Distributed Systems 1:
Distributed File Systems

Sam Kumar
CS 162: Operating Systems and System Programming
Lecture 23
https://inst.eecs.berkeley.edu/~cs162/su20

Read: OSTEP Ch 48-50
Recall: Internet Architecture

- Lower three layers implemented everywhere
- Top two layers implemented only at hosts
Recall: The Internet *Hourglass*

The Hourglass Model

Applications
Transport
Data Link
Physical

Sockets – the OS "Hourglass"

TCP | UDP
---|---
SMTP | HTTP | DNS | NTP

Ethernet | SONET | 802.11
---|---|---
Copper | Fiber | Radio

The Hourglass Model
Recall: Internet Protocol Non-Features

- Unreliable Delivery ("Best Effort")
  - IP packet delivery not guaranteed
  - May be lost by underlying physical layer (e.g., radio noise)
  - May be dropped in transit

- Out-of-order/duplicate delivery
  - Tolerance of physical layer retrying transmission
  - Tolerance of multiple paths
Recall: End-to-End Principle

• Seen as a guiding principle of the Internet

• Some types of network functionality can only be correctly implemented *end-to-end*
  • Reliability, security, etc.

• Implementing complex functionality in the network:
  • Doesn’t necessarily reduce complexity on end hosts
  • Does increase network complexity
  • Imposes a cost on all applications, *even if they don’t need the functionality*
Recall: TCP (Transport Control Protocol)

• Reliable, in-order, and at-most-once delivery

• Stream oriented: messages can be of arbitrary length

• Provides multiplexing/demultiplexing to IP

• Provides congestion and flow control

• Application examples: file transfer, chat, HTTP
Recall: 3-Way Handshake

- Server calls `listen()` to wait for a new connection
- Client calls `connect()` providing server’s IP address and port number
- Each side sends SYN packet proposing an initial sequence number (one for each sender) and ACKs the other
Recall: 4-Way Teardown

- Connection is not closed until both sides agree

- If multiple FDs on Host 1 refer to this connection, all of them must be closed
- Same for close() call on Host 2

Host 1

- close()
- OS discards data (no socket to give it to)
- Can retransmit FIN ACK if it is lost
- OS deallocates connection state

Host 2

- FIN
- FIN ACK
- data
- FIN
- FIN ACK
- Any calls to read() return 0
- close()
- OS deallocates connection state
Sockets in Schematic

Client

- Create Client Socket
- Connect it to server (host:port)
- Write request
- Read response
- Close Client Socket

Server

- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
- Read request
- Write response
- Close Connection Socket
- Close Server Socket
Problem: Packet Reordering

• Want to divide a message into packets / frames
• Think about downloading a file over IP
  • 64K max packet size
• IP might reorder these packets
  • Imagine receiving the end of a file before the beginning!
Solution: Sequence Numbers and Buffering

• Simulate ordered messages on top of unordered messages

• Assign each packet a sequence number: 0, 1, 2, 3, ...
  • If packets arrive out of order, hold on to them
  • Deliver their data *in order* to user (through socket interface)

• Example: Hold on to #3 until #2 arrives, etc.
Problem: Dropped Packets

• All physical networks can garble or drop packets
  • Physical hardware problems (bad wire, bad signal)

• Therefore, IP can garble or drop packets
  • It doesn't repair this itself (end-to-end principle!)

• Building reliable message delivery
  • Confirm that packets aren't garbled
  • Confirm that packets arrive exactly once
Solution: Acknowledgments

- Checksum: Detect garbled packets
- Receiver sends a packet to acknowledge when a packet received and ungarbled
  - No acknowledgement? **Resend** after timeout
- What if acknowledgement dropped?
  - Packet is resent (wasteful), second chance to acknowledge
Stop-and-Wait (No Packet Loss)

- Send; wait for ACK; repeat
- Round Trip Time (RTT): time it takes a packet to travel from sender to receiver and back
  - One-way latency ($d$): one way delay from sender and receiver
- For symmetric latency, $RTT = 2d$
Stop-and-Wait (No Packet Loss)

• How fast can you send data?
• Little’s Law applied to the network:
  \[ n = B \cdot \text{RTT} \]
• For Stop-and-Wait, \( n = 1 \) packet

• So bandwidth is 1 packet per RTT
  • Depends only on latency, not network capacity (!)
Stop-and-Wait (No Packet Loss)

- So bandwidth is 1 packet per RTT
  - Depends only on latency, not network capacity (!)

- Suppose RTT = 100 ms and 1 packet is 1500 bytes

- Throughput = \( \frac{1500 \cdot 8}{0.1} \) = 120 Kbps

- Very inefficient if we have a 100 Mbps link!
Stop-and-Wait with Packet Loss

• Loss recovery relies on timeouts
• How to choose a good timeout?

Sender

Receiver

RTT

timeout

Time

ACK 1

1

X

8/4/2020

Kumar CS 162 at UC Berkeley, Summer 2020
Moving Away From Stop-and-Wait

• Idea: don’t wait for ACK before sending next packet

• How many packets are in-flight now?

• How long does the sender have to keep the packets around?

• How long does the receiver have to keep the packets’ data?

• What if sender is sending packets faster than the receiver can process the data?
Recall: Communication Between Processes

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

• Data written by A is held in memory until B reads it
• Queue has a fixed capacity
  • Writing to the queue blocks if the queue if full
  • Reading from the queue blocks if the queue is empty
• POSIX provides this abstraction in the form of pipes
Buffering in a TCP Connection

- A single TCP connection needs **four** in-memory queues:
  - Send buffer: add data on `write` syscall, remove data when ACK received
  - Receive buffer: add data when packets received, remove data on `read` syscall
Window Size: Space in Receive Queue

• A host’s *window size* for a TCP connection is how much remaining space it has in its receive queue.

• A host advertises its window size in *every* TCP packet it sends!

• *Sender never sends more than receiver’s advertised window size*
Sliding Window Protocol

• TCP sender knows receiver’s window size, and aims never to exceed it
• But packets that it previously send may arrive, filling the window size!

Rule: TCP sender ensures that:

Number of Sent but UnACKed Bytes $\leq$ Receiver’s Advertised Window Size

• Can send new packets as long as sent-but-unacked packets haven’t already filled the advertised window size
Sliding Window (No Packet Loss)

- Window size ($w$) = 3 packets

- Window size to fill link is given by: $w = B \cdot RTT$

- Little’s Law once again!

<table>
<thead>
<tr>
<th>Time</th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>{1, 2}</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>{1, 2, 3}</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>{2, 3, 4}</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>{3, 4, 5}</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>{4, 5, 6}</td>
<td>6</td>
</tr>
</tbody>
</table>

Unacked packets that sender sent

Out-of-seq packets in receiver’s window
ACKs and Loss Recovery

• In TCP receiver always ACKs the packet before the first loss

• Duplicate ACKs are a signal for packet loss

• Rely on timeouts as well (in case the ACKs, too, are lost...)

8/4/2020
Sliding Window with Packet Loss

• In the actual TCP protocol, you actually ACK the sequence number you expect next…

• But that’s just a minor detail…

Window size (W) = 3 packets

Sender

Timeout Packet 4

Receiver

Why doesn’t sender retransmit packet 4 here?

Assume packet 4 lost!
Congestion

• Too much data trying to flow through some part of the network

• IP’s solution: Drop packets

• What happens to TCP connection?
  • Lots of retransmission – wasted work and wasted bandwidth (when bandwidth is scarce)
Congestion Management

- TCP artificially restricts the window size if it sees packet loss
- Careful control loop to make sure:
  1. We don’t send too fast and overwhelm the network
  2. We utilize most of the bandwidth the network has available
- In general, these are conflicting goals!

Announcements

• Congrats on finishing Quiz 3!
• Homework 5 is out (last required homework)
• Project 3 design docs are due tonight

• Reminder: vote in the special topics poll!
  • Current frontrunners: Mobile OS, Embedded OS, Containers/Orchestration
  • Other highly-voted options: skip a lecture, Cryptographic Systems
  • Not far off: Blockchains, the Internet, Networking, Cluster Computing...
Access to Storage (Files) Today

• File system over SSD or HDD on your local machine
• File Server in your organization (inst.eecs.Berkeley.edu)
  • Remote login (ssh), file transfer (scp) or mount
• Cloud storage
  • Accessed through web or app (drive, box, …)
  • Mounted on your local machine
  • Replicated and/or Distributed
Cloud Storage Options

• Storage Account / Share is like disk “partition”
  • hold file system: directory, index, free map, data blocks
• Access methods: mount, REST, file xfer, synch
• Security: credentials, encryption (xfer, storage)
• Performance: HDDs, SSDs, provisioning, bursting
• Redundancy
  • Local RAID
  • Storage cluster in a Data Center
  • Zone redundant (across data centers)
  • Geographic regions
Geographic Replication: Cluster, Zone, Geo

- Highly durable: Hard to destroy all copies
- Highly available for reads: Just talk to any copy
- What about for writes? Need every copy online to update all together?
Centralized vs. Distributed

- **Centralized System**: Major functions performed on one physical computer
  - Many cloud services logically centralized, but not physically so
- **Distributed System**: Physically separate computers working together to perform a single task

Peer-to-Peer Model
Client/Server Model
Distributed Systems: Motivation

• The *promise* of distributed systems
  • *Higher availability*: one machine goes down, use another
  • *Better durability*: store data in multiple locations
  • *More security*: each piece easier to make secure

• Other advantages too:
  • Cheaper/easier to build lots of simple computers
  • Allows for adding more resources incrementally
  • Users can have complete control over some components
  • Easier for users to collaborate
More Cynical View of Distributed Systems

• *Worse availability*: depend on every machine being up
  - Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
• *Worse reliability*: can lose data if any machine crashes
• *Worse security*: anyone in world can break into system

• *Coordination is much more difficult*...
Distributed Systems: Goals/Requirements

• **Transparency**: the ability of the system to mask its complexity behind a simple interface

• Possible transparencies:
  • **Location**: Can’t tell where resources are located
  • **Migration**: Resources may move without the user knowing
  • **Replication**: Can’t tell how many copies of resource exist
  • **Concurrency**: Can’t tell how many users there are
  • **Parallelism**: System may speed up large jobs by splitting them into smaller pieces
  • **Fault Tolerance**: System may hide various things that go wrong
Examples of Transparency

• RPC: Simple function-like interface
  • Masks complexity of marshalling/unmarshalling, sending data, using sockets...

• Sockets: Simple file-like interface
  • Masks complexity of segmentation, retransmissions, windowing, etc.

• But it’s difficult to mask the performance implications of and new failure modes introduced by the distributed setting!
  • Server fails midway through RPC
  • Your socket connection can “break” if there’s too much packet loss (OS gives up retransmitting after a while)
Recall: What is a Protocol?

• A protocol is an agreement on how to communicate
• Includes
  • Syntax: how a communication is specified & structured
    • Format, order messages are sent and received
  • Semantics: what a communication means
    • Actions taken when transmitting, receiving, or when a timer expires
• Described formally by a state machine
  • Often represented as a message transaction diagram
Recall: Client-Server Communication

- Client is “sometimes on”
  - Sends the server requests for services when interested
  - E.g., Web browser on laptop/phone
  - Doesn’t communicate directly with other clients
  - Needs to know server’s address

- Server is “always on”
  - Services requests from many clients
  - E.g., Web server for www.cnn.com
  - Doesn’t initiate contact with clients
  - Needs a fixed, well-known address

GET /index.html

“Site under construction”
Distributed System Protocols are Built by Message Passing

• Sending/receiving messages is **atomic**
  • Each message is either fully received exactly once...
  • or not received at all (!)

• Interface:
  • Mailbox: temporary holding area for messages
    • Includes both destination location and queue
  • Send(message,mbox)
    • Send message to remote mailbox identified by mbox
  • Receive(buffer,mbox)
    • Wait until mbox has message, copy into buffer, and return
    • If threads sleeping on this mbox, wake up one of them
But, doesn’t TCP give us reliable delivery???

• TCP provides a convenient interface to use an unreliable network...
  • ... but it does not make the network reliable!

• Messages can still be lost if you use TCP
  • After many retransmissions, the OS “gives up” and breaks the connection

• Losing messages is fundamental problem in distributed systems
  • TCP’s retransmissions turn packet losses into packet delays (even if it never “gives up”)
  • And very long delays look just like losses!
  • TCP makes the network easy to use, and it can help improve performance
  • But TCP doesn’t solve this fundamental problem (losing messages)
Distributed File Systems

• Transparent access to files stored on a remote disk

• Mount remote files into your local file system
  • Directory in local file system refers to remote files
  • e.g., /home/oksi/162/ on laptop actually refers to /users/oski on campus file server
Enabling Design: VFS

The System Call Interface

- Process Management
- Memory Management
- Filesystems
- Device Control
- Networking

- Concurrency, multitasking
- Virtual memory
- Files and dirs: the VFS
- TTYs and device access
- Connectivity

- Architecture Dependent Code
- Memory Manager
- File System Types
- Device Control
- Block Devices

- Connectivity
- Network Subsystem
- IF drivers
Recall:
Layers...

User App
User library

Application / Service

High Level I/O
Low Level I/O
Syscall
File System
I/O Driver

length = read(input_fd, buffer, BUFFER_SIZE);

ssize_t read(int, void *, size_t){
    marshal args into registers
    issue syscall
    register result of syscall to rtn value
};

void syscall_handler (struct intr_frame *f) {
    unmarshal call#, args from regs
    dispatch: handlers[call#](args)
    marshal results for syscall ret
}

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    User Process/File System relationship
    call device driver to do the work
    Device Driver

Exception U\(\rightarrow\)K, interrupt processing
VFS: Virtual Filesystem Switch

• Key idea: same system call interface is used to interact with many different types of file systems
  • Dispatch each system call to the appropriate filesystem-specific code

• Similar to device drivers: possible to plug in different implementations of the same interface

```c
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
          O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```
Simple Distributed File System

- Remote Disk: Opens, Reads, Writes, Closes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - Server may cache files in memory to respond more quickly
  - Server provides consistent view of file system to multiple clients
- Problem: performance (network slower than memory, server is bottleneck)
Local Caching

- Idea: Use caching to reduce network load (e.g., buffer cache)
- New Problem: Consistency across caches
Dealing with Failures: Liveness

• What if the server loses connectivity with one client?
  • Maybe client crashed; maybe all messages got lost
  • Other clients should continue to get service
  • Server shouldn’t “block” waiting for that one client
Dealing with Failures: Cache Consistency

• What if server crashes? Can client wait until it comes back and just continue making requests?
  • Changes in server's cache but not in disk are lost

• What if there is shared state across RPC's?
  • Client opens file, then does a seek
  • Server crashes
  • What if client wants to do another read?

• Similar problem: What if client removes a file but server crashes before acknowledgement?
Recall: Multiprocessor Cache Coherence

• Interconnect is a broadcast medium (clients can observe all writes)
• Can’t use this strategy in a distributed file system!
Recall: HTTP and State

• HTTP avoids this issue – *stateless protocol*

• Each request is self-contained
  • Treated independently of all other requests
  • Even previous requests from same client!

• So how do we get a *session*?
  • Client stores a unique ID locally – a *cookie*
  • Client adds this to each request so server can customize its response
Stateless Protocol

• A protocol in which all information required to service a request is included with the request

• Even better: Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)

• Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)
NFS is Stateless

• NFS servers are **stateless**; each request provides all arguments required for execution
  • E.g. reads include information for entire operation, such as `ReadAt(inumber, position)`, not `Read(openfile)`
  • No need to perform network `open()` or `close()` on file – each operation stands on its own

• **Idempotent**: Performing requests multiple times has the same effect as performing it exactly once
  • Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  • Example: Read and write file blocks: just re-read or re-write file block – no side effects
  • Example: What about “remove”? NFS does operation twice and second time returns an advisory error

• Failure Model: Transparent to client system
  • Is this a good idea? What if you are in the middle of reading a file and server crashes?
  • Options (NFS provides both):
    • Hang until server comes back up (next week?)
    • Return an error. (Of course, most applications don’t know they are talking over network)
Network File System (Sun)

- Defines an RPC protocol for clients to interact with a file server
  - E.g., read/write files, traverse directories, ...
  - Stateless to simplify failure cases
- Keeps most operations idempotent
  - Even removing a file: Return advisory error second time
- Don't buffer writes on server side cache
  - Reply with acknowledgement only when modifications reflected on disk
NFS Architecture
NFS Cache Coherence

- Clients flush local changes to server on close()
- Clients periodically contact server to check if local file version is out of date
  - 3-30 sec. intervals (configuration parameter)
- What if multiple clients write to same file?
  - No guarantees: could see either version, or parts of both
Cache Coherence in NFS: Summary

- Server allows multiple cached copies
- Update visibility by “flush on close” (and periodic flushing every 3 – 30 s)
- Clients check periodically for external modifications via GetAttr()
Network File System: Pros/Cons

+ Simple, highly portable
  - Just need to speak RPC protocol to participate

- Sometimes inconsistent

- Doesn’t scale well to lots of clients
  - Clients keep checking to see if their caches stale
  - Server becomes bottleneck due to polling messages
Moving Beyond NFS

• Stronger guarantees as to when you discover other clients’ updates
  • Have the server inform the client of changes, instead of clients polling the server

• Some way of getting stronger guarantees as to when your data reaches the server (so others don’t see partial updates)
  • Don’t flush until close
  • Also reduces network traffic
Andrew File System (AFS)

- Clients cache **entire files** (on local disk) rather than individual data blocks upon an open
- All reads/writes occur against local copy
  - Reduces network traffic
- Changes flushed to server on close
  - Clients don't see partial updates – all or nothing!
- **Callbacks** – server tracks who has copies of each file, **informs them** if their copy is now stale
  - Client will fetch new version on next open
Andrew File System (AFS)

• Clients no longer need to poll server for cache invalidation, less network traffic

• Client disk as cache: More files can be cached
  • Read only workload: No need to involve server

• Consistency still has issues, but easier to describe
  • Two clients have file open at same time and both write: last to close wins (overwrites other client's update)
Failure in AFS

• Client fails?
  • Need to double check validity of all cached files
  • May have missed callback alerts from server while down

• Server fails?
  • Clients must be made aware of this
  • Clients must reestablish callbacks

• Callbacks mean server maintains more state than in NFS design
Andrew File System: Pros/Cons

+ Less server load than NFS
  • Disk as cache: clients can cache more files locally
  • Callbacks: server is not involved for read-only files

+ Easier to reason about consistency model
  • Clients don’t see partial updates

- Server maintains more state

- Inefficient to download whole file on open
  • Especially if only part of it is used
NFS/AFS Issues

• Performance: Central file server is a bottleneck

• Availability: Server is a single point of failure

• Higher cost for server hardware, maintenance compared to client machines
Conclusion

• **Distributed File System:**
  - Transparent access to files stored on a remote disk
  - Caching for performance

• **VFS: Virtual File System layer**
  - Provides mechanism which gives same system call interface for different types of file systems

• **Cache Consistency:** Keeping client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated?
  - NFS: check periodically for changes
  - AFS: clients register callbacks to be notified by server of changes