2. **Stateless Servers.** In NFSv2, servers are stateless; what does it mean to be stateless? What are the advantages of having a stateless server? What does a client need to do when a server crashes? What does a server need to do when a client crashes? Is it okay to keep anything in server memory? What must be included in requests to a stateless server?

Servers contain no client-specific state. Client crashes are completely invisible to servers. Server crashes appear as sluggish performance to clients. No client information can be kept in memory; writes must flush to disk. All information necessary to ~~[illegible]~~ complete an operation must be included in the request.

3. **Idempotent Operations.** Most NFSv2 operations are idempotent; what does it mean for an operation to be idempotent? Why are idempotent operations a good match for stateless servers? Give examples of idempotent operations in NFSv2. Give an example of a non-idempotent operation. How can non-idempotent operations be handled?

Idempotent - Doing an op. multiple times is equivalent to doing once and does not affect correctness.

why? - Stateless servers are desirable for easy crash rec.
- Retrying an opn. should not affect the correctness of the opn. execution.

Example for id. ops: 1. read, getattr, null, write, setattr, statfs, readdir

Example for non-id ops: Create, mkdir, remove, rmdir, link, symlink, rename.

MKDIR - 1st call succeeded but ack dropped
2nd call should be handled.

We can handle in server (or) client:

i) Server:
```
status mkdir ( dirName ) {
    if ( Exists (dirName) return fh (dirName);
                                 attr (dirName); }
    else { // do new creation ; }
}
```

2) Client: Handle exception client code:
(Adv - flexibility, client decides what to do)

4. **Performance Implications.** NFSv2 clients perform caching to improve performance. When are writes sent from the client to the server? Why does the stateless NFSv2 server cause performance problems for client write requests? How could this problem be fixed using something on the server? For read requests, how does the client know if the data in its cache is up to date? What problems did that cause? How was this overhead reduced?

- Writes sent from client to server are flushed when file is closed.
- Because a ~~flus~~ server-side flush is required for each client write request.
- It can be fixed by batching ~~client~~ server side flushes. NVRAM

Solution
- By updating cache every 3 seconds for files. and 30 secs for directories. Also, the vnode cache's attributes should match the attributes of ~~the~~ being returned by the server.

Problem
- Unnecessary caching.

~~Reason~~ Cache is Fresh?
→ By using getattr() from server.

5. **Implementation.** What does an NFSv2 file handle look like? Why? What were some of the auxiliary services that are needed to support NFSv2? Why are they needed? Does NFSv2 always provide the same semantics as a UNIX local file system? Are there any examples of differences? What were some operations that were difficult to handle?

- File SystemID Volume number
- File ID - Inode number
- Generation number

---

- Network lock Manager (NLM)
- Actual filesystem to store files on
- mount - to get first file handle mountd
- Y/P for credentials

VFS/Vnode
Filesys abstract
NTP - clock sync

---

open file removal (rename instead)
- opened file deleted by another client
- if client crashes between rename and remove, garbage file is left on server

file permissions cached
- if file permissions change while a client has it open, read requests could fail

Solution: ↳ they save the client credentials in the file table at open time

6. **Example Protocol.** Describe the operations that take place on the two separate client machines and the server for the following operations (specifically, when messages must be sent). Focus on what is currently contained in each client's cache and attribute cache. Can you summarize the consistency semantics provided by NFSv2?

| Time | Client A | Client B | Server Action? |
|---|---|---|---|
| 0 | fd = open("file A"); | | lookup(). |
| 10 | read(fd, block1); | | read. |
| 20 | read(fd, block2); chk cache attr expired. | | read. |
| 30 | read(fd, block1); use local. | | get attr |
| 31 | read(fd, block2); attr not expired, use local | | |
| 40 | | fd = open("file A"); | |
| 50 | | write(fd, block1); local. | |
| 60 | read(fd, block1); attr expired, use local. | | get attr. |
| 70 | | close(fd); write back to server | write to disk. |
| 80 | read(fd, block1); attr expired, attr changed, use new file | | read |
| 81 | read(fd, block2); → read. | | read |
| 90 | close(fd); | | |
| 100 | fd = open("fileA"); | | lookup. |
| 110 | read(fd, block1); attr. expired, get new attr local | | getattr. |
| 120 | close(fd); | | |

Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., and West, M.J.
Scale and Performance in a Distributed File System
ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 51-81.

1. **Initial Prototype.** What were the primary goals of the Andrew File System? Why did the authors decide to implement a usable prototype first? What were the primary problems they found with their prototype and what are the general implications?

Primary goals of AFS: Scalability, ~~Transparency~~ (5000 - 10000 nodes)
- By improving the performance
- By simplifying the system operation / administration

Wanted to quickly validate the basic file system architecture
- obtain feedback on the design as rapidly
- Need to build a usable system enough to make the feedback meaningful.

Primary problems: (Test Auth, Get Stat)
- 1) Most were cache-validation calls & pathname resolution
- OS 4.2 BSD didn't allow sharing of address space & manipulation of data shown happ via files.
- Servers had processes for every client.
- File location database is present in stub directories so that found difficult to move user's ch

2) CPU overhead in serv
a) Pathname traversal
b) Context Switches and Paging overhead.

Implications:
- Stats increased the total running time of programs & the server load.
- Many process Context Switching
No sharing virtual memory paging.
- RPC package on top of reliable byte stream from Kernel & h/w related resources of kernel once

3) Load Im balancing across servers

Implement volumes

Hot & Cold files

2. **Whole File Caching.** Why does AFS use whole file caching? Where → local disk
are files cached? What are the pros and cons of this approach? For what workloads is this a good idea? When is it a bad idea?

whole file caching:
- Future ref. to file need no n/w access.
- contact server open / closing
- most files are read in entirety
- cache management - easy

Pros: low n/w b/w, reduces reboot time since data cached on disk.

Cons: Files larger than disk caches cannot be accessed.
Diskless workstations aren't possible
File ~~sma~~ large & making small changes only.

Workloads:
good - many reads + writes on a file
- sequential reads
bad - huge files (may not fit)
- very small reads / writes

3. **Client Caching.** AFS clients perform caching to improve performance. For read requests, how does a client know that its cached copy is up to date? When are writes sent from the client to the server? What happens when the server receives a write? What happens when a client crashes and reboots? What are the pros and cons of the AFS approach versus the NFS approach?

Callbacks - informs client of invalidated cached file

writes to server on file close

server notifies other clients with callback

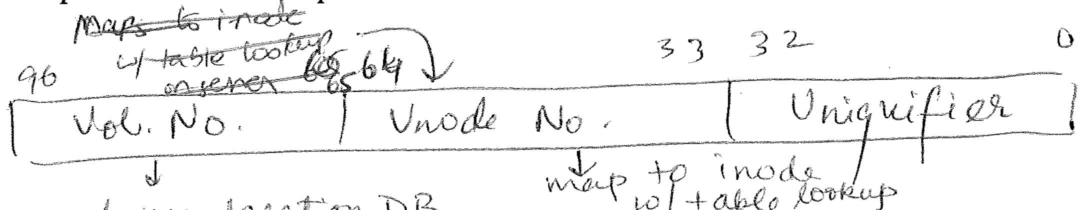
client crash - ~~[illegible]~~ invalidate callbacks
- check for valid files on open

low latency, if no concurrent writes/reads
fits in client cache, perf is local

con: means state

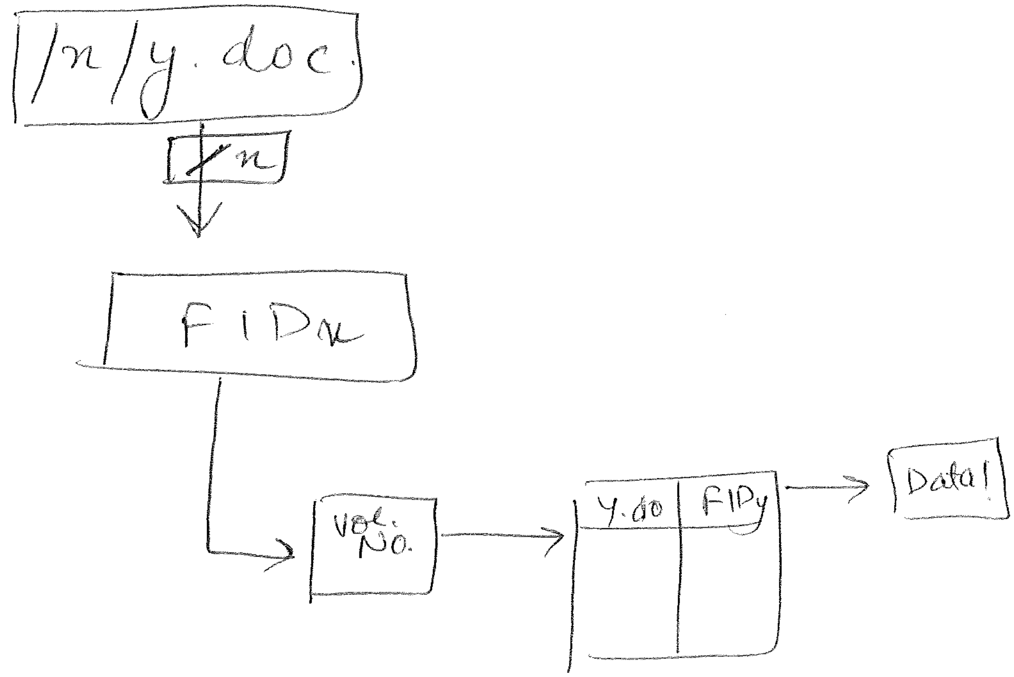
4. **Pathname Lookup.** AFS clients perform pathname lookups. What does an AFS fid look like? How does a client find the server that is responsible for a given volume? What steps take place when doing the pathname lookup for "/x/y.doc" (assume the client already has the root directory)? What portions of the needed information for a pathname lookup can be cached?

Fid.




- ① By the volume location database (Every server has it)
  ② By request redirection from original volume.

- Reading




Directories can be cached.

5. **Example Protocol.** Describe the operations that take place on the two separate client machines and the server for the following operations (specifically, when messages must be sent). Focus on the state of callbacks. Can you describe the consistency semantics of AFS? If two clients write to a file, which one will win (i.e., be store on the server)?

| Time | Client A | Client B | Server Action? |
|---|---|---|---|
| 0 | fd = open("file A"); | | Fetch (v1) |
| 10 | read(fd, block1); cached | | |
| 20 | read(fd, block2); cached | | |
| 30 | read(fd, block1); cached | | |
| 31 | read(fd, block2); cached | | |
| 40 | | fd = open("file A"); | Fetch (v1) |
| 50 | | write(fd, block1); | |
| 60 | read(fd, block1); cached | | |
| 70 | Callback | close(fd); | Store (v2) |
| 80 | read(fd, block1); cached | | |
| 81 | read(fd, block2); cached | | |
| 90 | close(fd); | | |
| 100 | fd = open("fileA"); | | Fetch (v2) |
| 110 | read(fd, block1); cached | | |
| 120 | close(fd); | | |