

Consistency & DFS

Lecture 8

Resume Example

- User table
 - Primary key: user_id
 - first_name and last_name as columns
- One-to-many relationship
 - >1 job, contact information
- Storing one-to-many
 - Normalized form
 - put positions, education, and contact information in separate tables
 - foreign key reference to the users table
- Retrieving data requires complex joins
 - Can slow down perf



Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

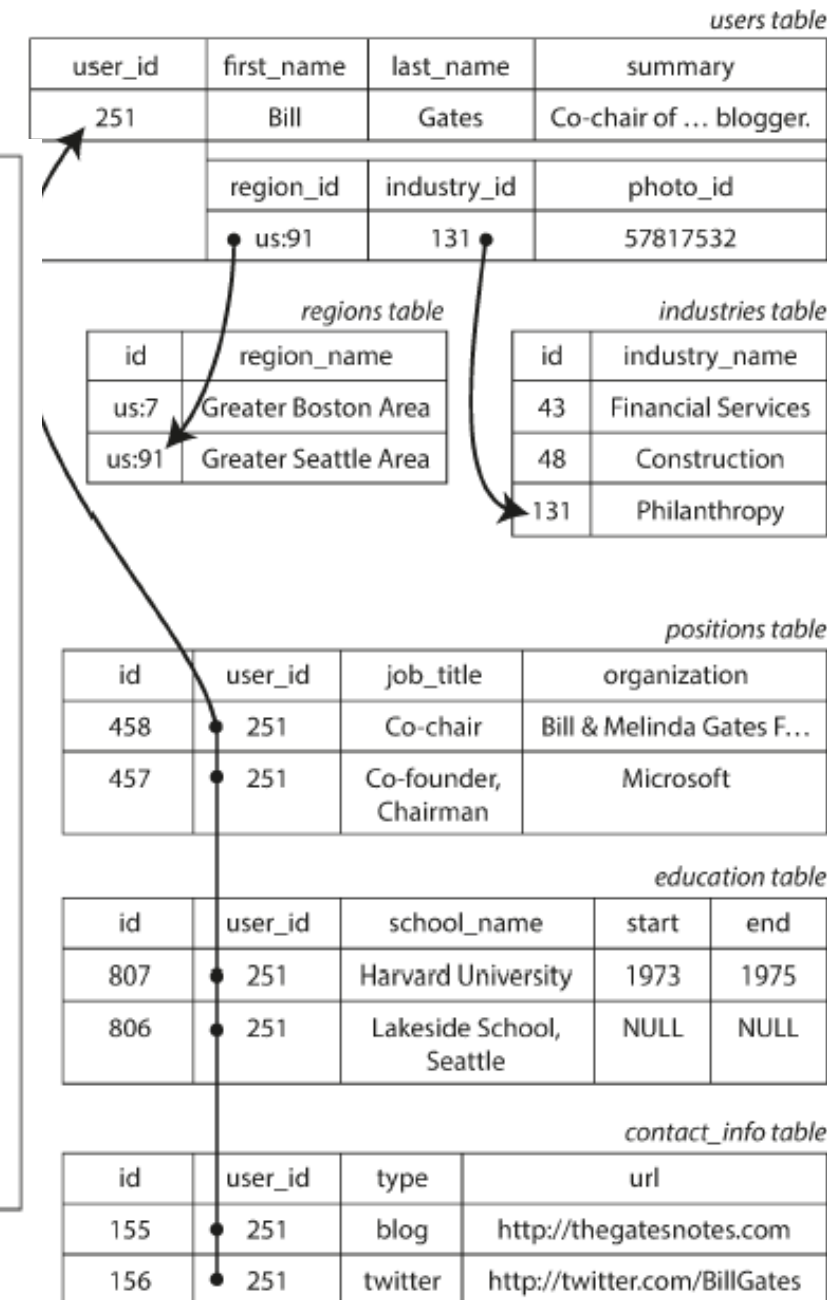
Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates



Resume Example: JSON

- Makes tree structure of one-to-many relationships explicit
- Could potentially provide better perf
 - No need for joins



Bill Gates
Greater Seattle Area | Philanthropy

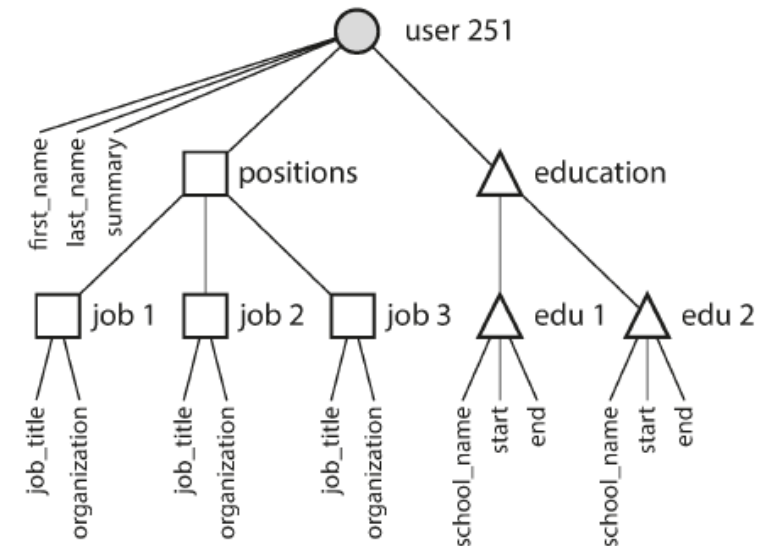
Summary
Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience
Co-chair • Bill & Melinda Gates Foundation
2000 – Present
Co-founder, Chairman • Microsoft
1975 – Present

Education
Harvard University
1973 – 1975
Lakeside School, Seattle

Contact Info
Blog: thegatesnotes.com
Twitter: @BillGates

```
{  "user_id": 251,  "first_name": "Bill",  "last_name": "Gates",  "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",  "region_id": "us:91",  "industry_id": 131,  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",  "positions": [    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}  ],  "education": [    {"school_name": "Harvard University", "start": 1973, "end": 1975},    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}  ],  "contact_info": {    "blog": "http://thegatesnotes.com",    "twitter": "http://twitter.com/BillGates"  }}
```



How about Many-to-one relationships?

- region_id and industry_id as IDs or strings ("Greater Seattle Area" and "Philanthropy")?
- Storing as ID
 - Avoid ambiguity & errors (data not copied)
 - Ease of updating (name is stored in only one place)
 - Localization support (multiple languages)
 - This is normalization!
- But normalization leads to many-to-one
 - many people live in one region
- How do we support many-to-one?
 - Databases can do joins across PK-FK
 - No support for join => app has to do a join
- Even simple example illustrates tradeoffs

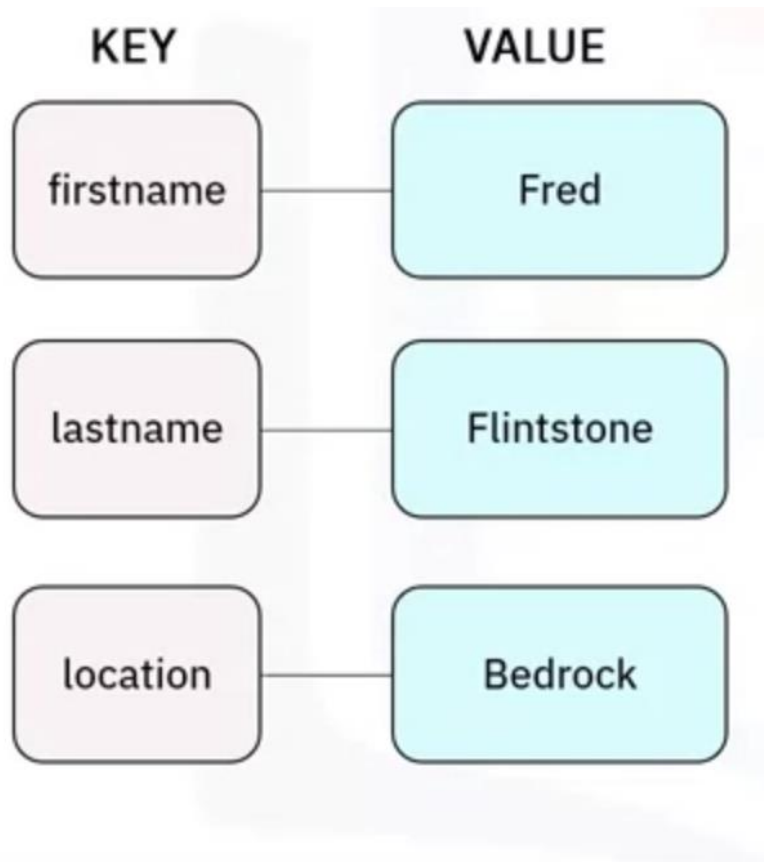
```
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University", "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

NoSQL

- Stands for Not-only SQL
 - originally intended simply as a catchy Twitter hashtag for a meetup on open source, distributed, nonrelational databases in 2009
- Why were they designed
 - Need for greater scalability than open-source relational databases
 - Preference for free and open source software
 - Desire for a more dynamic and expressive data model than relational model
 - Specialized query operations that are not well supported by the relational model

NoSQL Categories

- 4 types
 - **Key-value stores**, Document stores, wide column stores, Graph stores



- Store key-value pairs
 - Essentially a hashmap
- Ideal for basic Create-Read-Update-Delete (CRUD) ops
- Scales well
- Not meant for complex queries
- Atomic for single keys only!
 - No multi-ops transactions or ACID
- Value is opaque
 - No indexing and querying over values
- Ex: DynamoDB, Redis, Memcached
 - Use case: Storing and retrieving user session information, storing shopping cart data

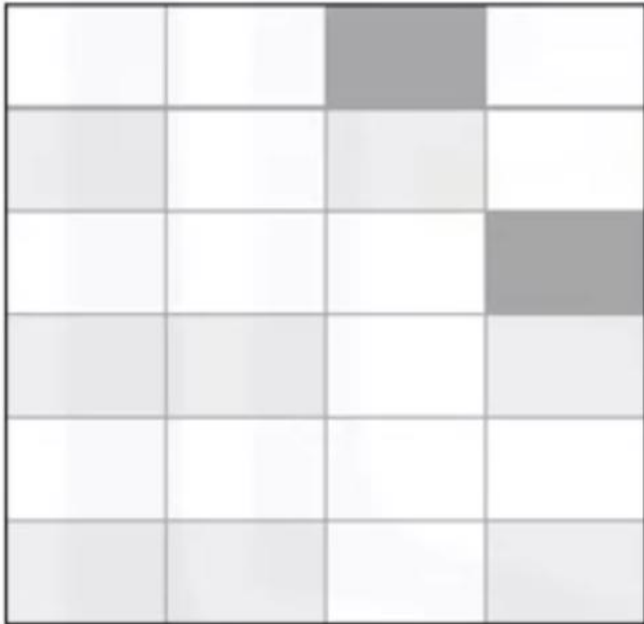
NoSQL Categories

- 4 types
 - Key-value stores, **Document stores**, wide column stores, Graph stores
- Extend KV model by making values visible and queryable
- Stores documents (JSON, XML usually)
 - Flexible schema, documents can be indexed
- Atomic operations on single document!
- Denormalize data
 - Add redundancy to make docs self contained
 - Avoid many-to-one problem
- Ex: MongoDB, CouchDB
 - Use case: Event logging for application, data for mobile/web apps



NoSQL Categories

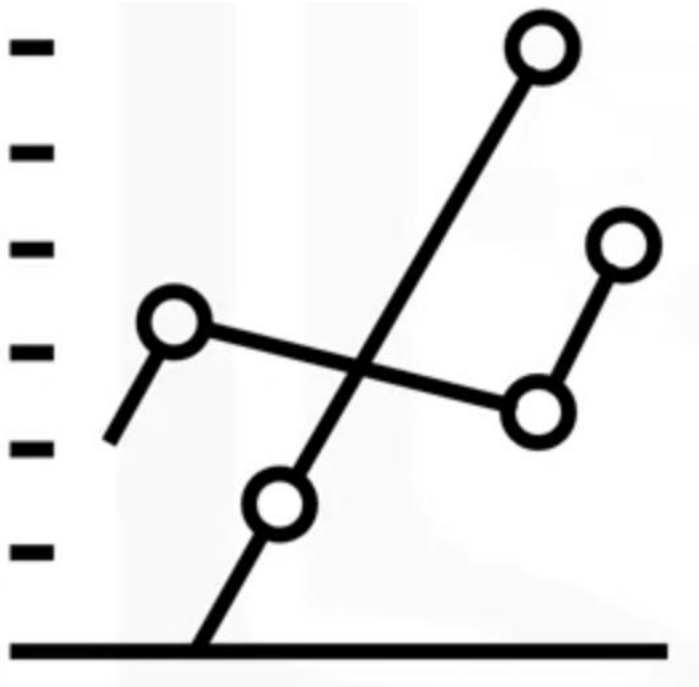
- 4 types
 - Key-value stores, Document stores, **wide column stores**, Graph stores



- Built off of Google BigTable
- Builds off relational model to accommodate rows with different number of columns
- Good for dealing with sparse data
 - Rows need not store empty columns
- Reads/writes atomic at row level!
- Ex: Cassandra, Hbase
 - Niche use cases

NoSQL Categories

- 4 types
 - Key-value stores, Document stores, wide column stores, **Graph stores**



- Excellent for graph-structured data
 - Entities as nodes, relationships as edges
 - Ex: Routing, spatial, map, social media data
- Can provide ACID updates to graph data
- Typically niche and not suitable in many cases
 - Graphs difficult to partition. So graphDBs are difficult to scale out
- Ex: Neo4j, Neptune

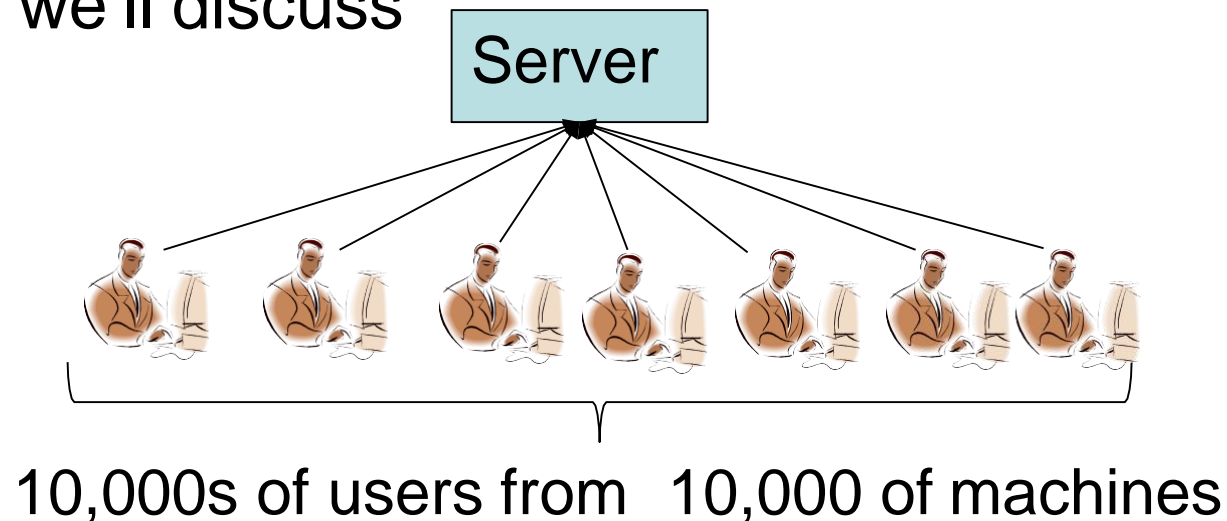
ACID vs BASE

- Relational databases support strong “consistency”
 - ACID properties we saw earlier
- NoSQL databases generally support weak “consistency”: BASE
 - Basically Available (BA): prioritize available over consistency
 - Soft state (S): Stores don't have to be write consistent, replicas don't have to be mutually consistent
 - Eventually Consistent (E): reads might be inconsistent for a long time
 - Inspired the famous CAP theorem, aggressively pursued by the famous Amazon DynamoDB.. More on this later
- But, we first need to understand this new notion of “consistency”
 - Lets start with distributed file systems first before we formalize consistency

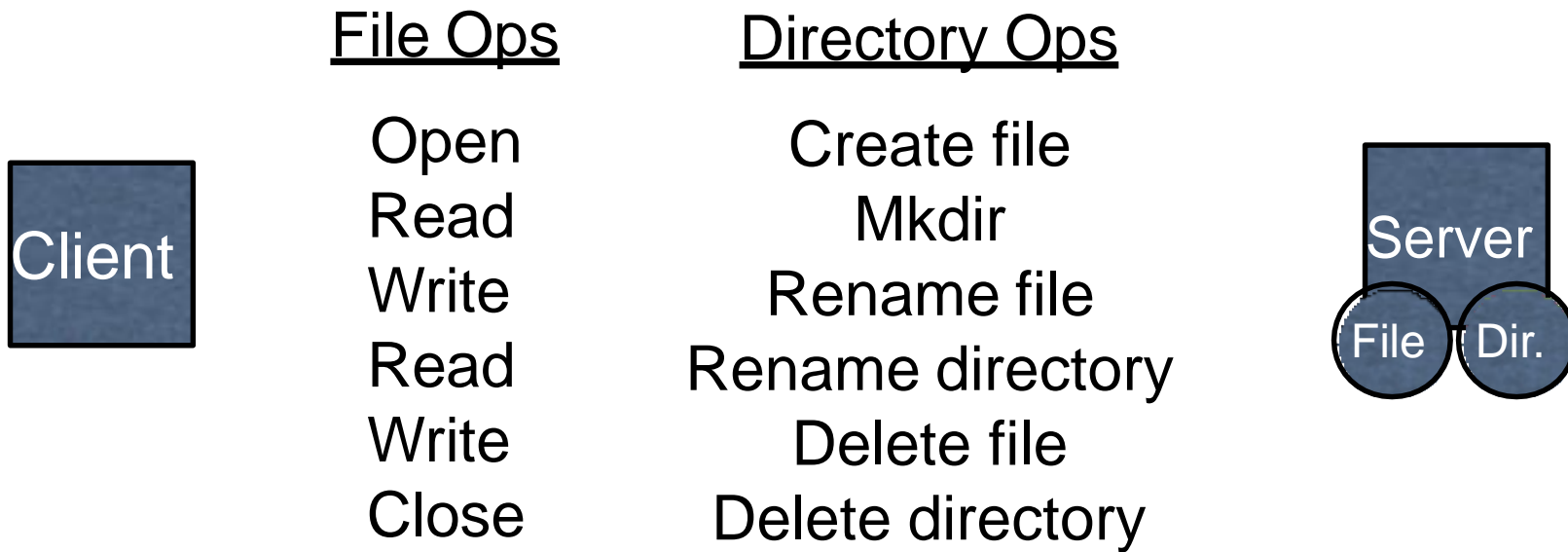
NFS Overview



- 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



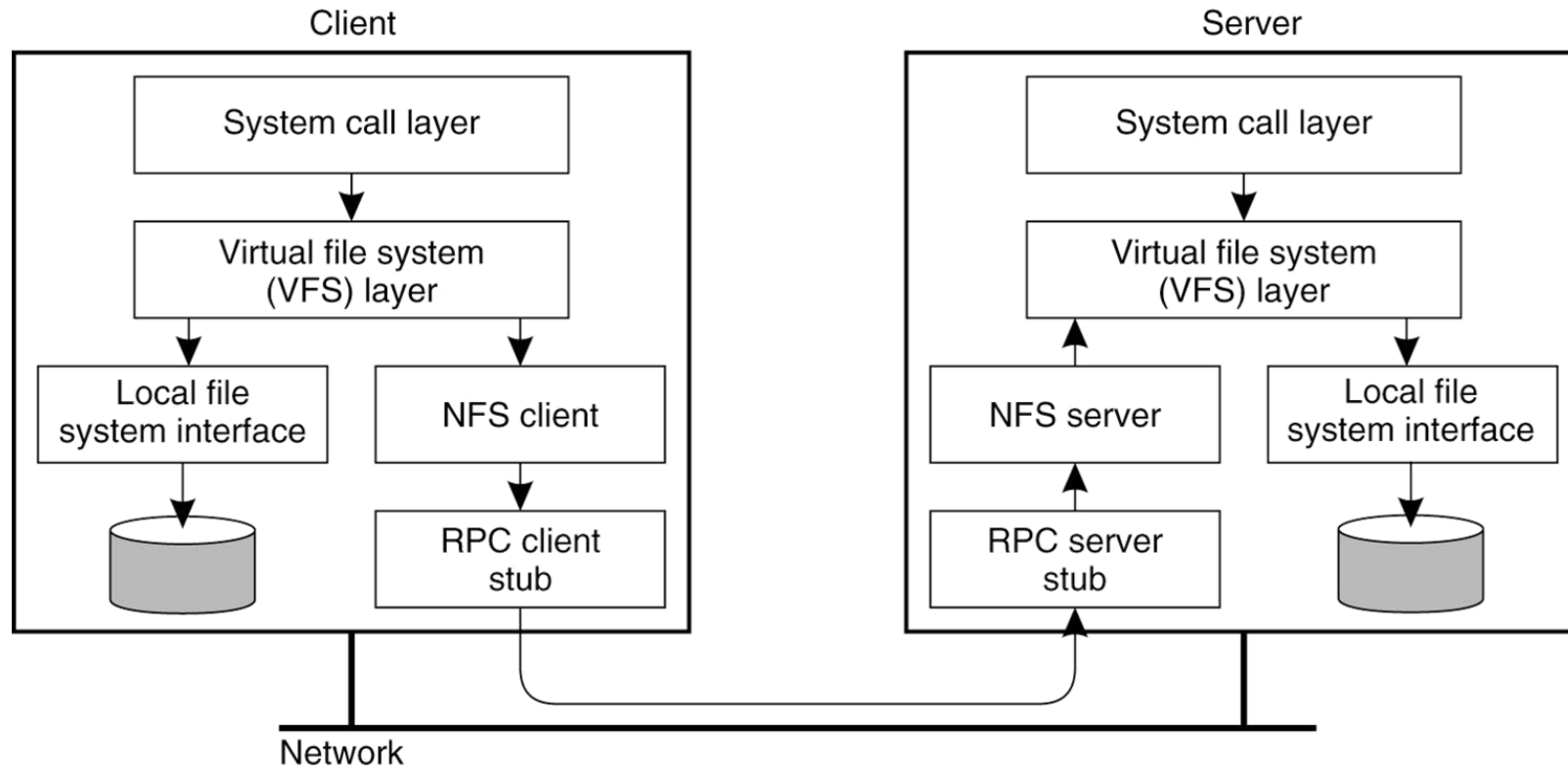
The FS Interface



Components in a DFS Implementation

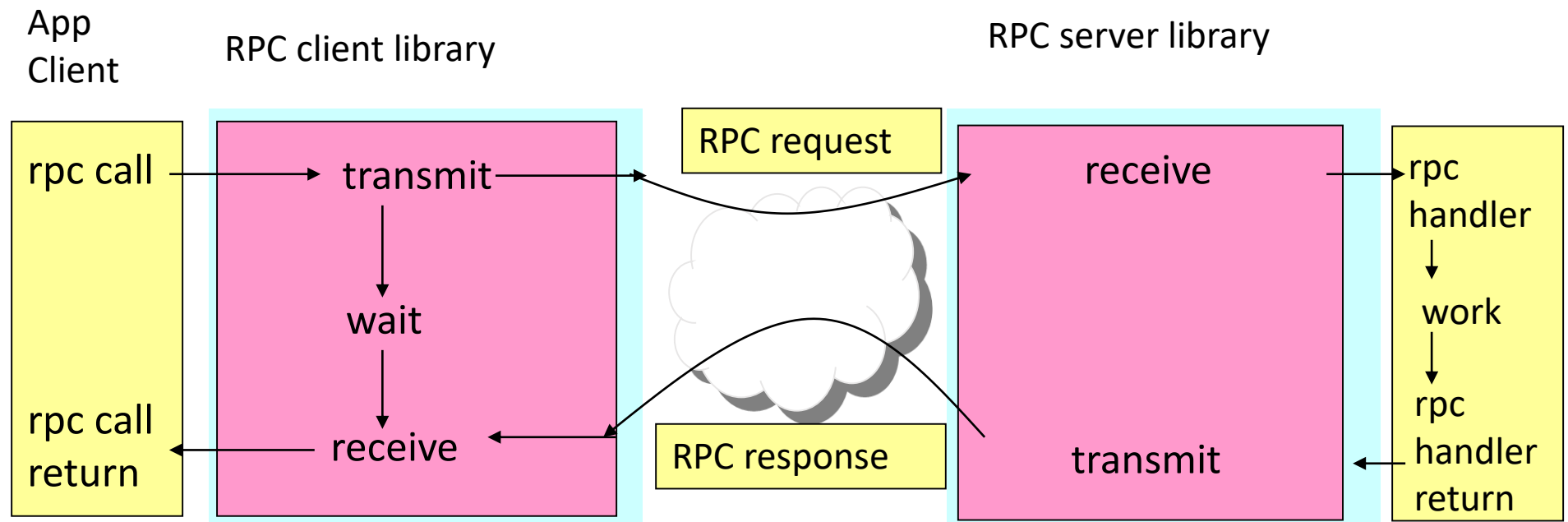
- Client side:
 - What has to happen to enable applications to access a remote file the same way a local file is accessed?
 - Accessing remote files in the same way as accessing local files → kernel support
- Communication layer:
 - Just TCP/IP or a protocol at a higher level of abstraction?
- Server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

NFS: Client—server architecture



RPC Overview

- RPC goal: make client/server communication look like local procedure calls
 - Servers **export** their local procedure APIs
 - On client, library generates requests over network to server
 - On server, called procedure executes, result returned in RPC response to client

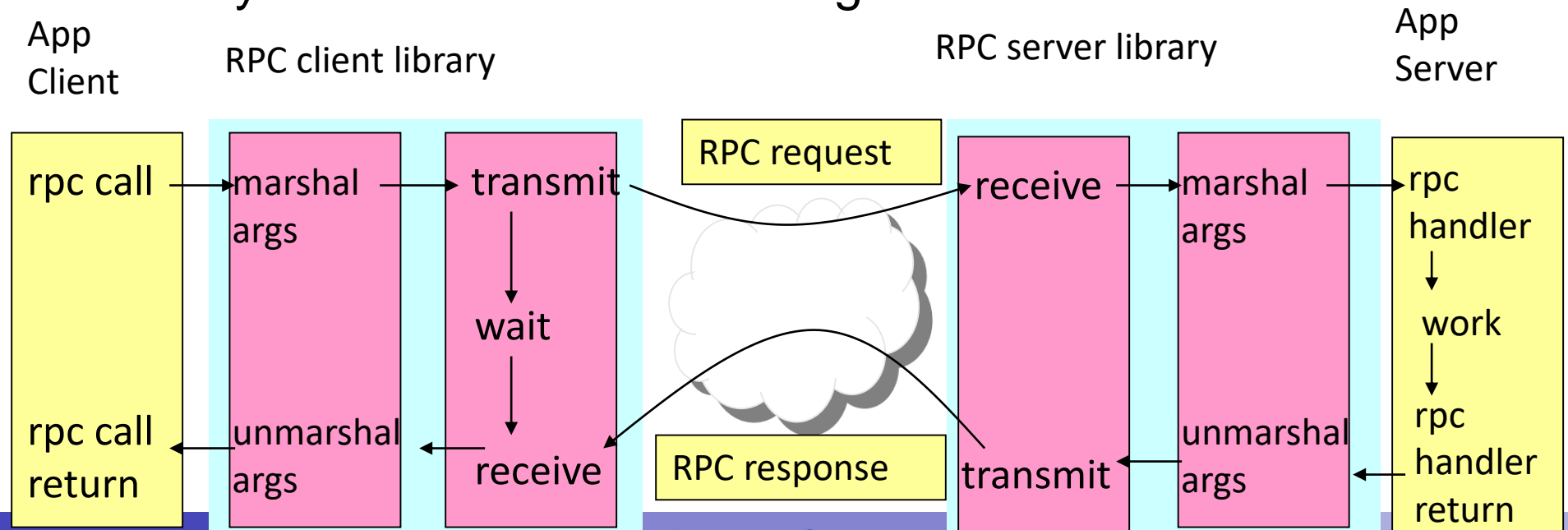


Key RPC Challenges

- 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
- 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**
 - 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 (track RPCs using IDs)
 -

Marshaling & Data Encoding

- 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, ...
 - Must convert to local representation of data (structs, strings, etc.)
- RPC fundamentally relies on a data encoding mechanism



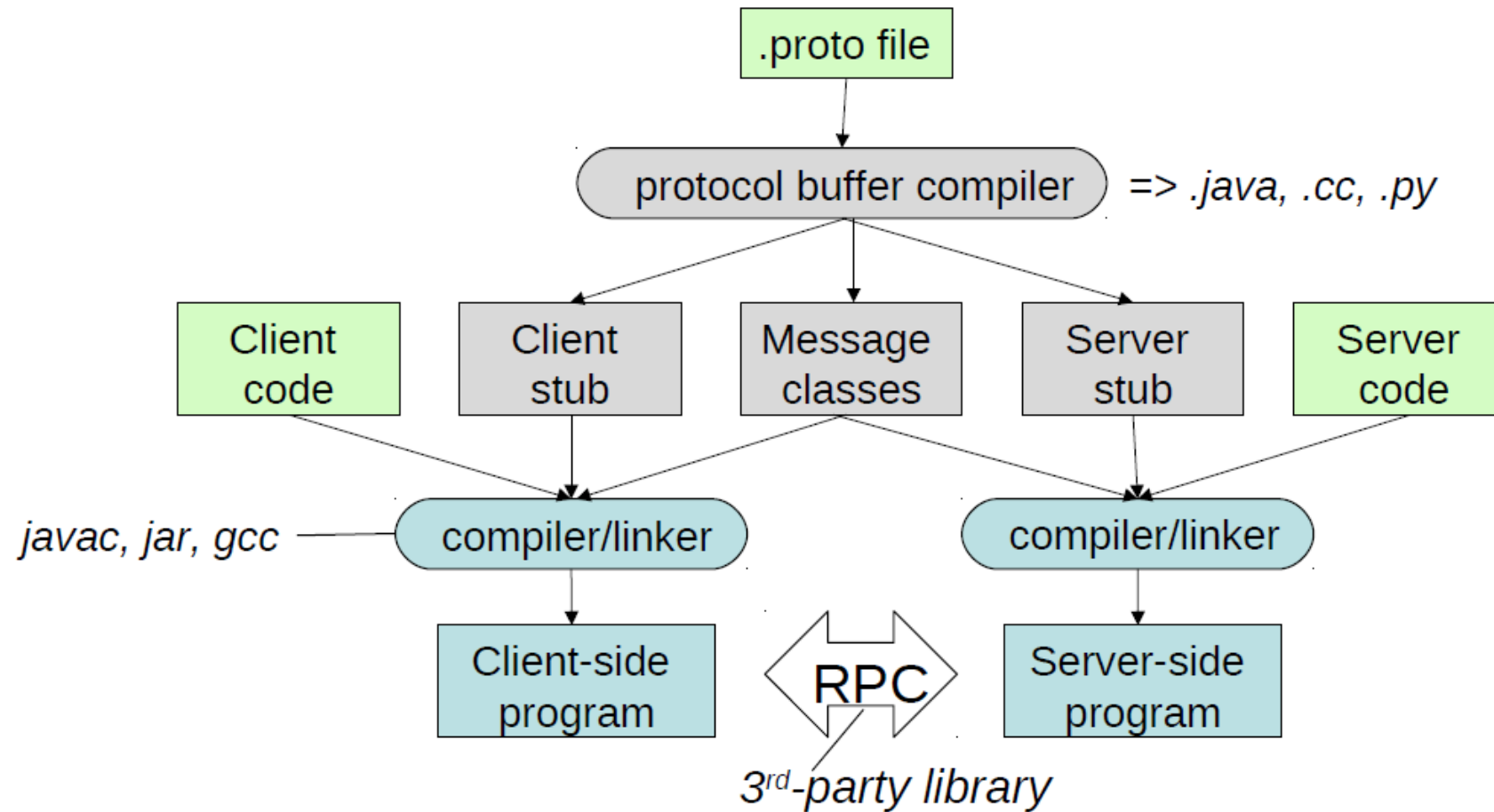
Popular RPC technologies

- 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);
    }
}
```

Protobuf Encoding Example

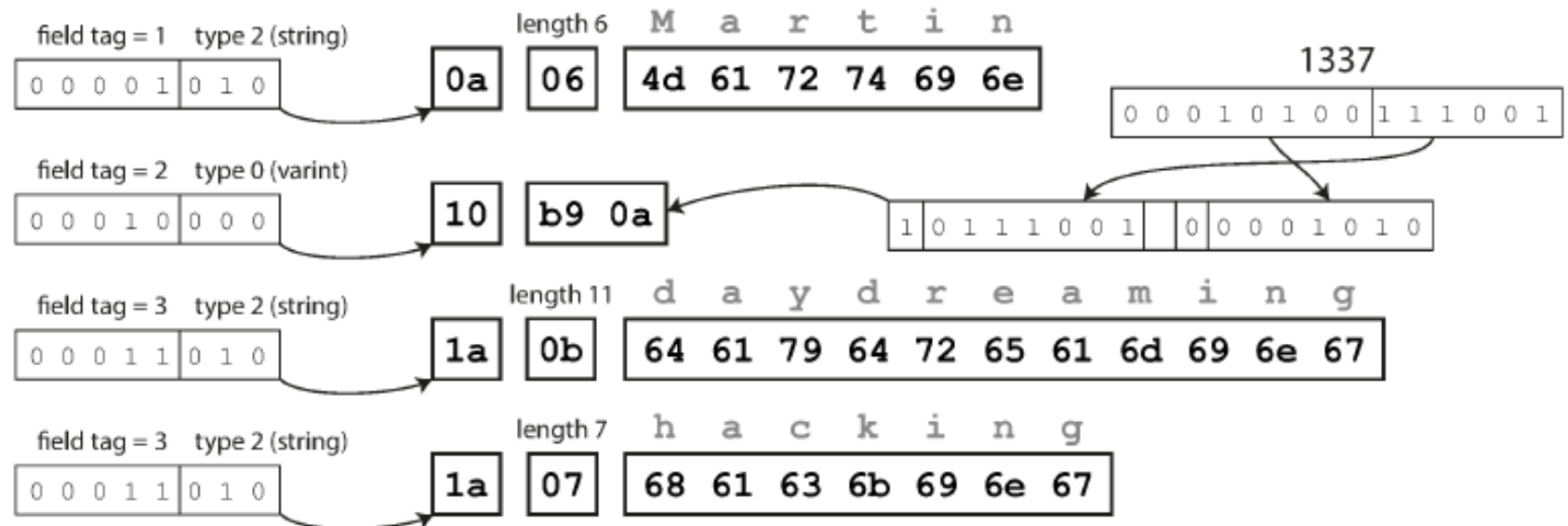
```
{  
  "userName": "Martin",  
  "favoriteNumber": 1337,  
  "interests": ["daydreaming", "hacking"]  
}
```

Protocol Buffers

Byte sequence (33 bytes):

0a	06	4d	61	72	74	69	6e	10	b9	0a	1a	0b	64	61	79	64	72	65	61
6d 69 6e 67				1a		07		68 61 63 6b 69 6e 67											

Breakdown:

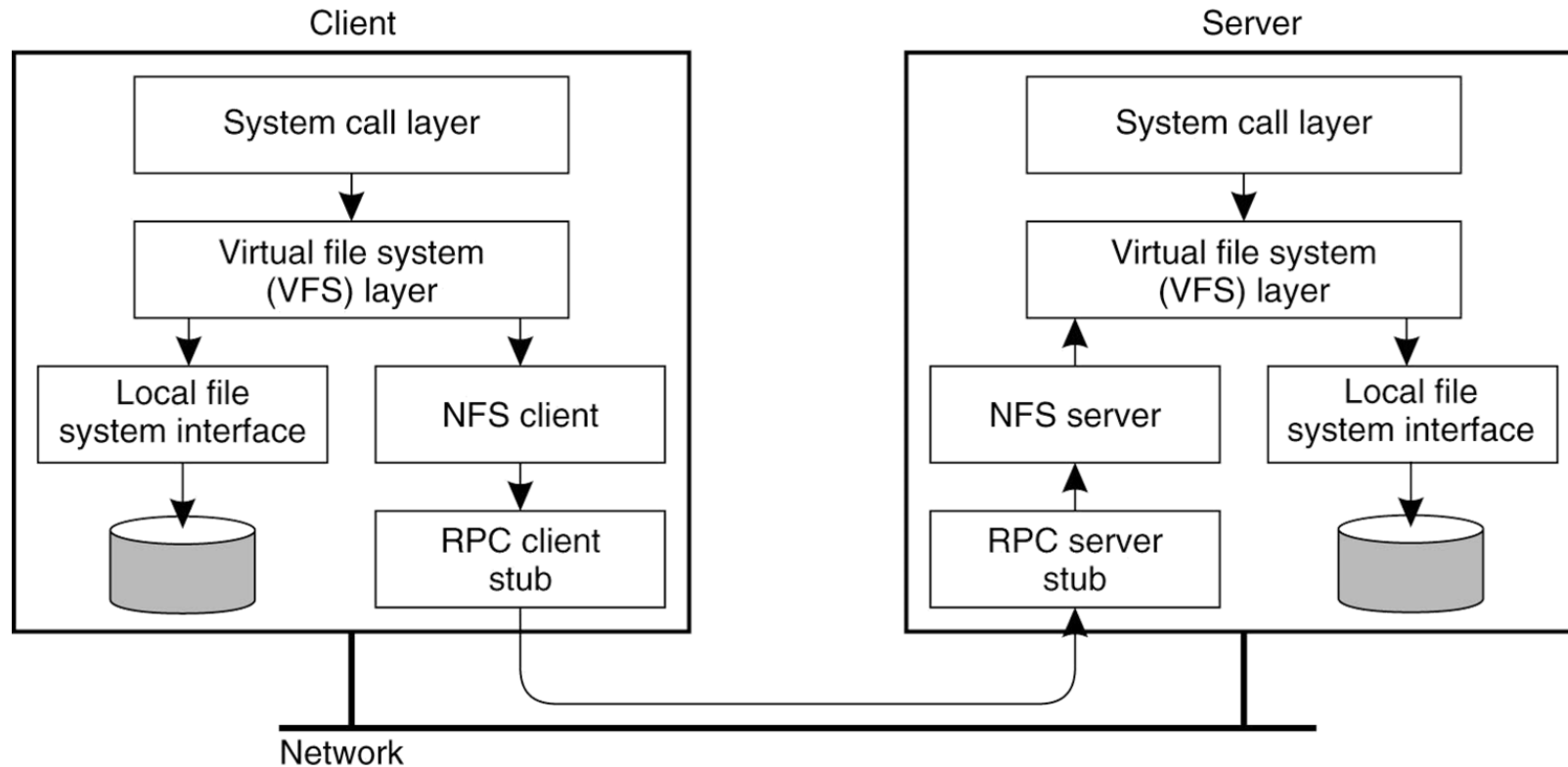


gRPC

- 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



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

Naïve FS Design

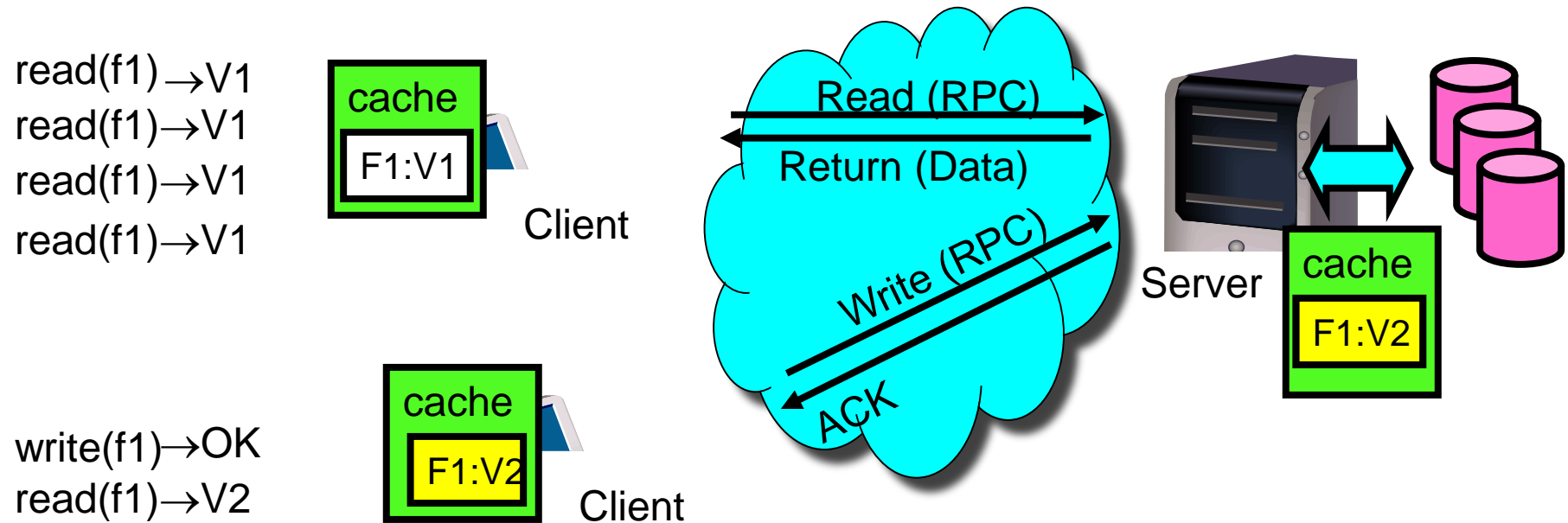
- Use RPC to forward *every* FS operation to the server
 - Server orders all accesses, performs them, and sends back result
- Good: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance will stink. Latency of access to remote server often much higher than to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

- 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
- Stateless protocol: requests specify exact state. `read()` → `read([position])`. no seek on server.

NFS's Failure Handling

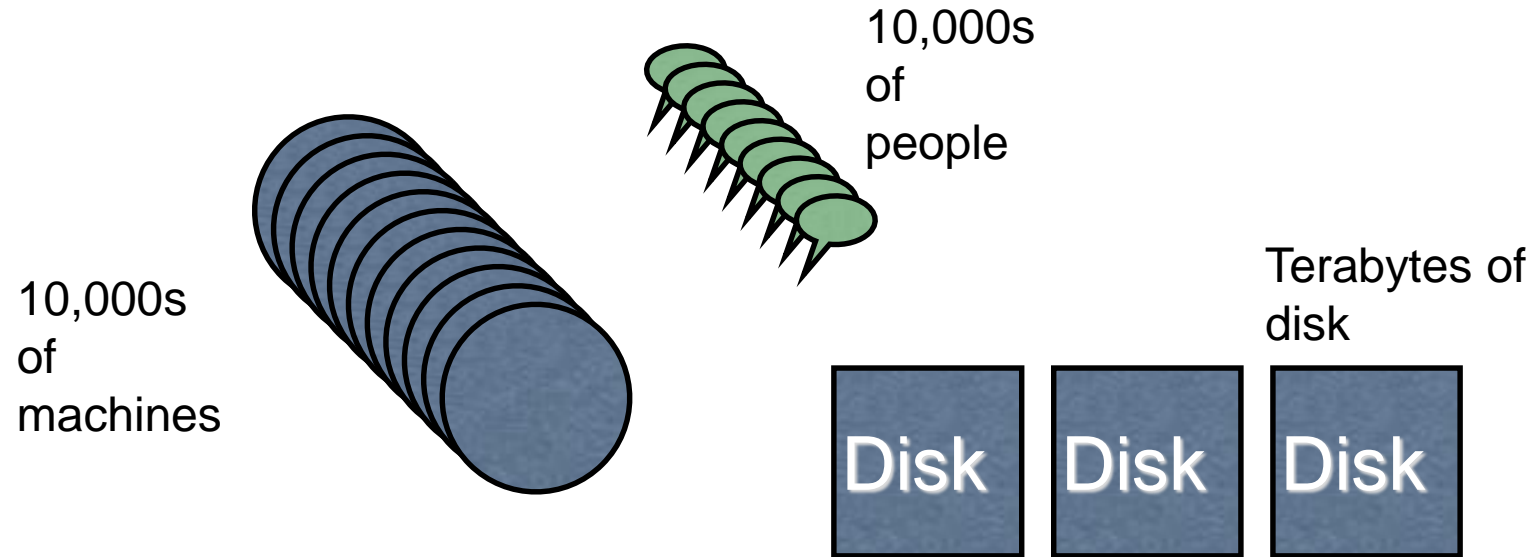
- Operations are idempotent
 - How can we ensure this? Unique IDs on files/directories. It's not `delete("foo")`, it's `delete(1337f00f)`, where that ID won't be reused.
- Write-through caching: When file is closed, all modified blocks sent to server. `close()` does not return until bytes safely stored.
 - 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
- 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
- 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)
 - 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 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

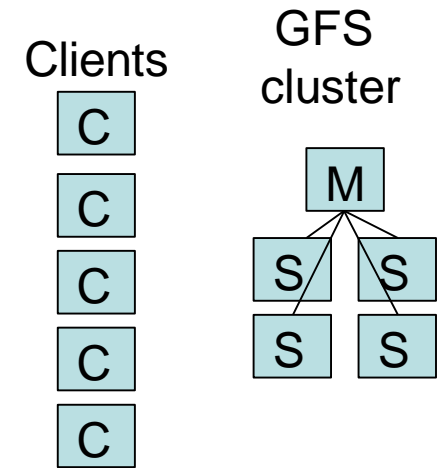
- Files are huge by traditional standards
 - Multi-GB files are common
- Most file updates are appends
 - Random writes are practically nonexistent
 - Many files are written once, and read sequentially
- High bandwidth is more important than latency lots of concurrent data accessing
 - E.g., multiple crawler workers updating the index file

GFS' design is geared toward apps' characteristics

Google apps have been geared toward GFS

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)



Master coordinates chunk servers

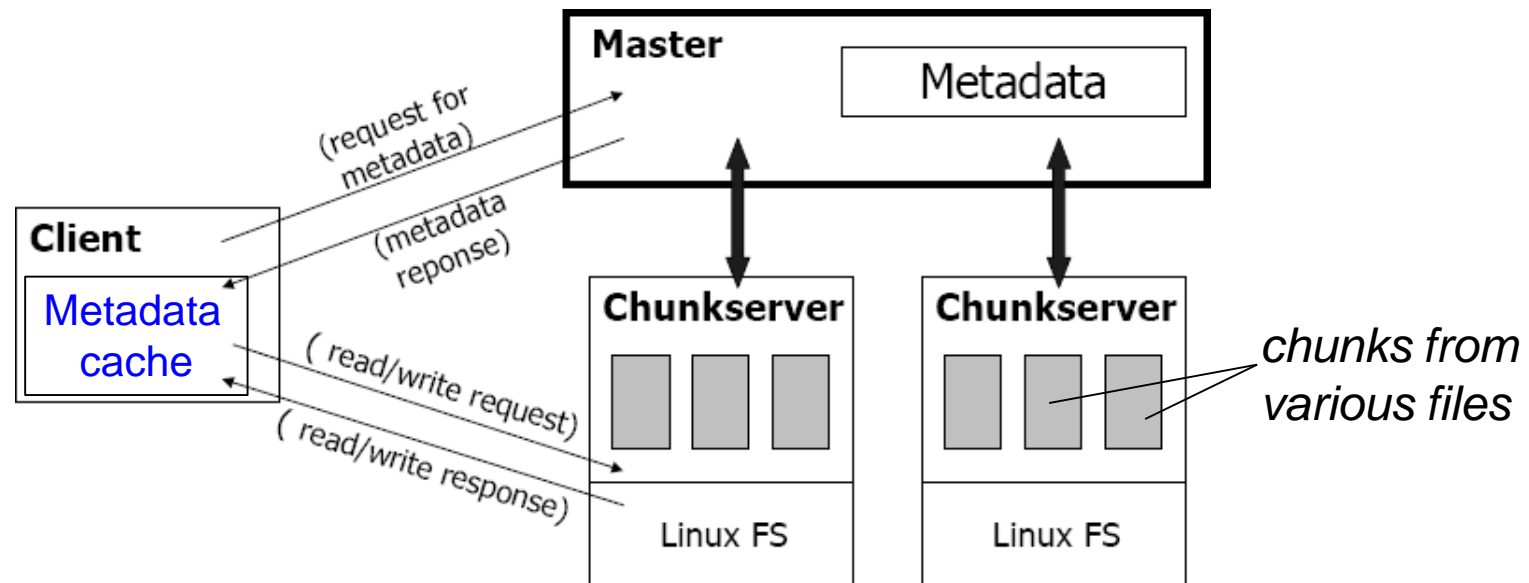
- Master and chunkserver communicate regularly (**heartbeat**):
 - Is chunkserver down?
 - Are there disk failures on chunkserver?
 - Are any replicas corrupted?
 - Which chunks does chunkserver store?
- Master sends **instructions** to chunkserver:
 - Delete a chunk
 - Create new chunk
 - Replicate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

GFS Interface

- Not POSIX compliant
 - Supports only few popular FS operations (read,write,...)
- Special 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

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



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: Formalizing consistency