if have open handles
        b. NFS may deny access
            i. Solution: Save client credentials at open time, use to check access later
            ii. NOTE: server doesn't do enforcement here

g. QUESTION: What happens when you open / read / write a file?
    i. Open: client checks with remote sever to fetch or revalidate cached inode (if older than 30 seconds)
    ii. Reads handled locally, writes written back after 30 seconds
    iii. Nothing happens on close
    iv. Data flushed after 30 seconds – may not be seen by other clients for another 30 seconds
    v. Write: delayed on client for 30 seconds, then written synchronously to server
h. Design essence
    i. Stateless server for easy crash recovery (keep system simple!)
    ii. Relax consistency (no guarantees) to get better performance
    iii. Pure client server; no distributed servers
i. Results:
    i. Bizarre consistency semantics
    ii. Higher server load – must interact with client on reads / writes
    iii. Less caching on client
    iv. Faster error recovery – can just reboot server
    v. More network packets
    vi. Lower latency – don't have to wait to download file on open. Better for large random access files

12. Approach to consistency / durability

a. Move requirement of **when** files are consistent / durable from system to application
    i. E.g. delayed write after 30 seconds
    ii. E.g. delayed check for consistency after 30 seconds
b. Following Unix semantics
    i. NFS gives up, tries to emulate on client
    ii. AFS weakens slightly with open-close instead of read-write consistency
    iii.
c. Naming: single global name space vs. per machine spaces
    i. QUESTION: Can you emulate per-machine with single global? Yes, use symlinks

d. Mounting
    i. How does AFS / NFS handle it?
        1. AFS resolves name via DNS?

13.
14.