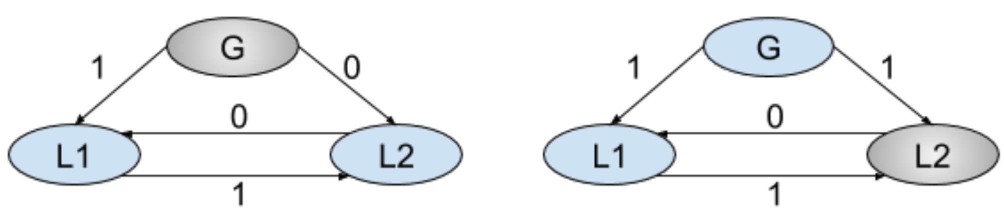
Distributed Systems

- 48 - Communication Basics
- 49 - NFS
- 50 - AFS
- GFS

Impossibility of consensus w/ 3

Can L1 tell who is faulty? Can L1 tell who is faulty?



- *all local lieutenants do the same thing*
- *any local lieutenant does what the general says*

Distributed consensus is the process by which a group of networked computers agree on a single data value or course of action, even in the presence of failures or unreliable communication.

Distributed Systems *reliable communication layers*

- Need to be able to detect and recover from packet loss:
  - *acknowledge* ("ack") receipt of a message

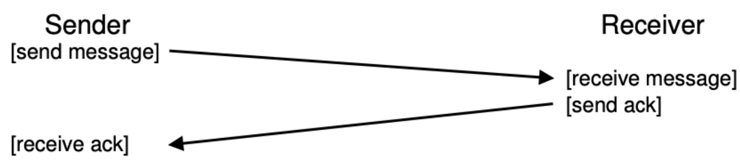


Figure 48.3: **Message Plus Acknowledgment**

Distributed Systems *reliable communication layers*

- Need to be able to detect and recover from packet loss:
  - *acknowledge* ("ack") receipt of a message
- What if we don't get the ack? How do we even know we don't get the ack?

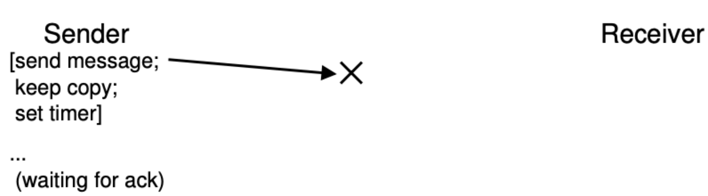


Figure 48.4: **Message Plus Acknowledgment: Dropped Request**

Distributed Systems *reliable communication layers*

- Need to be able to detect and recover from packet loss:
  - *acknowledge* (“ack”) receipt of a message
- What if we don’t get the ack? How do we even know we don’t get the ack?

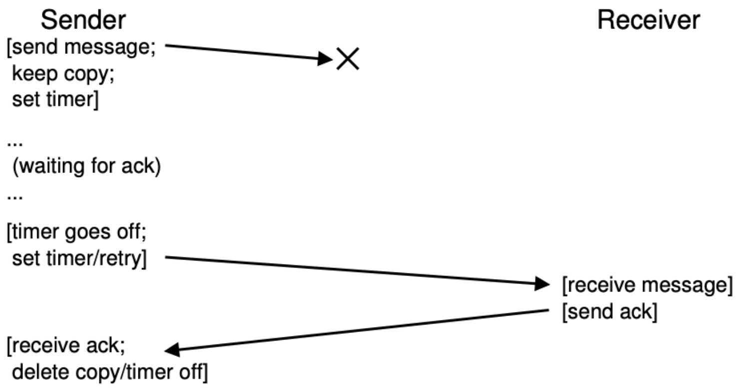
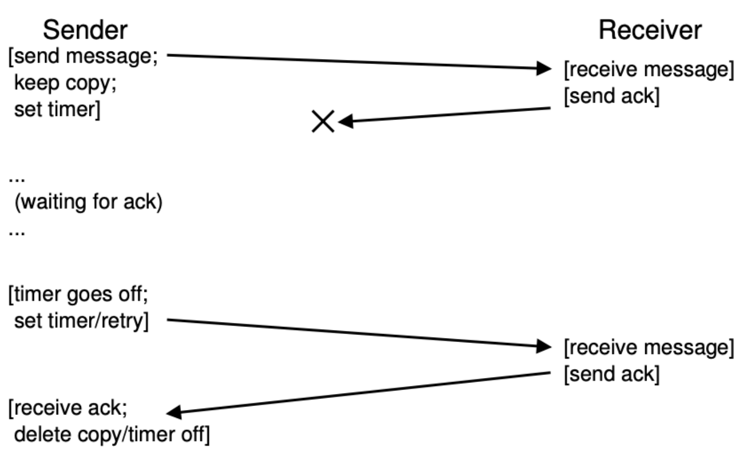


Figure 48.4: **Message Plus Acknowledgment: Dropped Request**

Distributed Systems *reliable communication layers*

- Need to be able to detect and recover from packet loss:
  - *acknowledge* (“ack”) receipt of a messaage
- What if we don’t get the ack? How do we even know we don’t get the ack?



Is this ok?

…no.

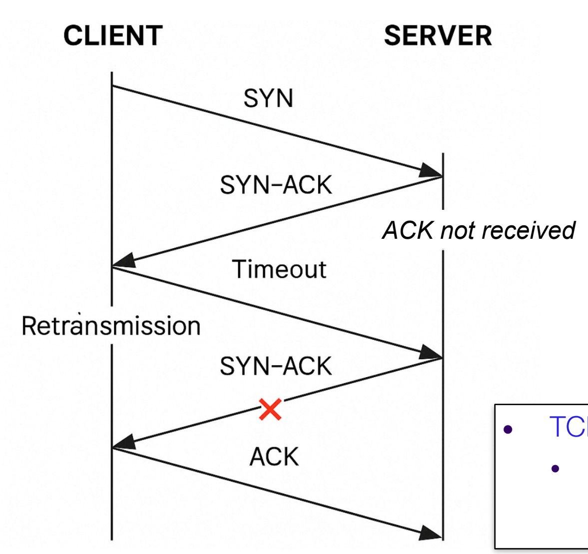
Figure 48.5: **Message Plus Acknowledgment: Dropped Reply**

Distributed Systems *reliable communication layers*

- 48.4 and 48.5 appear the same to the server…
  - but the msg was received in 48.4, and not in 48.5
  - this is bad, as server's default is to repeat the message, not good if messages are not idempotent
- fix is to include *sequence numbers* in messages
  - receiver could track every number ever seen, but expensive.
- *monotonically increasing* sequence numbers better
  - receiver tracks highest received sequence number
    - acks, but does not execute duplicate messages
    - dealing with out-of-order messages (42, 44, 43, 45…) app-dependent

Distributed Systems *TCP*

- client returns exact same ack after syn-ack transmission
- process continues for a bit and then server gives up



- TCP builds on seq numbers:
  - selective acknowledgments, sliding windows, PAWS (seq wraparound protection), congestion control…

Remote Procedure Calls

- turn remote requests into procedure calls to local functions
- need interface definition:

```
interface {
  int func1(int arg1);
  int func2(int arg1, int arg2);
};
```

- *client* stub generator uses interface def to:
  - create a msg buffer
  - pack (*marshal*) request into buffer
  - send to destination
  - synchronously wait for reply
  - unpack (*unmarshal*) return values
  - return return values to caller

- *server* stub generator uses interface def to:
  - unpack (unmarshal) the message
  - call local func w/ arguments
  - pack the return values into a reply buffer
  - send the reply

Remote Procedure Calls

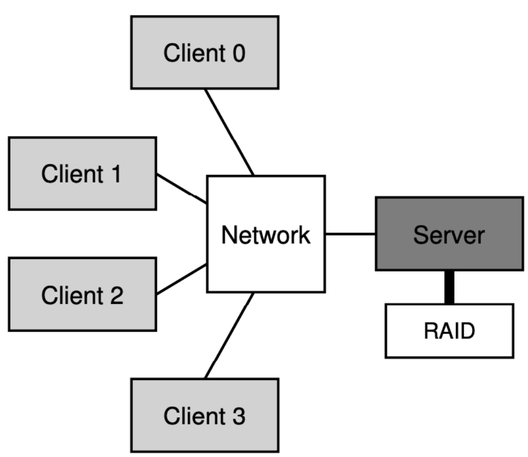
- What about pointers, or other complex data data types?
  - architecture- and language-independent encodings
    - JSON
    - protocol buffers
    - etc.
- What about concurrency in server?
  - want the server to be multi-threaded
  - need to ensure no data races between server stubs and the functions they call
- RPC generally doesn't need reliable communication (TCP)
  - "ack" is not needed, as RPC ("the app") generally returns a response
- gRPC/protobufs is your friend if you are working with micro-services

Distributed Systems

- *48 - Communication Basics*
- 49 - NFS
- 50 - AFS
- GFS

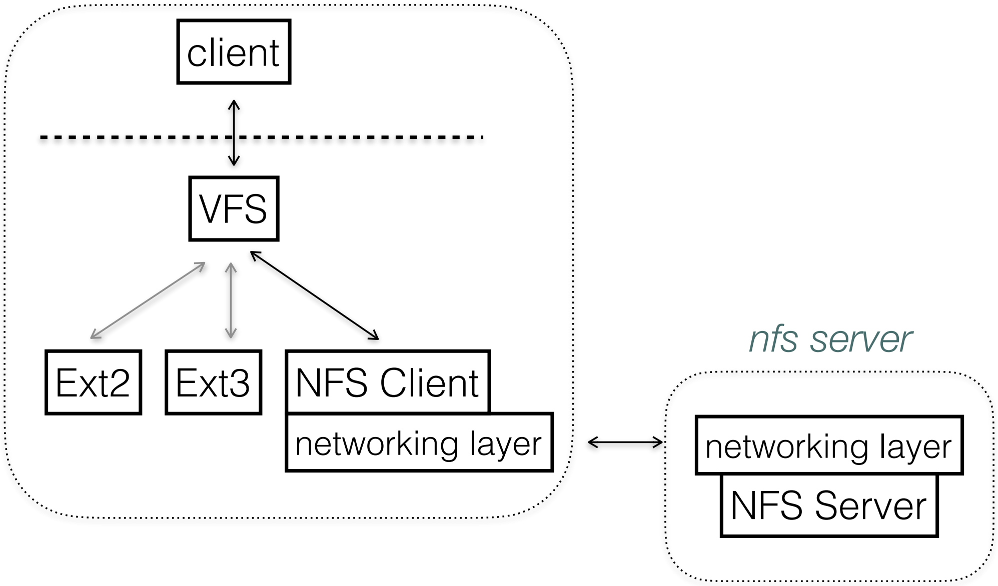
NFS *Sun Microsystems*

- first widely used distributed file system
  - clients diskless
    - easy sharing (consistency easy)
    - centralized admin
    - security



NFS

- distributed file system should be *transparent*
  - except possibly in performance
  - client issues same file-system calls as standalone system



NFS *actually NFSv2*

- NFS goals:
  - simple and fast file recovery
  - *stateless* protocol : *server keeps no client state*
    - server scales well
    - client crashes transparent
    - server crashes transparent
    - client must maintain all state the the server needs for any communication

“a distributed system is one where a machine I've never heard of goes down and I can't read my email”

- Leslie Lamport: Turing Award Winner for his work on distributed systems

NFS *actually NFSv2*

- file handle : uniquely describe file or directory
  - volume ID
  - inode number
  - generation number (inumbers get re-used)

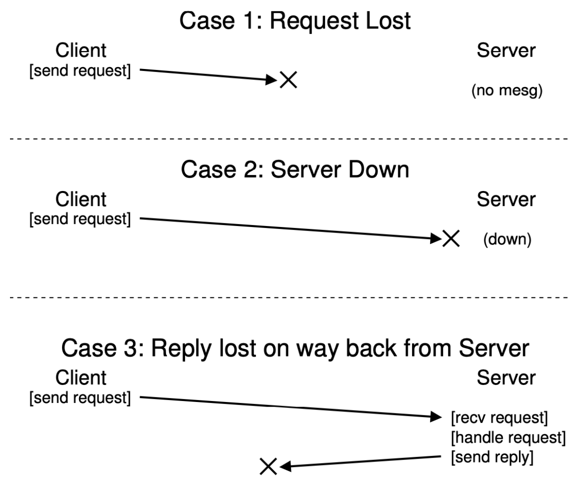
NFSPROC_GETATTR	file handle returns: attributes
NFSPROC_SETATTR	file handle, attributes returns: –
NFSPROC_LOOKUP	directory file handle, name of file/dir to look up returns: file handle
NFSPROC_READ	file handle, offset, count data, attributes
NFSPROC_WRITE	file handle, offset, count, data attributes
NFSPROC_CREATE	directory file handle, name of file, attributes –
NFSPROC_REMOVE	directory file handle, name of file to be removed –
NFSPROC_MKDIR	directory file handle, name of directory, attributes file handle
NFSPROC_RMDIR	directory file handle, name of directory to be removed –
NFSPROC_READDIR	directory handle, count of bytes to read, cookie returns: directory entries, cookie (to get more entries)

NFS *reading a file :*

Client	Server
fd = open("/foo", ...); Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	
read(fd, buffer, MAX); Index into open file table with fd get NFS file handle (FH) use current file position as offset Send READ (FH, offset=0, count=MAX)	
	Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client
Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app	
read(fd, buffer, MAX); Same except offset=MAX and set current file position = 2*MAX	
read(fd, buffer, MAX); Same except offset=2*MAX and set current file position = 3*MAX	
close(fd); Just need to clean up local structures Free descriptor "fd" in open file table (No need to talk to server)	

NFS *server failures*

- server crashes / restarts, knowing nothing about clients:
  - most client requests are *idempotent*
    - lookups, reads don't change server state
    - writes contain data and exact offset to write to
- client handles all timeouts in the same way



NFS *performance*

- client-side caching
  - read file data (and metadata) cached by client
  - all good unless the file changes on the server
- client-side write buffers
  - coalescing
  - aggregating disparate messages
  - writes sent back to server asynchronously (but before close())

- However : cache consistency!

NFS *cache consistency*

Problems:

- update visibility
  - C_1 writes `foo.c`, but does not immediately push to server
  - C_2 reads, sees old version
  - C_1 flushes to server
- stale cache
  - C_2 closes and reads again, sees old version (`foo.c` locally cached)

Fixes:

- *close-to-open* consistency
  - every open guaranteed to see every prior write to the server
    - must validate cache before use (GETATTR)
    - but maybe not all the time

NFS consistency is weak…   (like most other FS's)

NFS *server caching*

- tons of memory
  - wants to use it for disk cache   (satisfy reads)
  - wants to use it for write buffer   (quickly ack writes)
    - what could go wrong?
- server could ack a write before writing to disk!
  - say file initially has three 4k blocks of data:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

  - client overwrites with:

```
write(aaa…, 0)., write(bbb…, 4k), write(ccc…, 8k):
```

    - server crashes after acking second block, before writing:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- oops
ccccccccccccccccccccccccccccccccccccccccccccc
```

  - client never evens knows that the server crashed

NFS *cache consistency*

Problem: *poor performance for client$_i$ the same file again*

- fix: allow *client$_i$* to cache data and attributes on client
  - *but when client$_i$ re-opens not guaranteed most recent version*
- fix: have clients re-validate on open
  - *but slow*
- fix: time out the cached attributes
  - means data can all be cached, attributes *sometimes* validated w/ server before accesses
  - *but when client$_i$ re-opens not guaranteed most recent version (still)*

non-fix: *NFS consistency is weak…* *(like most other FS's)*

NFS *innovations*

- stateless protocol
  - minimizes state server needs to track
  - server can crash and recover w/o clients being aware
- itempotent requests
  - necessary for statelessness
  - client treats network message drops, server failure the same
  - client does not need to know which is which
- client and server buffering
  - essential for performance
  - cache consistency issues
    - server flushes writes before acking
    - client attribute cache times out
- VFS interface
  - makes application API independent of underlying FS