```
* NFS intro, cont.

NFSd on server calls lower f/s (e.g., ext4):
- NFSd treats the disk based f/s like a stackable f/s

NFS clients/servers often in kernel
- but there's a number of user-level implementations (e.g., NFS Ganesha)

NFSv2:
- No "seek" message? b/c it'd be state
- no open/close

NFS_READ can return updated attributes, to let client know that some stat(2)
data of file has changed
- o/w, clients have to set cache expirations for file/dir meta/data, and no
  one value works well: either you send too much network updates, or you read
  stale data.

* How do applications' syscalls translate into client/server protocol msgs

Application     NFS Client      NFS Server
---------------+---------------+-------------------
read(2) NFS_READ        nfsd_read -> ext4_read
---------------+---------------+-------------------
open            NFS_LOOKUP      nfsd_lookup -> ext4_lookup
                NFS_GETATTR     ->getattr
                ->permission

read            NFS_READ        nfsd_read -> ext4_read
                                (will update inode atime)

close           NFS_SETATTR     nothing...
                (if attrs changed)

* Implications

Client has to perform permission checking.  Anyone can hack a client OS to
bypass permission checking entirely.

Filehandle construction:
- fsid: small number, starting at 1
- inode number: typically start at 1, small number, can go up to millions on
  large f/s.
- gen. no.: typically 1
Meaning: easy to fake valid fhandles and read data you have no right to.

Proposed "solution" fill fh w/ random bits:
- harder for bad clients to guess
- but now server has to keep state, and map random numbers to
  <inum,fsid,igen>

When server sees first ->lookup:
- lookup and cache dentry+inode on server
when server sees NFS_READ
- need a struct file object, has to filp_open, so can issue fop->read
- problem: file remains closed, no client msg will cause it to close(2)
- Solution:
1. on NFS_READ: do filp_open, ->read, filp_close: too much burden on
   server for repeated reads
2. keep file structs open for some time T, then close them.  May have to
   reopen, if file is being re-read.


Application     NFS Client      NFS Server
```

```
---------------+---------------+-------------------
unlink          NFS_REMOVE      ->unlink
---------------+---------------+-------------------
>>PROBLEM SUPPORTING POSIX (read/write an open-unlinked file)
open            NFS_LOOKUP      ->lookup
                NFS_GETATTR     ->getattr
                ->permission

unlink          NFS_REMOVE      ->unlink

read            NFS_READ        -ESTALE (violated POSIX)

close           nothing
---------------+---------------+-------------------
>>Solution
open            NFS_LOOKUP      ->lookup
                NFS_GETATTR     ->getattr
                ->permission

unlink          NFS_RENAME("file", ".nfsXXXXXX")
                                ->rename("file", ".nfsXXXXXX")

stat("file")    NFS_LOOKUP("file")      -ENOENT

read            NFS_READ        ->read() successful

close           NFS_UNLINK(".nfsXXXXXX")
                                ->unlink(".nfsXXXXXX")

What if client program or OS dies before close(2)
- lots of .nfs* leftovers

* One more state, at RPC level
1. client issues RPC msg, waits for response N seconds
2. if no response, resend message and wait N seconds
3. after M tries, abort and return error

Scenario:
1. client sends RPC to server
2. server receives RPC message, unpacks it
3. server performs the op on disk f/s: op succeeds
        if op is NFS_GETATTR
        if op is NFS_UNLINK
4. server sends reply to client "NFS_OK"
5. reply never gets to client!
6. after N seconds, client will reissue RPC
7. server gets resent RPC, unpacks it
8. server performs the op on disk f/s: NOW WHAT HAPPENS?
        if op is NFS_GETATTR: server sends NFS_OK
        if op is NFS_UNLINK: server gets ENOENT
9. server replies back to client based on step 8.
        NFS_GETATTR: reply is ok to client
        NFS_UNLINK: reply is ENOENT even though file was successfully
                unlinked the first time.

The problem has to do with two classes of operations:
1. Idempotent: op can be repeated gets same result: NFS_GETATTR/READ
2. Non-Idempotent: op cannot be repeated: NFS_UNLINK/RENAME/MKDIR/etc.

Solution to non-idempotent ops:
- keep state on server, in memory
- every RPC, has an "ID" that a client assigns
- that way client can match replies with requests
- server caches results for every reply of non-idempotent op
- when server sees request to perform an op, for a cached RPC ID that was
```

```
   preformed before, it responds with the CACHED result (not performing the
   op on disk again).
 - have to keep cache for some time, to ensure that it won't be shorter than
   what the client RPC timeout is.
 - but if server crashes, all that RPC state is lost, and clients get errors
   upon server reboot.

 * "root squash"
 1. in NFSv2, clients send the effective UID+GID as part of the RPC header
 - this is called "UNIX permissions model"
 - problem: anyone can fake UIDs and GIDs (e.g., sudo, su)
 - can fake on client as any user
 - worse: can you fake being UID 0 (root)?
 2. Solution:
 - on server, you configure a file called /etc/exports
 format:
 /path   IPaddr(s)       flags
 /home   130.245.0.0/16  ro,root_squash

 flag: root_squash
 - if server gets an op w/ UID 0
 - change effective uid to -2 (user nobody)
 - so root on client has very limited file access
 - unless server turns off root_squash, by setting "no_root_squash" option

 Delete fh?
```