Other File Systems

CS439: Principles of Computer Systems

April 22, 2015

Last Time

- Parallel and Distributed Computing
  - Parallel: Tightly-coupled, same machine
  - Distributed: Loosely-coupled, different machines
- Parallel Programming Models
  - Shared Memory: threads, OpenMP, explicit synchronization
  - Message Passing: processes, MPI, explicit communication
- Distributed Computing
  - Event Ordering: Happened-Before Relationship, Logical Clocks
  - Atomicity of Transactions: Distributed Consensus, Two Phase Commit (2PC)

Today’s Agenda

Distributed File Systems

- Consistency Models
- NFS
- GFS

Other File Systems

Network and Distributed File Systems

- Provide transparent access to files stored on remote disks
- Issues:
  - *Naming*: How do we locate a file?
  - *Performance*: How well does a distributed file system perform as compared to a local file system?
  - *Failure handling*: How do applications deal with remote server failures?
  - *Consistency*: How do we allow multiple remote clients to access the same files?

Background: Consistency Models

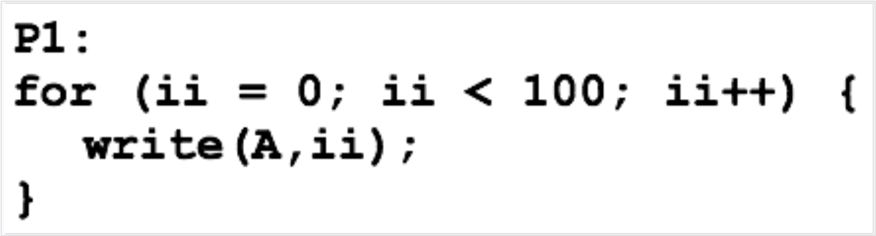
- Shared data is kept in a *data store*
  - a register, a file system, a database...
- Clients access the data store through *read* and *write* operations
- *Consistency Semantics*: a contract between the data store and its clients that specifies the results that clients can expect to obtain when accessing the data store

Coherence, Staleness, and Consistency

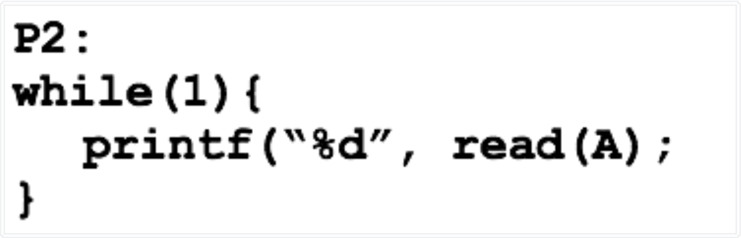
- *Coherence*: restricts order of reads and writes to one location
- *Staleness*: bounds maximum (real-time) delay between writes and reads to one location
- *Consistency*: restricts orders of reads and writes across locations

Coherence

- A read should return the result of the latest write to a memory location



```
P1:
for (ii = 0; ii < 100; ii++) {
   write(A,ii);
}
```



```
P2:
while(1){
   printf("%d", read(A);
}
```

- A memory that is not coherent may return

  1 2 3 3 3 4 8 10 9 11 12 13
- How could this happen?
  - writer sends updates via Internet; updates are reordered in route
  - reader switches between two servers (e.g. redirected to a different Akamai node)

Coherence: Plain Text

- A read should return the result of the latest write to a memory location
  - P1 code:

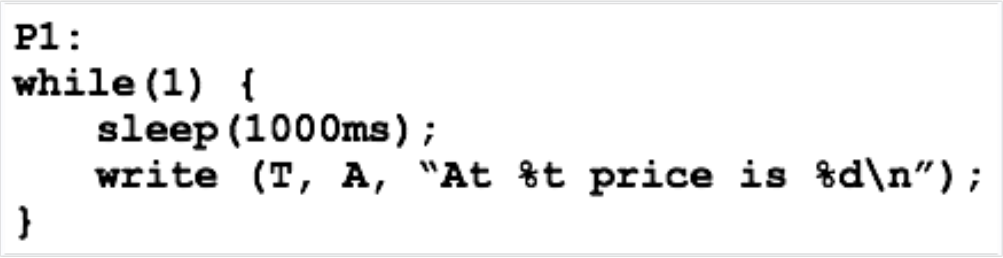
    ```
    for(ii = 0; ii < 100; ii++) {
            write(A, ii);}
    ```

  - P2 code:

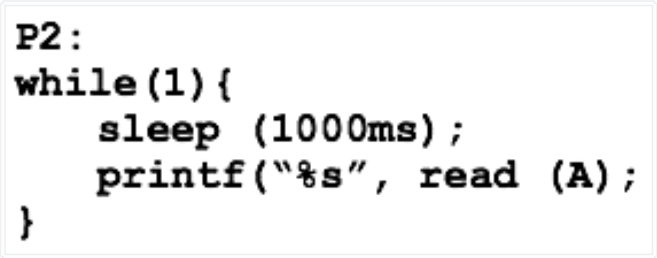
    ```
    while(1){
            printf(“%d”, read(A));}
    ```

- A memory that is not coherent may return: 1 2 3 3 3 4 8 10 9 11 12 13
- How could this happen?
  - Writer sends updates via Internet; updates are reordered in route
  - Reader switches between 2 servers (e.g., redirected to a different Akamai node)

Staleness



```
P1:
while(1) {
    sleep(1000ms);
    write (T, A, “At %t price is %d\n”);
}
```



```
P2:
while(1){
    sleep (1000ms);
    printf(“%s”, read (A);
}
```

- Assume perfectly synchronized clocks and a real time OS
- Reads may return

```
At 1:00:00 price is 10.50
At 1:00:01 price is 10:55
At 1:00:02 price is 10.65
At 1:00:02 price is 10:65
At 1:00:04 price is 13:18....
```

- Why staleness?
  - Polling interval
  - cache update/invalidation delayed by network

Staleness: Plain Text

- P1 code:

```
while(1) {
        sleep(1000 ms);
        write(T, A, “At %t prices is %d\n”); }
```

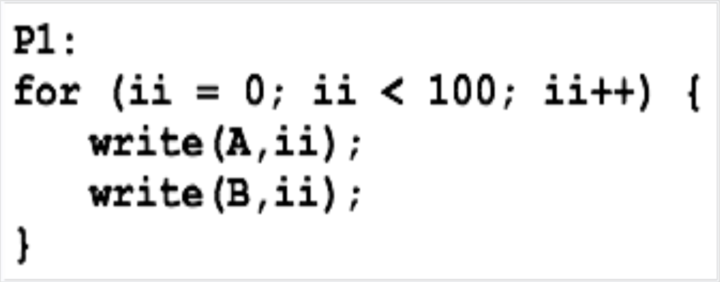
- P2 code:

```
while(1) {
        sleep(1000ms);
        printf(“%s”, read(A)); }
```

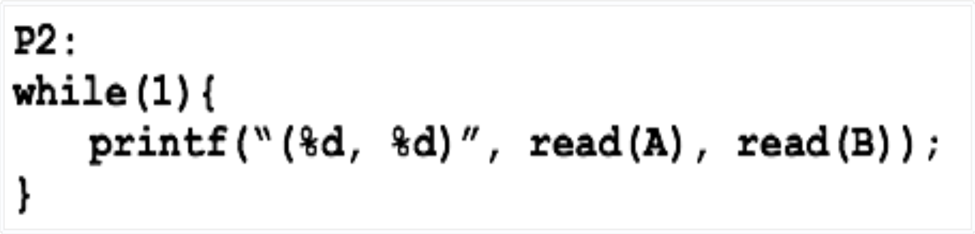
- Assume perfectly synchronized clocks and real time OS
- Reads may return
  - at 1:00:00 price is 10..50
  - at 1:00:01 price is 10.55
  - at 1:00:02 price is 10.65
  - at 1:00:02 price is 10.65
  - at 1:00:04 prices is 13.18 …
- Why staleness?
  - Polling interval
  - Cache update/invalidation delayed by network

Consistency

- Restricts order of reads and writes across locations



```
P1:
for (ii = 0; ii < 100; ii++) {
   write(A,ii);
   write(B,ii);
}
```



```
P2:
while(1){
   printf("(%d, %d)", read(A), read(B));
}
```

- A memory that is not consistent may return

(0,0),(0,1),(1,2),(4,3),(4,8),(8,9),(9,9),(9,10),(10,10),(11,10),(11,11),(12,12)

Consistency: Plain Text

- Restricts order of reads and writes across locations
- P1 code:

```
for(ii = 0; ii < 100; ii++) {
    write(A, ii);
    write(B, ii); }
```

- P2 code:

```
while(1)
    print("(%d, %d)", read(A), read(B));
```

- A memory that is not consistent may return:

  (0,0), (0,1), (1,2), (4,3), (4,8), (8,9), (9,9), (10, 10), (11, 11), (12, 12)

Sequential Consistency

- "The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program" (Lamport, 1979)
- In other words: create a total order that includes all the operations of the execution, such that:
  - the total order respects the local history of each process
  - every read returns the result of the latest write, according to the total order (data coherence)

Limitations of Strong Consistency

- Implementing strong semantics has intrinsic costs
  - Lipton and Sandberg: in sequential consistency can have either fast reads or fast writes, but not both
- CAP theorem: it is impossible in a distributed system to provide simultaneously
  - Consistency: all nodes see the same data at the same time, even in the presence of updates
  - Availability: every request receives a response
  - Partition Tolerance: the system properties hold even when the system is partitioned
- One can only have two properties out of three...

Weakening Sequential Consistency: Causal Consistency

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines. (Hutto and Ahamad, 1990)

More Weakening: FIFO Consistency

“Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes” (PRAM consistency, Lipton and Sandberg 1988)

Back to File Systems…

Finding Files: Naming

Two approaches to file naming:

- Explicit naming: <file server: file name >
    - E.g., windows file shares
    - //anslaptop/Users/ans/Desktop
- Implicit naming
    - Location transparency: file name does not include name of the server where the file is stored

Either way, server must be identified

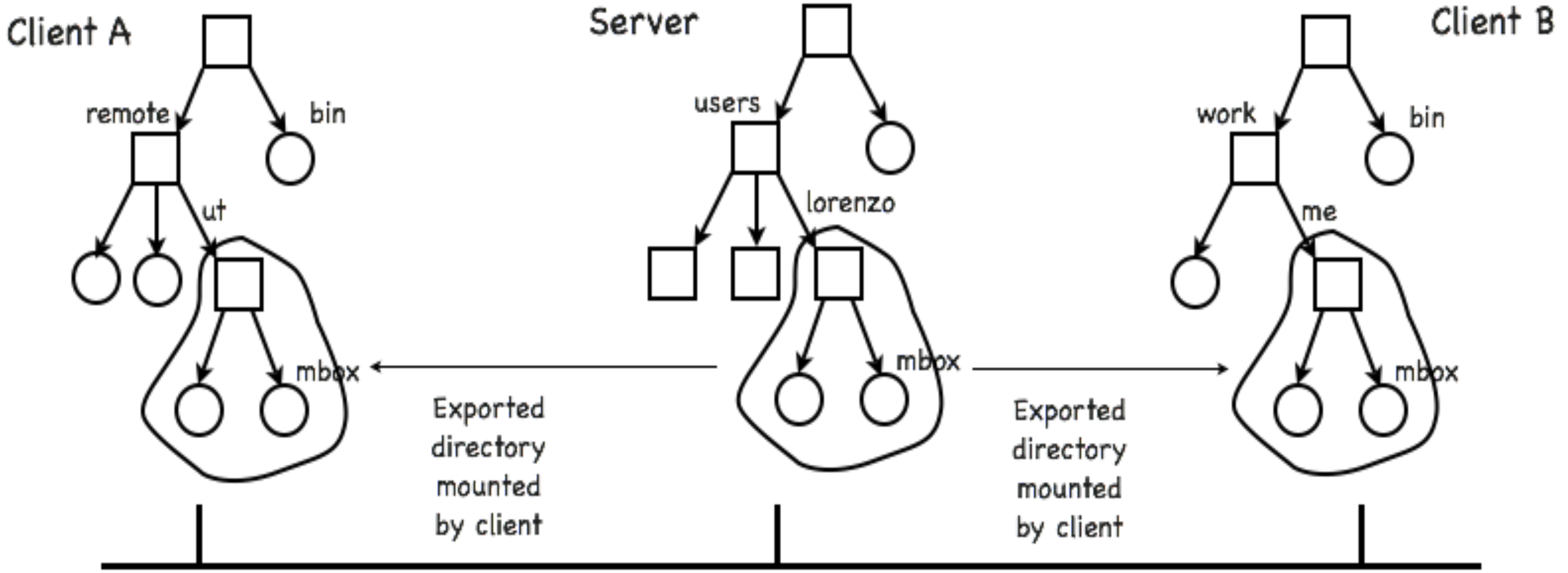
Finding Files: Identifying the Server

Most common solution (e.g., NFS):

- Static, location-transparent mapping
- Example: NFS Mount protocol
  - Mount/attach remote directories as local directories
  - Maintain a mount table with directory → server mapping, e.g.,

```
mount zathras:/vol/vol0/users/ans  /home/ans
```

Example: Mounting a Directory

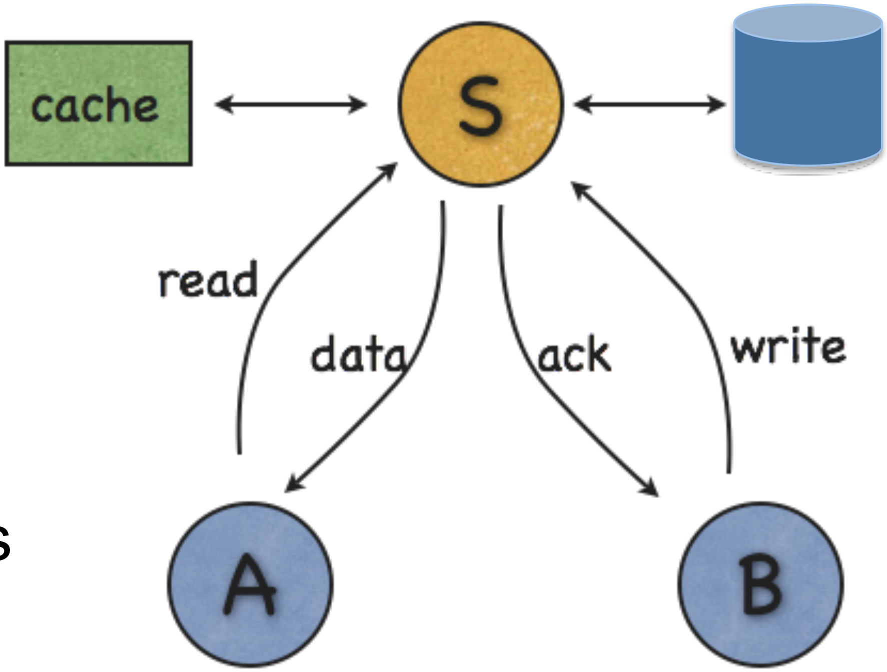

Client A
remote
bin
ut
mbox
Server
users
lorenzo
mbox
Client B
work
bin
me
mbox
Exported directory mounted by client
Exported directory mounted by client

Example: Mounting a Directory Text Description

- Have 3 different file systems: Server, Client A, and Client B
- We have a directory in the server’s file system that we want to mount on Client A and Client B
  - It is the directory that manages Lorenzo's UT account
  - One of the files contained in the directory is mbox - his email
- The server exports the directory and it is mounted into Client A and Client B under specified directory names
  - It is UT for Client A and me for Client B
- Now Lorenzo can access his account, and more importantly his email, from either Client A or Client B

Performance: Simple Case

- Simple case: straightforward use of RPC
  - Use RPC to forward every file system request (e.g., open, seek, read, write, close, etc.) to the remote server
  - Remote server executes each operation as a local request
  - Remote server responds back with the result
- Advantage: Server provides a consistent view of the file system to distributed clients.
  - What does *consistent* mean?
- Disadvantage: Poor performance



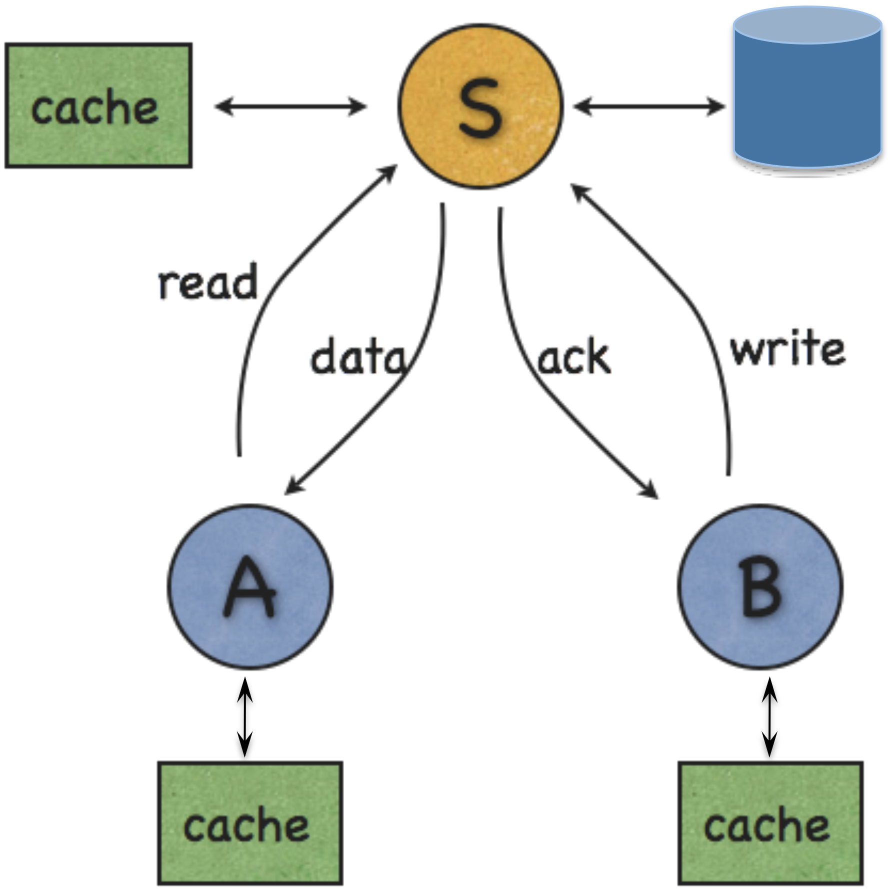
Solution: Caching at the Clien

Performance: Simple Case Plain Text

- Simple case: straightforward use of RPC
  - Use RPC to forward every file system request (e.g., open, seek, write, close, etc.) to the remote server
  - Remove server executes each operation as a local request
  - Remote server responds back with the result
- Example
  - Server can reference either cache or disk for each request
  - Process A requests to read data and gets data back from the server
  - Process B requests to write data and gets an ack back from the server
- Advantage: server provides a consistent view of the file system to distributed clients
  - What does consistent mean?
- Disadvantage: poor performance
- Solution: caching at the client

Sun's Network File System (NFS)

- Cache data blocks, file headers, etc. both at client and server
  - Generally, caches are maintained in memory; client-side disk can also be used for caching
  - Cache update policy: write-back or write-through
- *Advantage*: read, write, stat, etc. can be performed locally
  - Reduce network load and
  - Improve client performance



Sun’s Network File System (NFS): Plain Text

- Cache data blocks, file headers, etc. both at client and server
  - Generally, caches are maintained in memory; client-side disk can also be used for caching
  - Cache update policy: write-back or write-through
- Advantage: read, write, stat, etc. can be performed locally
  - Reduce network load and
  - Improve client performance
- In our example, processes A and B now each have their own cache

Sun’s NFS Issues: Consistency

- What if multiple clients share the same file?
  - Easy if both are reading files …
  - But what if one or more clients start modifying files?
- *Client-initiated weak consistency protocol*
  - Clients poll the server periodically to check if the file has changed
  - When a file changes at a client, server is notified
    - Generally, using a delayed write-back policy
  - Clients on detecting a new version of file at the server obtain a new version
- Consistency semantics determined by the cache update policy and the file-status polling frequency
- Other possibility: server-initiated consistency protocol

Sun’s NFS Issues: Failures

- What if the server crashes? Can client wait for the server to come back up and continue as before?
  - – Data in server memory can be lost
  - – Client state maintained at the server is lost (e.g., seek + read)
  - – Messages may be retried
- What if clients crash?
  - – Lose modified data in client cache

NFS Protocol: Statelessness

- NFS uses a *stateless* protocol
  - Server maintains *no state* about clients or open files
    - Except as hints to improve performance
  - Each file request must provide complete information
    - Example: ReadAt(inode, position) rather than Read(inode)
- *Idempotent operations*: All requests can be repeated without any adverse effects
  - "at least once" semantics
- Server failures are (almost) transparent to clients
  - When server fails, clients:
    - hang until the server recovers or
    - crash after a timeout
  - Why not return an error?

iClicker Question

Accessing files over NFS is much more complicated for user applications than if they were using the local file system.

A. True

B. False

NFS: Summary

- Key features
  - Location-transparent naming
  - Client-side and server-side caching for performance
  - Stateless architecture
  - Client-initiated weak consistency
- Advantages
  - Simple
  - Highly portable
- Disadvantages
  - Lack of strong consistency

Google File System

Google File System: Setting the Scene

Warehouse scale systems

- 10K-100K nodes
- Power efficient
  - 50MW (1 MW = 1,000 houses)
  - Located near cheap power
- CPU, storage network, power:
  - With 10K-100K-node data center 3-5x cheaper (per node) than for 100-1K-node data center

Google File System: Design Considerations

- Files are huge by traditional standards
  - Multi-GB files are common
  - Billions of objects
- 10K-100K nodes
- Component failures are the norm
  - 1000s of components
  - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
  - Monitoring, error detection, fault tolerance, automatic recovery

More Design Considerations

- Most modifications are appends
  - Random writes are practically nonexistent
  - Many files are written once, and read sequentially
- Two types of reads
  - Large streaming reads
  - Small random reads (in the forward direction)
- Sustained bandwidth more important than latency
- File system APIs are open to changes

Goal: Scaling!

Three main ways to achieve scalability:

- Distribute
  - partition data and computation across multiple machines
- Replicate
  - make copies of data available at different machines
- Cache
  - allow client processes to access local copies

GFS: Interface

- POSIX-like
  - – create, delete, open, close, read, write
- Additional operations
  - – Snapshot
  - – Record append
    - Supports concurrent append
    - Guarantees atomicity of each append

GFS: Architectural Design

- A file
  - Divided in fixed-sized *chunks*
    - Chunks are 3-way replicated and stored at chunkservers
    - Each chunk has a 64-bit unique global IDs
- A GFS cluster
  - A single *master*
  - Multiple *chunkservers* per master
    - Accessed by multiple *clients*
  - Running on commodity Linux machines

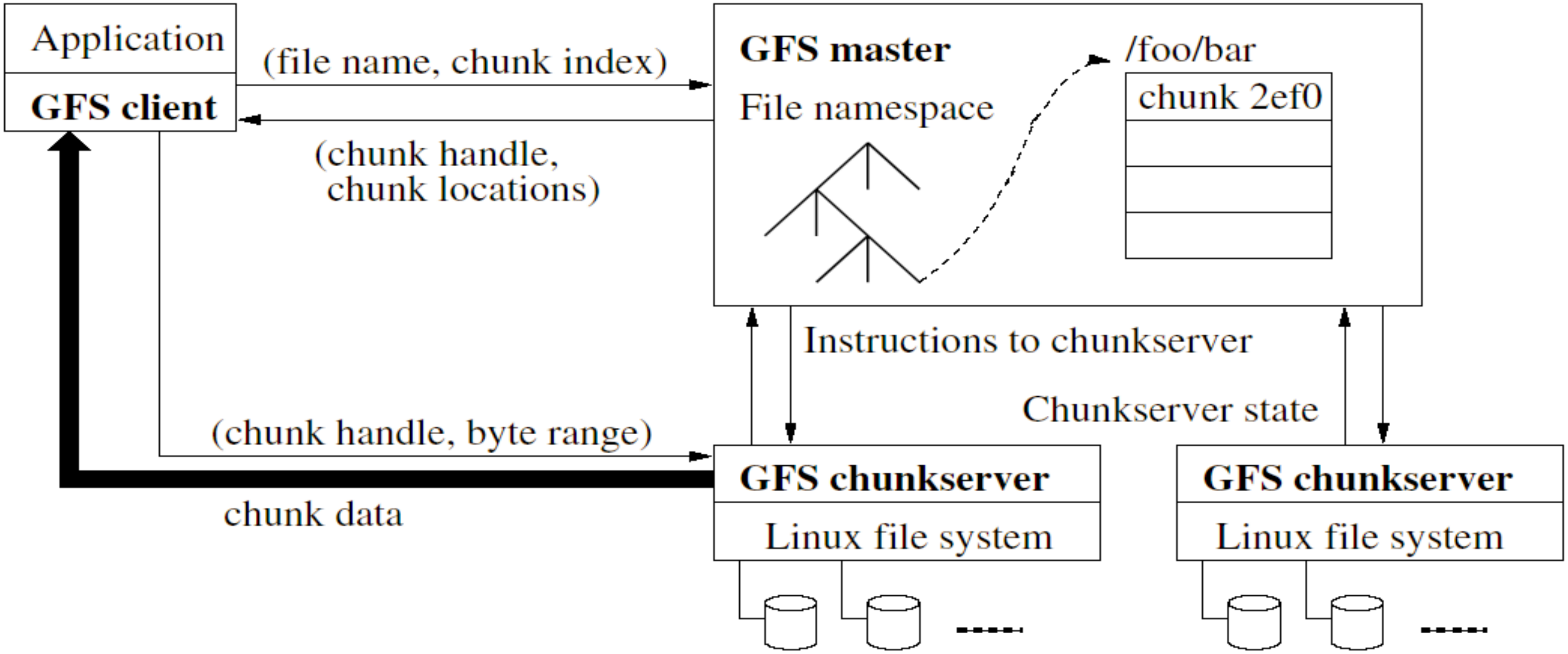
GFS: Single Master

- Disadvantages:
  - Single point of failure
  - Bottleneck (scalability)
  - Typical for single *anything* in distributed systems
- Solution?
  - Clients use Master only for metadata
    - Name spaces
    - Access control
  - Reading/writing goes directly through the chunkservers

GFS: File Access

- Client translates file name and byte offset to chunk index
- Sends request to master
- Master replies with chunk handle and location of replicas
- Client caches this information
- Sends request to a close chunkserver, specifying chunk handle and byte range
- Requests to master are typically buffered

GFS Architectural Design



GFS Architectural Design: Text Description

- Components
  - GFS client: the application
  - GFS chunkservers: hold a Linux file system, where the data actually lives
  - GFS master: holds the file namespace and the file system structure information
    - Knows which chunkserver stores which data
- Component Interactions
  - The client sends a file name and chunk index to the master
  - The master determines which chunkserver it needs to query and checks permissions, etc.
  - The master sends chunk handle and chunk locations to the client
  - The master sends instructions to chunkserver
  - The client queries the chunkserver with a chunk handle and a byte range
  - The chunk server sends chunk data back to the client
  - Periodically, the chunkserver sends its state to the master

GFS: Chunk Size

64 MB: much larger than a normal FS block

- Fewer chunk location requests to master
- Fewer metadata entries for a client (or for the Master) to cache in memory
- Potential issue with fragmentation
- Why did they choose a larger chunk size?

GFS: Metadata

- Three major types
  - File and chunk namespaces
  - File-to-chunk mappings
  - Locations of chunk replicas
- All metadata is in memory
  - System capacity limited by memory of master
    - Memory is cheap
  - On startup or recovery:
    - Name space and mapping determined through an operation log (kept on disk and replicated remotely)
    - Location info built by querying chunkservers

Metadata: Operation Log

- Serves as logical timeline for when files and chunks are created
- Replicated on remote machines
- Server reads log at startup to determine
  - File/chunk namespaces
  - File to chunk mapping
- Server *checkpoints* its state periodically to truncate logs (and thus reduce startup time)

Metadata: Chunk Locations

- Not saved at the Master
  - Chunkservers polled at startup
  - Information periodically refreshed using heartbeat messages to chunkservers
- Simplicity
  - Chunkservers may come and go---voluntarily or because of a failure
    - Master does not need to know or track this information
  - With many failures, hard to keep consistency anyway

Consistency Model

- Two types of updates
  - Writes (at application-specified offset)
  - Record appends (at offset of GFS's choosing)
- Relaxed consistency
  - Concurrent changes are consistent but undefined
    - *undefined* means that all clients may not see modification in its entirety (change may not appear to be atomic)
  - A record append is atomically committed at least once
    - Even during concurrent updates
    - Offset returned to the client marks beginning of defined region
    - Occasional padding/duplications
- Updates are applied in the same order at all replicas
  - Use chunk version number to detect stale replicas
    - They are garbage collected ASAP
- Client read may temporarily see stale values
  - Staleness limited by timeout on cache entry

Leases and mutation order

- Master grants chunk lease to *primary* replica
- Primary determines order of updates to all replicas
- Lease:
  - 60 second timeouts
  - Can be extended indefinitely
  - Extension request are piggybacked on heartbeat messages
  - Can be revoked

Fault Tolerance and Diagnosis

- Fast recovery
  - Master and chunkserver are designed to restore their states and start in seconds regardless of termination conditions
- Chunk replication
- Master replication
  - Shadow masters provide read-only access when the primary master is down

Fault Tolerance and Diagnosis (Continued)

- Data integrity
  - – A chunk is divided into 64-MB blocks
  - – Each with its checksum (logged persistently)
  - – Verified at read and write times
  - – Also background scans for rarely used data

iClicker Question

The Google File System would be a good choice for the file system on your home computer.

A. True
B. False

Summary

- Distributed file systems have more challenges than local file systems
- NFS (Sun’s version, 1984) uses RPC and a level of abstraction to provide seamless access for the OS kernel
- NFS uses mount to allow multiple file systems to appear to the user as a single file system
- GFS is optimized for very large files with particular access patterns and for a very large number of simultaneous clients
- Consistency models provide the user with guarantees about how and when updates will be seen in a distributed file system.

Announcements

- Homework 10 due Friday, 8:45a
- Project 4 (Last one!) due Friday, 5/8, 11:59p
  - No slip days!
- If you have a conflict for the final, you should have already contacted me (email, please!)
  - Thursday, May 14, 7p-10p in UTC 2.102A