slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems
Spring 2019

Christopher Godley

PhD Student

Department of Computer Science

University of Colorado Boulder

Distributed File System

BLOCKTECH in FINANCIAL SERVICES VIRTUALscape

by William Mougayar

APPLICATIONS & SOLUTIONS

Brokerage
coinbase, BITPagos, Unocoin, BTCC, BITFINEX, CIRCLE, COINJAR, QUADRIGACX, bitFlyer, safello, volabit, coinfloor, coins.ph

Exchanges
BTer.com, coinbase, kraken, HUOBI.com, BITSTAMP, POLONIEX, BTC-e, bitcoin.de, GEMINI, mexbt, CAMP BX, BITSO, Coinffeine, PAYMIUM, CEX.IO, BitOasis, SHAPESHIFT, BTC express, coinsecure, coinsetter, Bits of Gold

Soft Wallets
BLOCKCHAIN, airBitz, coinbase, ARMORY, ELECTRUM, xapo, bread wallet, Coinkite, Mycelium, MultiBit HD, coinprism

Hard Wallets
TREZOR, case, Ledger Wallet, keepkey

Microtransactions
BitMesh, BitWall, ChangeTip, ProTip, Strawpay

Investments
Grayscale, magnr, Loanbase, string, Yuanbao, KOIBANX, Bitbond, WeiFund, WEALTHCOIN, lighthouse, BSAVE.IO, dangpu.com, BTCjam, CHROMA.FUND

Merchants
bitpay, Bitnet, Coinkite, Coinify, PEY, CoinPayments, coinsnap, coinbase, CoinSimple, BITPagos

Compliance
ELLIPTIC, thirdkey solutions, PROTUS, CHAINALYSIS, Sig3, BLOCKSEER, CryptoCorp, IdentityMind GLOBAL, Tradle, vogogo, COINALYTICS, BLOCKVERIFY, Merkle Tree

Trading Platforms
COINIGY, HEDGY, OrderBook, tradewave, COINUT, AltOptions, MAKER, BITNOMIAL, TERA EXCHANGE, BitMEX, Mirror, CRYEX, 1Broker, TABTRADER, DXmarkets, AlphaPoint, NOBLE MARKETS, HitFin

Capital Markets
Chain, symbiont, NASDAQ Private Market, Digital Asset Holdings, clearmatics, itBit, TradeBlock, t0, epiphyte

Money Services
CRYPTOPAY, cashila, ABRA, Fuzo, tether, BitwalaBTC, coins.ph, Simplex, ATLAS, BITX, coinx, REBIT, uphold, SecuraCoin, DUO MONEY, BITNEXO, CoinPip, LocalBitcoins.com, BitPesa, BlinkTrade, COINAPULT, MELOTIC, Glidera, bridge21

Financial Data
bitcoinity, CoinMarketCap, CryptoCoin, BRAVE NEW COIN, BlockJockey, CRYPTRADER, BitcoinWisdom, TradeBlock, CoinGecko, Coinhills

Payments
Align Commerce, GOCOIN, About Payments, BLADE, GAZEBO.IO, GemPay, cuber, SETL.io, safecash

ATMs
LocalBitcoins.com, Robocoin, bitxatm, bitaccess, Project Skyhook, btcpoint, SERY, LAMASSU BITCOIN VENTURES, GB, COINOUTLET, genesiscoin, Modenero Concierge

Trade Finance
GAZEBO.IO, everledger, CHRONICLED, WAVE, digix, skuchain, PROVENANCE, thingchain

Payroll & Insurance
paybits, bitWAGE, DYNAMIS

Banks
BBVA, UBS, LHV, London Stock Exchange, secco, BNY MELLON, BARCLAYS, fidor BANK, citibank, moni

MIDDLEWARE & SERVICES

Services
CRYPTONOMEX, B9, CONSENSYS, SolidX, appliedblockchain, RUBIX

Software Development
chainscript, HydraChain, Blockstack.io, PEERNOVA, openchain, CREDITS, eris INDUSTRIES, MultiChain, Manifold Technology, Blockstream

General APIs
BitGo, neuroware, coinbase, bitcore, Gem, BLOCKCYPHER, Coinkite

Special APIs
TIERION, Open Assets, bitbind.io, COLOREDCOINS, colu, factom, ChromaWay

Platforms
Counterparty, Monetas, Omni, Mercoin, blockstack, HYPERLEDGER, Tendermint, BLOCKAPPS, appliedblockchain

Smart Contracts
SmartContract, ETH BaaS, CoinSpark, ROOTSTOCK, bitShares, Tembusu Systems

INFRASTRUCTURE & BASE PROTOCOLS

Public
bitcoin, bitShares, ethereum

Special
ripple, stellar

Payment
Amiko Pay, MONERO, Lightning Network

Miners
ANTPOOL, BitFury, 21 INC, BTCC, BITCOINCZ

Source: http://startupmanagement.org/blog

Distributed Systems

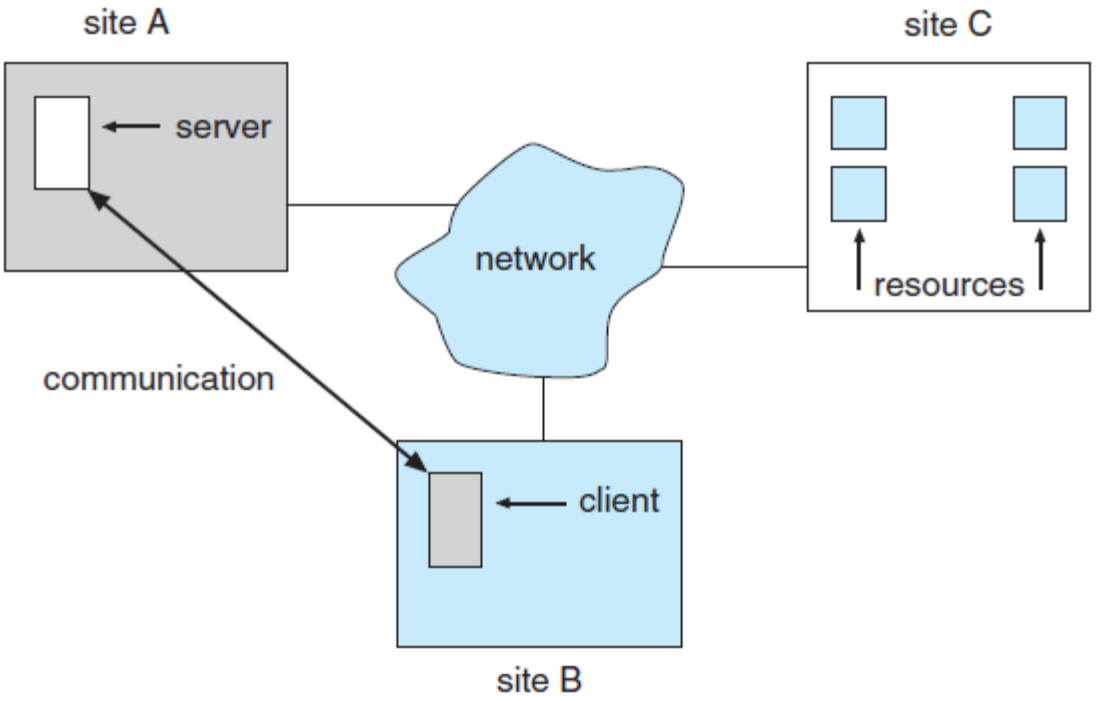
- …are widely used!
- We need (want?) to share physical devices (e.g., printers) and information (e.g., files)
- Many applications are distributed in nature (e.g., ATM machines, Airline reservations, Entertainment contents,)
- **Many large problems can be solved by decomposing into smaller problems that run in parallel (e.g., MapReduce)**
- Next slides: go over three distributed file system models
  - **Client-server**
  - **Peer-to-peer**
  - **Cloud(cluster) computing**

Distributed System

- A distributed system is a collection of processors that do not share memory or a clock.
- Each node has its own local memory
- The nodes communicate with one another through various networks
- Applications of distributed systems range from providing transparent access to files inside an organization, to large-scale cloud file and photo storage services, to business analysis of trends

Distributed System

- Advantages of Distributed System
  - Resource Sharing
  - Computation Speedup
  - Reliability



A client-server distributed system

Distributed File Systems (DFS)

- To explain the structure of DFS, we need to define the terms *service*, *server*, and *client*
  - A *service* is a software entity running on one or more machines and provide particular type of function to clients
  - A *server* is the service running on a single machine
  - A *client* is a process that can invoke a service using a set of operation that form its client interface.

- **A file system provides file services to clients**. A client interface for a file service is formed by a set of primitive file operations such as (1) create a file, (2) delete a file, (3) read from a file, and (4) write to a file

- So, the primary component that a file server is **controls and access a set of local secondary-storage devices** (i.e., hard disks or solid-state drives).

Distributed File Systems (DFS)

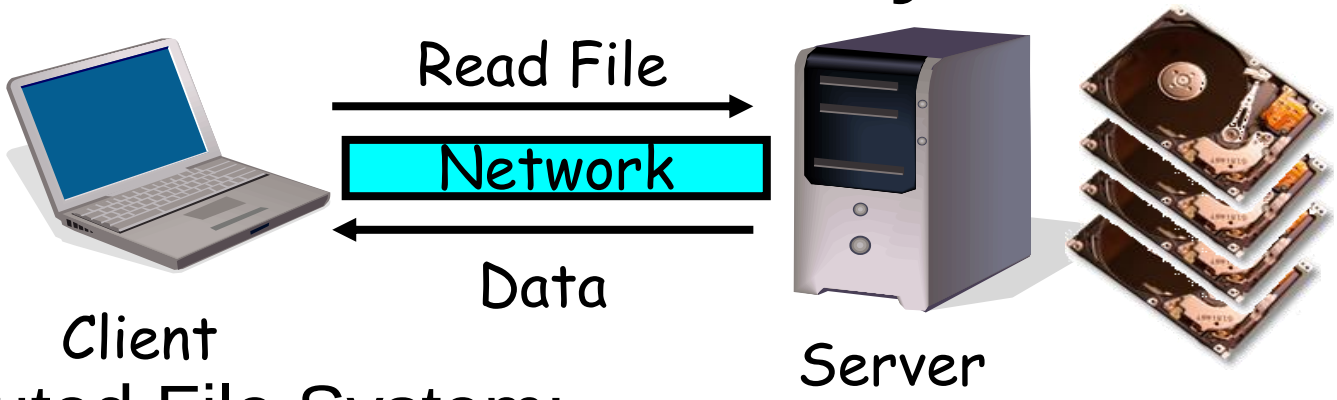
- DFS are multiplicity and autonomy of clients and servers in the system.
  - Ideally, though, a DFS should appear to its clients to be a conventional, centralized file system.
  - The client interface of a DFS should not distinguish between local and remote files
  - It is up to DFS to locate the files and to arrange for the transport of the data

Distributed File Systems (DFS)

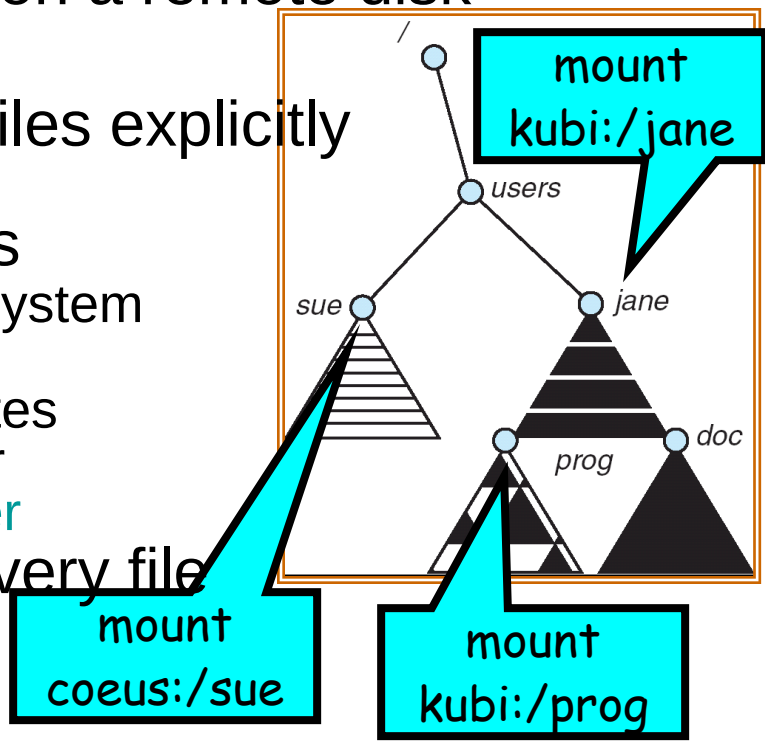
- **Response Time:** The most important performance measure of a DFS is the amount of time needed to satisfy service requests
  - In conventional systems, this consists of storage-access time and CPU-processing time (small amount)
  - In DFS, **the additional overhead** is.
  - This overhead includes
    - (1) the time to deliver the request to server
    - (2) the time to get the response across the network back to the client

  In other words, ideal DFS performance would be comparable to that of a conventional file system.

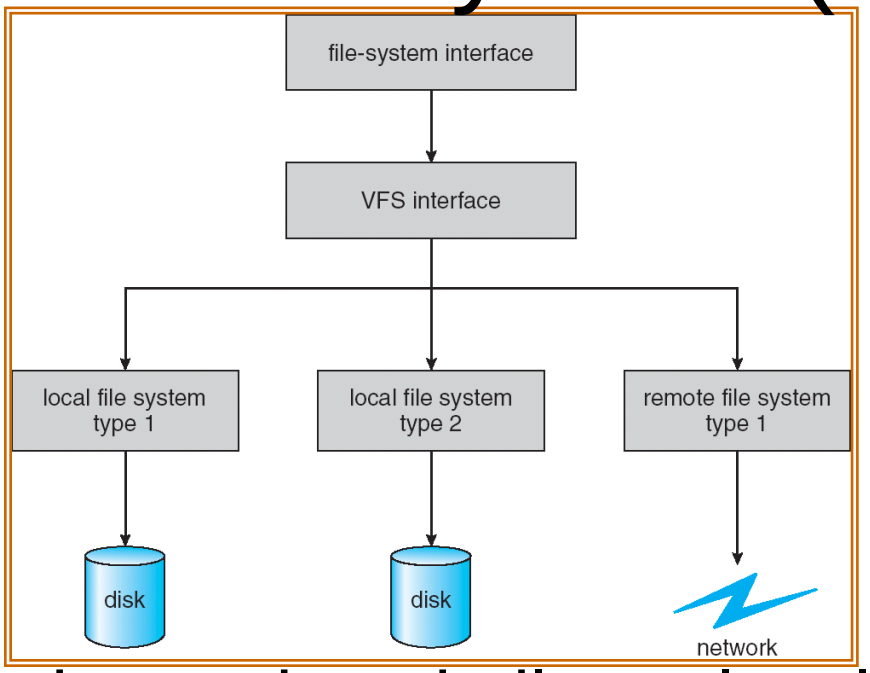
Distributed File Systems



- Distributed File System:
  - Transparent access to files stored on a remote disk
- Naming options: :
  - *(1) Hostname:*local_name: Name files explicitly
    - No location or migration transparency
  - *(2) Mounting* of remote file systems
    - System manager mounts remote file system by giving name and local mount point
    - Transparent to user: all reads and writes look like local reads and writes to user e.g. /users/sue/foo→/sue/foo on server
  - *(3) A single, global name space:* every file in the world has unique name
    - Location Transparency: servers can change and files can move without involving user

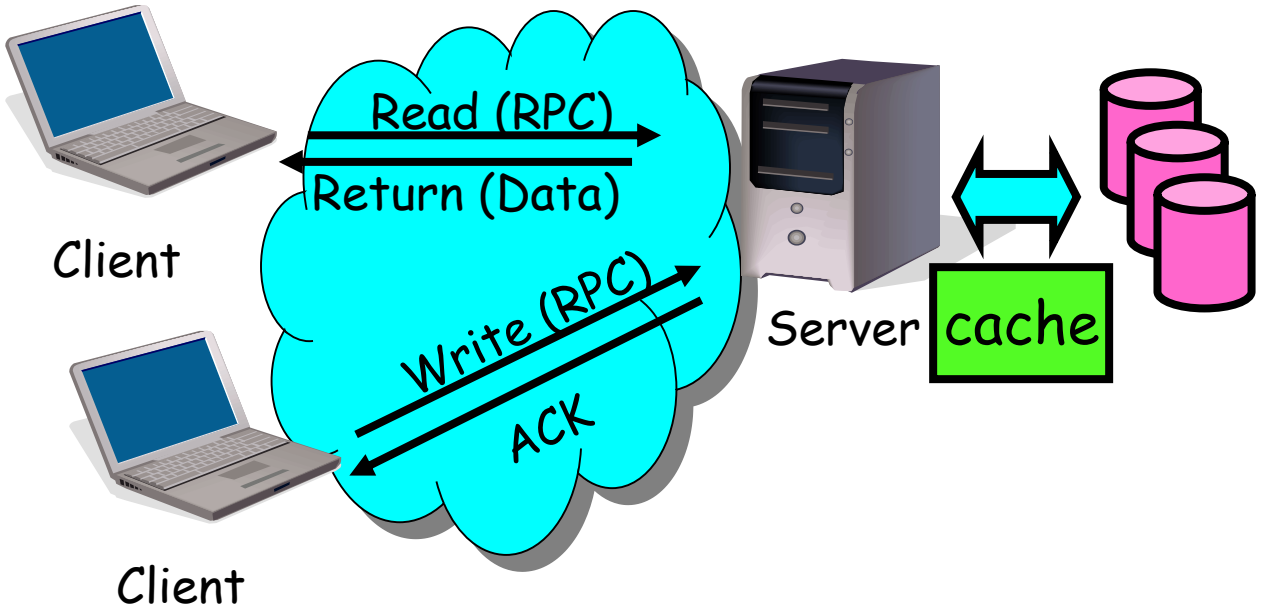


Virtual File System (VFS)



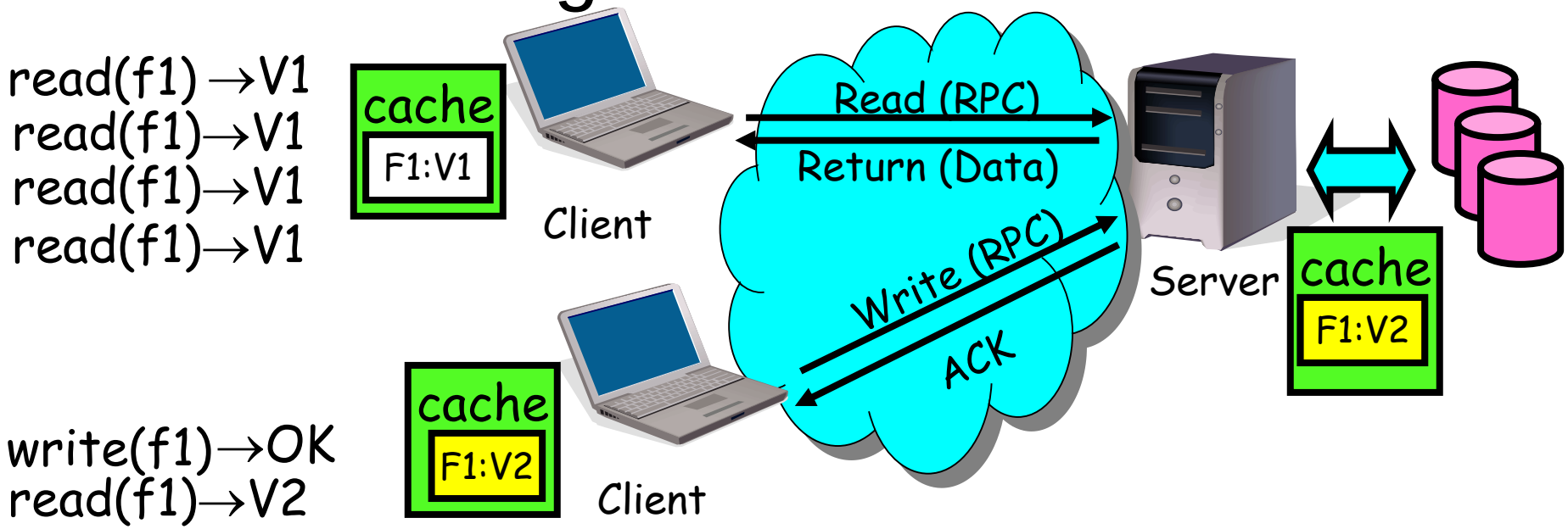
- VFS: Virtual abstraction similar to local file system
  - Instead of “inodes” has “vnodes”
  - Compatible with a variety of local and remote file systems
    - provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Simple Distributed File System



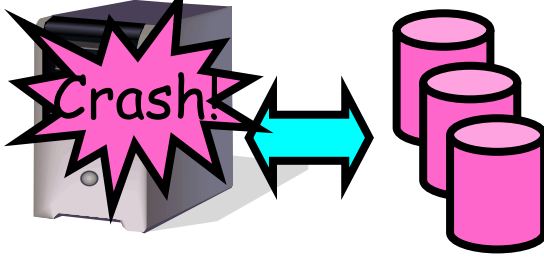
- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching/can be caching at server-side
- Advantage:
  - Server provides completely consistent view of file system to multiple clients
  - Can have “thin client”
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

Use of caching to reduce network load



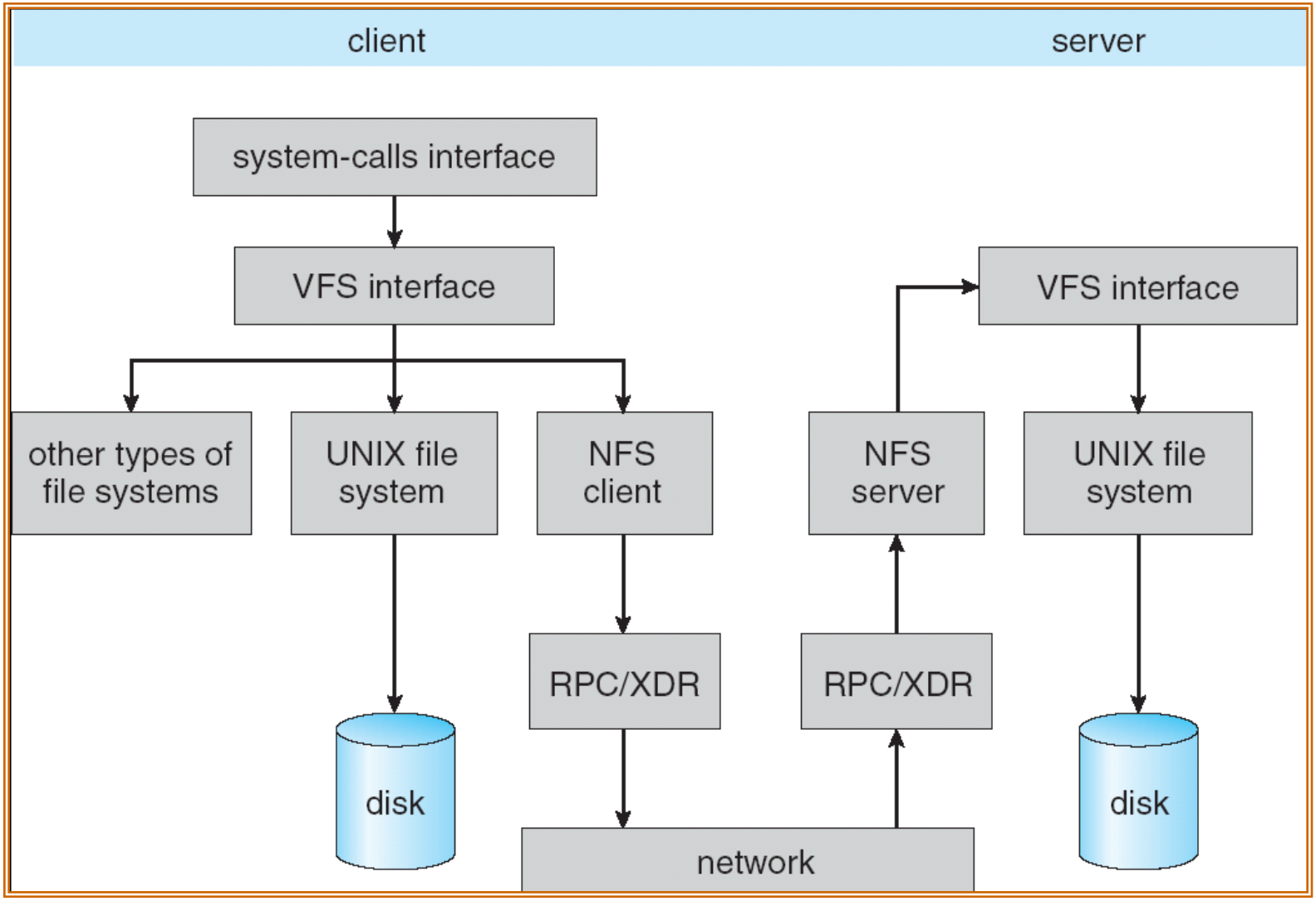
- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - Client caches have data not committed at server
  - Cache consistency!
    - Client caches not consistent with server/each other

Failures



- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does “read”, it will fail
  - Message retries: suppose server crashes after it does UNIX “rm foo”, but before acknowledgment?
    - Message system will retry: send it again
    - How does it know not to delete it again? (could solve with two-phase commit protocol, but NFS takes a more ad hoc approach)
- Stateless protocol: A protocol in which all information required to process a request is passed with request
  - Server keeps no state about client, except as hints to help improve performance (e.g. a cache)
  - Thus, if server crashes and restarted, requests can continue where left off (in many cases)
- What if client crashes?
  - Might lose modified data in client cache

Schematic View of NFS



Network File System (NFS)

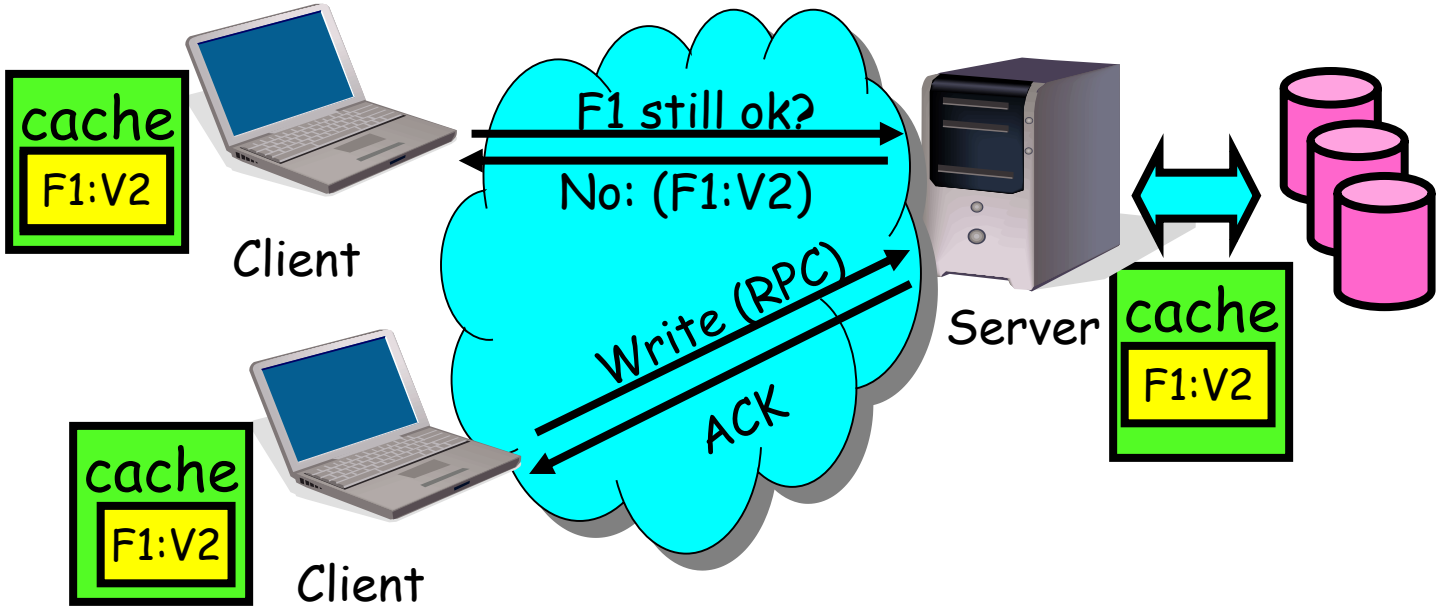
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes!

NFS Continued

- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: remove a file: NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - Hang until server comes back up (next week?)
    - Return an error. (Of course, most applications don't know they are talking over network)

NFS Cache consistency

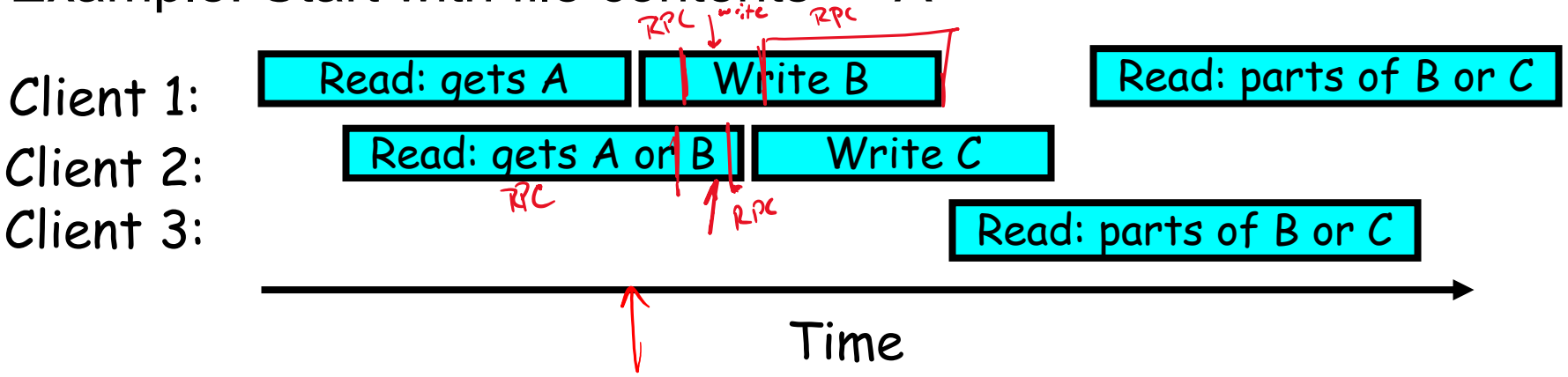
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - In NFS, can get either version (or parts of both)
    - Completely arbitrary!

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, another CPU reads file before it's done, ?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - 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:
    - E.g. if read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

NFS Pros and Cons

- NFS Pros:
  - Simple, Highly portable
- NFS Cons:
  - Sometimes inconsistent
  - Doesn’t scale to large # clients
    - Must keep checking to see if caches out of date

Andrew File System

- Andrew File System (AFS, late 80's)

- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - As a result, do not get partial writes: all or nothing!
    - Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

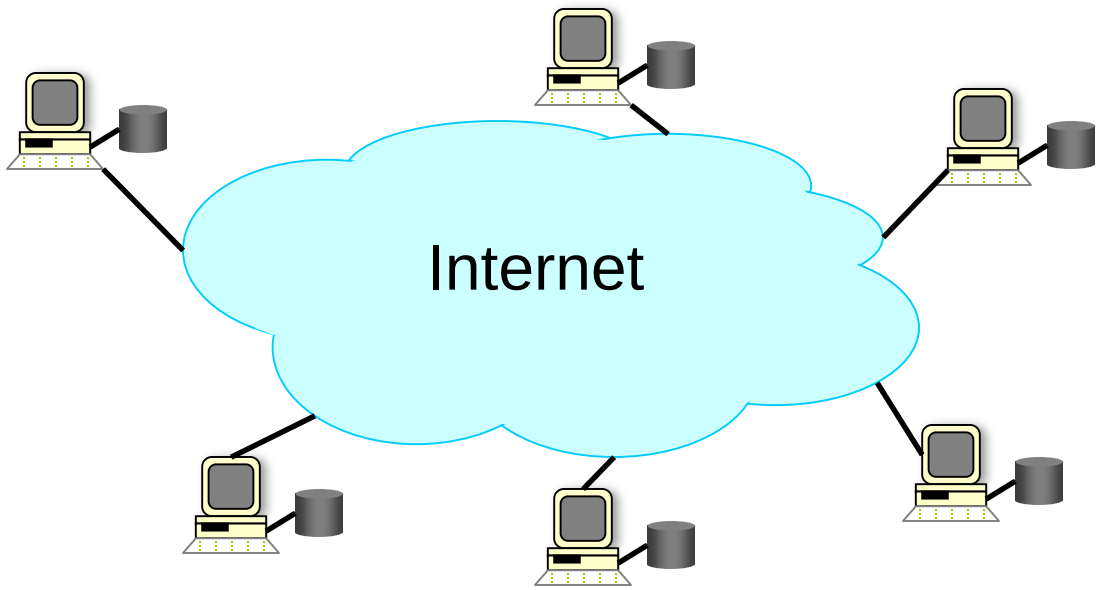
Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - Get file from server, set up callback with server
  - On write followed by close:
    - Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pros: Relative to NFS, less server load:
  - Disk as cache $\Rightarrow$ more files can be cached locally
  - Callbacks $\Rightarrow$ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes$\rightarrow$server, cache misses$\rightarrow$server
  - Availability: Server is single point of failure
  - Cost: Need servers - machine's high cost relative to normal workstation

Peer-to-Peer Distributed Systems

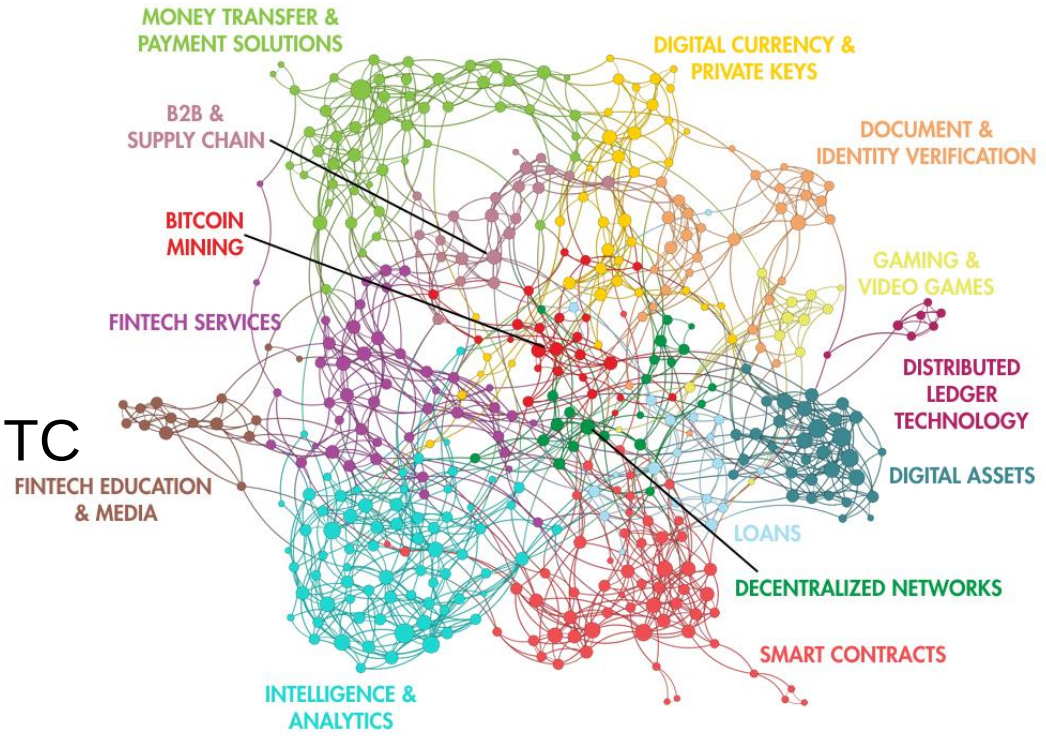
How Did it Start?

- A killer application: Naptser
  - Free music over the Internet
- Key idea: share the **storage *and* bandwidth** of individual users



What is it evolved to? Blockchain: 10,000 foot view

- New bitcoins are “created” every ~10 min, owned by “miner”
- Thereafter, just keep record of transfers
  - e.g., Alice pays Bob 1 BTC



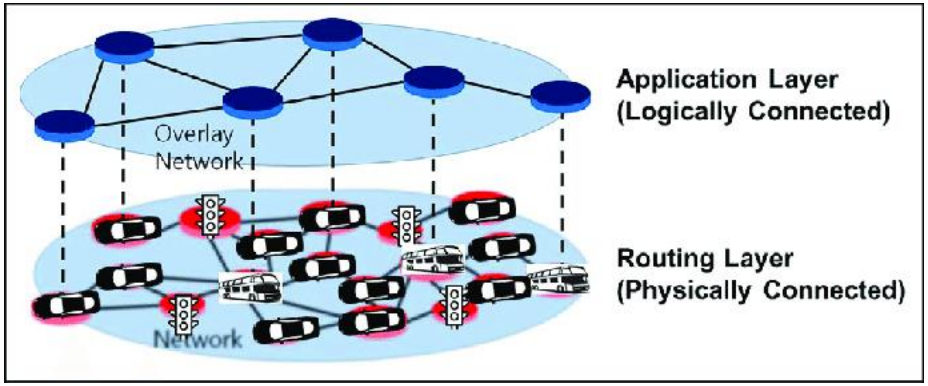
- Basic protocol:
  - Alice signs transaction: txn = $Sign_{Alice}$ (BTC, PK_{Bob})
  - Alice shows transaction to others…

What is it?

- A network of peers - without the need for central coordination instances.
- Used largely for sharing of content files such as audio, video, data or anything in a digital format.
- There are many p2p protocols such as Ares, Bittorrent, or eDonkey.
- Network can be very large
- Can also be used for business solutions - may not have resources available to implement a server solution.

Application Overlay Network

- P2P applications like Bittorrent create these overlay networks over the existing internet in order to perform indexing and peer collection functions.
- Have no control over physical networks.
- Weak resource coordination, as well as weak response to fairness of network resource sharing.

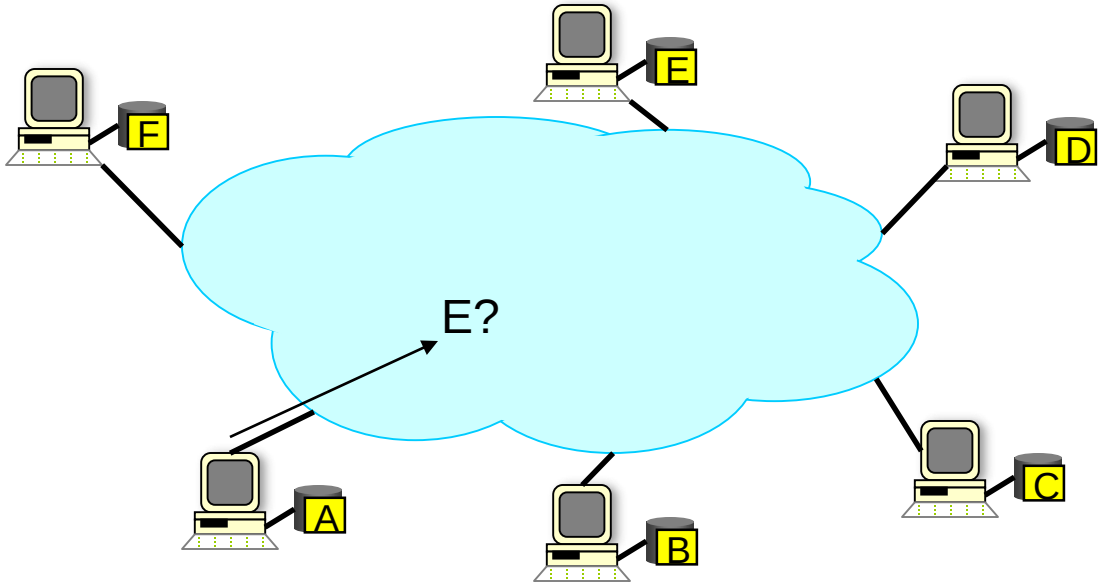


Peer-to-peer File System Model

- Each user stores a subset of files
- Each user has access (can download) files from other users in the system

Main Challenges

- **Find where a particular file is stored**



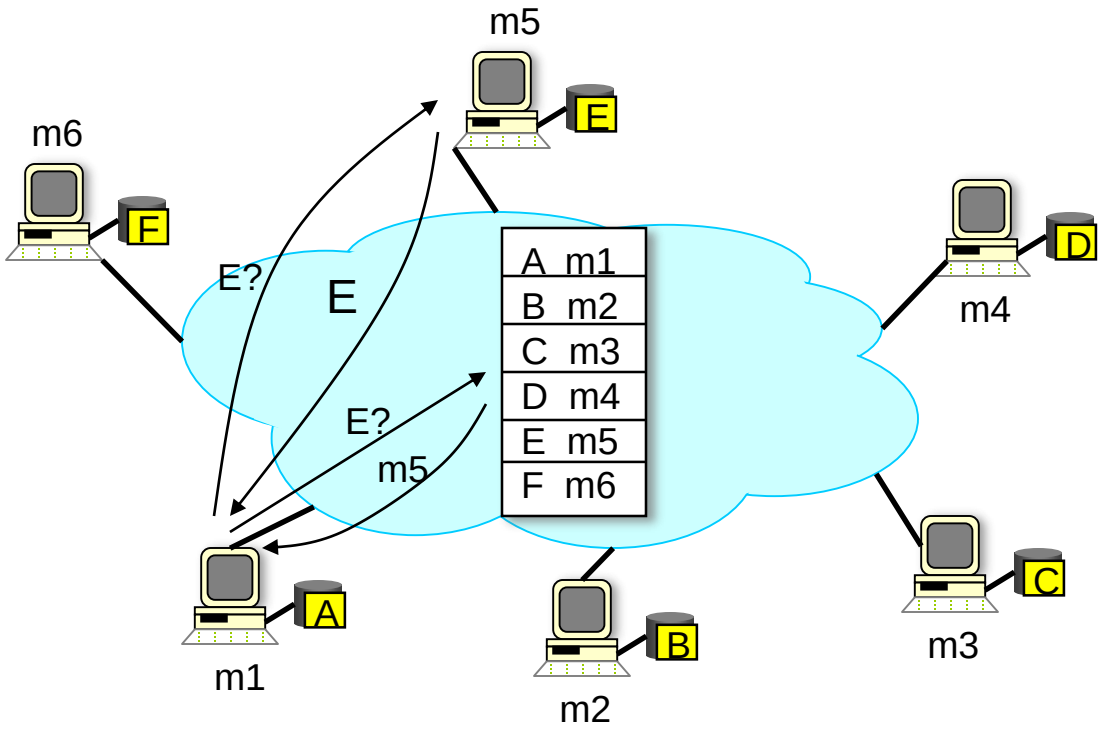
Other Challenges

- Scale: up to millions of machines
- Dynamicity: machines can come and go any time

Napster

- Assume a **centralized index system** that maps files (songs, shows, sw) to machines that are ***alive***
- How to find a file:
  - Query the index system → return a machine that stores the required file
    - Ideally this is the closest/least-loaded machine
  - ftp the file from that machine
- Advantages:
  - Simplicity, easy to implement sophisticated search engines on top of the index system
- Disadvantages:
  - Robustness, scalability (?)

Napster: Example

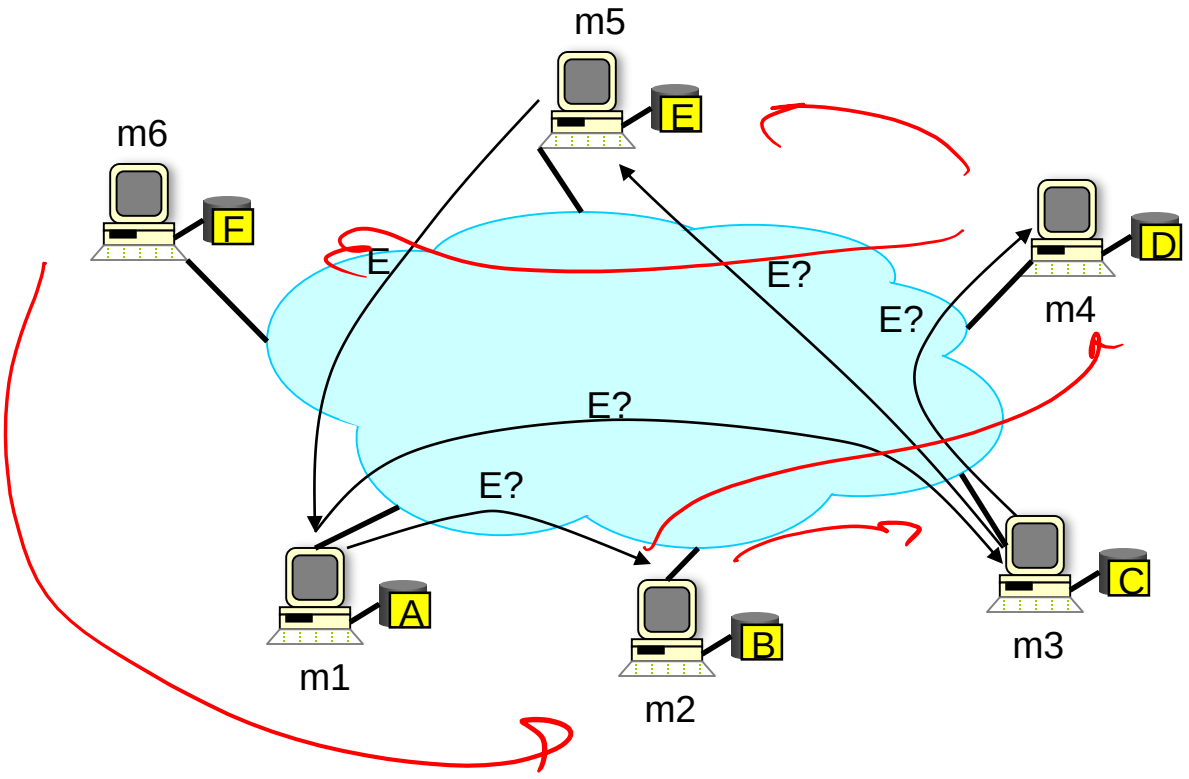


Gnutella

- Distribute file location
- Idea: broadcast the request
- How to find a file?
  - Send request to all neighbors
  - Neighbors recursively multicast the request
  - Eventually a machine that has the file receives the request, and it sends back the answer
- Advantages:
  - Totally decentralized, highly robust
- Disadvantages:
  - Not scalable; the entire network can be swamped with requests
    - When to stop relaying?
    - How to fix the problem?

Gnutella: Example

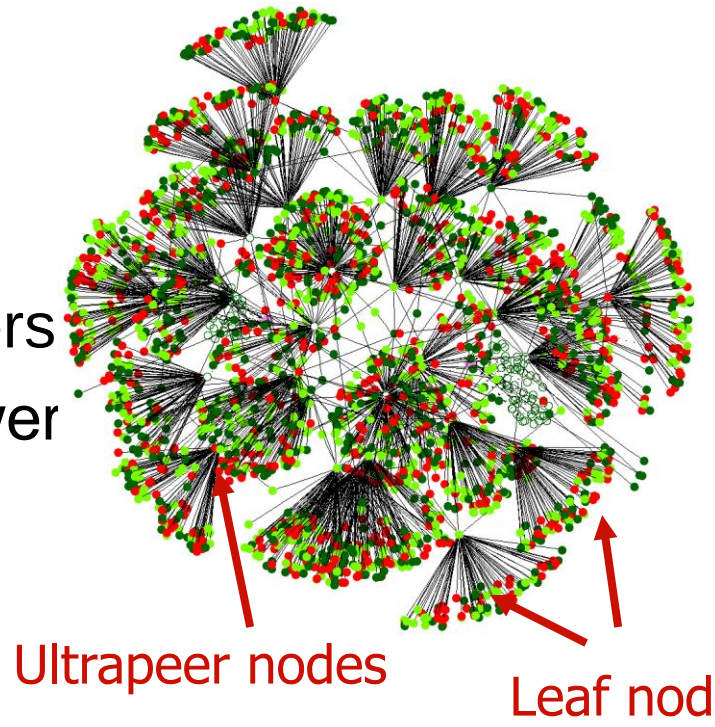
- Assume: m1's neighbors are m2 and m3; m3's neighbors are m4 and m5;…



Two-Level Hierarchy

- Gnutella implementation, KaZaa
- Leaf nodes are connected to a small number of ultra-peers (supernodes)
- Query
  - A leaf sends query to its ultra-peers
  - If ultra-peers don't know the answer they flood the query to other ultrapeers
- More scalable:
  - Flooding only among ultra-peers

Oct 2003 Crawl on Gnutella



Advantages

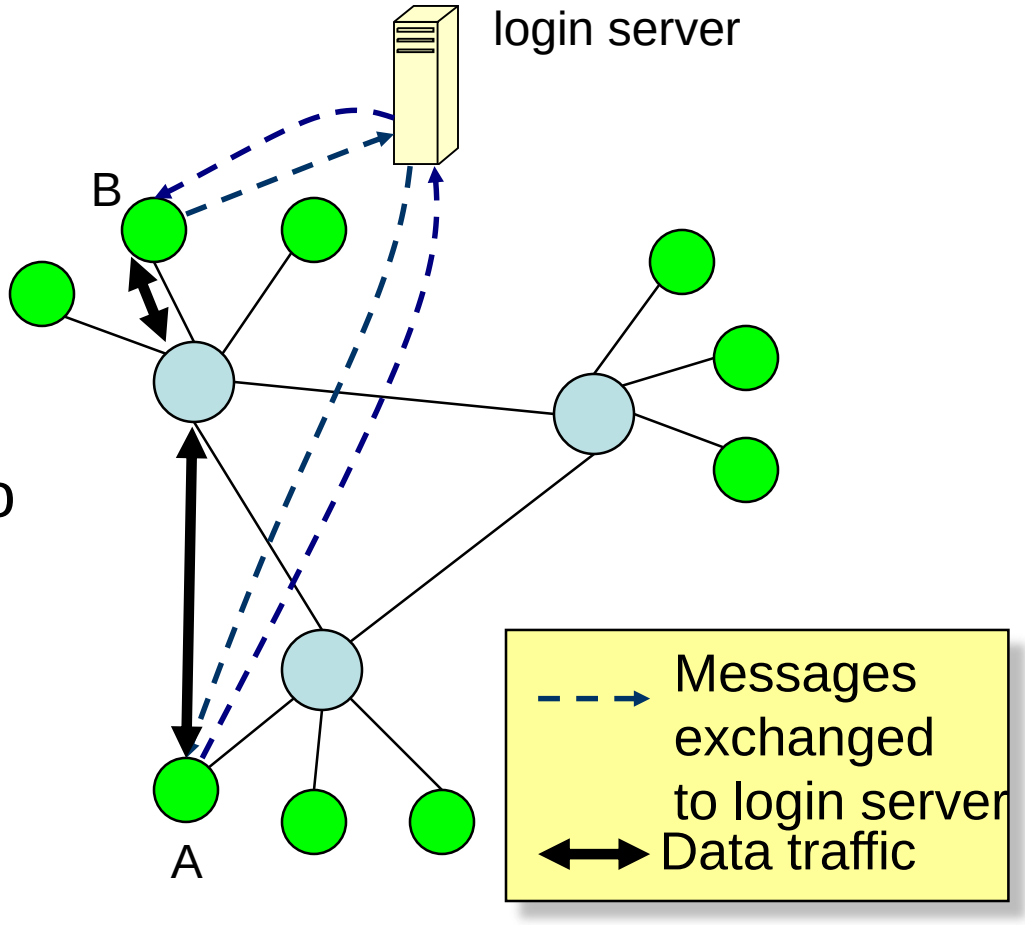
- Scalable:
  - The more nodes that are part of the system, demand increases and total capacity of the system also increases. Where in client-server network architectures as more clients are added to the system, the system resources decreases.
- Failure:
  - There is no single point of failure, due to robustness of the system.
- Distribute cost
  - All clients contribute to the system

Disadvantages

- Security is a major concern, not all shared files are from benign sources. Attackers may add malware to p2p files as an attempt to take control of other nodes in the network.
- Heavy bandwidth usage
- Anti-P2P companies have introduced faked chunks into shared files that rendered shared files useless upon completion of the download.
- ISP throttling of P2P traffic
- Potential legal/moral concerns

Skype

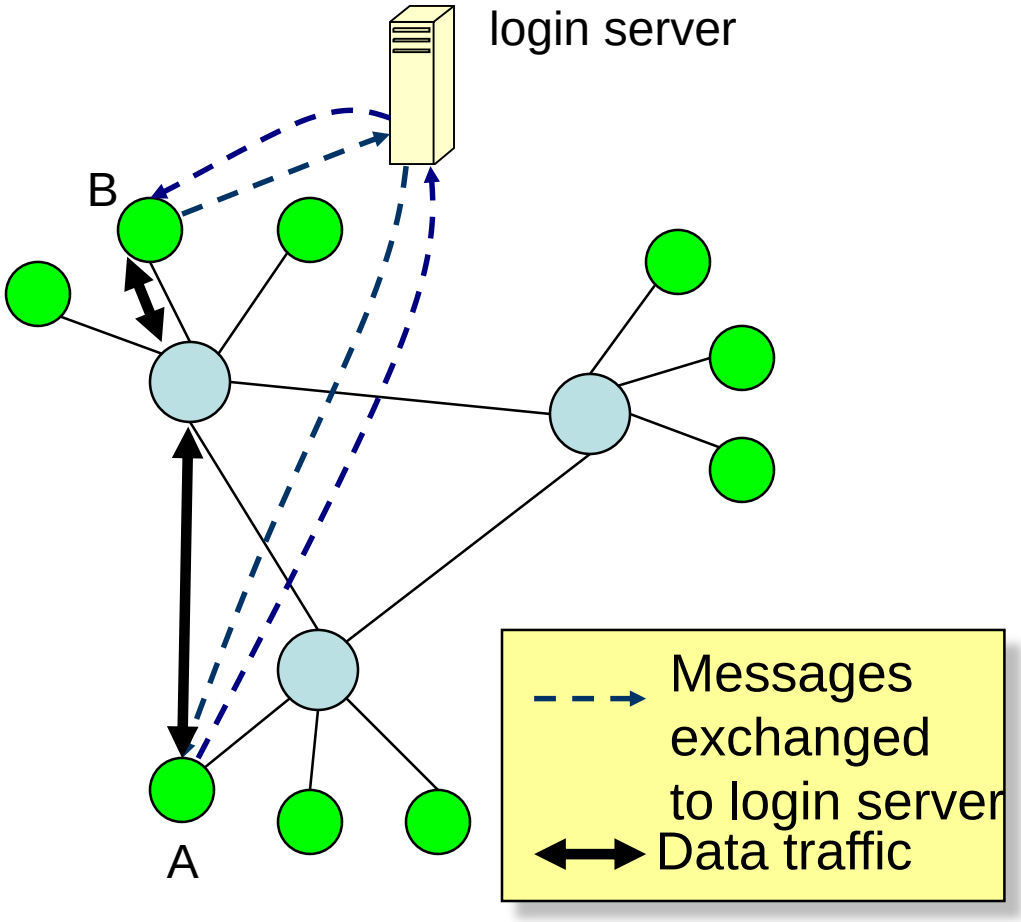
- Peer-to-peer Internet Telephony System
- Two-level hierarchy like KaZaa
  - Ultrapeers used mainly to route traffic between NATed end-hosts …
  - … plus a login server to
    - authenticate users
    - ensure that names are unique across network



(Note*: probable protocol; Skype protocol is not published)

Skype

- Peer-to-peer Internet Telephony System
- Distributed - server no longer is bottleneck!
  - Performance: Only do look up through Server
  - Availability: Server is single point of failure
  - Cost: Much lower cost and distributed cost
- With a price!!!!
  - Privacy is at risk



(Note*: probable protocol; Skype protocol is not published)

The Cluster-Based DFS Model

What is Cloud Computing?

- “Cloud” refers to large Internet services running on 10,000s of machines (Google, Facebook, etc)

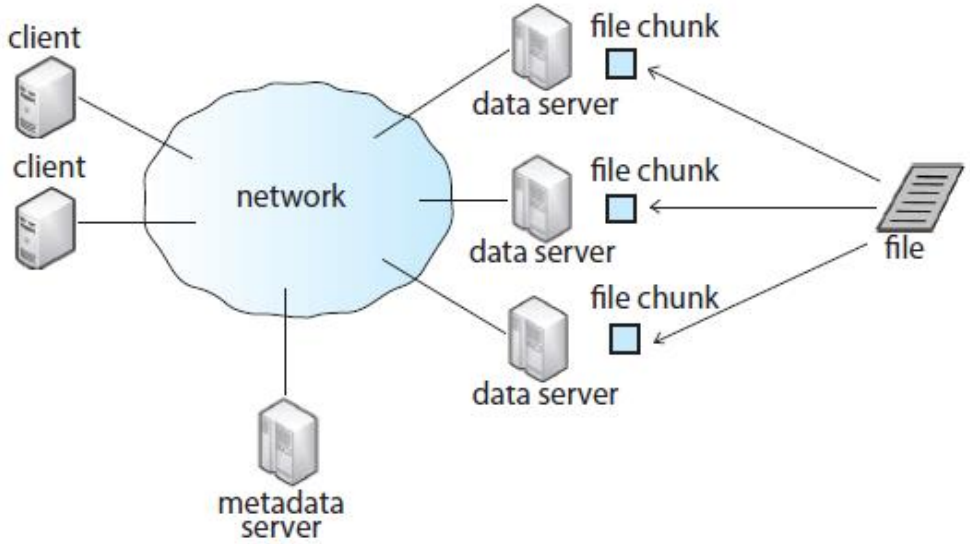
- “Cloud computing” refers to services by these companies that let external customers rent cycles
  - Amazon EC2: virtual machines at 8.5¢/hour, billed hourly
  - Amazon S3: storage at 10-15¢/GB/month
  - Windows Azure: applications using Azure API

- Attractive features:
  - Scale: up or down 100s of nodes available in minutes
  - Fine-grained billing: pay only for what you use
  - Ease of use: get root access

What Can You Run in Cloud Computing?

- Virtual Machine instances
- Storage services
  - Simple Storage Service (S3)
  - Elastic Block Storage (RBS)
- Databases:
  - Database instances (e.g., mySQL, SQL Server, …)
  - SimpleDB
- Content Distribution Network: CloudFront
- **MapReduce**: Amazon Elastic MapReduce
- …

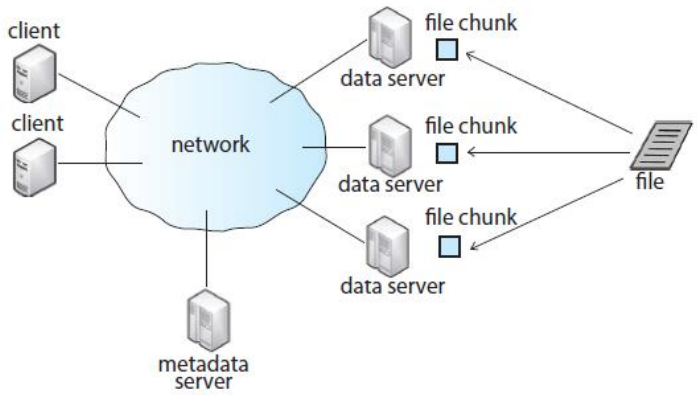
The Cluster-Based DFS Model



- DFS model presented (e.g. GFS and HDFS)
  - One ore more clients are connected via a network to a master metadata server and several data servers that house “chunks” of files
  - The metadata server keeps a mapping of **which data servers hold chunks of which files** and **a traditional hierarchical mapping of directories and files**.
  - Each file chunk is stored on a data server and is replicated a certain number of times to protect against component failure and for faster access to data

The Cluster-Based DFS Model

- Procedure of obtaining a file
  - A client must first contact the meta server
  - Metadata server then returns to the client the ID of the data servers that hold the requested file chunks
  - The client can then contact the closest data server(s) to receive the file information
  - Different chunks of the file can be read or written to in parallel if they are stored on different data server, and the metadata server may need to be contacted only once in the entire process
  - This helps metadata server less likely to be a performance bottleneck



Challenges

- Cheap nodes fail, especially when you have many
  - MTBF(Mean time between failures) for 1 node = 3 years
  - MTBF for 1 out of 1000 nodes = 1 day
  - **Solution:** Build fault-tolerance into system

- Programming distributed systems is hard
  - **Solution:** Restricted programming model:
    users write data-parallel “map” and “reduce” functions, system handles work distribution and failures

The Google File System Motivation

- Need for a scalable DFS
- Large distributed data-intensive applications
- High data processing needs
- Performance, Reliability, Scalability and Availability

The Google File System

Environment

- Commodity Hardware
  - Inexpensive and therefore…
- Component Failure
  - the norm rather than the exception
- Thousand of TBs of Space

Applications & Usecase

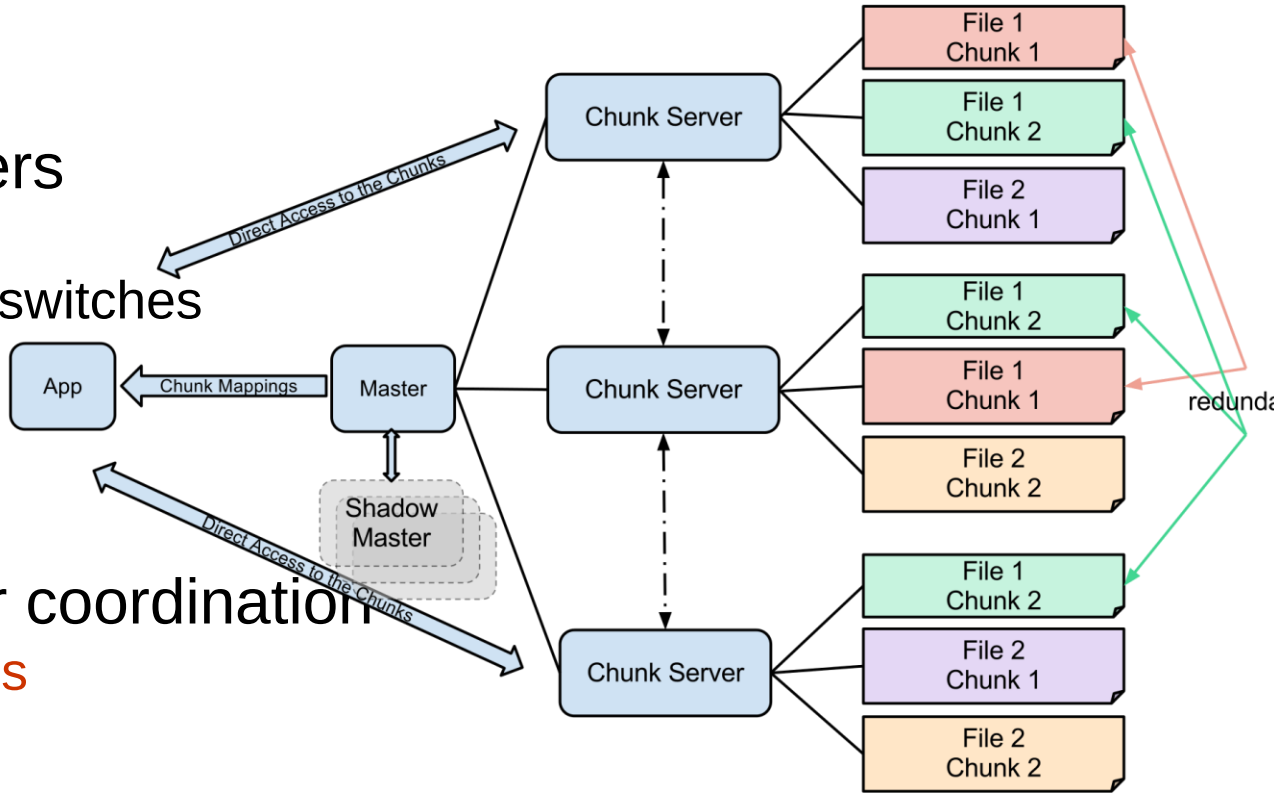
- **Multi-GB & Multi-TB files**
  - **Common**
- **Workloads**
  - Large streaming reads
  - Small random reads
  - Large, sequential writes that append data to file
  - Multiple clients concurrently append to one file
- **High sustained bandwidth**
  - **More important than low latency**

Architecture

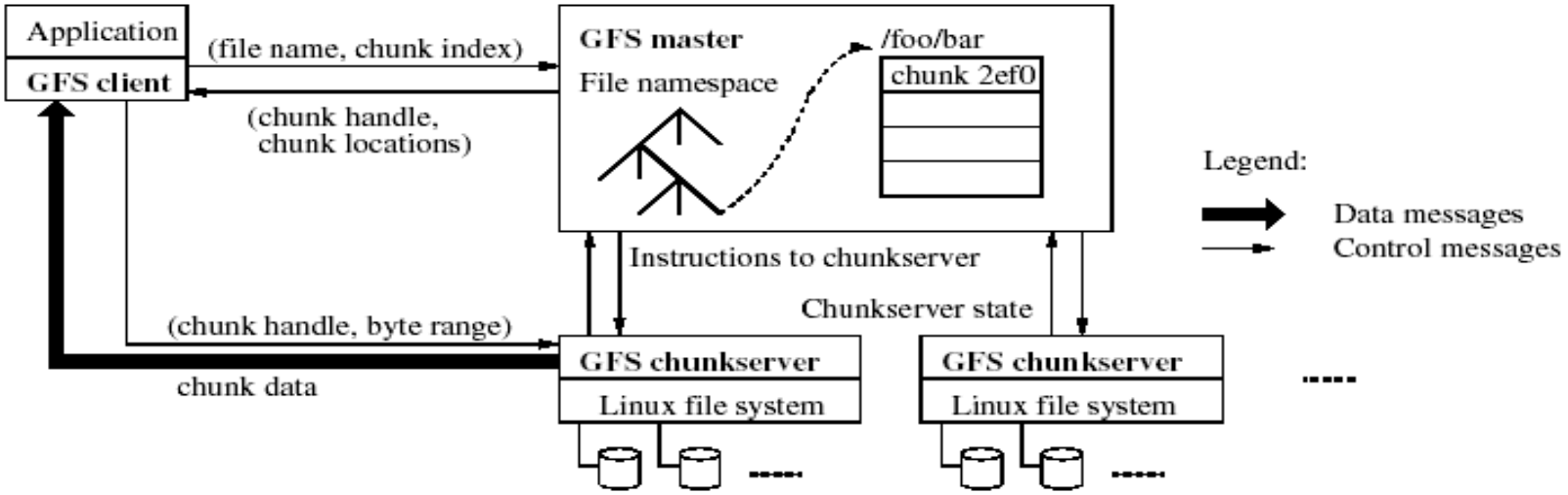
- Files are divided into *chunks*
- Fixed-size chunks (64MB)
- Replicated over *chunkservers*, called *replicas*
- Unique 64-bit *chunk handles*
- Chunks as Linux files

Architecture

- Single master
- Multiple chunkservers
  - Grouped into Racks
  - Connected through switches
- Multiple clients
- Master/chunkserver coordination
  - HeartBeat messages



Architecture



- Contact single *master*
- Obtain chunk locations
- Contact one of chunkservers
- Obtain data

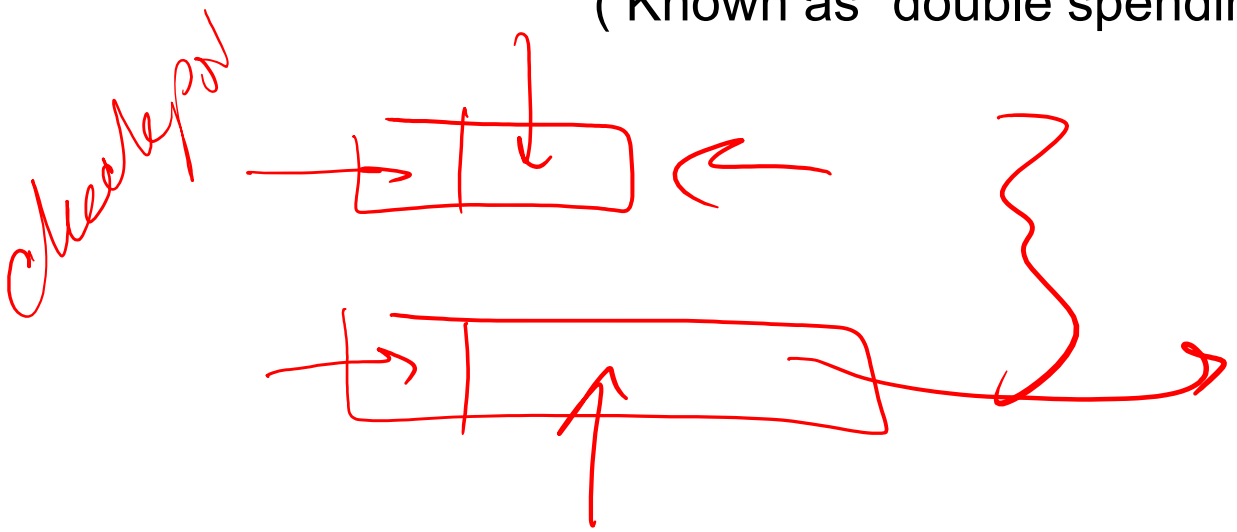
Features

- High availability and component failure
  - Fault tolerance, Master/chunk replication, HeartBeat, Operation Log, Checkpointing, Fast recovery
- TGs of Space
  - 100s of chunkservers, 1000s of disks
- Networking
  - Clusters and racks
- Scalability
  - Simplicity with a single master
  - Interaction between master and chunkservers is minimized

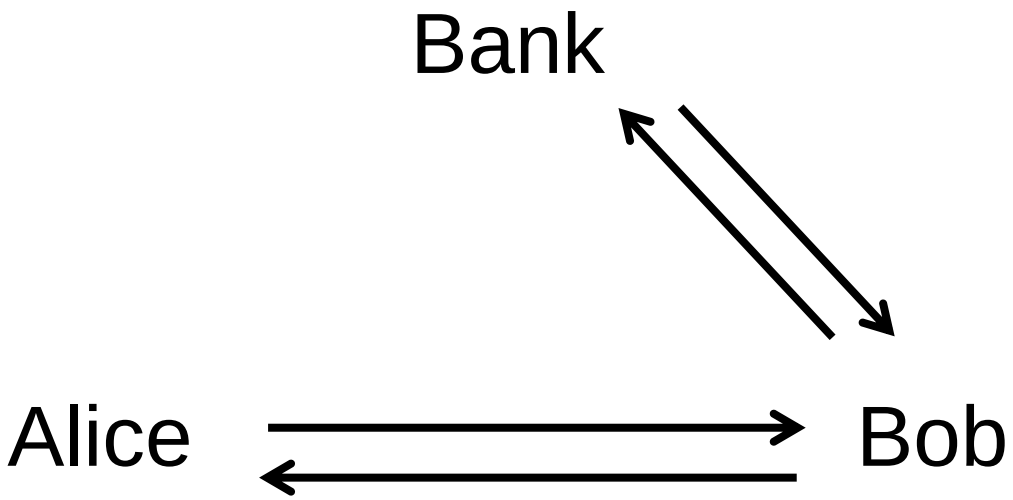
Problem: Equivocation!

Can Alice “pay” both Bob and Charlie with same bitcoin ?

(Known as “double spending”)

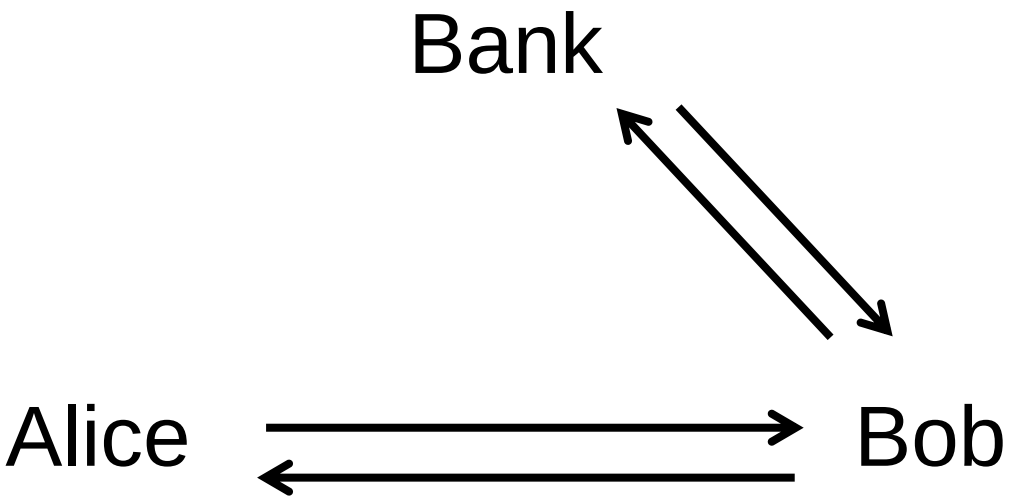


How traditional e-cash handled problem



- When Alice pays Bob with a coin, Bob validates that coin hasn't been spend with trusted third party
- Introduced "blind signatures" and "zero-knowledge protocols" so bank can't link withdrawals and deposits

How traditional e-cash handled problem



- When Alice pays Bob with a coin, Bob validates that coin hasn't been spend with trusted third party

Bank maintains linearizable log of transactions

PROBLEM: EQUIVOCATION!

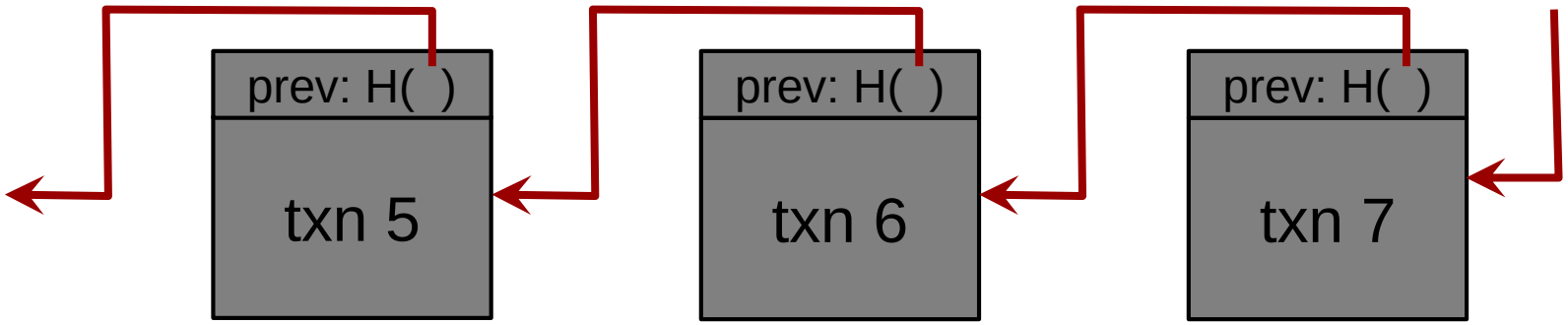
Goal: No double-spending in decentralized environment

Bitcoin: 10,000 foot view

- Public
  - Transactions are signed: $txn = Sign_{Alice}(BTC, PK_{Bob})$
  - All transactions are sent to all network participants

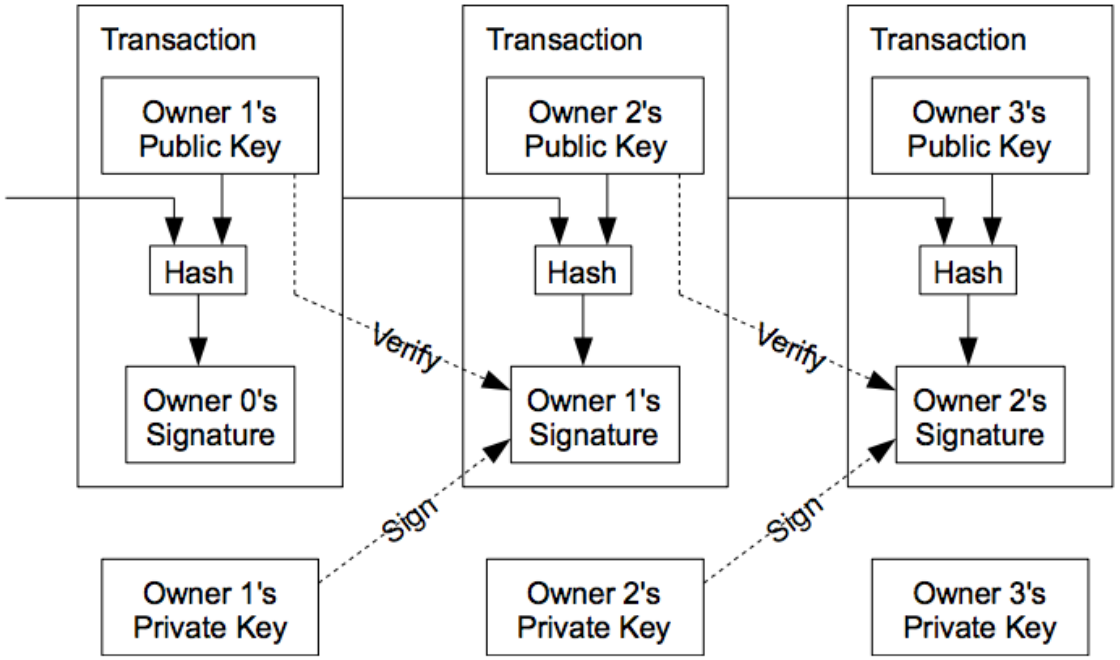
- No equivocation: Log append-only and consistent
  - All transactions part of a hash chain
  - Consensus on set/order of operations in hash chain

Blockchain: Append-only hash chain



- Hash chain creates “tamper-evident” log of txns
- Security based on collision-resistance of hash function
  - Given m and h = hash(m), difficult to find m’ such that h = hash(m’) and m != m’

Blockchain: Append-only hash chain



Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main

GFS

Master

- **Metadata**
  - Three types
    - File & chunk namespaces
    - Mapping from files to chunks
    - Locations of chunks' replicas
  - Replicated on multiple remote machines
  - Kept in memory

- **Operations**
  - Replica placement
  - New chunk and replica creation
  - Load balancing
  - Unused storage reclaim

Implementation – Consistency Model

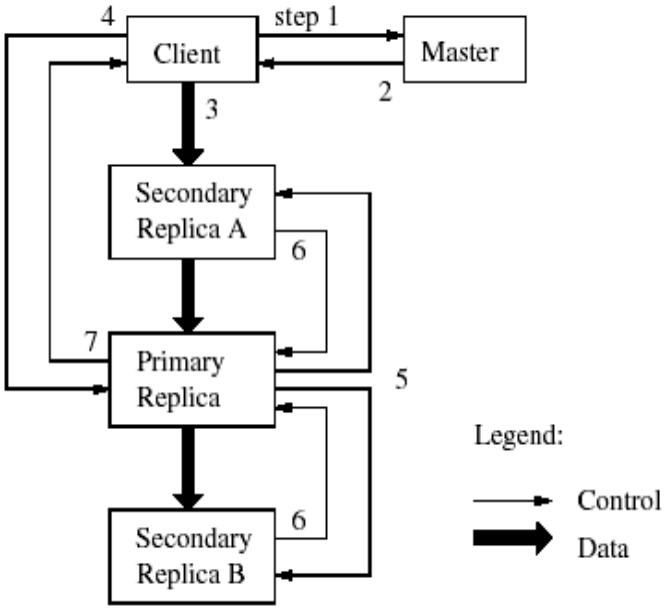
- Relaxed consistency model
- Two types of mutations
  - **Writes**
    - Cause data to be written at an application-specified file offset
  - **Record appends**
    - Operations that append data to a file
    - Cause data to be appended atomically at least once
    - Offset chosen by GFS, not by the client

- States of a file region after a mutation
  - *Consistent*
    - All clients see the same data, regardless which replicas they read from
  - *Defined*
    - *consistent* + all clients see what the mutation writes in its entirety
  - *Undefined*
    - *consistent* +but it may not reflect what any one mutation has written
  - *Inconsistent*
    - Clients see different data at different times

Implementation – Leases and Mutation Order

- Master uses *leases* to maintain a consistent mutation order among replicas
- *Primary* is the chunkserver who is granted a chunk lease
- All others containing replicas are *secondaries*
- Primary defines a mutation order between mutations
- All *secondaries* follows this order

Implementation – Writes



Mutation Order

→ *identical replicas*

→ File region may end up containing *mingled fragments* from different clients (*consistent* but *undefined*)

Implementation – Atomic Appends

- **The client specifies only the data**

- **Similar to writes**
  - Mutation order is determined by the *primary*
  - All *secondaries* use the same mutation order

- **GFS appends data to the file *at least once atomically***
  - The chunk is padded if appending the record exceeds the maximum size → *padding*
  - If a record append fails at any replica, the client retries the operation → *record duplicates*
  - File region may be *defined* but interspersed with *inconsistent*

Implementation – Snapshot

- Goals
  - To quickly create branch copies of huge data sets
  - To easily checkpoint the current state

- Copy-on-write technique
  - Metadata for the source file or directory tree is duplicated
  - Reference count for chunks are incremented
  - Chunks are copied later at the first write

Implementation – Operation Log

- contains historical records of metadata changes
- replicated on multiple remote machines
- kept small by creating checkpoints
- checkpointing avoids interfering other mutations by working in a separate thread

Implementation – Namespace Management and Locking

- Namespaces are represented as a lookup table mapping full pathnames to metadata
- Use locks over regions of the namespace to ensure proper serialization
- Each master operation acquires a set of locks before it runs

Implementation – Example of Locking Mechanism

- Preventing /home/user/foo from being created while /home/user is being snapshotted to /save/user
  - Snapshot operation
    - Read locks on /home and /save
    - Write locks on /home/user and /save/user
  - File creation
    - read locks on /home and /home/user
    - write locks on /home/user/foo
  - Conflict locks on /home/user

Implementation – Guarantees by the consistency model

	Write	**Record Append**
Serial Success	*defined*	*Defined* interspersed with *inconsistent*
Concurrent Successes	*consistent* but *undefined*	
Failure	*inconsistent*	

- File namespace mutations (e.g., file creation) are atomic
  - Namespace management and locking
  - The master's operation log

- After a sequence of successful mutations, the mutated file is guaranteed to be defined and contain the data written by the last mutation. This is obtained by
  - Applying the same mutation order to all replicas
  - Using chunk version numbers to detect stale replica

Implementation – Implications for Applications

- **Relying on appends rather on overwrites**
- **Checkpointing**
  - to verify how much data has been successfully written
- **Writing self-validating records**
  - Checksums to detect and remove *padding*
- **Self-identifying records**
  - Unique Identifiers to identify and discard *duplicates*

Other Issues – Data flow

- Decoupled from control flow
  - to use the network efficiently
- Pipelined fashion
  - Data transfer is pipelined over TCP connections
  - Each machine forwards the data to the “closest” machine
- Benefits
  - Avoid bottle necks and minimize latency

Other Issues – Garbage Collection

- Deleted files
  - Deletion operation is logged
  - File is renamed to a hidden name, then may be removed later or get recovered
- Orphaned chunks (unreachable chunks)
  - Identified and removed during a regular scan of the chunk namespace
- Stale replicas
    - *Chunk version numbering*

Other Issues – Replica Operations

- Creation
  - Disk space utilization
  - Number of recent creations on each chunkserver
  - Spread across many racks

- Re-replication
  - Prioritized: How far it is from its replication goal…
  - The highest priority chunk is cloned first by copying the chunk data directly from an existing replica

- Rebalancing
  - Periodically

Other Issues – Fault Tolerance and Diagnosis

- Fast Recovery
  - Operation log
  - Checkpointing
- Chunk replication
  - Each chunk is replicated on multiple chunkservers on different racks
- Master replication
  - Operation log and check points are replicated on multiple machines
- Data integrity
  - Checksumming to detect corruption of stored data
  - Each chunkserver independently verifies the integrity
- Diagnostic logs
  - Chunkservers going up and down
  - RPC requests and replies

Current status

- Two clusters within Google
  - Cluster A: R & D
    - Read and analyze data, write result back to cluster
    - Much human interaction
    - Short tasks
  - Cluster B: Production data processing
    - Long tasks with multi-TB data
    - Seldom human interaction

Measurements

- Read rates much higher than write rates
- Both clusters in heavy read activity
- Cluster A supports up to 750MB/read, B: 1300 MB/s
- Master was not a bottle neck

Cluster	**A**	**B**
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

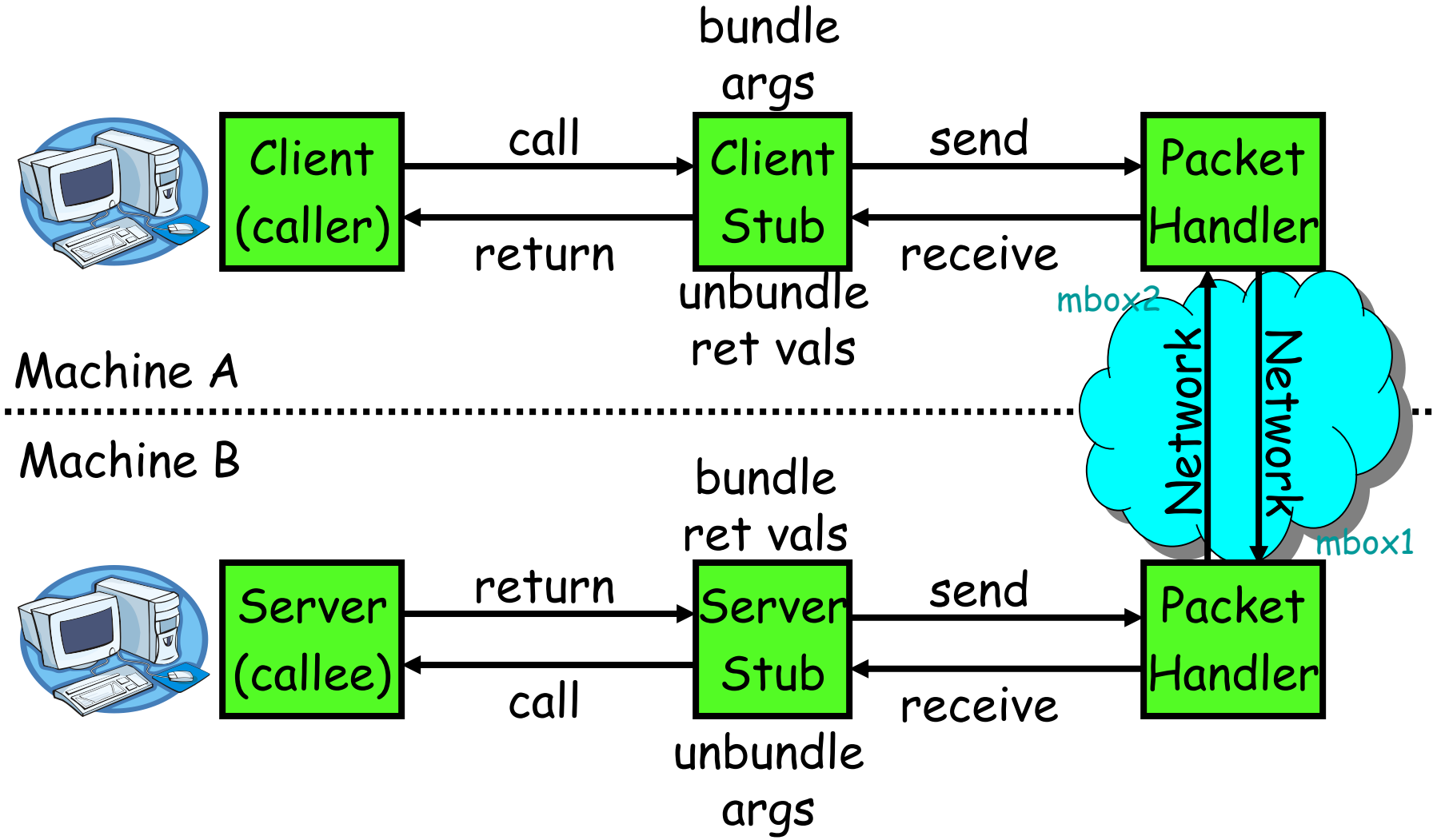
Measurements

- Recovery time (of one chunkserver)
  - 15,000 chunks containing 600GB are restored in 23.2 minutes (replication rate ≅ 400MB/s)

Recall: Remote Procedure Call

- Using raw messaging for programming is a bit too low-level
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
- Better option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    ```
    remoteFileSystem→Read("parameter");
    ```
  - Translated automatically into call on server:
    ```
    fileSys→Read("parameter");
    ```
- Implementation:
  - Request-response message passing
  - "Stub" provides glue on client/server
    - Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
    - Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Information Flow



RPC Details

- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an “interface definition language (IDL)”
    - Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    - Convert everything to/from some canonical form
    - Tag every item with an indication of how it is encoded (avoids unnecessary conversions).

RPC Details (continued)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - This is another word for “naming” at network level
    - Static: fixed at compile time
    - Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - Name service provides dynmaic translation of service→mbox
  - Why dynamic binding?
    - Access control: check who is permitted to access service
    - Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - Choose unloaded server for each new request
    - Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC

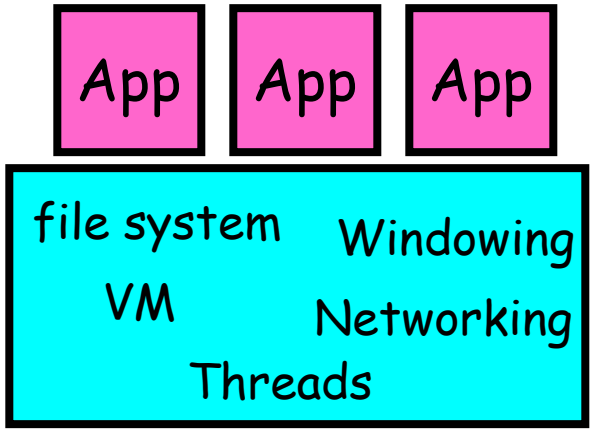
- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    - User-level bug causes address space to crash
    - Machine failure, kernel bug causes all processes on same machine to fail
    - Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    - Did my cached data get written back or not?
    - Did server do what I requested or not?
  - Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    - Caching can help, but may make failure handling complex

Cross-Domain Communication/Location Transparency

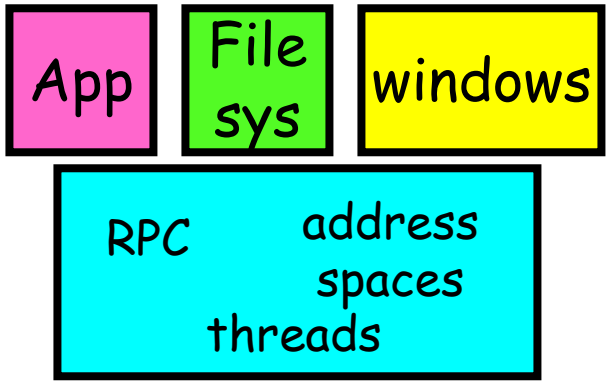
- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - “Remote” procedure call (2-way communication)
- RPC’s can be used to communicate between address spaces on different machines on the same machine
  - Services can be run wherever it’s most appropriate
  - Access to local and remote services looks the same
- Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

Microkernel operating systems

- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine



Monolithic Structure



Microkernel Structure

- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.

Summary

- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)
- VFS: Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems
- Distributed File System:
  - Transparent access to files stored on a remote disk
    - NFS: Network File System
    - AFS: Andrew File System
  - Caching for performance
- Cache Consistency: Keeping contents of client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated will affect performance vs. consistency
  - NFS: check periodically for changes
  - AFS: clients register callbacks so can be notified by server of changes