Logistics

• Proj 3 Milestone Tonight
• HW3 Due on August 13
  • Last Tuesday of the class
• HW/Project Party on Friday, 5PM, 405 Soda
• Course Evaluations at https://course-evaluations.berkeley.edu/
Recall: Go

• "Goroutines": Lightweight, user-level threads

• Channels: Named message queues for communication among threads
  • Given a type (send and recv instances)

• Key Idea: Prefer message passing over shared memory
Recall: Go Channels

buf

recv\text{x}

Next slot to read from

send\text{x}

Next slot to write to
Recall: Go's Scheduler

Global Run Queue

Newly created goroutines

Local Run Queue

OS Thread (M)

CPU Core

Local Run Queue

OS Thread (M)

CPU Core

Local Run Queue

OS Thread (M)

CPU Core

...
Recall: Centralized vs Distributed

- **Centralized System:** Major functions performed on one physical computer

- **Distributed System:** Physically separate computers working together to perform a single task
Recall: Challenge of Coordination

- Components communicate over the network
  - Send messages between machines

- Need to use messages to agree on system state
  - This issue does not exist in a centralized system
Recall: What is a Protocol?

• An agreement on how to communicate
  • Syntax: Format, order messages are sent and received
  • Semantics: Meaning of each message

• Described formally by a state machine

• A distributed system is embodied by a protocol
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/oski/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
- Architecture Dependent Code
- Memory Manager
- Files and dirs: the VFS
- File System Types
- Block Devices
- TTYs and device access
- Connectivity
- Network Subsystem
- IF drivers
VFS (Virtual Filesystem Switch)

• Similar to device drivers: possible to plug in different implementations of the same interface
  • Just need to provide inodes, files, directories, etc.
  • Doesn't matter if these are on local disk or remote!

• **Key Idea:** Same system call interface is used to interact with many different types of filesystems
Simple Distributed File System

- Remote Disk: Reads and writes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
- No local caching
- Server may cache files in memory to respond more quickly
Simple Distributed File System

- Advantage: Server acts as final authority on file contents
Simple Distributed File System

- Performance issues
  - Server is a bottleneck
  - Going across network is much slower than local memory
Local Caching to Reduce Network Traffic

- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don’t need to do any network traffic
- New problem: Consistency across caches

```
read(f1) → V1
read(f1) → V1
read(f1) → V1
read(f1) → V1
read(f1) → OK
write(f1) → V2
write(f1) → OK
```

![Diagram](image-url)
Dealing with Failures

• 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?
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)

• Recall HTTP: Also a stateless protocol
  • Include cookies with request to simulate a session
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 Cache consistency

- 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
Sequential Ordering Constraints

Client 1:  
- Read: gets A  
- Write B  
- Read: parts of B or C

Client 2:  
- Read: gets A or B  
- Write C  

Client 3:  
- Read: parts of B or C

Time

- What if we wanted to match single-machine case?  
  - If read finishes before write starts, get old copy  
  - If read starts after write finishes, get new copy  
  - Otherwise, get either new or old copy

- For NFS:  
  - If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update
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
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
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
Break
Key Value Storage

Simple interface

• `put(key, value);` // Insert/write "value" associated with key

• `get(key);` // Retrieve/read value associated with key
Why Key Value Storage?

• Easy to Scale
  • Handle huge volumes of data (e.g., petabytes)
  • Uniform items: distribute easily and roughly equally across many machines

• Simple consistency properties

• Used as a simpler but more scalable "database"
  • Or as a building block for a more capable DB
Key Values: Examples

• Amazon:
  • Key: customerID
  • Value: customer profile (e.g., buying history, credit card, ..)

• Facebook, Twitter:
  • Key: UserID
  • Value: user profile (e.g., posting history, photos, friends, …)

• iCloud/iTunes:
  • Key: Movie/song name
  • Value: Movie, Song
KV Storage Systems in the Wild

• Amazon
  • DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  • Simple Storage System (S3)

• BigTable/HBase/Hypertable: distributed, scalable data storage

• Cassandra: “distributed data management system” (developed by Facebook)

• Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)
Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: **partition** set of key-value pairs across many machines
Challenges

• **Fault Tolerance:** handle machine failures without losing data and without degradation in performance

• **Scalability:**
  • Need to scale to thousands of machines
  • Need to allow easy addition of new machines
Challenges

- **Consistency**: maintain data consistency in face of node failures and message losses

- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32 Kb/s to 1 Gb/s
Important Questions

• put(key, value):
  • where do you store a new (key, value) tuple?

• get(key):
  • where is the value associated with a given “key” stored?

• And, do the above while providing
  • Fault Tolerance
  • Scalability
  • Consistency
### Directory-Based Architecture

Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K5</td>
<td>V5</td>
</tr>
<tr>
<td>K14</td>
<td>V14</td>
</tr>
<tr>
<td>K105</td>
<td>V105</td>
</tr>
</tbody>
</table>
```

- `put(K14, V14)`
- `put(K105, V105)`
Iterative vs. Recursive Query

- Recursive Query: Directory Server Delegates
- Iterative Query: Client Delegates
Iterative vs Recursive Query

**Recursive**
+ Faster, as directory server is typically close to storage nodes
+ Easier for consistency: directory can enforce an order for all puts and gets
- Directory is a performance bottleneck

**Iterative**
+ More scalable, clients do more work
- Slower
- Harder to enforce consistency
Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures
Aside: Data Centers
Aside: Data Centers
Scalability: How easy is it to make the system bigger?

- **Storage:** Use more nodes
- **Number of Requests**
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular item on more nodes
- **Master/Directory Scalability**
  - Replicate it (multiple identical copies)
  - Partition it, so different keys are served by different directories
Scalability: Load Balancing

- Directory tracks available storage at each node
  - Prefer to insert at nodes with more storage available

- What happens when a new node is added?
  - Cannot insert only new values at new node
  - Move values from heavily loaded nodes to new node

- What happens when a node fails?
  - Replicate values from failed node to other nodes
Scaling Up Directory

• Directory contains number of entries equal to number of key/value pairs in entire system
  • Could be tens or hundreds of billions of pairs

• Solution: **Consistent Hashing**
  • Assign each node a unique ID in $[0..2^m-1]$  
  • Assume we can hash keys to same range of IDs  
  • Each (key,value) stored at node with smallest ID larger than hash(key)

• Important property: Adding a new bucket doesn't require moving lots of existing values to new buckets
Partitioning example with $m = 6 \rightarrow$ ID space: 0..63
Node 8 maps keys [5,8]
Node 15 maps keys [9,15]
Node 20 maps keys [16, 20]
... 
Node 4 maps keys [59, 4]
Performing a Lookup

• Fully decentralized
  • Any node can act as a directory for clients
  • Still works if a node leaves the network
• Each node knows about its successor and predecessor in the "circle"
  • All that is strictly needed for correctness
• Faster lookups: Each node maintains a routing table, allows client to get closer to destination in one hop
Example: Chord
Consistency

• Need to make sure a value is replicated correctly

• How do you know a value is replicated on every expected node?

• Wait for acknowledgements from all expected nodes
Consistency

• What happens if a node fails during replication?
  • Pick another node and try again

• What happens if a node is slow?
  • Slow down entire put? Pick another node?

• In general with multiple replicas: slow put and fast get operations
Consistency

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order

Master/Directory

- `put(K14, V14')` and `put(K14, V14'')` reach `N1` & `N3` in reverse order
- What does `get(K14)` return?
  - Undefined!
Quorum Consensus

• Improve put and get operation performance

• Define a replica set of size N
  • put waits for acknowledgements from at least $W$ replicas
  • get waits for responses from at least $R$ replicas
  • $W + R > N$

• Why does it work?
  • There is at least one node that contains the update

• Why might you use $W+R > N+1$?
Quorum Consensus Example

- \( N=3, W=2, R=2 \)
- Replica set for \( K14: \{N1, N2, N4\} \)
Quorum Consensus Example

• Now, issuing `get` to any two nodes out of three will return the answer

```
get(K14) V14
```
Summary

• Distributed File Systems: Transparent access to files located on remote disks
  • Caching for performance
  • But this now introduces consistency issues!
  • NFS: Check periodically for changes to server copy
  • AFS: Server notifies client of changes

• Key Value Store: Simple put and get operations
  • Fault tolerance: replication
  • Scalability: Add nodes, balance load, no central directory
  • Consistency: Quorum consensus for better performance