Cloud Data Management: Distributed File Systems

Lecture 6

Last time..

- Relational databases
  - Relational model: Logical data independence
  - Relational algebra: Algebraic optimization, declarative querying
  - Optimized access paths: Indexing, materialized views, …
  - Transactional semantics: ACID guarantees

- What is the link with MapReduce? Spark?

Relational operations over MR: Selection

- **In practice, selections do not need a full-blown MapReduce implementation**
  - They can be implemented in the map phase alone
  - Actually, they could also be implemented in the reduce portion
- **A MapReduce implementation of** $\sigma_C(R)$
- **Map:**
  - For each tuple t in R, check if t satisfies C
  - If so, emit a key/value pair *(t, t)*
- **Reduce:**
  - Identity reducer

Relational operations over MR: Projections

- **A MapReduce implementation of $\pi_S(R)$**
- **Map:**
  - For each tuple t in R, construct a tuple t' by eliminating those components whose attributes are not in S
  - Emit a key/value pair *(t', t')*
- **Reduce:**
  - For each key t' produced by any of the Map tasks, fetch *t', [t', ,t']*
  - Emit a key/value pair *(t', t')*
- **NOTE: the reduce operation is duplicate elimination**
  - This operation is associative and commutative, so it is possible to optimize MapReduce by using a Combiner in each mapper

Relational operations over MR: Unions

- **Suppose relations** R **and** S **have the same schema**
  - Map tasks will be assigned chunks from either R or S
  - Mappers don't do much, just pass by to reducers
  - Reducers do duplicate elimination
- **A MapReduce implementation of union**
- **Map**
  - For each tuple t in R or S, emit a key/value pair *(t, t)*
- **Reduce:**
  - For each key t there will be either one or two values
  - Emit *(t, t)* in either case

Relational operations over MR: Intersection

- **Very similar to computing unions**
  - Suppose relations R and S have the same schema
  - The map function is the same (an identity mapper) as for union
  - The reduce function must produce a tuple only if both relations have that tuple
- **A MapReduce implementation of intersection**
- **Map:**
  - For each tuple t in R or S, emit a key/value pair *(t, t)*
- **Reduce:**
  - If key t has value list *[t, t]* then emit the key/value pair *(t, t)*
  - Otherwise, emit the key/value pair *(t, NULL)*

Relational operations over MR: Difference

- **Assume we have two relations** R **and** S **with the same schema**
  - The only way a tuple t can appear in the output is if it is in R but not in S
  - The map function passes tuples from R and S to the reducer
  - NOTE: it must inform the reducer whether the tuple came from R or S
- **A MapReduce implementation of difference**
- **Map:**
  - For a tuple t in R emit a key/value pair *(t, ‘R’)* and for a tuple t in S, emit a key/value pair *(t, ‘S’)*
- **Reduce:**
- For each key t, do the following:
  - If it is associated to [‘R’], then emit *(t, t)*
  - If it is associated to *[‘R’; ‘S’]* or *[‘S’,‘R’]*, or *[‘S’]*, emit the key/value pair *(t, NULL)*

Relational operations over MR: Natural Join

- **Let's look at two relations** R(A, B) **and** S(B, C)
  - We must find tuples that agree on their B components
  - We shall use the B-value of tuples from either relation as the key
  - The value will be the other component and the name of the relation
  - That way the reducer knows each tuple's relation
- **Map:**
  - For each tuple *(a, b)* of R emit the key/value pair *(b, ('R', a))*
  - For each tuple *(b, c)* of S emit the key/value pair *(b, ('S', c))*
- **Reduce:**
  - Each key b will be associated to a list of pairs that are either *('R', a)* or *('S', c)*
  - Emit key/value pairs of the form *(b, [(a_1, b, c_1), (a_2, b, c_2),   , (a_n, b, c_n)])*

MR DBMS Comparison

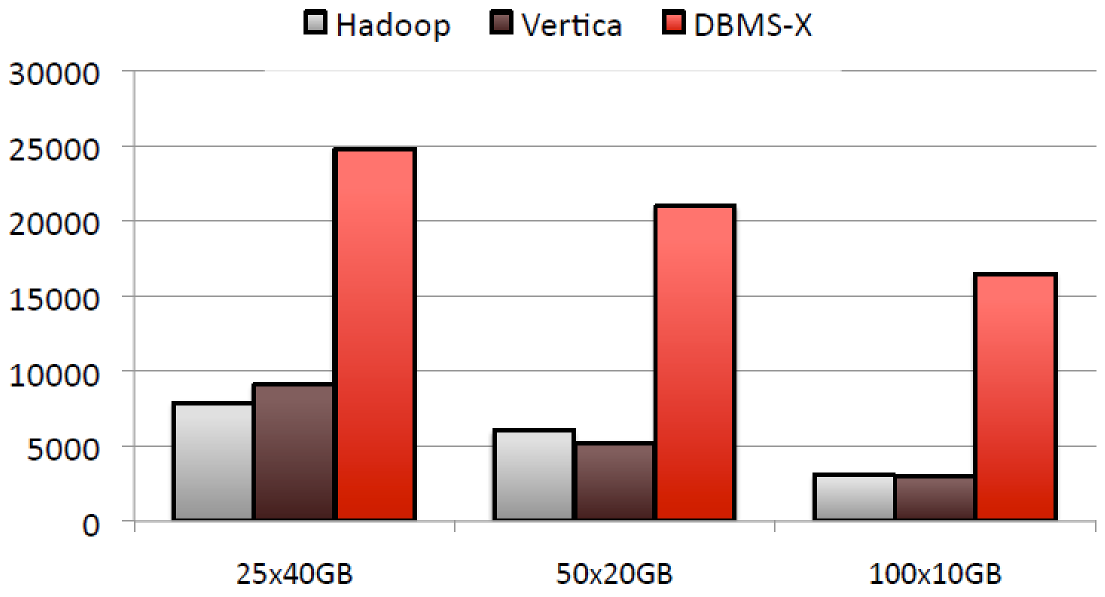
“MapReduce and Parallel DBMS: A comparison of approaches to large-scale data analysis” – Andy Pavlo ’09

- Tested Systems
  - Hadoop (MR)
  - Vertica (Columnar DBMS)
  - DBMS-X (Rowstore)

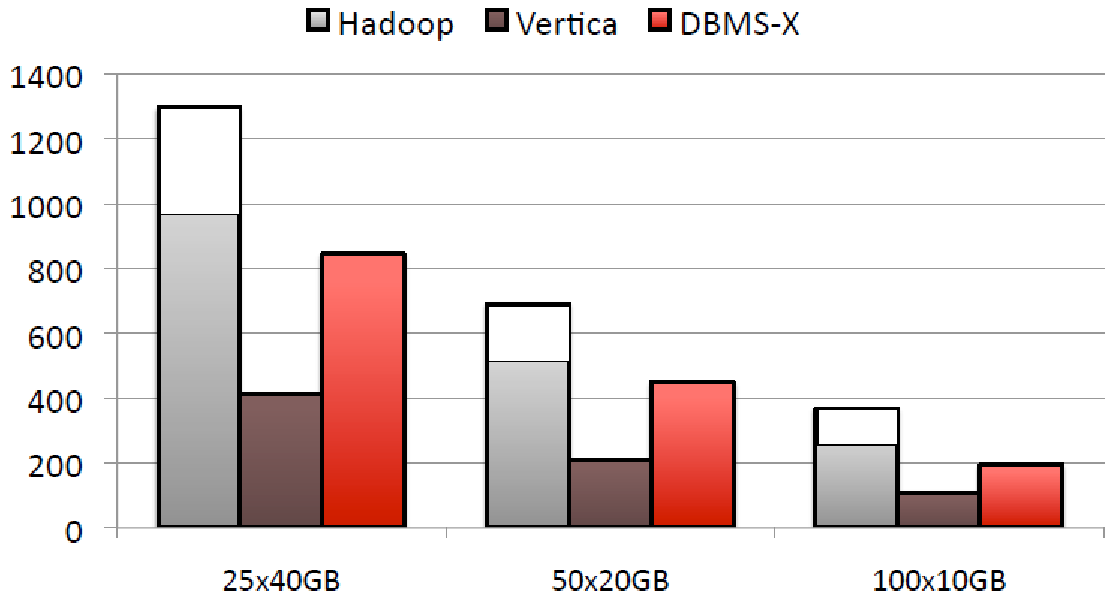
Benchmark 1: Grep Task

- Grep task: Shows overhead of data loading in DBMS
  - Search 3 byte pattern across 10Billion records
  - Each record: 100 bytes (10-byte key, 90-byte value)
  - 1TB across 25,50 or 100 nodes

DBMS slow during data Loading

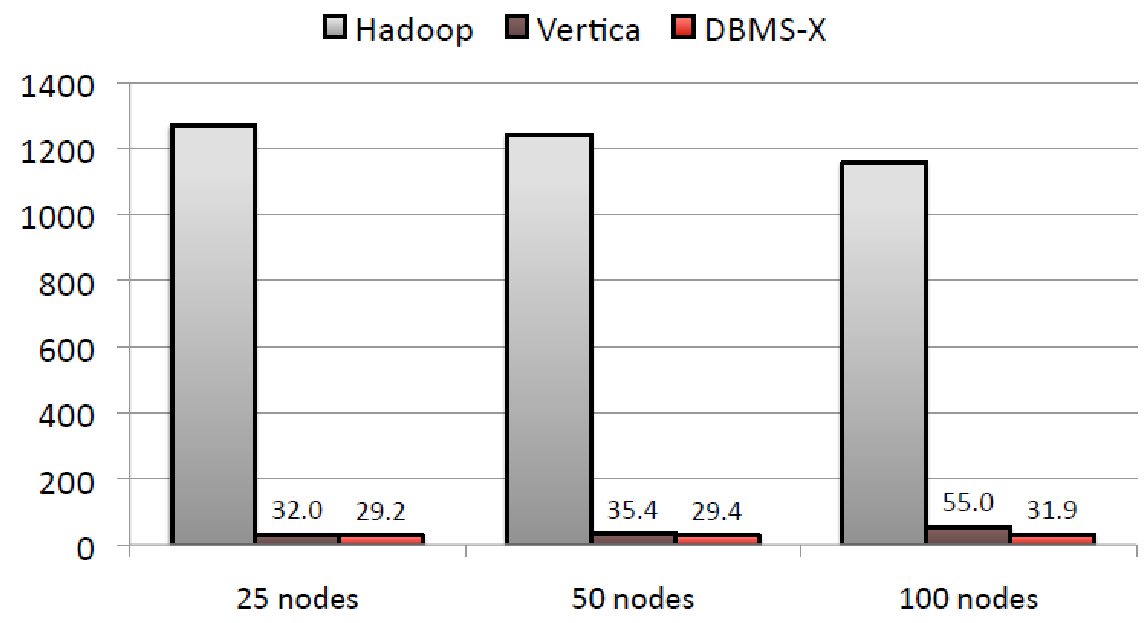


DBMS fast during execution



Analytical Task

- Web processing: Shows the benefit of query optimization
  - 600k html documents
  - 155Million uservisits records, 18 million rankings records
  - Task: Find sourceIP that generated most revenue with avg. pagerank
  - DBMS: Complex SQL join query, MR: 3 separate MR programs



RDBMS vs MapReduce: Summary

- Systems designed to meet different requirements
- Traditional relational databases
    - Fine-grained updates to shared data
        - Guaranteeing ACID properties despite concurrent access and failures
    - Interactive SQL analytics
        - Optimized for point queries (random access) and range queries (scans)
    - Built for enterprises (dedicated DB admin, few DB servers)
        - No need to scale to 1,000 or more nodes
        - Proprietary and paid products
- MapReduce
    - Latency-insensitive batch analytics
        - Sequential scans of Petabytes of data
    - Built for the cloud: Fault tolerance across commodity servers
        - Focus on faults during query rather than recovery after updates
    - Open source and “One person” deployment
        - Turn any Java developer into a distributed analytics engineer

Spark and Relational DBMS

- Spark can support interactive workloads
  - Exploiting memory hierarchy, pipelining transformations, …
- But SQL as a high-level programming language
  - Offers expressiveness, succinctness
  - Enables compatibility with existing tools, e.g. BI using JDBC
  - Large pool of engineers proficient in SQL
- Also need to support variety data formats
  - JSON, CSV, ORC, Parquet, JDBC, …
- SparkSQL was born: SQL as a library over Spark
  - Use SparkContext to interact with Spark
  - DataSource API to deal with input data formats
  - DataFrame API to support SQL extensibly

DataSource API

- Unified interface for reading and writing data

```
df = sqlContext.read \
.format(''json'') \
.option(''samplingRatio'', ''0.1'') \
.load(''data.json'')

df.write \
.format(''parquet'') \
.mode(''append'') \
.partitionBy(''year'') \
.saveAsTable(''myData'')
```

DataFrame

- **General idea borrowed from Python Pandas**
  - Tabular data with an API
- **Schema to the rescue**
  - A distributed collection of rows organized into named columns
  - Schema inference can be automatic
- **Relation to a low-level RDD**
  - Introduces structure to the data
  - Specific relational operators
  - An abstraction for selecting, filtering, aggregating and plotting structured data

DataFrame API

- Example using RDDs

```
data = sc.textFile(...).split(" ")
data.map(lambda x: (x[0], [int(x[1]), 1]))
      .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]])
      .map(lambda x: [x[0], x[1][0] / x[1][1]])
      .collect()
```

- Example using SQL

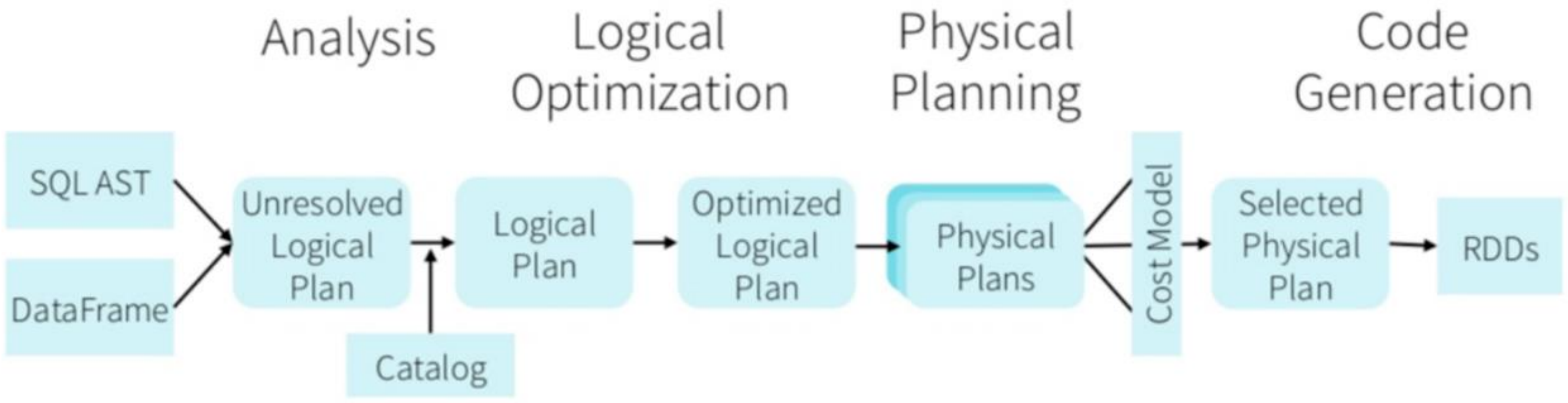
```
SELECT name, avg(age)
FROM people
GROUP BY name
```

- Example using DataFrames

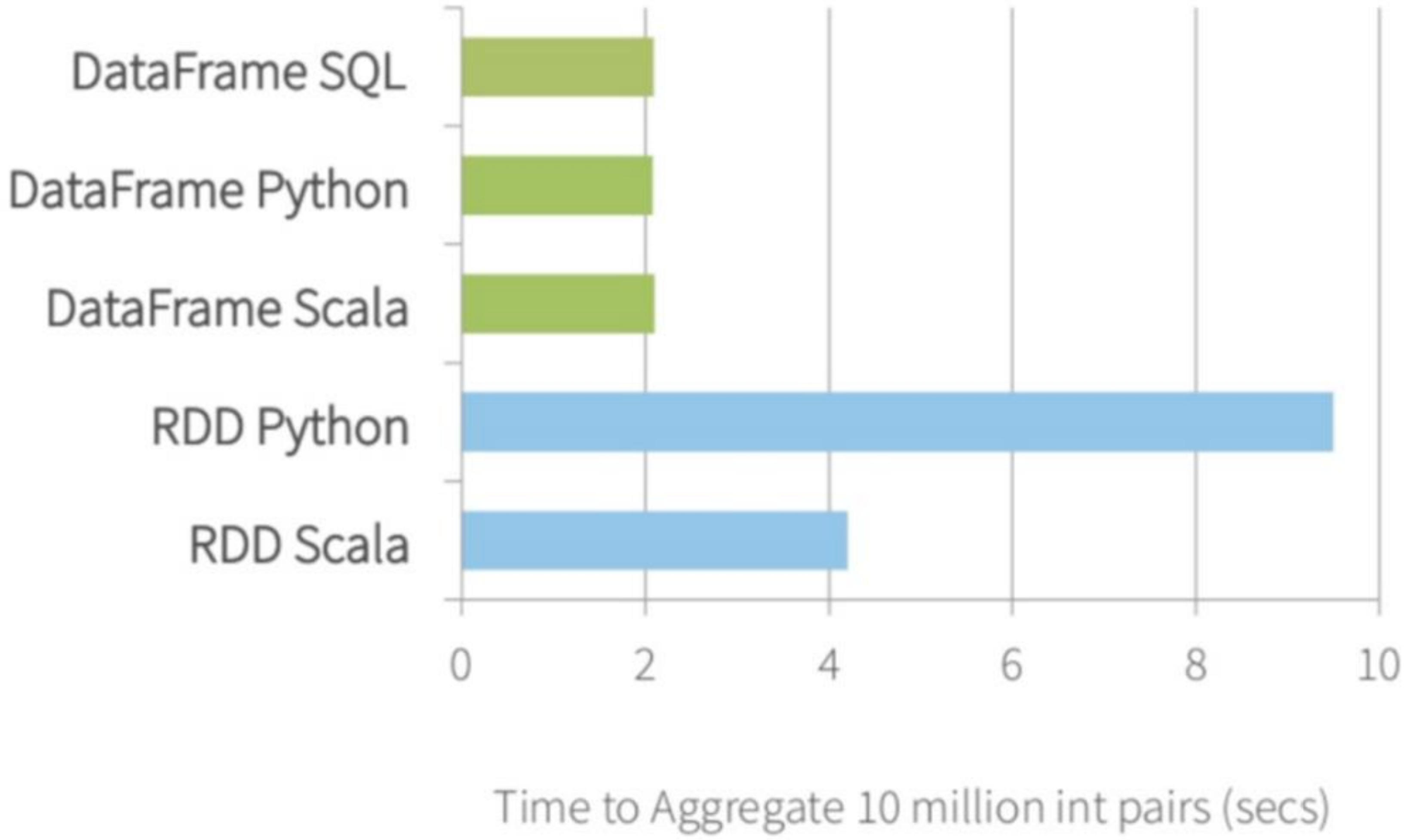
```
sqlContext.table(''people'') \
.groupBy(''name'') \
.agg(''name'', avg(''age'')) \
.collect()
```

Catalyst (Optimizer) & Tungsten (Codegen)

- **Reminiscent of traditional database systems**
  - Abstract representation of SQL expressions
  - Optimizations for efficiency and performance
  - Sophisticated cost model



Performance



Spark & RDBMS: Summary

- Spark: unified analytics engine
  - Quickly adopted RDBMS concepts to optimize SQL analytics
  - Other libraries developed for machine learning (Mlib), graph analytics(GraphX),…
  - RDD: an underlying abstraction that supports several libraries

- DBMSs have also evolved
  - Disk-based to in-memory to NVM
  - One-size-fits-all “OldSQL” DBMS to customized “NewSQL” engines
    - column stores for Business Intelligence
    - highly parallel transaction engines for OLTP
    - Array databases for scientific applications
    - …
  - NewSQL still king for structured data management

What about new applications?

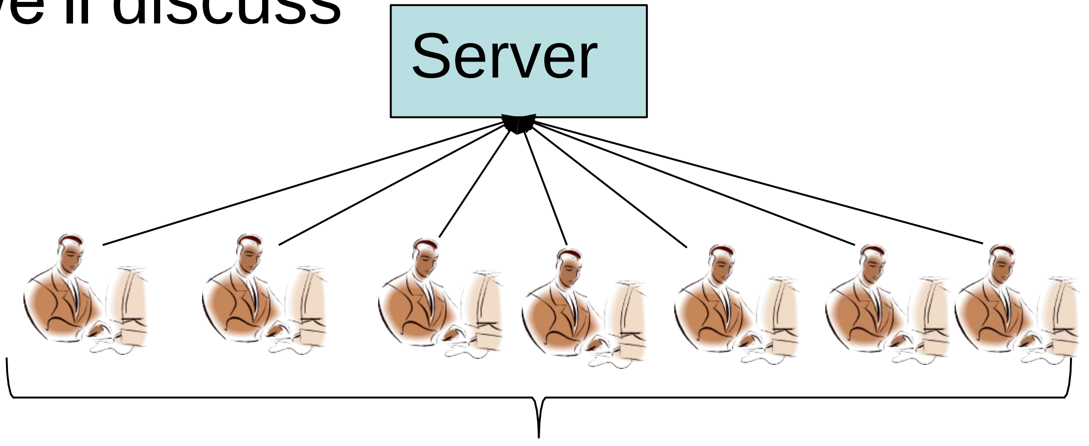
- What if your application stores unstructured data?
  - Photos and videos
  - Tweets and status updates
  - Shared documents
  - ….

- What if your users don't care about ACID semantics?
  - Your status update is not immediately visible
  - The edits you make to a document are visible with some delay
  - Conflicts in dropbox file

- This lecture: Let's look at Distributed File Systems (DFS)
  - NFS and AFS to understand tradeoffs in designing/choosing file systems
  - Google File System (GFS): the cloud-scale storage system used by original MapReduce implementation

NFS Overview

- Networked file system developed by Sun Microsystems in 80s
- Goal:
  - Have a consistent namespace for files across computers
  - Let authorized users access their files from any computer
- Fses like NFS and AFS are different in properties and mechanisms, and that’s what we’ll discuss



10,000s of users from 10,000 of machines

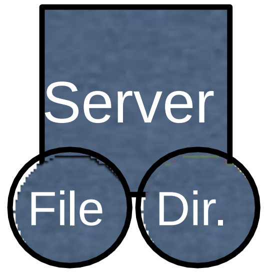
The FS Interface

File Ops

Open
Read
Write
Read
Write
Close

Directory Ops

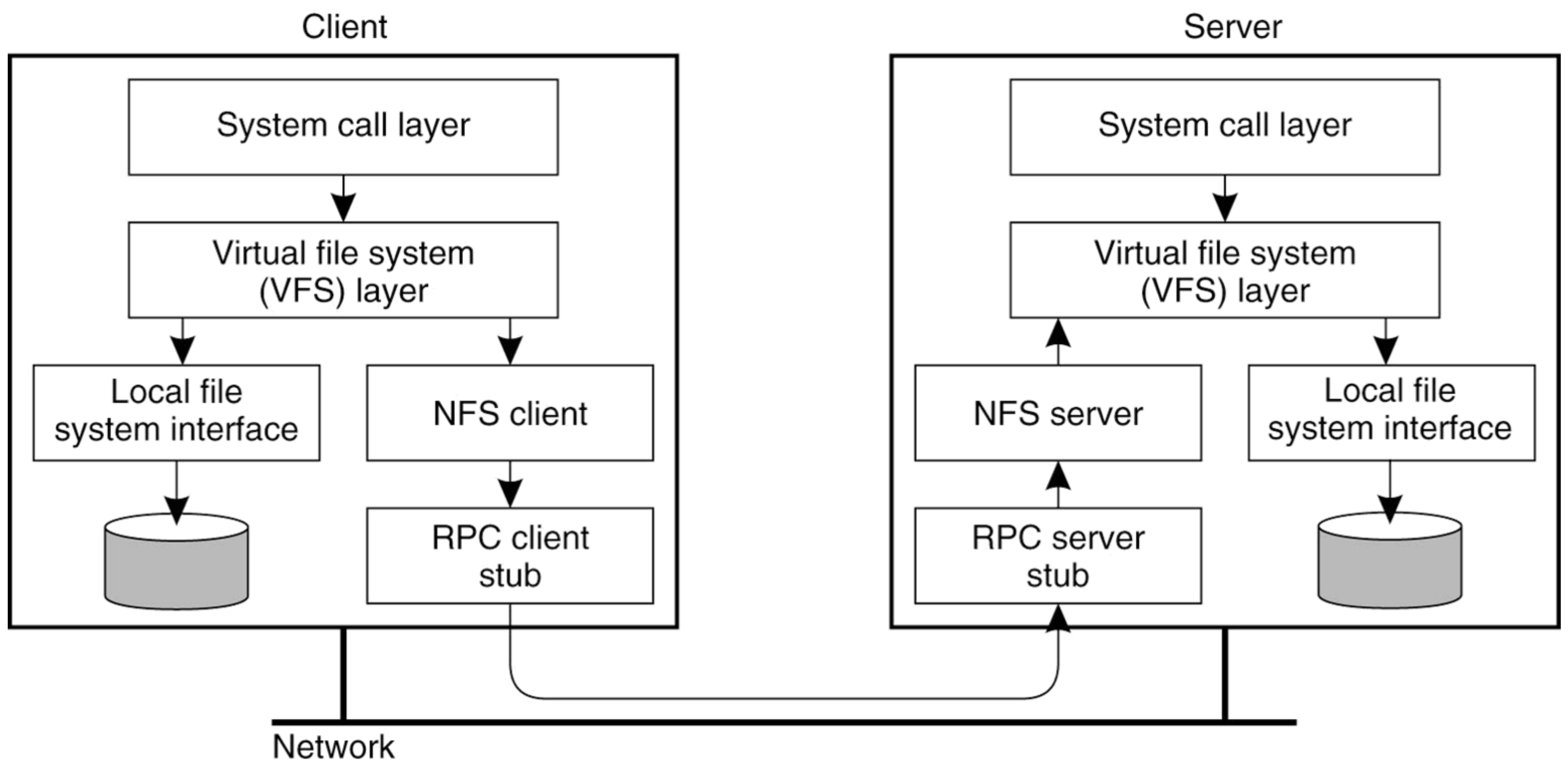
Create file
Mkdir
Rename file
Rename directory
Delete file
Delete directory



Components in a DFS Implementation

- Client side:
  - What has to happen to enable applications to access a remote file the same way a local file is accessed?
  - Accessing remote files in the same way as accessing local files → kernel support
- Communication layer:
  - Just TCP/IP or a protocol at a higher level of abstraction?
- Server side:
  - How are requests from clients serviced?

NFS: Client—server architecture



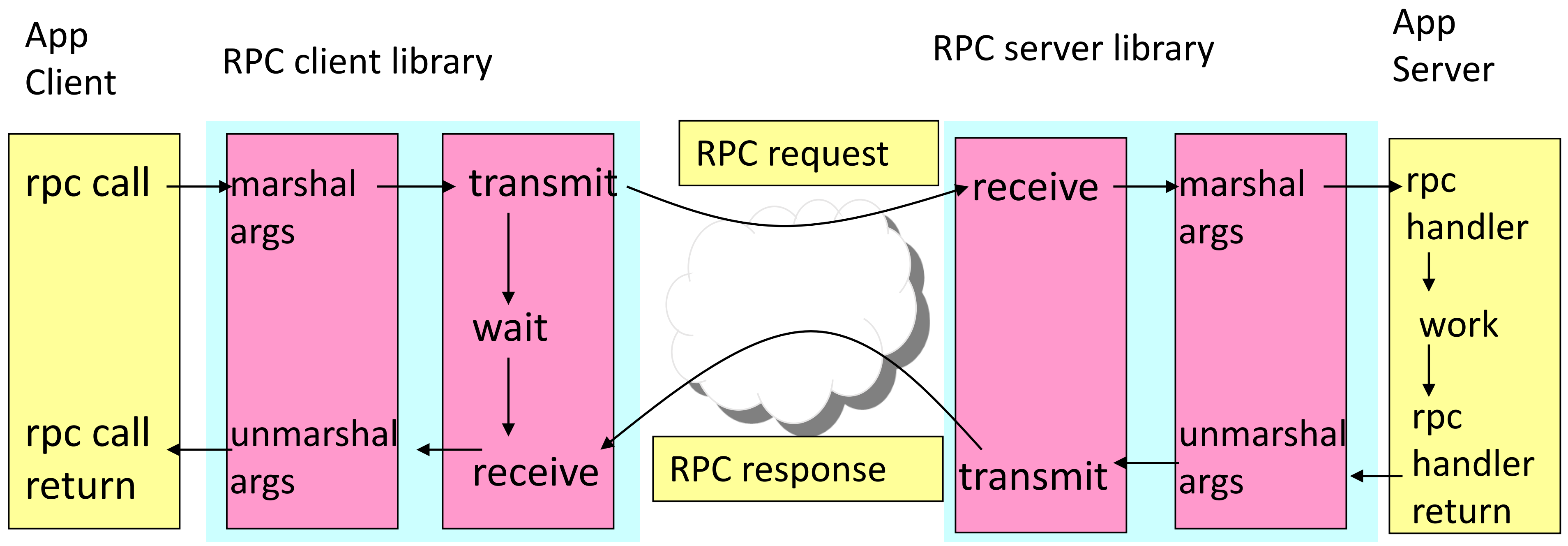
Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

A word about RPC

- Everyone loves procedure calls
  - Transfer control and data on local programs
- RPC goal: make client/server communication look like procedure calls
- Easy to write programs with
  - Procedure calls are a well-understood model
  - RPC hides details of passing data between nodes

RPC Overview

- Servers export their local procedure APIs
- On client, RPC library generates RPC requests over network to server
- On server, called procedure executes, result returned in RPC response to client



Marshaling

- Calling and called procedures run on different machines, with different address spaces
  - Therefore, pointers are meaningless
  - Plus, perhaps different environments, different operating systems, different machine organizations, …
  - E.g.: the endian problem: If I send a request to transfer $1 from my little-endian machine, the server might try to transfer $16M if it's a big-endian machine
- Must convert to local representation of data (structs, strings, etc.)
- That's what marshaling does for you

Key RPC Challenges

- Communication failures
  - delayed and lost messages
  - connection resets
  - expected packets never arrive
- Machine failures
  - Server or client failures
  - Did server fail before or after processing the request?
- Might be impossible to tell communication failures from machine failures

RPC failure semantics

- What are the possible outcomes in the face of failures?
  - Procedure did not execute / executed once / executed many times / partially executed
- Desired semantics: **at-most-once**
- Implementing at-most-once
  - Server might get same request twice...
  - Must re-send *previous* reply and not process request (implies: keep cache of handled requests/responses)
  - Must be able to identify requests
  - Strawman: remember *all* RPC IDs handled. -> Requires infinite memory.
  - Real: Keep sliding window of valid RPC IDs, have client number them sequentially.

Popular RPC technologies

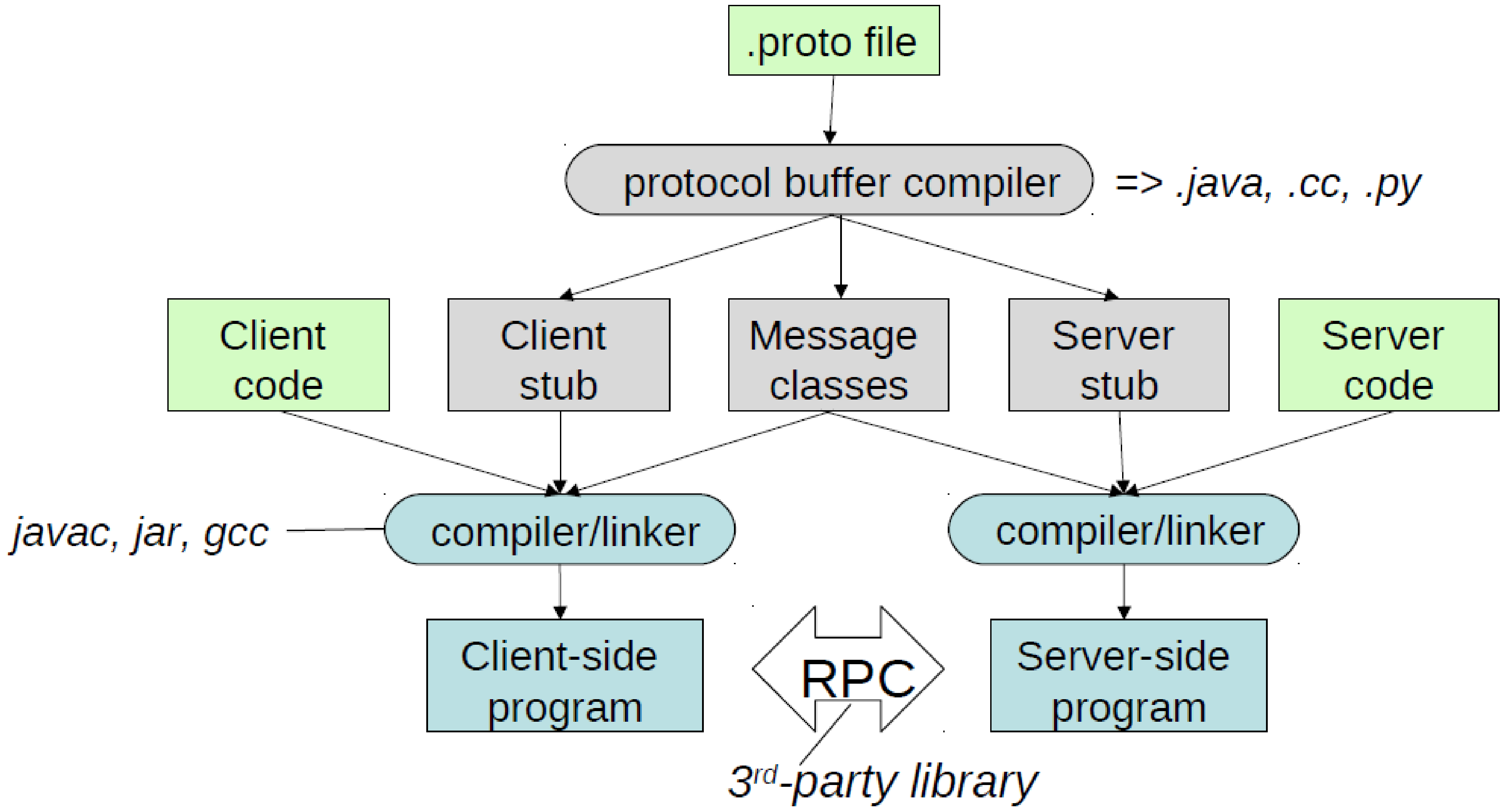
- SOAP
  - Designed for web services via HTTP, huge XML overhead
- gRPC & Protocol Buffers
  - Lightweight, developed by Google
- Thrift
  - Lightweight, supports services, developed by Facebook

Protocol Buffers

- Interface Definition Language (IDL)
  - Describe service interface and structure of payload messages

- Properties:
  - Efficient, binary serialization
  - Support protocol evolution
    - Can add new parameters
    - Order in which I specify parameters is not important
  - Supports types, which give you compile-time errors!
  - Supports complex structures

Protocol Buffers workflow



Example proto file: Address book

```
package tutorial
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
  required string name  = 1;
  required int32 id     = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME   = 1;
    WORK   = 2;
  }

  message PhoneNumber {
    required string number  = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
    repeated Person person = 1;
}
```

Compiling proto file

- The protocol-buffer library provides a compiler, which takes in a **.proto** file and generates corresponding classes in a language of your choice, e.g. Java, Python, or C++

```
# $PROTOC_HOME/bin/protoc –java_out $PWD AddressBook.proto
```

Generates com.example.tutorial.AddressBookProtos.java, with two classes: **Person** and **AddressBook**

- Nested within each class is a class for each message you specified in addressbook.proto
- Each class has its own Builder class that you use to create instances of that class.

Example using protocol buffer

We can serialize and de-serialize protocol buffer structures to/from input and output streams. These streams can be backed either by some network channel or by files or even by database connections.

```
package tutorial
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos"

message Person {
  required string name  = 1;
  required int32 id     = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME   = 1;
    WORK   = 2;
  }

  message PhoneNumber {
    required string number  = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
    repeated Person person = 1;
}
```

```
import com.example.tutorial.AddressBookProtos.Person;
import com.example.tutorial.AddressBookProtos.AddressBook;

public class HandleAddressBook {
  public static void createAndSerializeAddressBook(OutputStream output) {
    Person.Builder person = Person.newBuilder();
    person.setId(1234);
    person.setName(“John Doe”);

    Person.PhoneNumber.Builder phoneNumber =
        Person.PhoneNumber.newBuilder().setNumber(“102-203-4005”);
    phoneNumber.setType(Person.PhoneType.MOBILE);
    person.addPhone(phoneNumber);
    // Can add other phone numbers.
```

```
    // person.setEmail(“johndoe@email.com”);  // this is optional –may or may not add
it.

    Person person = person.build();  // generate the Person object.
    AddressBook.Builder addressBook = AddressBook.newBuilder();
    addressBook.addPerson(person);
    // Add other persons.

    // Write the new address book to an OutputStream (can be backed by a file, a
    // socket stream, etc.).
    addressBook.build().writeTo(output);
  }
```

gRPC

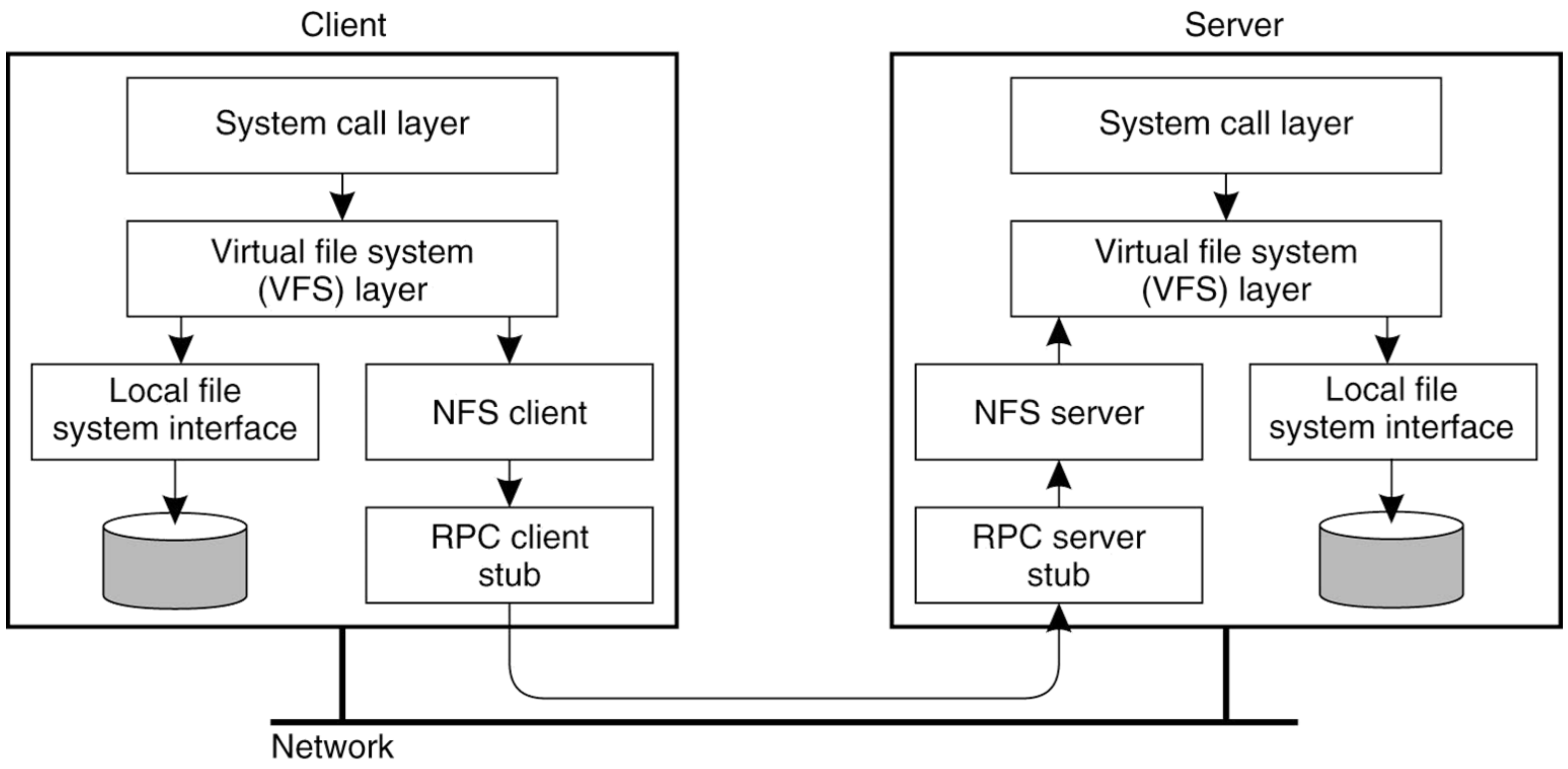
- Uses protocol buffers to define services and methods
  - You specify services and stub code is autogenerated

```
service HelloService {
        rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
        string greeting = 1;
}

message HelloResponse {
        string reply = 1;
}
```

Back to NFS: Client—server architecture


Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

Some NFS V2 RPC Calls

- NFS used SunRPC
  - Defines “XDR” (“eXternal Data Representation”) -- C-like language for describing structures and functions
  - Provides a compiler that creates stubs

Proc.	Input args	Results
LOOKUP	dirfh, name	status, fhandle, fattr
READ	fhandle, offset, count	status, fattr, data
CREATE	dirfh, name, fattr	status, fhandle, fattr
WRITE	fhandle, offset, count, data	status, fattr

NFS Server Side : mountd and nfsd

- mountd: provides the initial file handle for the exported directory
  - Client issues nfs_mount request to mountd
  - mountd checks if the pathname is a directory and if the directory should be exported to the client
- nfsd: answers the RPC calls, gets reply from local file system, and sends reply via RPC
  - Usually listening at port 2049
- Both mountd and nfsd use underlying RPC implementation

Naïve FS Design

- Use RPC to forward *every* FS operation to the server
  - Server orders all accesses, performs them, and sends back result
- Good: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance will stink. Latency of access to remote server often much higher than to local memory.
- Really bad: Server would get hammered!

Question 1: How can we avoid going to the server for everything? *What* can we avoid this for? What do we lose in the process?

Client-Side Caching

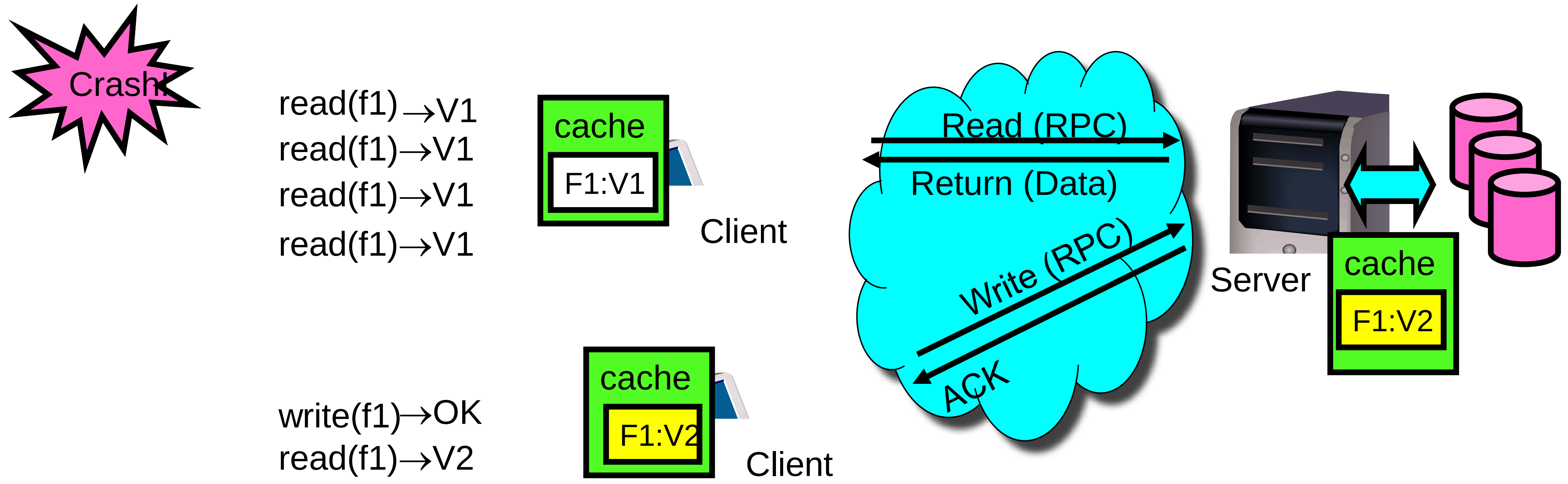
- Huge parts of systems rely on two solutions to every problem:

  “All problems in computer science can be solved by adding another level of indirection. But that will usually create another problem.”

  David Wheeler

- So add caching! But what do we cache?
  - Read-only file data and directory data → easy
  - Data written by the client machine → when is data written to the server? What happens if the client machine goes down?
  - Data that is written by other machines → how to know that the data has changed?
- And if we cache... doesn’t that risk making things inconsistent?

Caching Problem 1: Consistency



Client Caching in NFS v2

- Cache both clean and dirty file data and file attributes
- File attributes in the client cache expire after 60 seconds (file data doesn’t expire)
- File data is checked against the modified-time in file attributes (which could be a cached copy)
  - Changes made on one machine can take up to 60 seconds to be reflected on another machine
- Dirty data are buffered on the client machine until file close or up to 30 seconds
  - If the machine crashes before then, the changes are lost

Implication of NFS v2 Client Caching

- Advantage: No network traffic if open/read/write/close can be done locally.
- But…. Data consistency guarantee is very poor
  - Simply unacceptable for some distributed applications
  - Productivity apps tend to tolerate such loose consistency
- Generally clients do not cache data on local disks

Design Choice

- Clients can choose a stronger consistency model:
    - *close-to-open* consistency

– How? Always ask server for updates before open()

– Trades a bit of scalability / performance for better consistency

- What about multiple writes?
    - NFS provides no guarantees at all!
    - Might get one client’s writes, other client’s writes, or a mix of both!

Caching Problem 2: Failures

- Server crashes
  - Data in memory but not disk lost
  - So... what if client does
    - seek() ; /* SERVER CRASH */; read()
    - If server maintains file position, this will fail. Ditto for open(), read()
- Lost messages: what if we lose acknowledgement for delete(“foo”)
  - And in the meantime, another client created foo anew?
- Client crashes
  - Might lose data in client cache

NFS's Failure Handling: Stateless Server

- Files are state, but...
- Server exports files without creating extra state
  - No list of "who has this file open" (permission check on each operation on open file!)
  - No "pending transactions" across crash
- Crash recovery is "fast"
  - Reboot, let clients figure out what happened
- State stashed elsewhere
  - Separate MOUNT protocol
  - Separate NLM locking protocolStateless design
- Stateless protocol: requests specify exact state. read() → read([position]). no seek on server.

NFS's Failure Handling

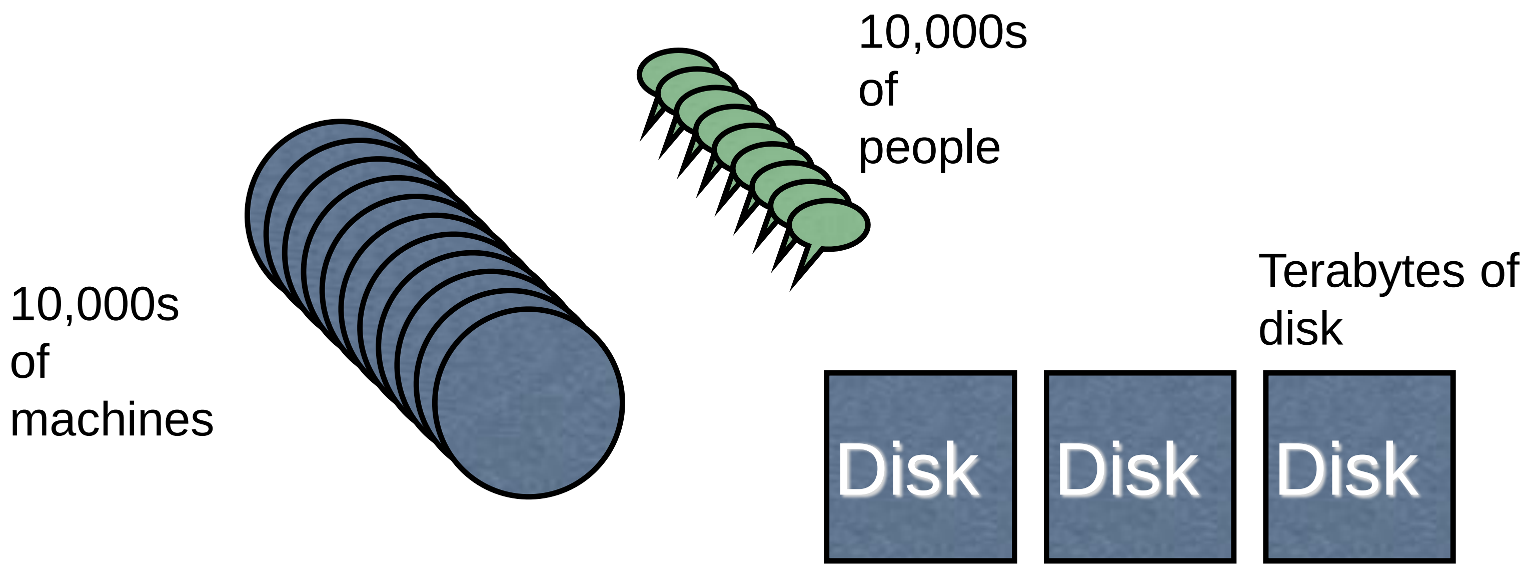
- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories. It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused.
- Write-through caching: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Close failures?
    - retry until things get through to the server
    - return failure to client
  - Most client apps can't handle failure of close() call.
  - Usual option: hang for a long time trying to contact server

NSF Summary

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it's gotten a little more complex over time, but...)
- Weak consistency semantics
- Requires hefty server resources to scale (write-through, server queried for lots of operations)

andrew.cmu.edu

- Andrew File System (AFS)



Goal: Have a consistent namespace for files across computers. Allow any authorized user to access their files from any computer

AFS Assumptions

- Client machines are untrusted
  - Must **prove** they act for a specific user
    - Secure RPC layer
  - Anonymous “system:anyuser”
- Client machines have disks(!!)
  - Can use them also for caching!
- Write/write and write/read sharing are rare
  - Most files updated by one user, on one machine

Aggressive caching in AFS

- More aggressive caching (AFS caches on disk in addition to RAM)
- Prefetching (on open, AFS gets entire file from server)
  - When would this be more efficient than prefetching N blocks?
  - With traditional hard drives, large sequential reads are much faster than small random reads.
  - So it’s more efficient read whole files. Improves scalability, particularly if client is going to read whole file anyway eventually.

Client Caching in AFS

- Close-to-open consistency only
  - Makes sense based on read/write sharing assumptions

- Invalidation callbacks: Clients register with server that they have a copy of file;
  - Server tells them: “Invalidate!” if the file changes
  - This trades state for improved consistency

- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone “who has which files cached?”

AFS Write Policy

- Writeback cache
  - Opposite of NFS “every write is sacred”
  - Store chunk back to server
    - When cache overflows
    - On last user close()
  - ...or don't (if client machine crashes)
- Is writeback crazy?
  - Write conflicts “assumed rare”
  - Who wants to see a half-written file?

AFS Summary

- Lower server load than NFS
  - More files cached on clients
  - Cache invalidation callbacks: server not busy if files are read-only (common case)
- But maybe slower: Access from local disk is much slower than from another machine's memory over a LAN
- For both, central server is:
  - – A bottleneck: reads and writes hit it at least once per file use;
  - – A single point of failure;
  - – Expensive: to make server fast, beefy, and reliable, you need to pay $$$.

DFS always involve a tradeoff: consistency, performance, scalability.

GFS Design Constraints

1. Machine 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 must be integral parts of a design

2. Design for big-data workloads
   - Search, ads, Web analytics, Map/Reduce, …

Workload Characteristics

- Files are huge by traditional standards
  - Multi-GB files are common

- Most file updates are appends
  - Random writes are practically nonexistent
  - Many files are written once, and read sequentially

- High bandwidth is more important than latency lots of concurrent data accessing
  - E.g., multiple crawler workers updating the index file

GFS' design is geared toward apps' characteristics

– And Google apps have been geared toward GFS

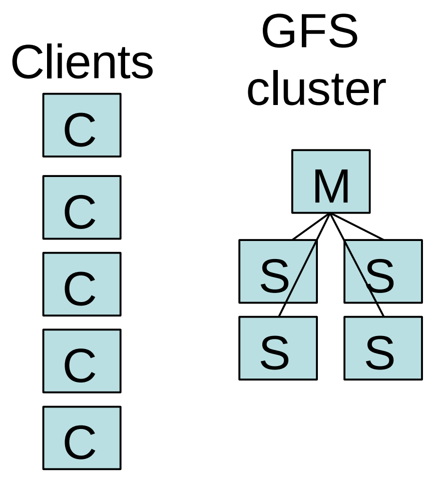
GFS Interface

- Not POSIX compliant
  - Supports only few FS operations, and semantics are different
  - That means you wouldn’t be able to mount it…

- Additional operation: record append
  - Frequent operation at Google:
    - Merging results from multiple machines in one file (Map/Reduce)
    - Using file as producer - consumer queue
    - Logging user activity, site traffic
  - Order doesn’t matter for appends, but atomicity and concurrency matter

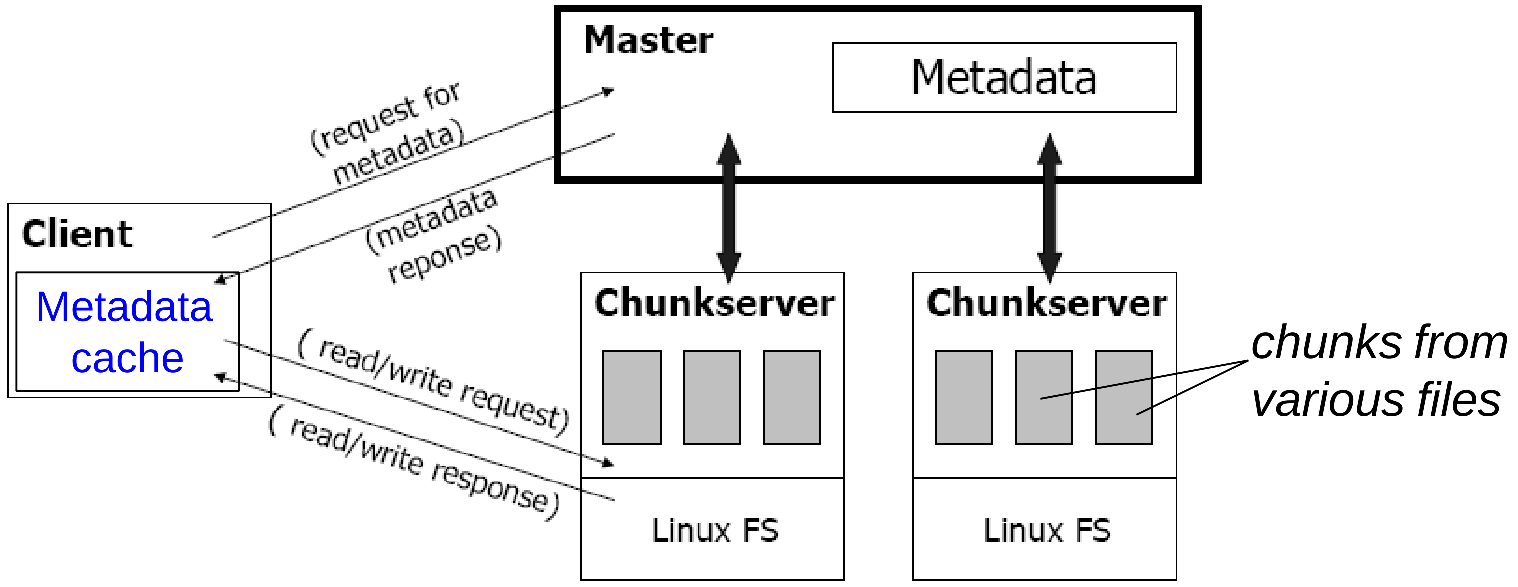
Architectural Design

- A GFS cluster
  - A single master (replicated later)
  - Many chunkservers
    - Accessed by many *clients*
- A file
  - Divided into fixed-sized chunks (similar to FS blocks)
    - Labeled with 64-bit unique global IDs (called *handles*)
    - Stored at chunkservers
    - 3-way replicated across chunkservers
    - Master keeps track of metadata (e.g., which chunks belong to which files)

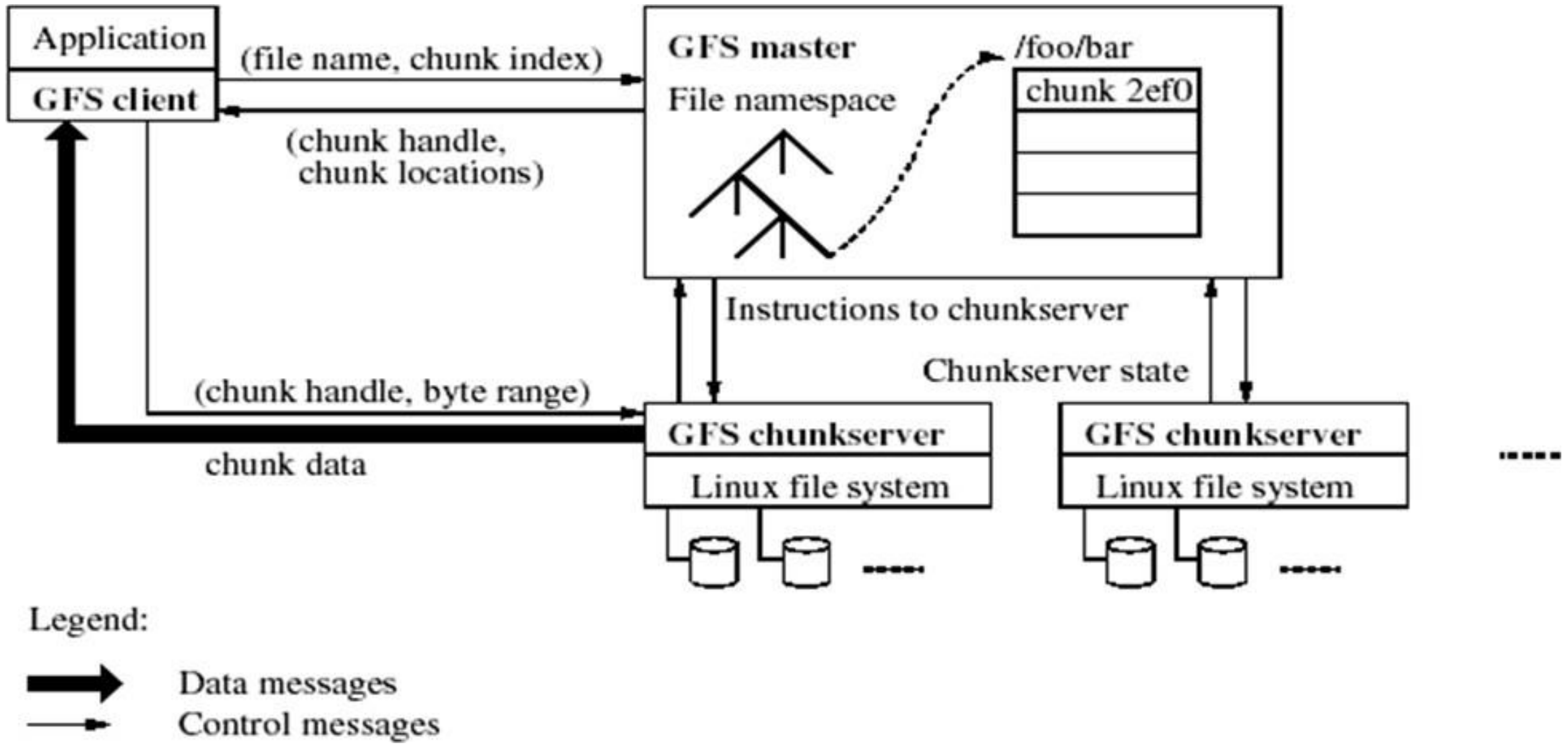


GFS Basic Operation

- Client retrieves metadata for operation from master
- Read/Write data flows between client and chunk server
- Minimizing the master's involvement in read/write operations alleviates the single-master bottleneck



Architecture in depth



Chunks

- Analogous to FS blocks, except larger
- Size: 64 MB!
  - Normal FS block sizes are 512B - 8KB
- Pros of big chunk sizes:
  - Less load on server (less metadata, hence can be kept in master's memory)
  - Suitable for big-data applications (e.g., search)
  - Sustains large bandwidth, reduces network overhead
- Cons of big chunk sizes:
  - Fragmentation if small files are more frequent than initially believed

GFS Master

- A process running on a separate machine
  - Initially, GFS supported just a single master, but then they added master replication for fault-tolerance in other versions/distributed storage systems
  - The replicas use Paxos, an algorithm that we'll talk about later to keep coordinated, i.e., to act as one
- Stores all metadata
  - Namespaces (Hierarchical namespace for files, flat namespace for chunks)
  - File-to-chunk mappings
  - Locations of a chunk’s replicas

Chunk Locations

- Kept in memory, no persistent states
  - Master polls chunk servers at startup
- What does this imply?
  - Upsides: master can restart and recover chunks from chunkservers
    - Note that the hierarchical file namespace is kept on durable storage in the master
    - Downside: restarting master takes a long time
- Rationale
  - Design for failures
  - Simplicity
  - Scalability – the less persistent state master maintains, the better

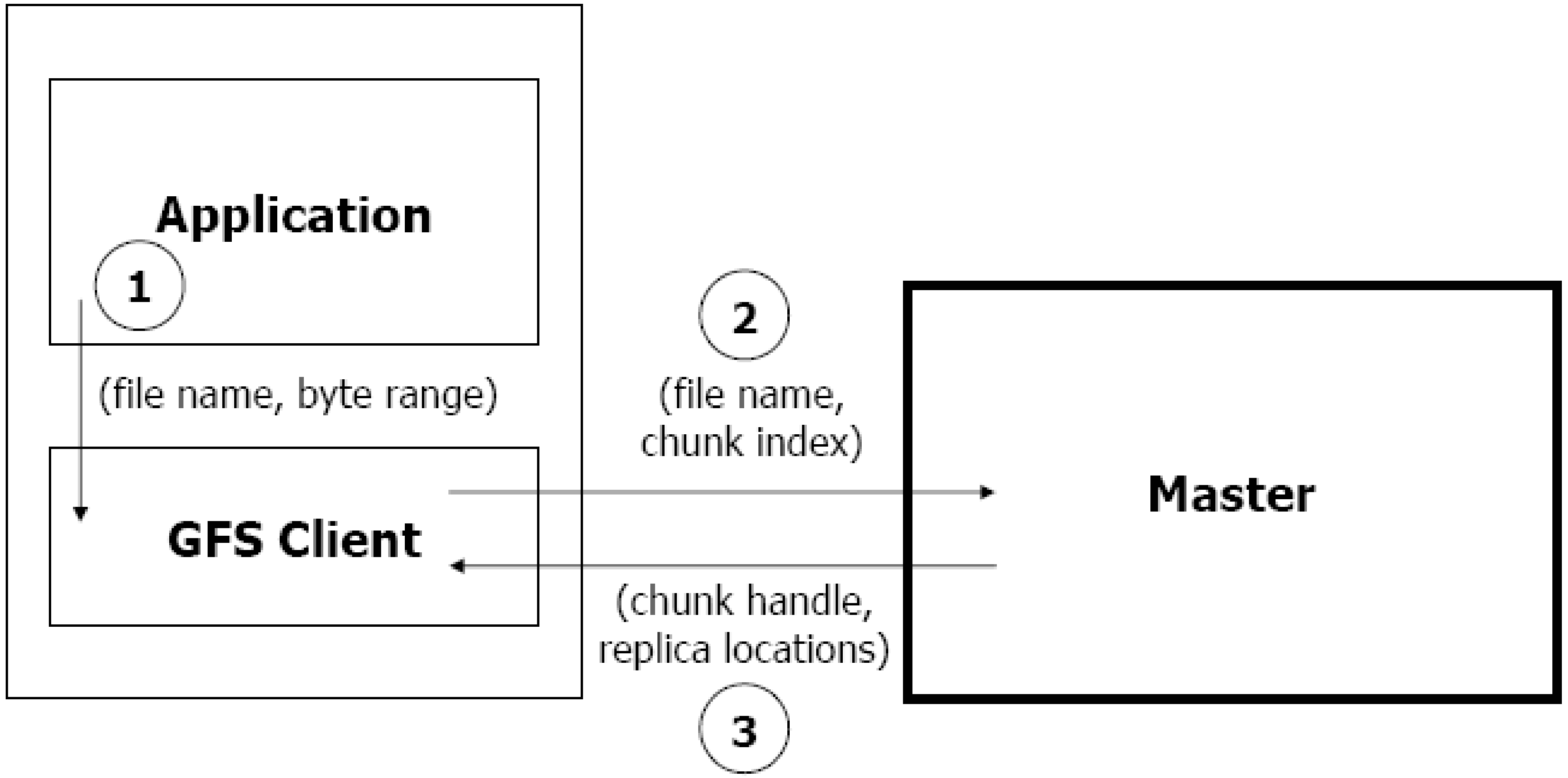
Master coordinates chunk servers

- Master and chunkserver communicate regularly (heartbeat):
  - Is chunkserver down?
  - Are there disk failures on chunkserver?
  - Are any replicas corrupted?
  - Which chunks does chunkserver store?
- Master sends instructions to chunkserver:
  - Delete a chunk
  - Create new chunk
  - Replicate and start serving this chunk (chunk migration)
    - Why do we need migration support?

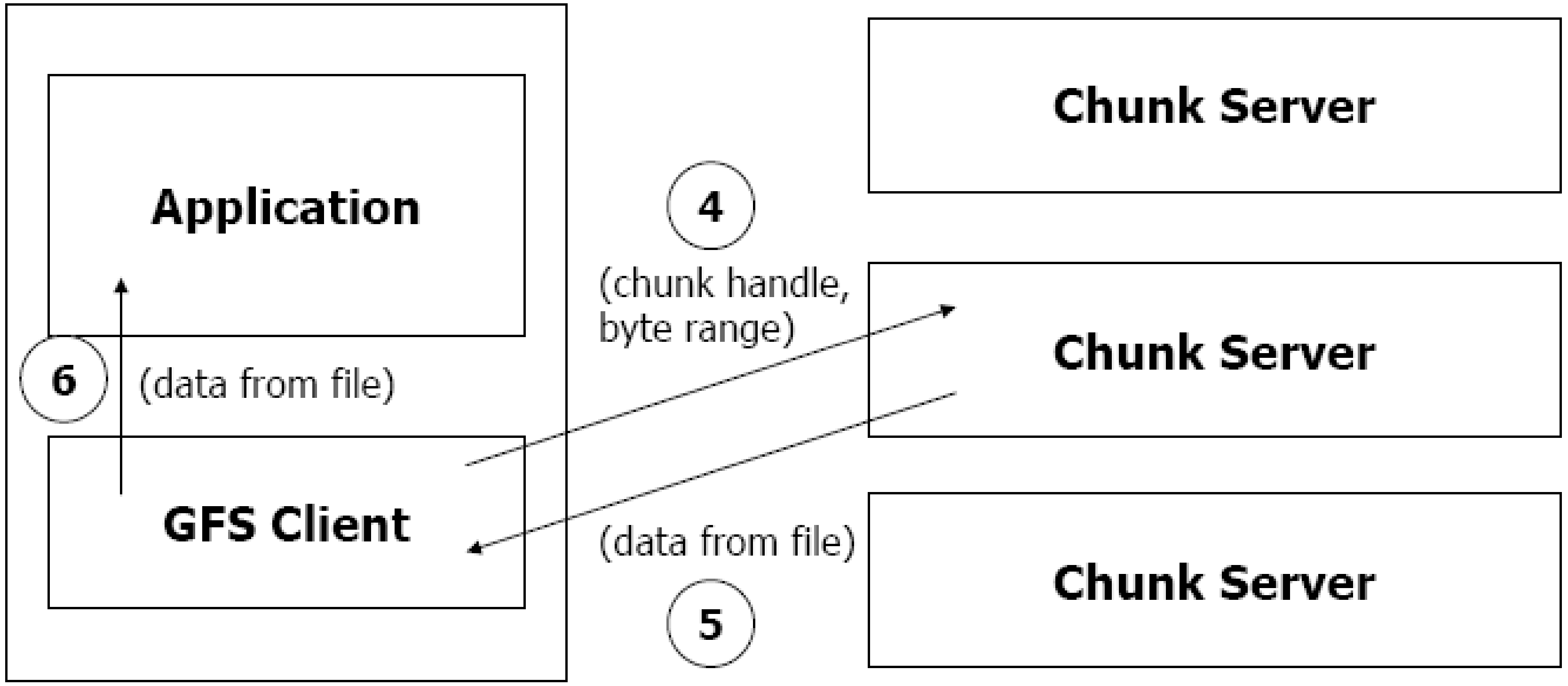
GFS Operations

- Reads
- Updates:
  - Writes
  - Record appends

Read Protocol



Read Protocol



Updates

- Writes:
  - Cause data to be written at application-specified file offset
- Record appends:
  - Operations that append data to a file
  - Cause data to be appended atomically at least once
  - Offset chosen by GFS, not by the client
- Goals:
  - Clients can read, write, append records at max throughput and in parallel
  - Some consistency that we can understand (kinda)
  - Hence, favor concurrency / performance over semantics

Update order

- For consistency, updates to each chunk must be ordered in the same way at the different chunk replicas

- Consistency means that replicas will end up with the same version of the data and not diverge

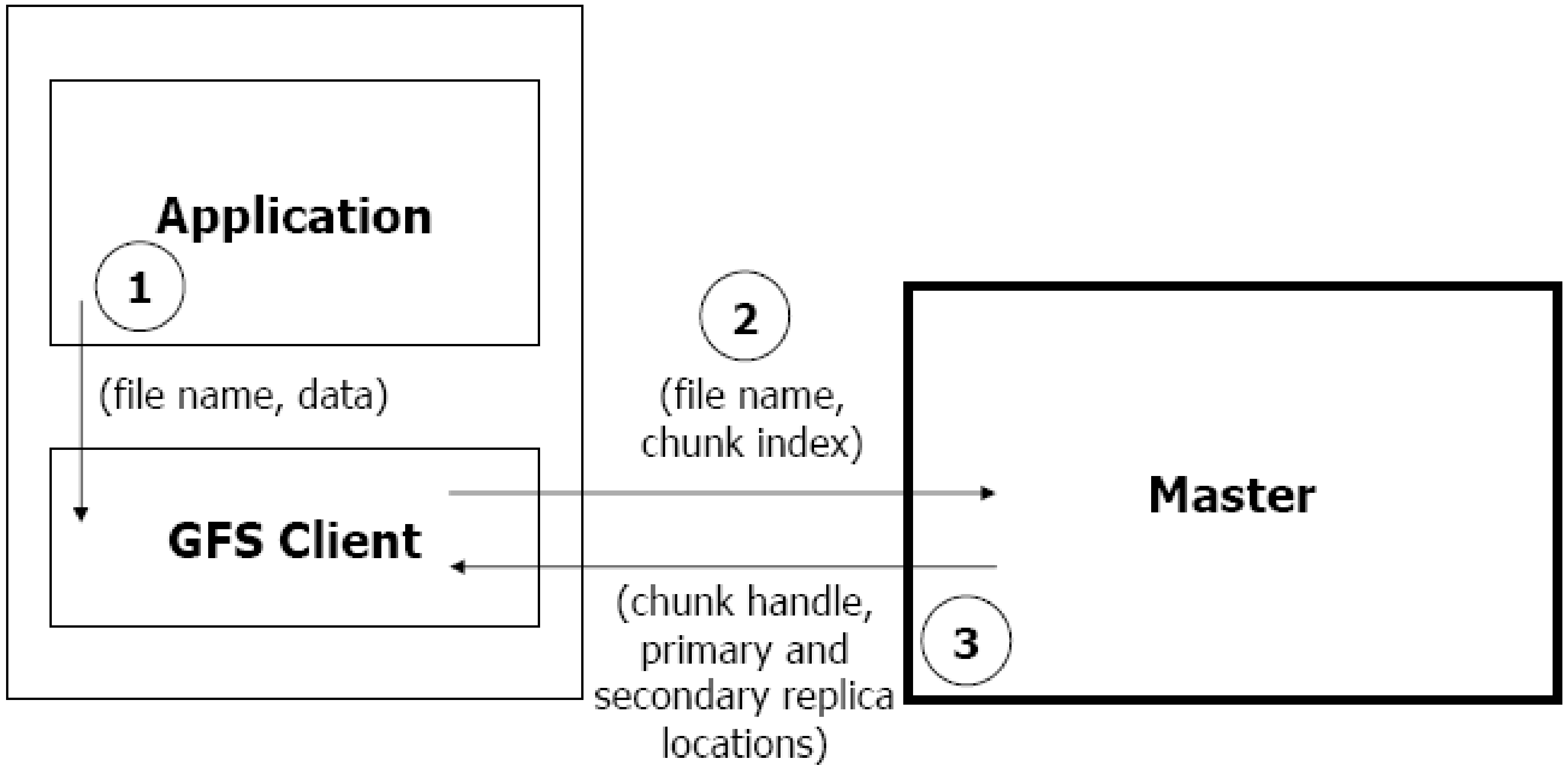
- For this reason, for each chunk, one replica is designated as the **primary**. The other replicas are designated as **secondaries.**

- Primary defines the update order. All secondaries follows this order

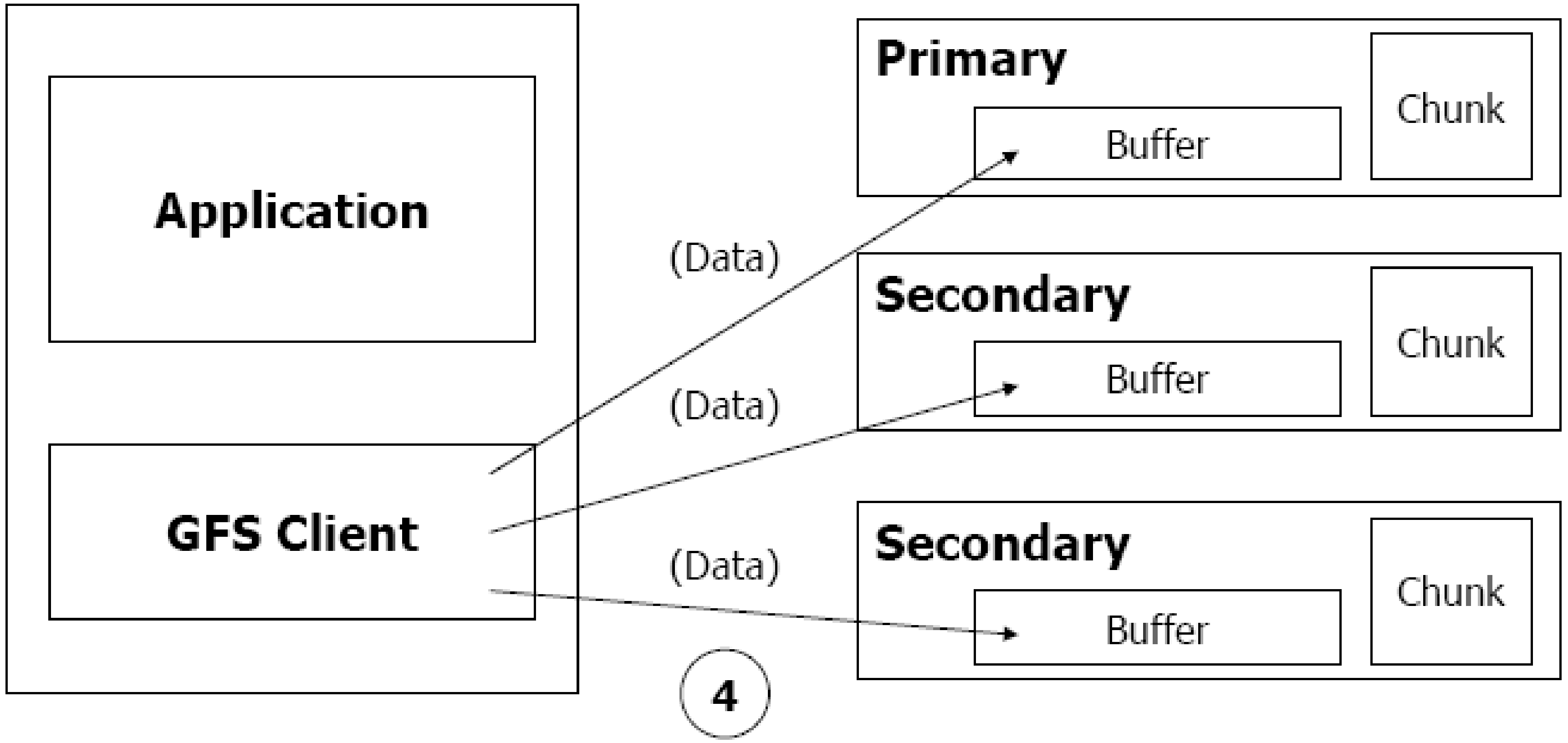
Primary Leases

- For correctness, at any time, there needs to be one single primary for each chunk
  - Or else, they could order different writes in different ways
- To ensure that, GFS uses leases
  - Master selects a chunkserver and grants it lease for a chunk
  - The chunkserver holds the lease for a period T after it gets it, and behaves as primary during this period
  - The chunkserver can refresh the lease endlessly
  - But if the chunkserver can't successfully refresh lease from master, it stops being a primary
  - If master doesn't hear from primary chunkserver for a period, it gives the lease to someone else
- So, at any time, at most one server is primary for each chunk
  - But different servers can be primaries for different chunks

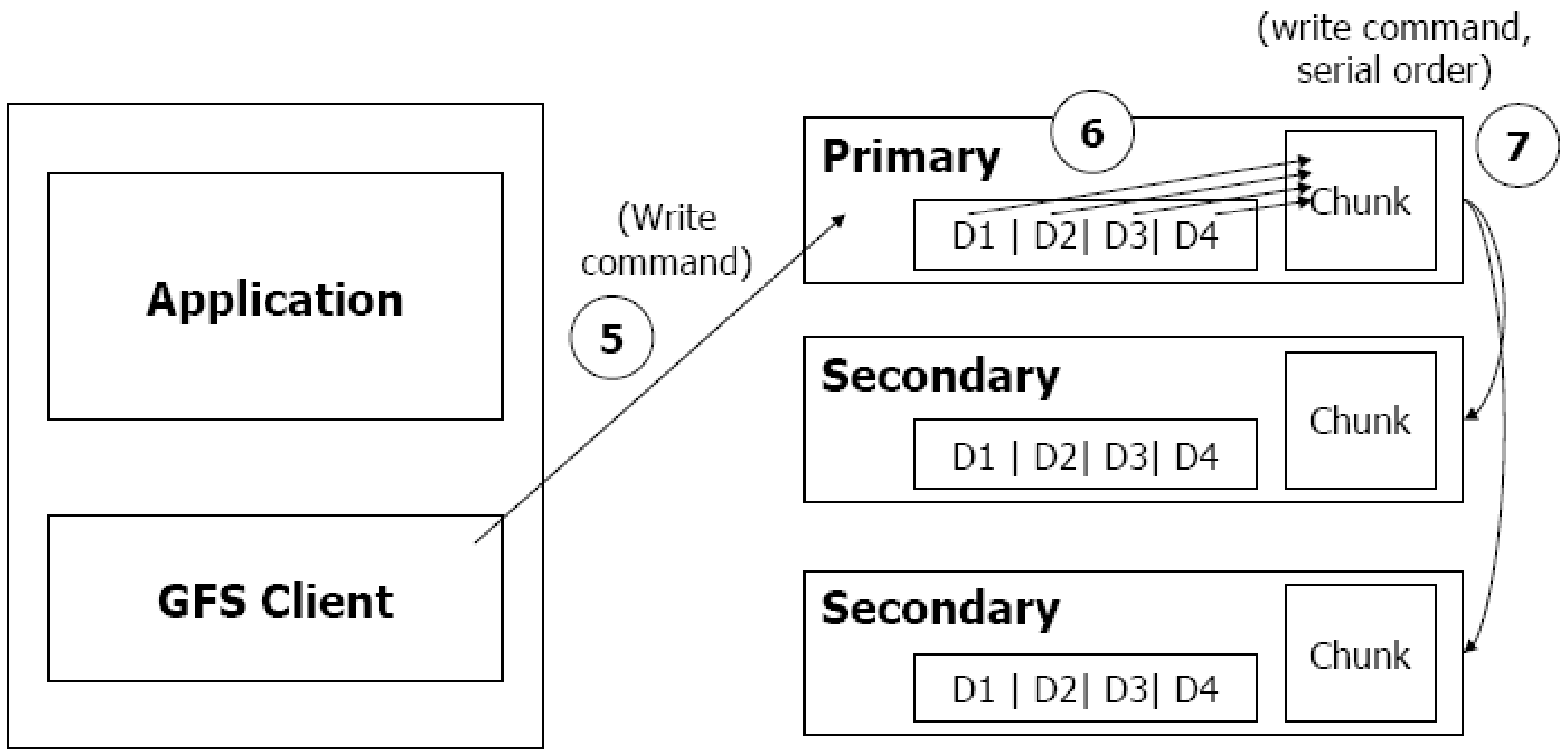
Write Algorithm



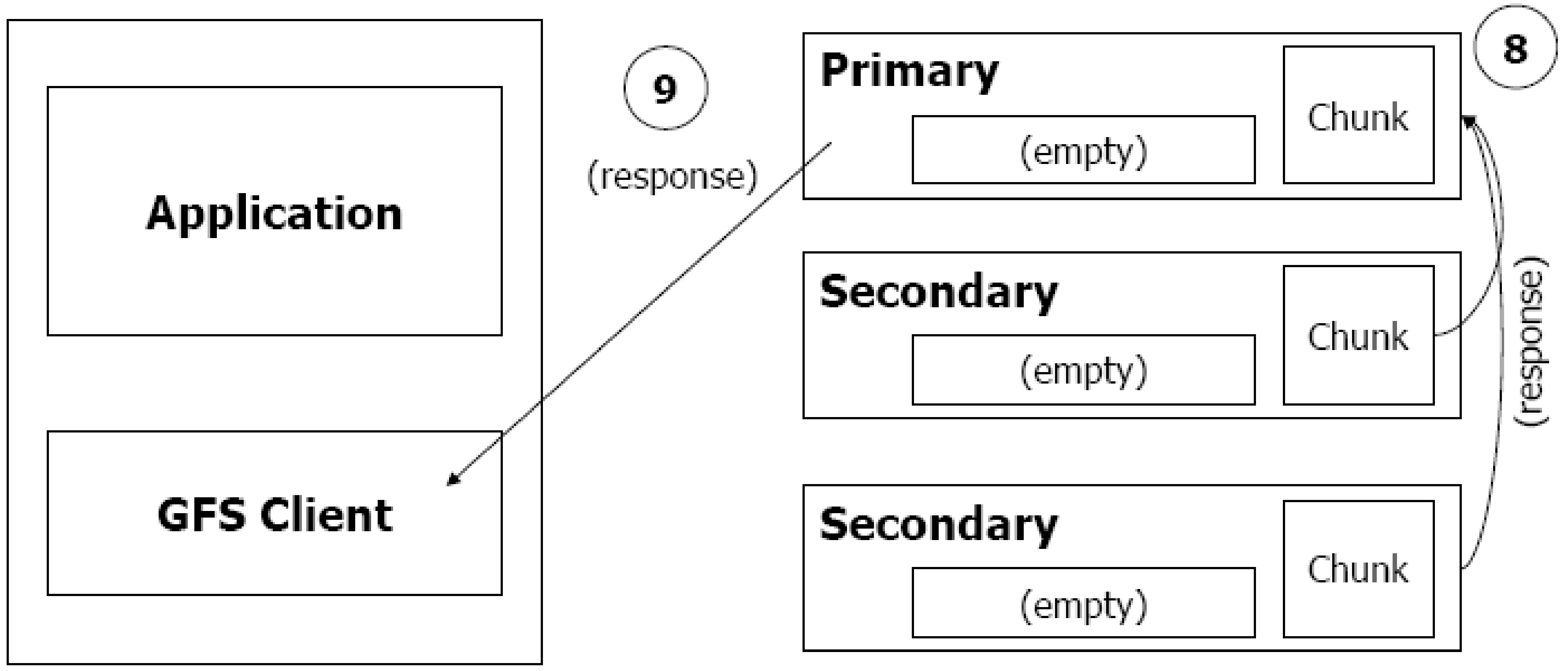
Write Algorithm



Write Algorithm



Write Algorithm



Write consistency

- Primary enforces one update order across all replicas for concurrent writes
- It also waits until a write finishes at the other replicas before it replies
- Therefore:
  - We'll have identical replicas
  - But, file region may end up containing mingled fragments from different clients
    - E.g., writes to different chunks may be ordered differently by their different primary chunkservers
  - Thus, writes are consistent but undefined in GFS

Record Appends

- The client specifies only the data, not the file offset
  - File offset is chosen by the primary
- Provide meaningful semantic: at least once atomically
  - Because FS is not constrained where to place data, it can get atomicity without sacrificing concurrency
- Rough mechanism:
  - If record fits in chunk, primary chooses the offset and communicates it to all replicas: *offset is arbitrary*
  - If record doesn’t fit in chunk, the chunk is padded and client gets failure: *file may have blank spaces*
  - If a record append fails at any replica, the client retries

Record Append Algorithm

1. Application originates record append request.
2. GFS client translates request and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations.
5. Primary checks if record fits in specified chunk.
6. If record does not fit, then:
   - Primary pads chunk, tells secondaries to do the same, and informs client.
   - Client then retries the append with the next chunk.
7. If record fits, then the primary:
   - appends the record at some offset in chunk
   - tells secondaries to do the same (specifies offset)
   - receives responses from secondaries
   - and sends final response to the client.

Implications of weak consistency

- Relying on appends rather on overwrites
- Applications need to write self-validating records
  - Checksums to detect and remove *padding*
- And self-identifying records
  - Unique Identifiers to identify and discard *duplicates*

- Hence, applications need to adapt to GFS and be aware of its inconsistent semantics
  - We'll talk soon about several consistency models, which make things easier/harder to build apps against

GFS Summary

- Optimized for large files and sequential appends (large chunk size)

- File system API tailored to stylized workload

- Single-master design to simplify coordination
  - But minimize workload on master by not involving master in large data transfers

- Implemented on top of commodity hardware
  - Unlike AFS/NFS, which for scale, require a pretty hefty server

NFS-AFS-GFS Takeaway

- Distributed (file)systems always involve a tradeoff: consistency, performance, scalability.
- Often times one must define one’s own consistency model & operations, informed by target workloads
  - AFS provides close-to-open semantic, and, they argue, that’s what users care about when they don't share data
  - GFS provides atomic-at-least-once record appends, and that’s what some big-data apps care about

But how do these compare? What’s “right”?

What do these mean for applications/users?

Coming up next: NoSQL Key-value stores, Formalizing consistency