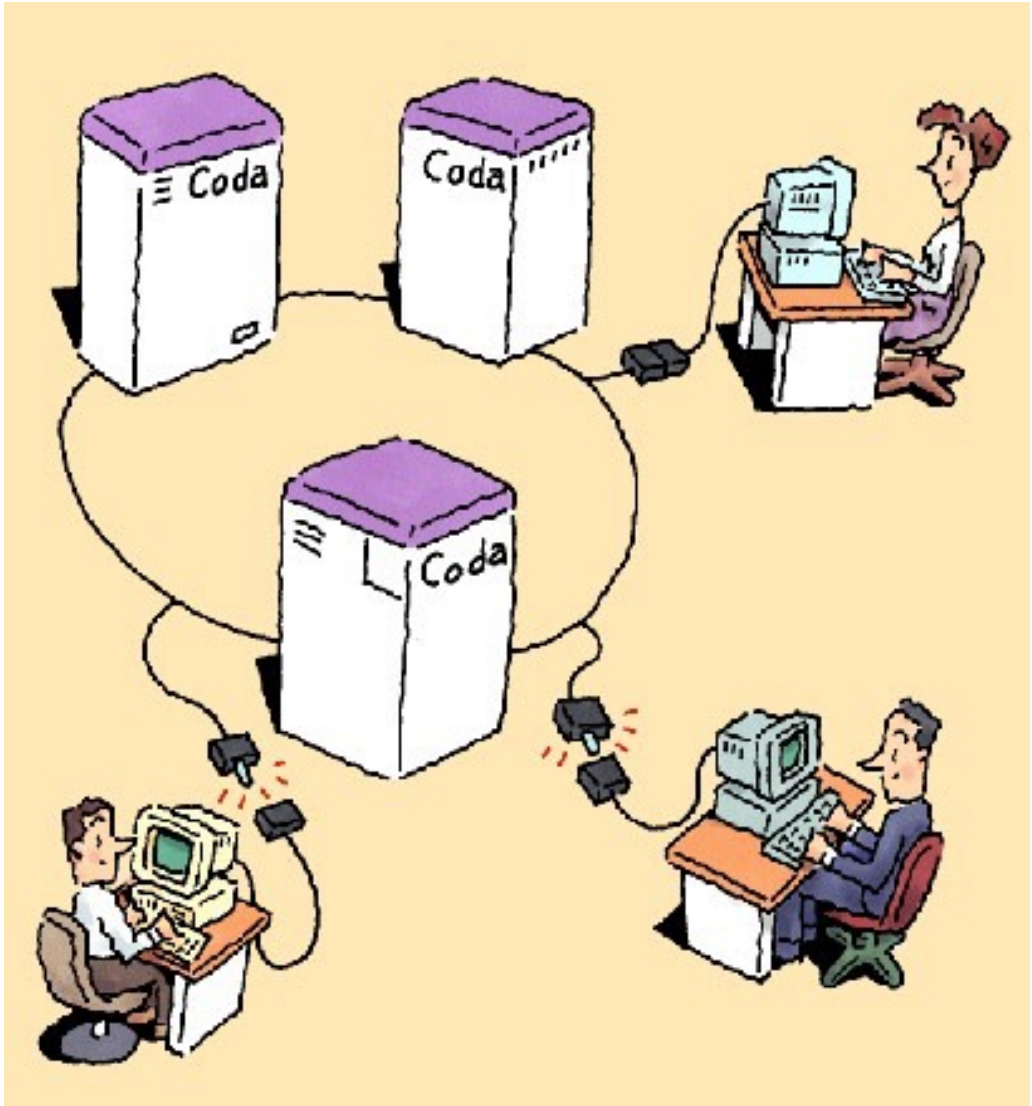
Distributed File Systems

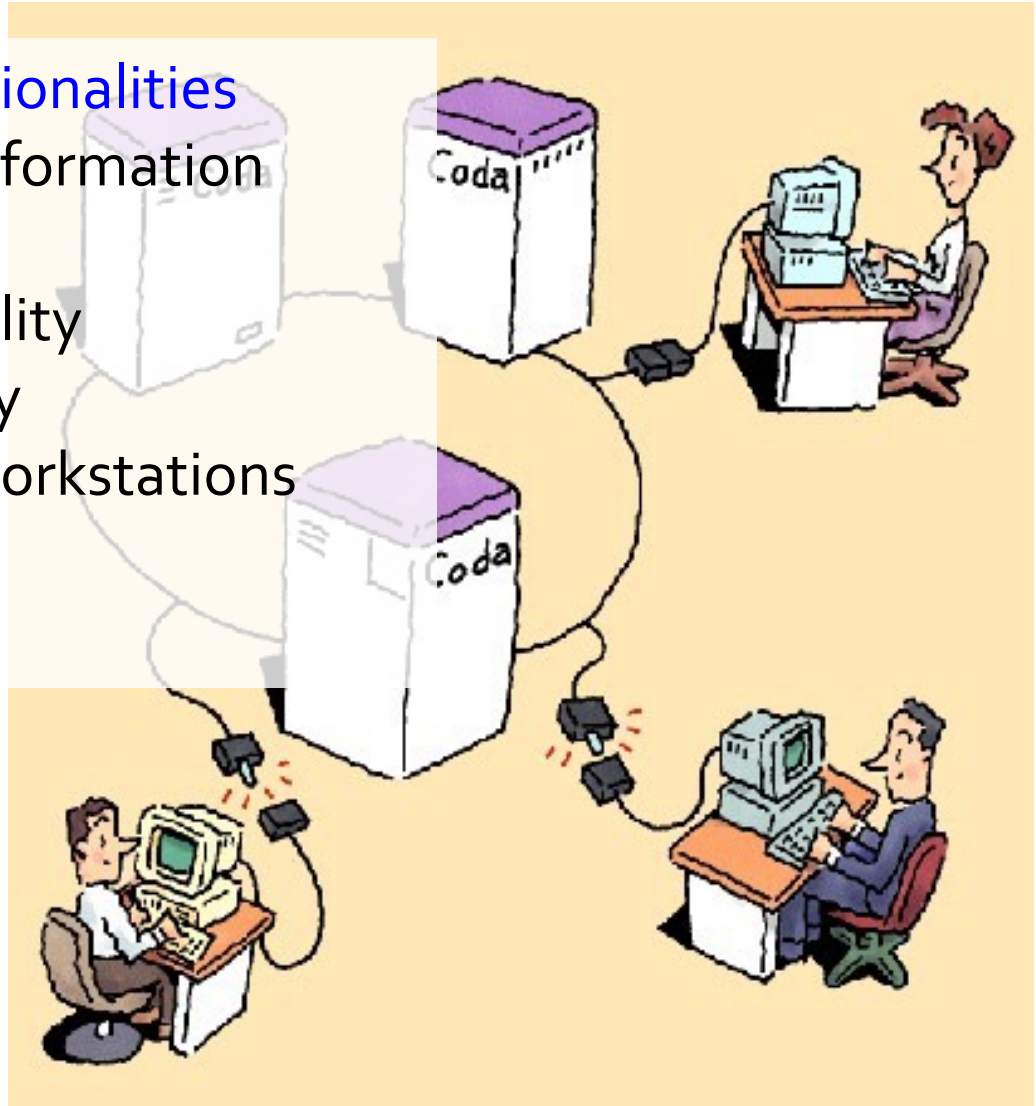
Introduction



Introduction

Functionalities

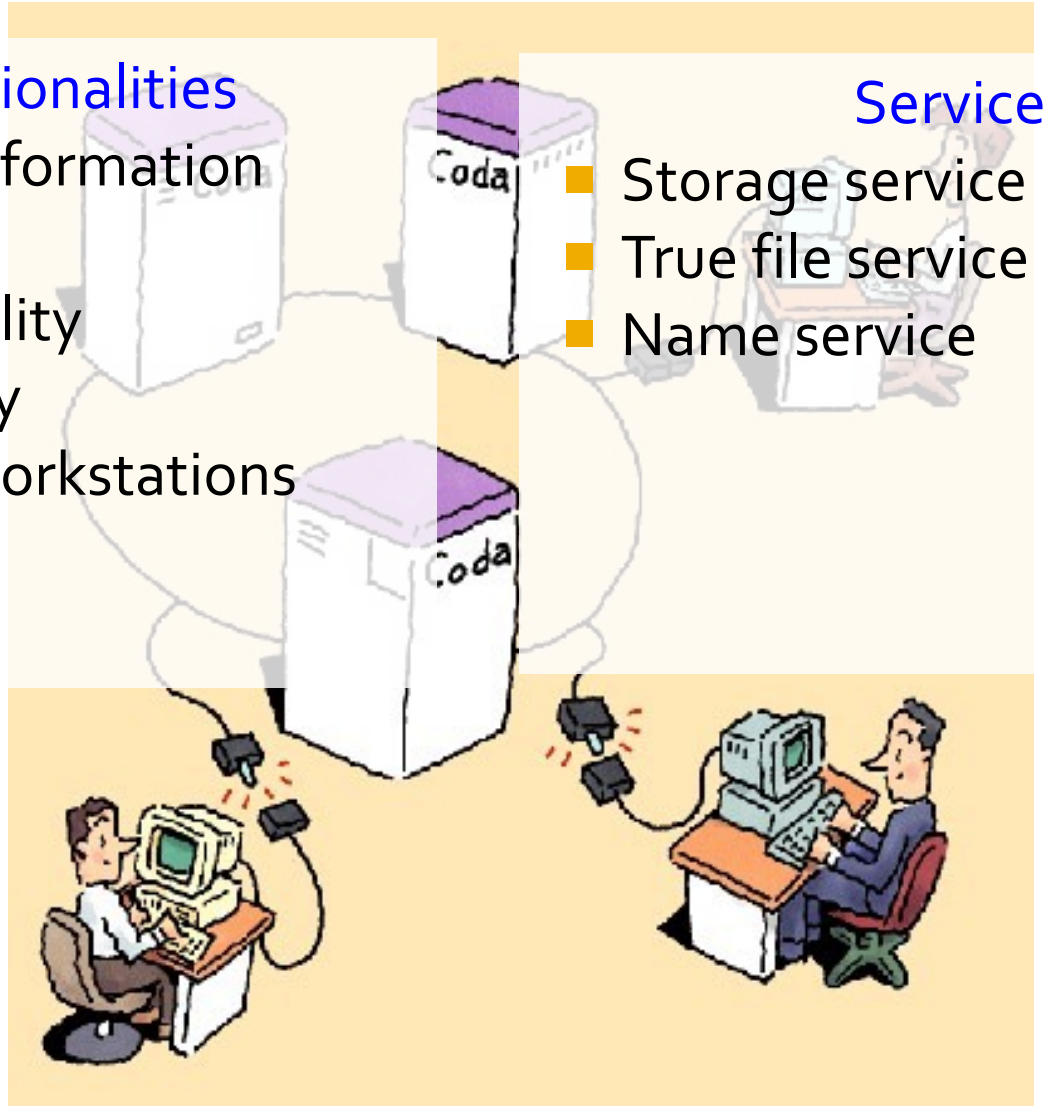
- Remote information sharing
- User mobility
- Availability
- Diskless workstations



Introduction

Functionalities

- Remote information sharing
- User mobility
- Availability
- Diskless workstations



Services

- Storage service
- True file service
- Name service

Topics

- Desirable Features
- File Models
- File-Accessing Models
- File-Sharing Semantics
- File-Caching Schemes
- File Replication
- Fault Tolerance
- Case Study: CODA File System
- Case Study: Google File System andBigTable

Discussion: Desirable Features

Desirable Features

- Transparency:
  - Structure transparency
  - Access transparency
  - Naming transparency
  - Replication transparency
- User Mobility
- Performance
- Simplicity and ease of use
- Scalability
- High availability
- High reliability
- Data integrity
- Security
- Heterogeneity

File Models – Unstructured/Structured Files

- Unstructured files:
  - A file is an unstructured sequence of data.
  - The operating system is not interested in the information stored in the files.
  - The interpretation of the meaning and structure of the data stored in the files are entirely up to the application programs.
  - Who are using this model: UNIX and DOS.
- Structured files:
  - Non-indexed records
  - Indexed records
  - This model was planned to be deployed in Windows 7.

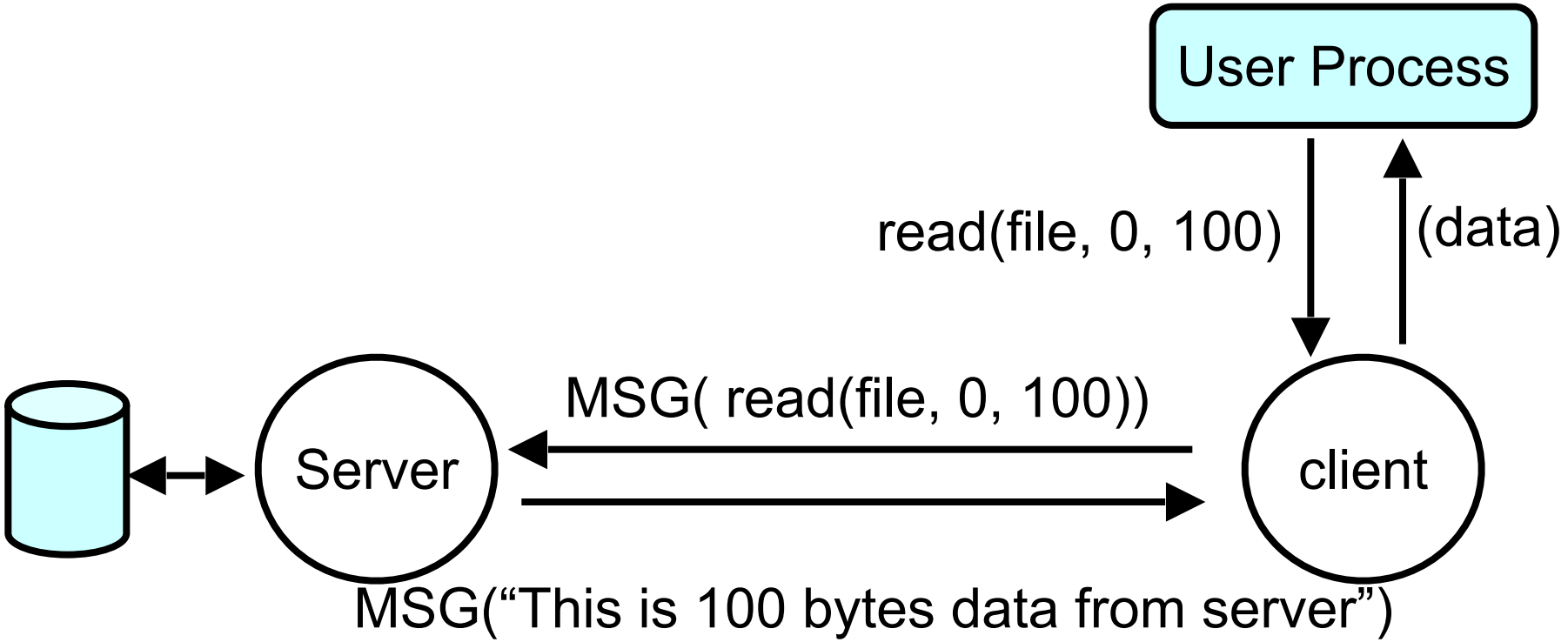
File Models: Mutable/Immutable Files

- Mutable file model: an update overwrites on its old contents to produce the new contents.
- Immutable file model:
  - A history of the files or file changes are stored.
  - Pros:
    - Easier to support file caching and replication
  - Cons:
    - Increased use of disk space and
    - Increased disk allocation activity
  - Cedar File System, 1988:
    - Only a limited number of historical files are stored.
    - The users can specify a historical file for file access.

File Access Models

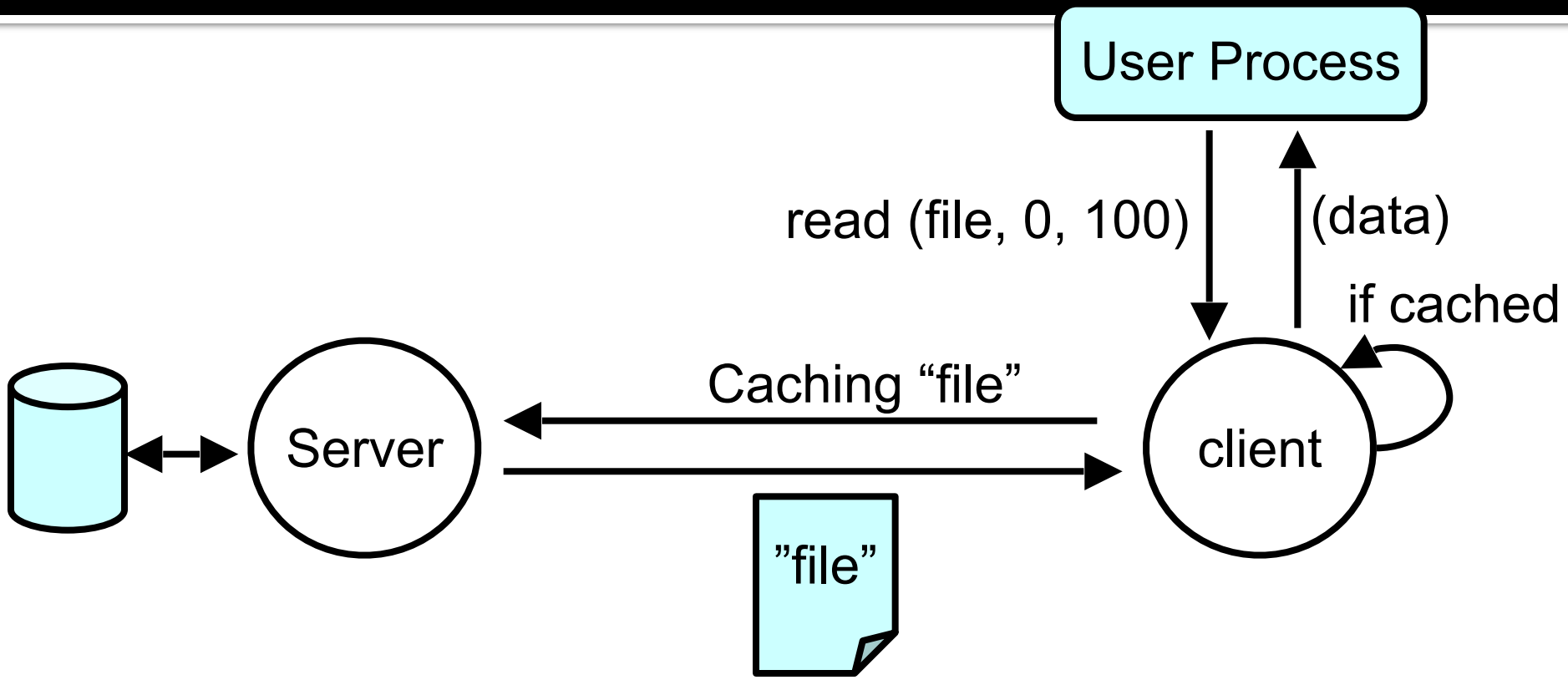
- How the users access the files depends on the file access models used by the DFS.
- Two factors: access method and data units
  - the method used for accessing remote files
    - Remote Service Model: network overhead should be minimized.
    - Data-Caching Model: concurrence control must be considered.
    - Hybrid method:
      - LOCUS and NFS use the remote service model but add caching for better performance.
      - Sprite uses the data-caching model but employs remote access under certain circumstances.

Accessing Remote Files - Remote Service Model



Pros and Cons

Accessing Remote Files – Data-Caching Model



Pros and Cons

- Write operation may incur substantial overhead.
- It could reduce network traffic, contention for the network, and contention for the file servers.

Class Discussion

- How you determine the access model for your distributed file systems?

Unit of Data Transfer

	Pros	Cons	Examples
File-level			
Block-level			
Byte-level			
Record-level			

Unit of Data Transfer

	Pros	Cons	Examples
File-level	• Better efficiency • Better scalability • Low disk access overhead • Reliability	• Greater storage space on client side • wasteful of storage space under certain case	Ameoba, Cedar File System (CFS), and Andrew File System (AFS-2)
Block-level			
Byte-level			
Record-level			

Unit of Data Transfer

	Pros	Cons	Examples
File-level	• Better efficiency • Better scalability • Low disk access overhead • Reliability	• Greater storage space on client side • wasteful of storage space under certain case	Ameoba, Cedar File System (CFS), and Andrew File System (AFS-2)
Block-level	• Less storage space on client sides	• Poor performance when the entire file is requested	Apollo domain file system, LOCUS and Sun's NFS
Byte-level			
Record-level			

Unit of Data Transfer

	Pros	Cons	Examples
File-level	• Better efficiency • Better scalability • Low disk access overhead • Reliability	• Greater storage space on client side • wasteful of storage space under certain case	Ameoba, Cedar File System (CFS), and Andrew File System (AFS-2)
Block-level	• Less storage space on client sides	• Poor performance when the entire file is requested	Apollo domain file system, LOCUS and Sun's NFS
Byte-level	• Maximum flexibility	• Difficult for cache management	Cambridge File Server
Record-level			

Unit of Data Transfer

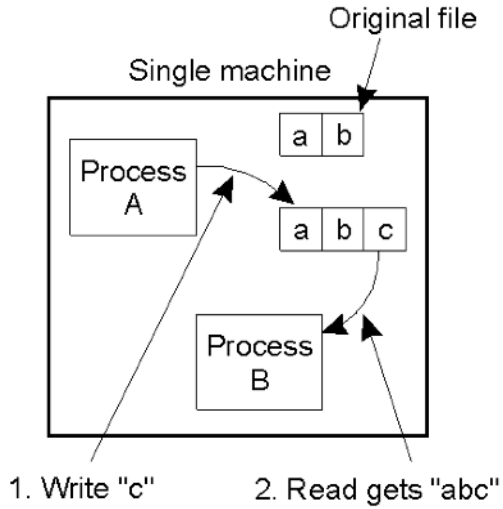
	Pros	Cons	Examples
File-level	• Better efficiency • Better scalability • Low disk access overhead • Reliability	• Greater storage space on client side • wasteful of storage space under certain case	Ameoba, Cedar File System (CFS), and Andrew File System (AFS-2)
Block-level	• Less storage space on client sides	• Poor performance when the entire file is requested	Apollo domain file system, LOCUS and Sun's NFS
Byte-level	• Maximum flexibility	• Difficult for cache management	Cambridge File Server
Record-level	• Best for structured file model		RSS

Unit of Data Transfer

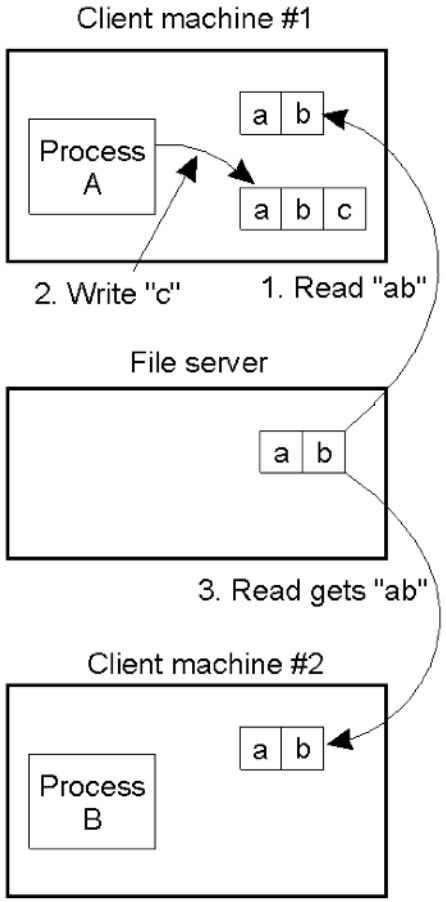
	Pros	Cons	Examples
File-level	• Better efficiency • Better scalability • Low disk access overhead • Reliability	• Greater storage space on client side • wasteful of storage space under certain case	Amoeba, Cedar File System (CFS), and Andrew File System (AFS-2)
Block-level	• Less storage space on client sides	• Poor performance when the entire file is requested	Apollo domain file system, LOCUS and Sun's NFS
Byte-level	• Maximum flexibility	• Difficult for cache management	Cambridge File Server
Record-level	• Best for structured file model		RSS

Semantics of File Sharing (1)

- On a single processor, when a *read* follows a *write*, the value returned by the *read* is the value just written.
- In a distributed system with caching, obsolete values may be returned.



(a)



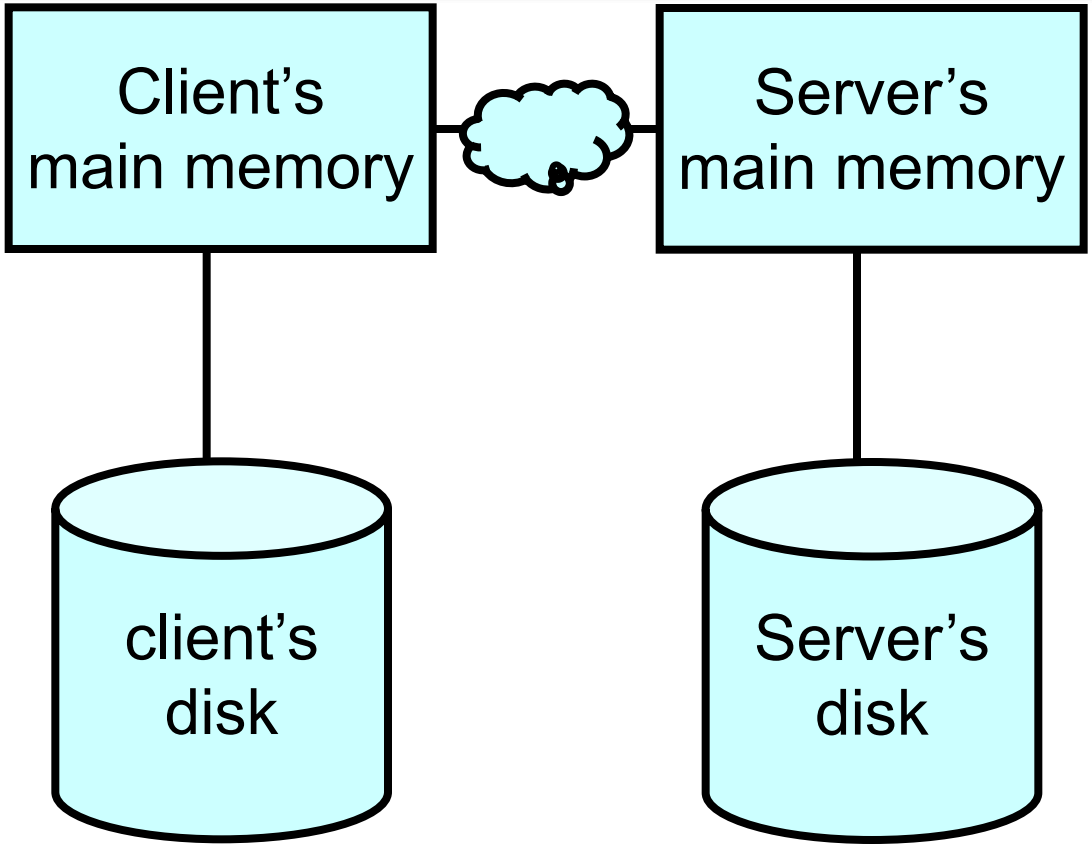
(b)

Semantics of File Sharing (2)

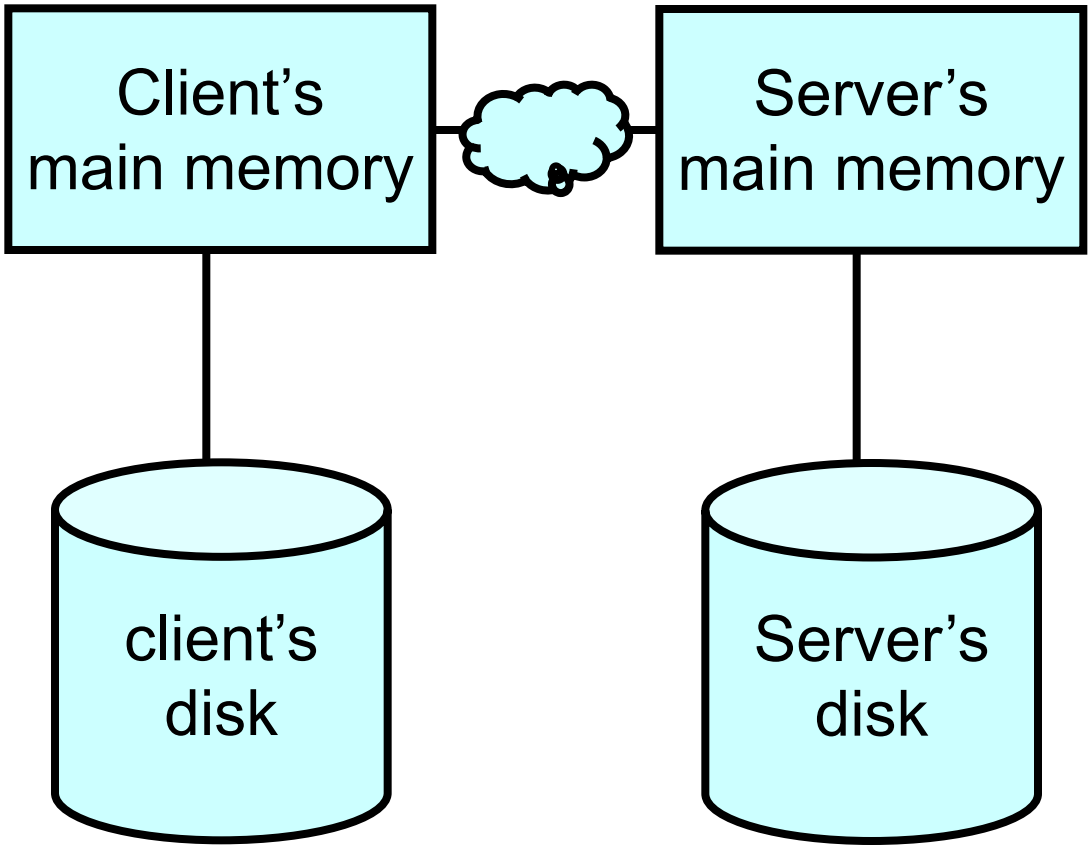
Method	Remark
UNIX semantics	**Every operation on a file is instantly visible to all processes**
Session semantics	**No changes are visible to other processes until the file is closed**
Immutable files	**No updates are possible; simplifies sharing and replication**
Transaction	**All changes occur atomically**

- Four ways of dealing with the shared files in a distributed system.
- UNIX semantics is desirable for distributed file systems but is difficult to implement due to poor performance, poor scalability, and poor reliability.
- Relaxed semantics of file sharing are usually used.

File-Caching Schemes- Cache location



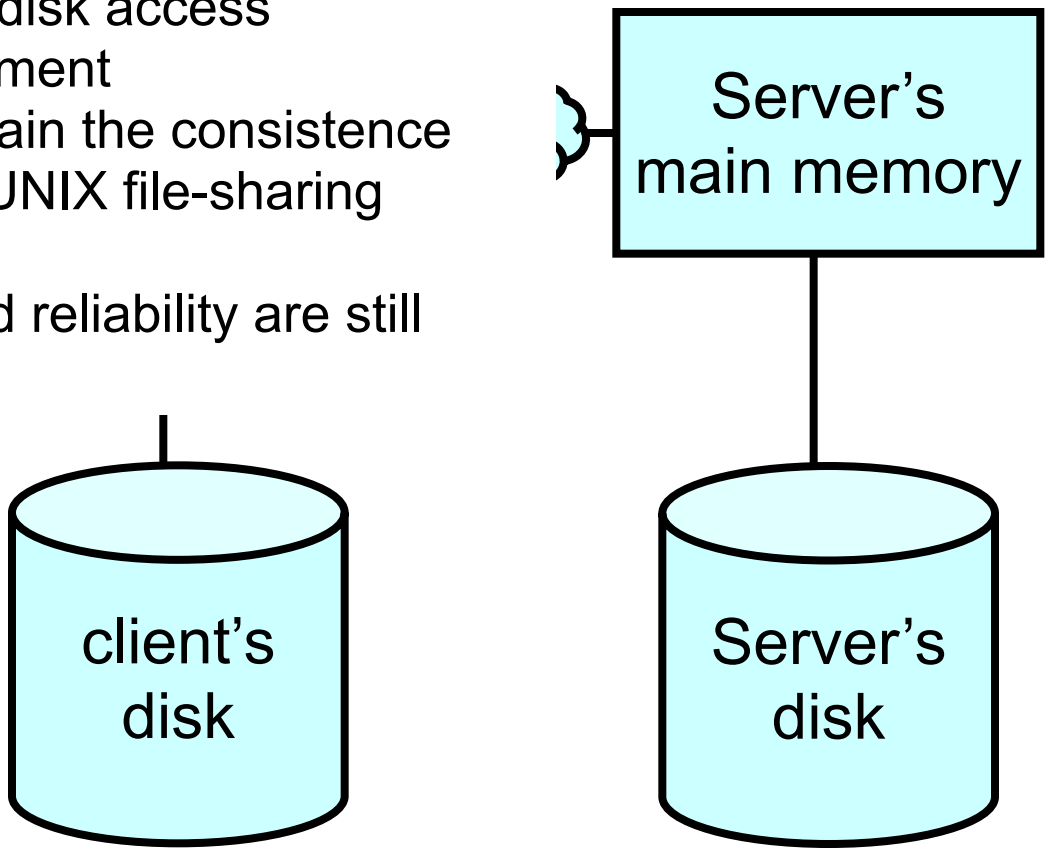
File-Caching Schemes - Cache location



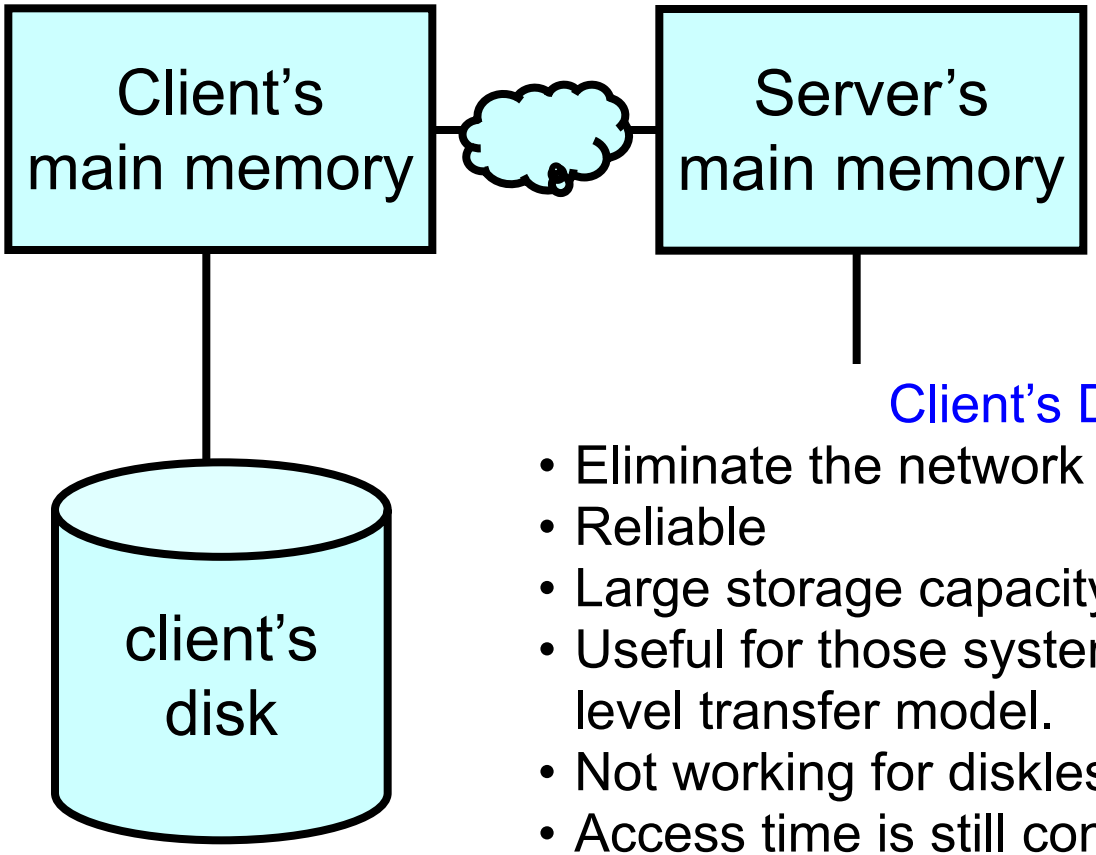
File-Caching Schemes - Cache location

Server's Main Memory

- Eliminate the disk access
- Easy to implement
- Easy to maintain the consistence
- May support UNIX file-sharing semantics
- Scalability and reliability are still open.



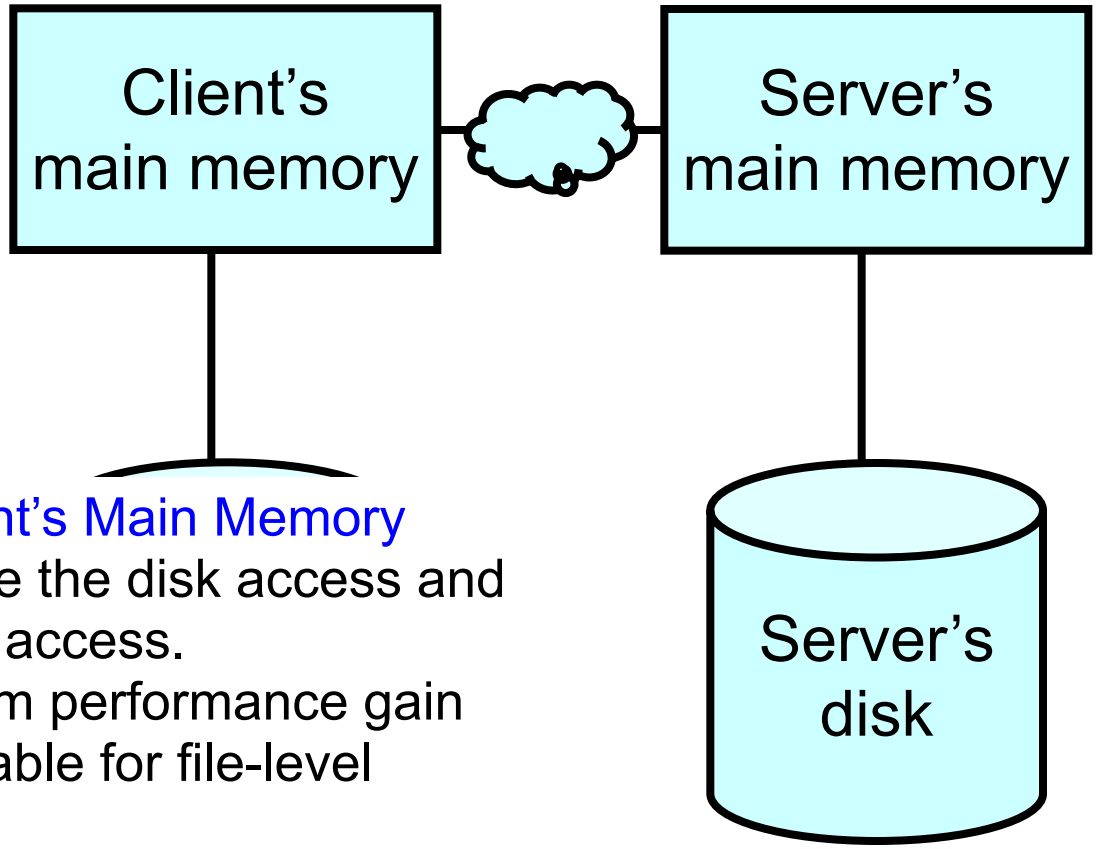
File-Caching Schemes - Cache location



Client's Disk

- Eliminate the network access
- Reliable
- Large storage capacity
- Useful for those system using the file-level transfer model.
- Not working for diskless clients.
- Access time is still considerable large due to the local disk access.

File-Caching Schemes - Cache location



Client's Main Memory

- Eliminate the disk access and network access.
- Maximum performance gain
- Not suitable for file-level sharing.

File Cache vs. Memory Cache

- Size:
  - Memory cache are located in CPU and are limited.
    - L1 ~ L4: 4KB ~ 128MB (on Intel Iris Pro Graphics).
  - File cache are located in memory or file systems and can be as large as the full file.
- Impedance Mismatch
  - Memory access delay: local bus + memory access
  - File access delay: communication (mostly network) + secondary storage access.

File-Caching Schemes – Modification Propagation

- Two design issues:
  - When to propagate modifications?
  - How to verify the validity of cached data?
- The modification propagation scheme used has a critical effect on the system's performance.

How to propagate the modified data?

File-Caching Schemes – Modification Propagation

- Two design issues:
  - When to propagate modifications?
  - How to verify the validity of cached data?
- The modification propagation scheme used has a critical effect on the system's performance.
- Write-Through Scheme
  - The new value is immediately sent to the server.
  - It has high reliability and suitability for UNIX-like semantics and poor write performance.
  - Only suitable for fairly large read-to-write ratio.

Delay-Write Scheme

- Delay-Write Scheme
  - When the cache is modified, the new value is written only to the cache and the client only makes a note that the cache entry has been updated.
  - All updated cache entries corresponding to a file are gathered together and sent to the server at a time.
- Different types of writing back
  - Write on ejection from cache
  - Periodic write
  - Write on close
- It improves the performance but may suffer from reliability problem.

File Replication-
What's the difference between caching and replication?

File Replication-
What's the difference between caching and replication?

- A replica is associated with a server; a cache is normally associated with a client.

File Replication-
What's the difference between caching and replication?

- A replica is associated with a server; a cache is normally associated with a client.
- A replica exists to improve the availability and performance; the existence of a cache depends on the locality in file access pattern.
  - We barely discuss caching for DSM. Can you think of the reasons?

File Replication-
What's the difference between caching and replication?

- A replica is associated with a server; a cache is normally associated with a client.
- A replica exists to improve the availability and performance; the existence of a cache depends on the locality in file access pattern.
  - We barely discuss caching for DSM. Can you think of the reasons?
- A replica is more persistent than a cache.

File Replication-
What's the difference between caching and replication?

- A replica is associated with a server; a cache is normally associated with a client.
- A replica exists to improve the availability and performance; the existence of a cache depends on the locality in file access pattern.
  - We barely discuss caching for DSM. Can you think of the reasons?
- A replica is more persistent than a cache.
- A cached copy is contingent upon a replica.

Multicopy Update Problem (1)

Multicopy Update Problem (1)

- Read-Only Replication

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.
- Read-Any-Write-All Protocol

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.
- Read-Any-Write-All Protocol
  - Allows the replication of both immutable and mutable files.

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.
- Read-Any-Write-All Protocol
  - Allows the replication of both immutable and mutable files.
  - Requires to read any copy of the replicated files and to write to all copies of the replicated files.

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.
- Read-Any-Write-All Protocol
  - Allows the replication of both immutable and mutable files.
  - Requires to read any copy of the replicated files and to write to all copies of the replicated files.
  - Requires lock to perform the write operations.

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.
- Read-Any-Write-All Protocol
  - Allows the replication of both immutable and mutable files.
  - Requires to read any copy of the replicated files and to write to all copies of the replicated files.
  - Requires lock to perform the write operations.
- Available-Copies Protocol

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.
- Read-Any-Write-All Protocol
  - Allows the replication of both immutable and mutable files.
  - Requires to read any copy of the replicated files and to write to all copies of the replicated files.
  - Requires lock to perform the write operations.
- Available-Copies Protocol
  - Allows to write on "**available**" copies.

Multicopy Update Problem (1)

- Read-Only Replication
  - Allows the replication of only immutable files.
  - Is suitable for frequently read and modified only once in a while, such as object codes of the system programs and reverse-indexed web search database.
- Read-Any-Write-All Protocol
  - Allows the replication of both immutable and mutable files.
  - Requires to read any copy of the replicated files and to write to all copies of the replicated files.
  - Requires lock to perform the write operations.
- Available-Copies Protocol
  - Allows to write on "**available**" copies.
  - Does not prevent inconsistence in the presence of communication failure.

Multicopy Update Problem (2)

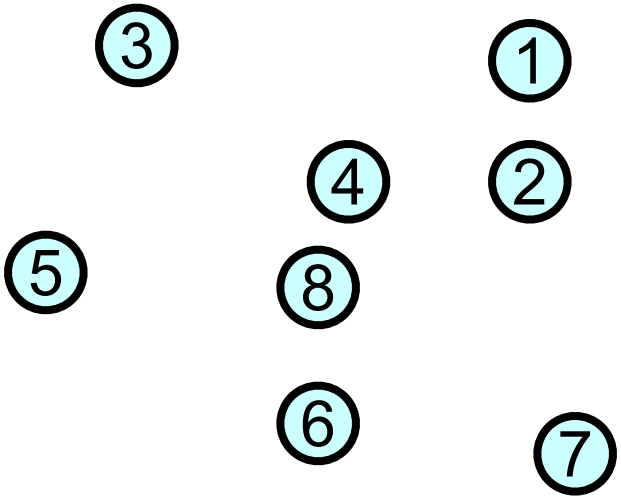
Multicopy Update Problem (2)

- Primary-Copy Protocol
  - Each file has one primary copy and several secondary copies.
  - Write operations are done on primary copy; Secondary copies are updated later.

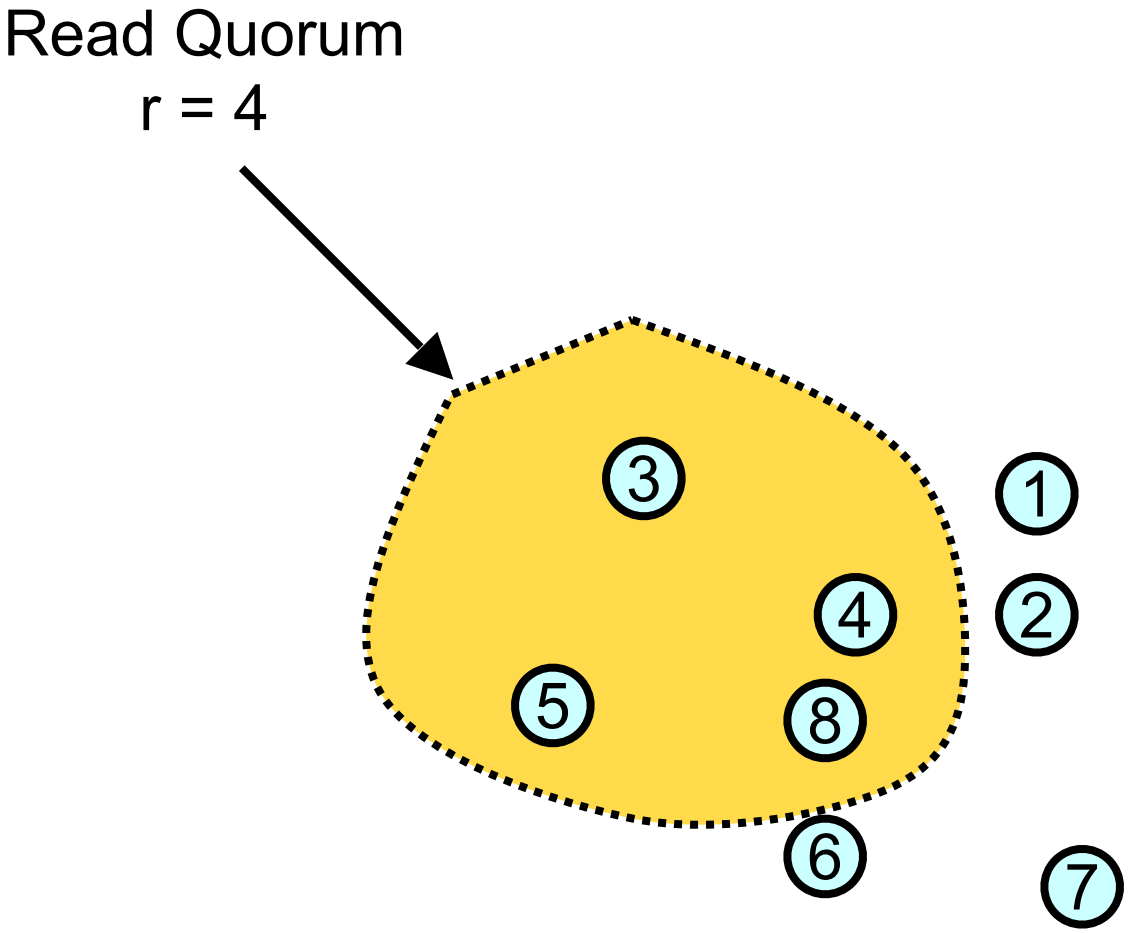
Multicopy Update Problem (2)

- Primary-Copy Protocol
  - Each file has one primary copy and several secondary copies.
  - Write operations are done on primary copy; Secondary copies are updated later.
- Quorum-Based Protocol:
  - Read-any-write-all and available-copies suffer from the network partition problem.
  - We may increase the availability at the expense of read operations.

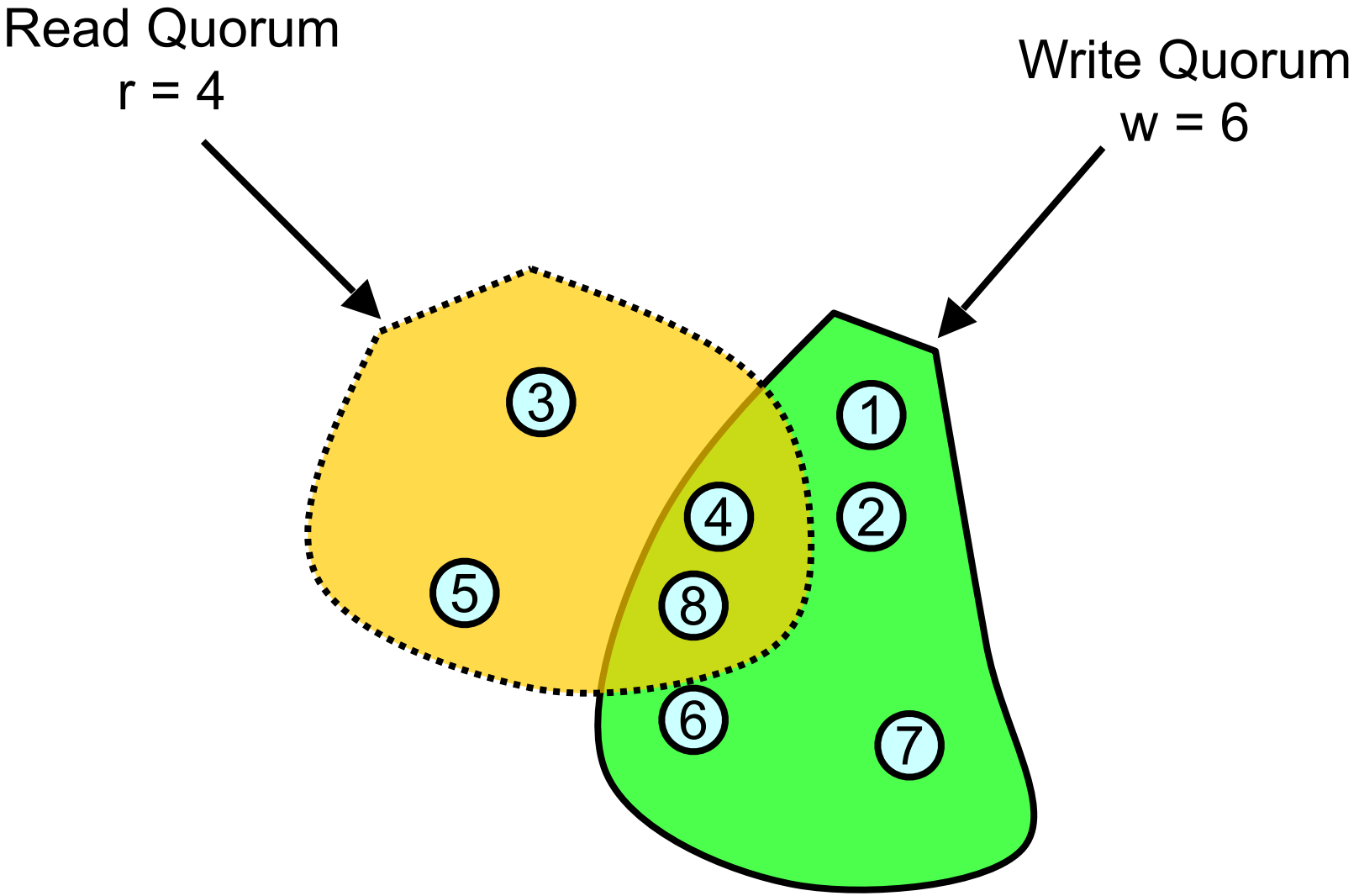
Quorum-Based Protocol



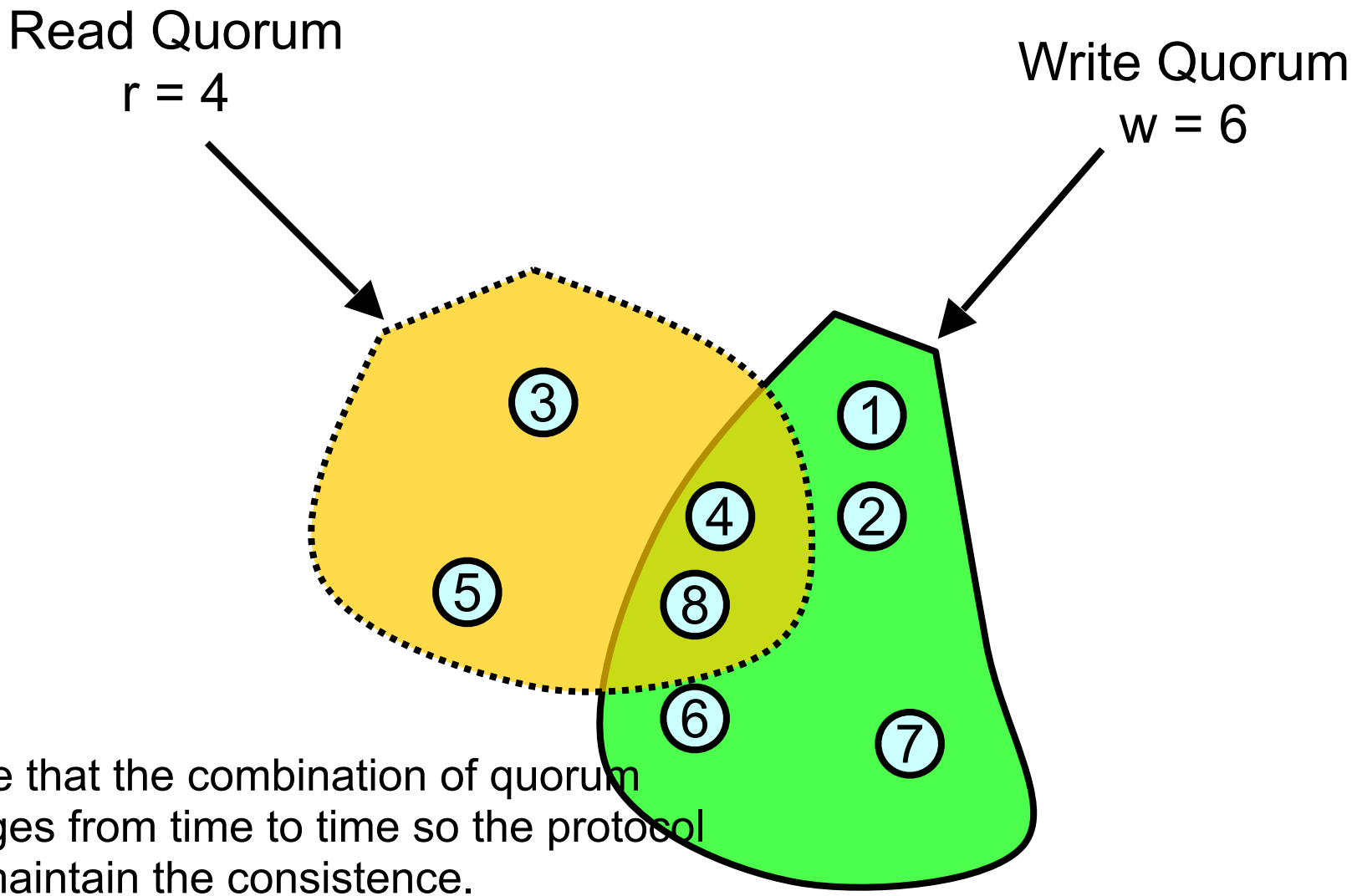
Quorum-Based Protocol



Quorum-Based Protocol

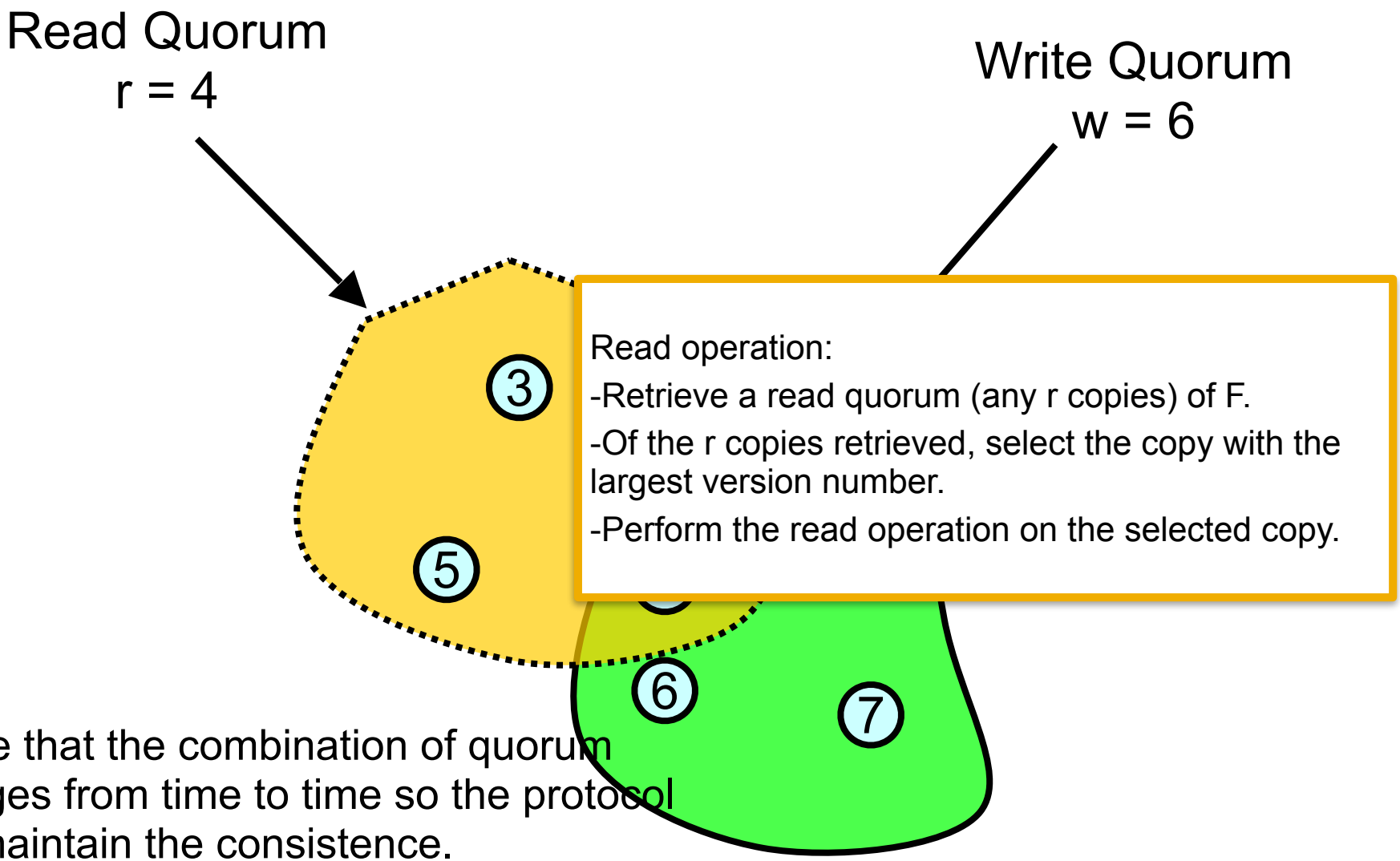


Quorum-Based Protocol



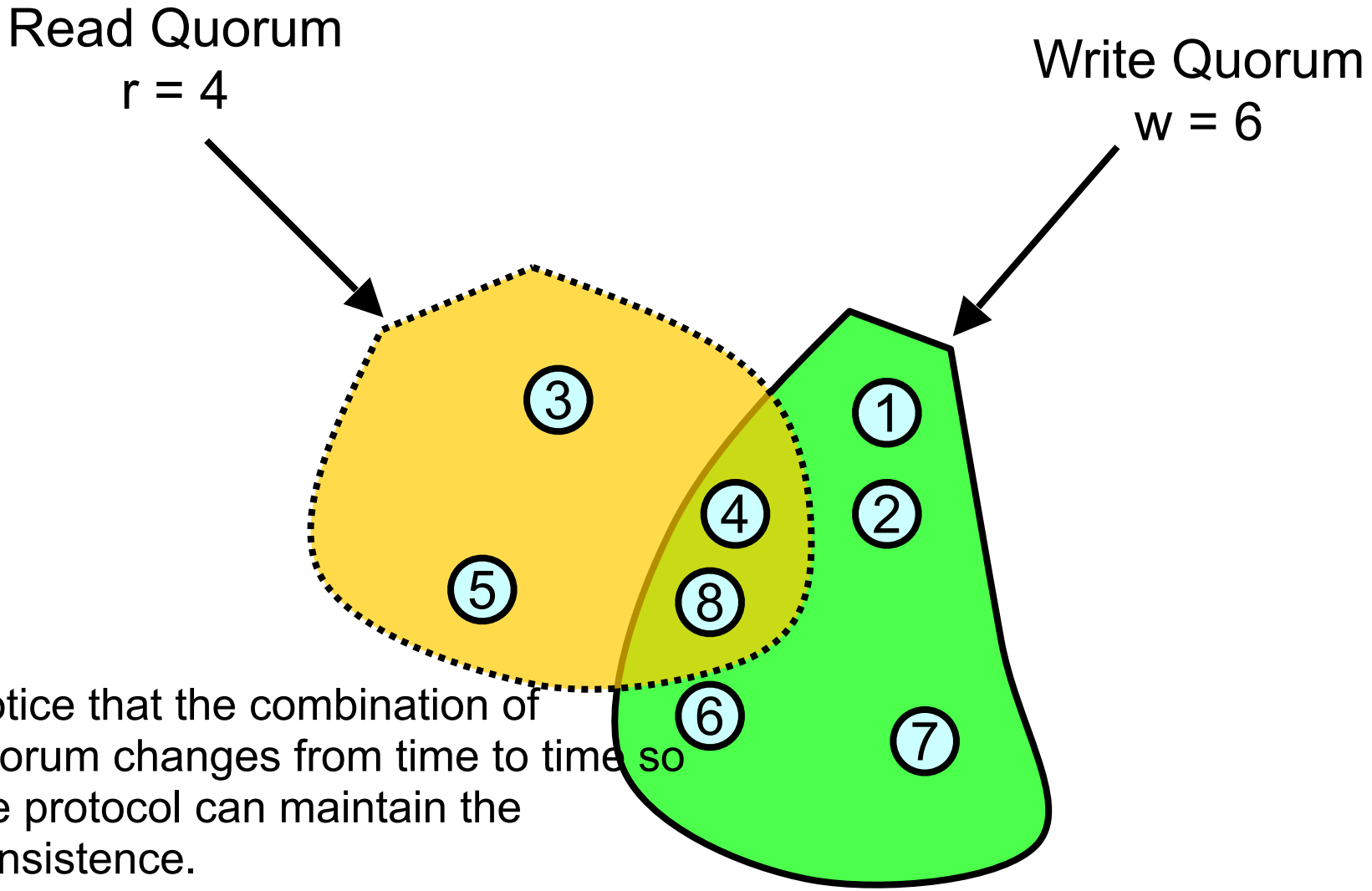
Notice that the combination of quorum changes from time to time so the protocol can maintain the consistence.

Quorum-Based Protocol



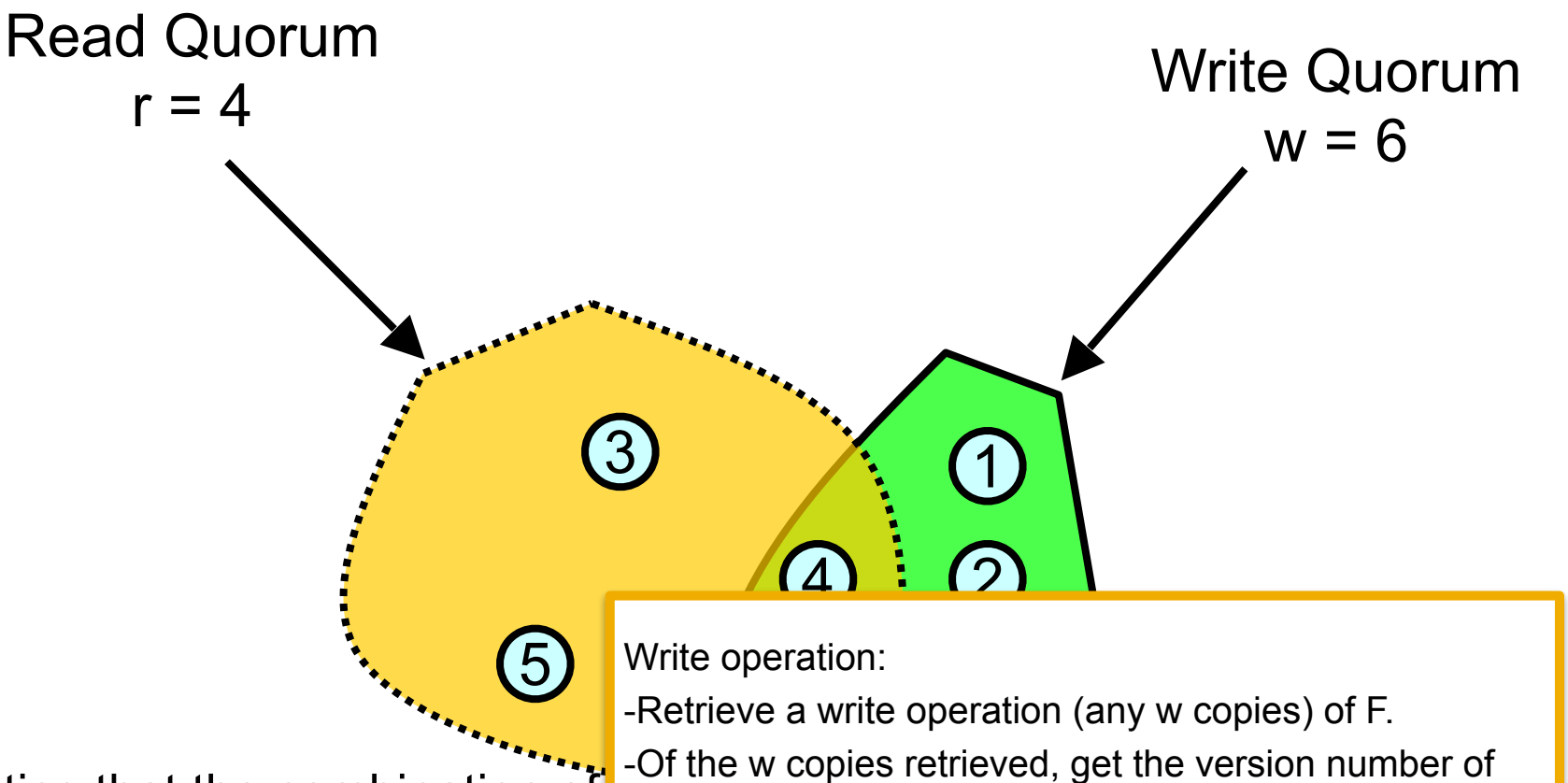
Notice that the combination of quorum changes from time to time so the protocol can maintain the consistence.

Quorum-Based Protocol



Notice that the combination of quorum changes from time to time so the protocol can maintain the consistence.

Quorum-Based Protocol



Notice that the combination of
quorum changes from time to
the protocol can maintain the
consistence.

Write operation:

-Retrieve a write operation (any w copies) of F.

-Of the w copies retrieved, get the version number of the copy with the largest version number.

-Increment the version number.

-Write the new value and the new version number to all the w copies of the write quorum.

Fault Tolerance

- What kind of failures are subtle to distributed file systems?
  - A server loses the contents of its main memory in the event of a crash.
  - During a request processing, the server or client may crash, resulting in the loss of state information of the file being accessed.
  - Transient faults caused by electromagnetic fluctuations
  - Decay of disk storage device

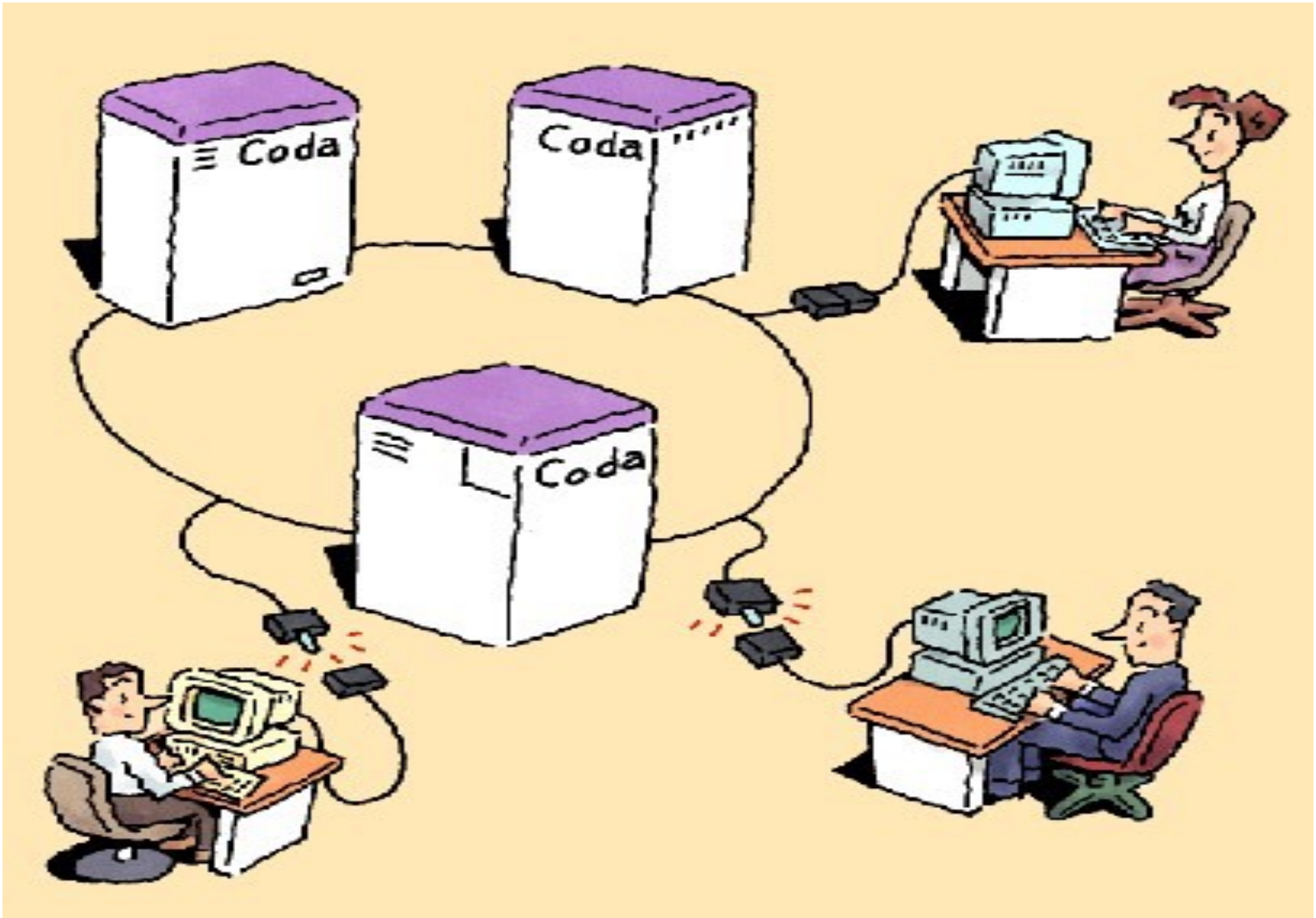
Effect of Service Paradigm on Fault Tolerance
Stateful File Servers vs. Stateless File Servers

- Stateful IO Functions:
  - `fid = Open( filename, mode )`
  - `read( fid, n, buffer ), write(fid, n, buffer), seek( fid, position), and`
  - `close( fid ).`
- Stateless IO Functions:
  - `read( filename, position, buffer )`
  - `write( filename, position, buffer)`
- State information shall be maintained for a certain amount of time, which is called session.
- A session starts with `open()` and ends with `close()`.
- Stateful service requires complex crash recovery procedures.

But, there is no free lunch.

- Stateless services have the following constraints:
  - Each file should have a system-wide low-level name associated with it.
  - Operations including read, write and delete files on stateless servers have to be **idempotent** to protect the server from duplicate requests.
- Stateless services suffer from
  - longer request messages and
  - slower processing of requests.
- In some cases, stateful service becomes necessary.
  - The packets transmitted over the network may be received out of its sending order.
  - State info will be useful to maintain the correct order.

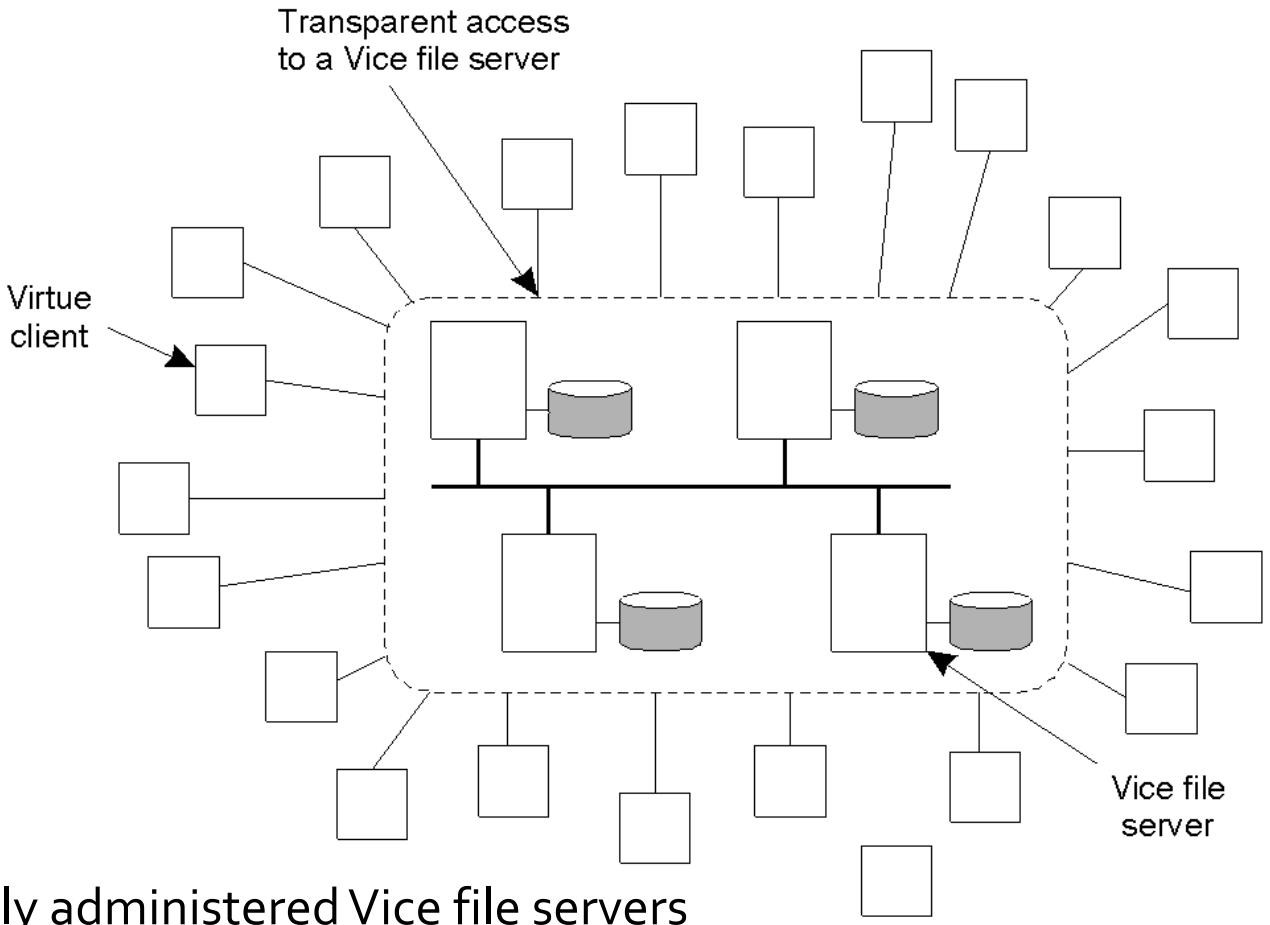
Case Study: CODA File System



Coda

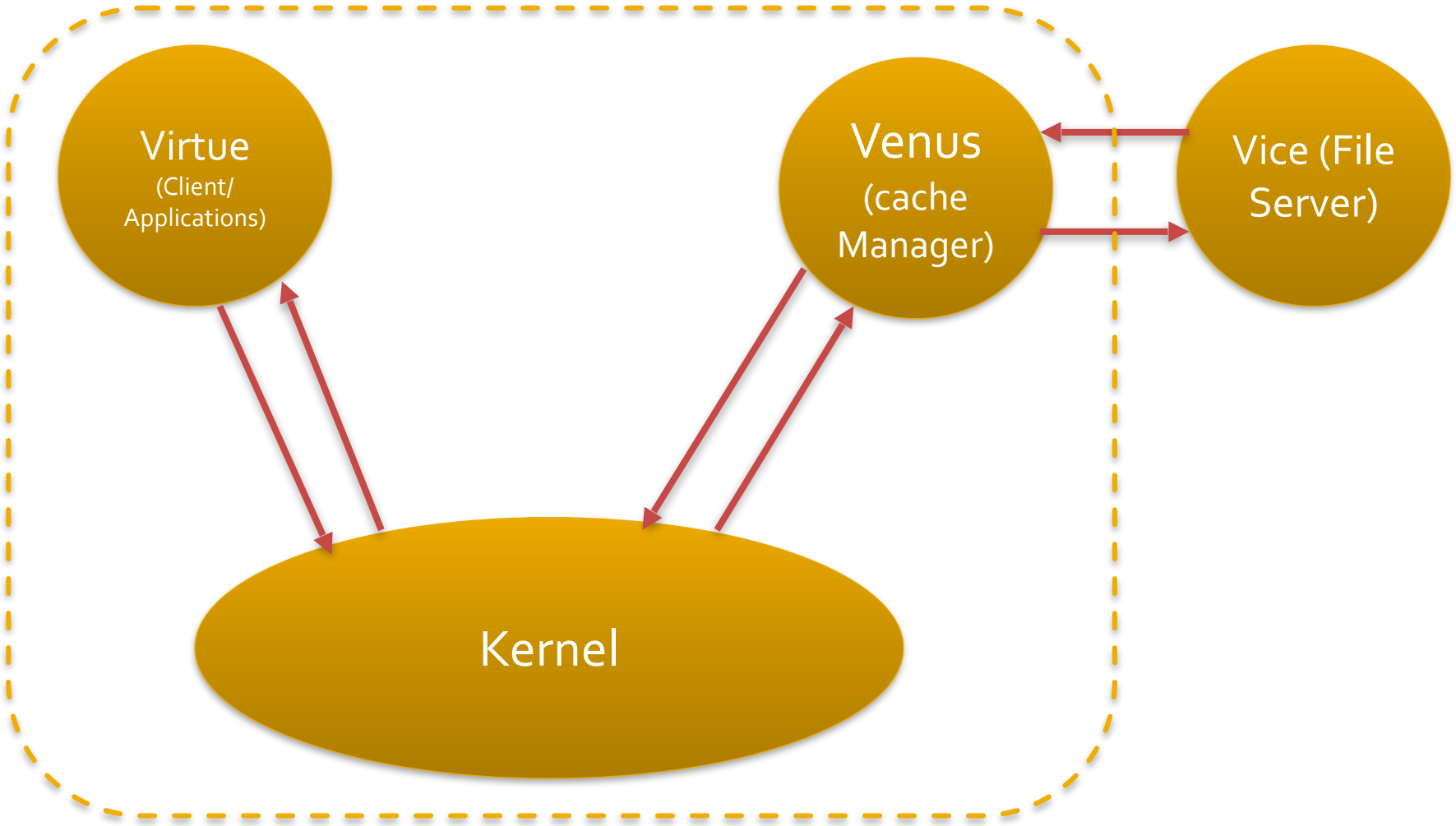
- Coda: descendent of the Andrew file system at CMU
  - Andrew designed to serve a large (global) community.
  - Started in 1987, and the last change was in 2011.
- Salient features:
  - Support for disconnected operations
    - Desirable for mobile users
  - Support for a **large** number of users
  - **Not** support for
    - highly concurrent and
    - fine granularity data access.

Overview of Coda

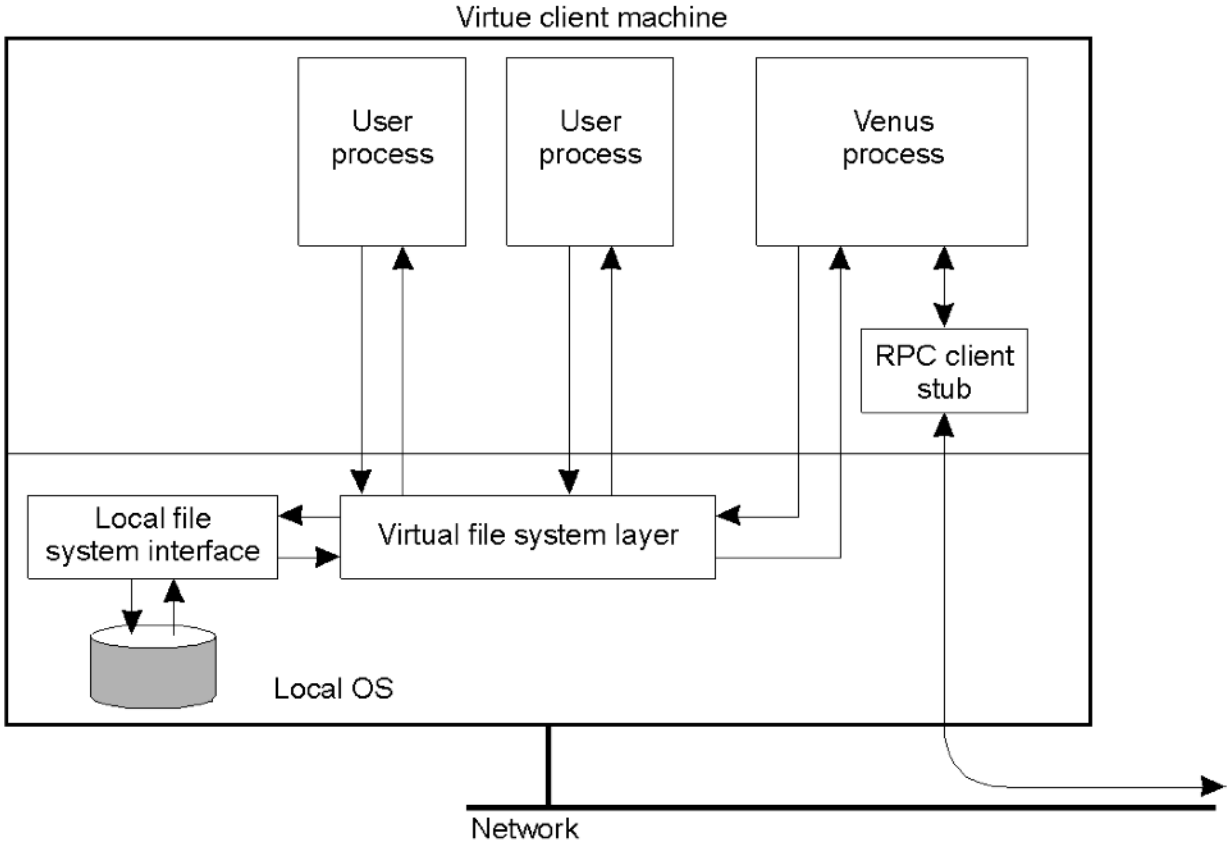


- Centrally administered Vice file servers
- Large number of virtue clients

Client/Venus/Vice in Coda

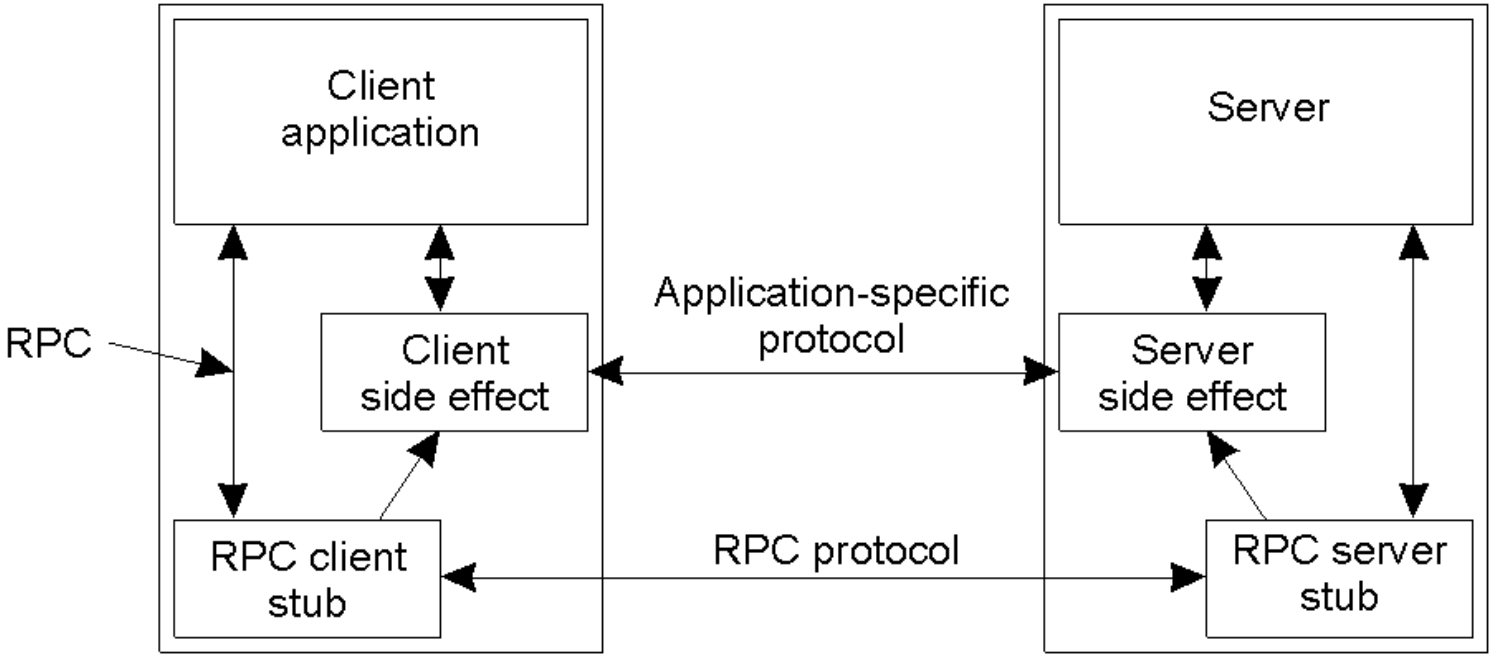


Virtue: Coda Clients



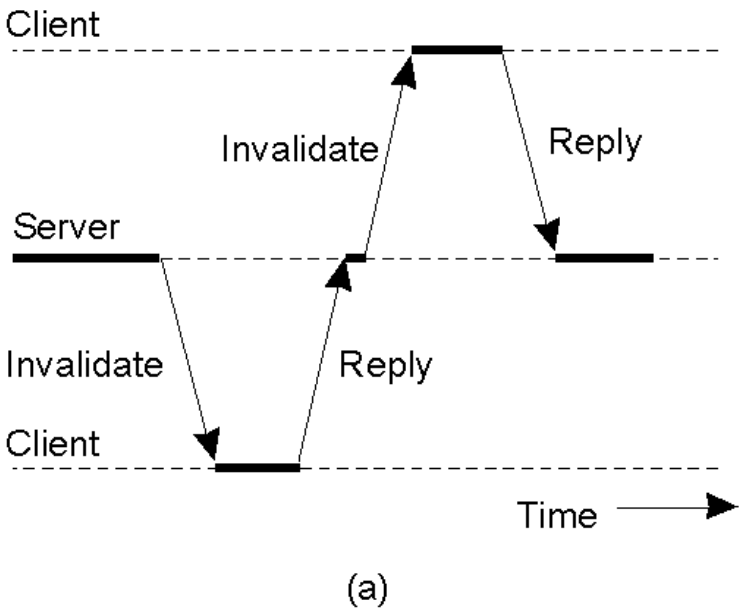
- The internal organization of a Virtue workstation.
  - Designed to allow access to files even if server is unavailable
  - Uses VFS and appears like a traditional Unix file system

Communication in Coda

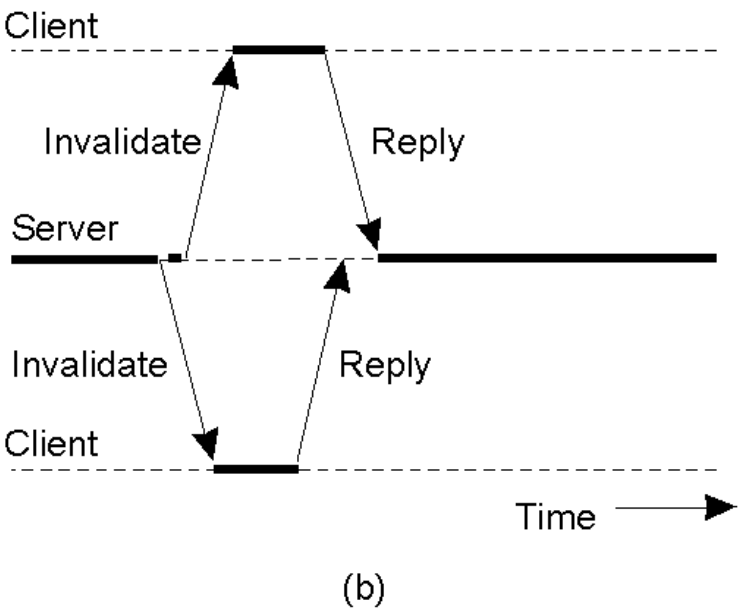


- Coda uses RPC2: a sophisticated *reliable* RPC system
  - Start a new thread for each request, server periodically informs client it is still working on the request.
- RPC2 supports *side-effects:* application-specific protocols
  - Useful for video streaming [where RPCs are less useful]
- RPC2 also has multicast support.

Communication: Invalidations



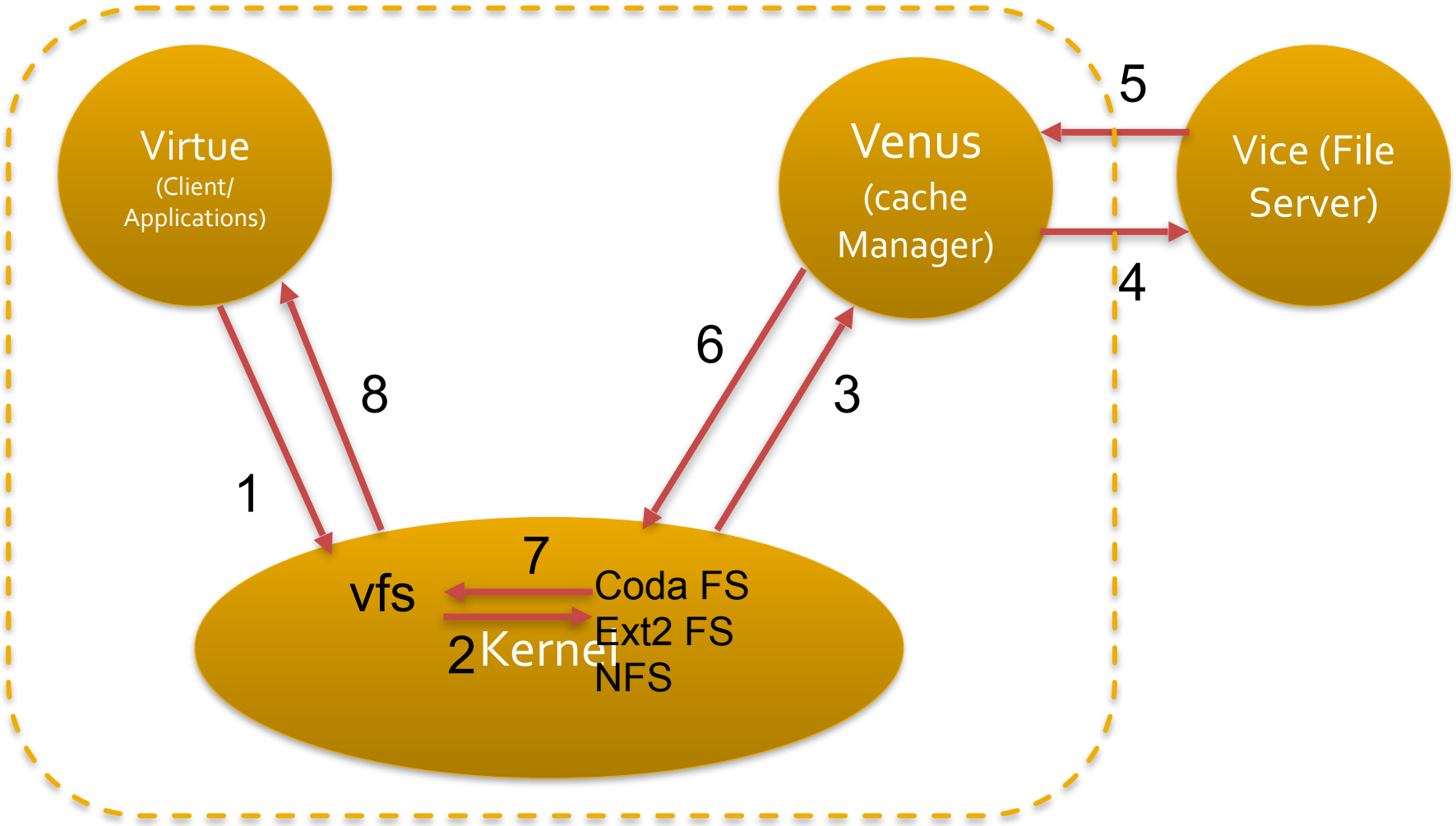
(a)



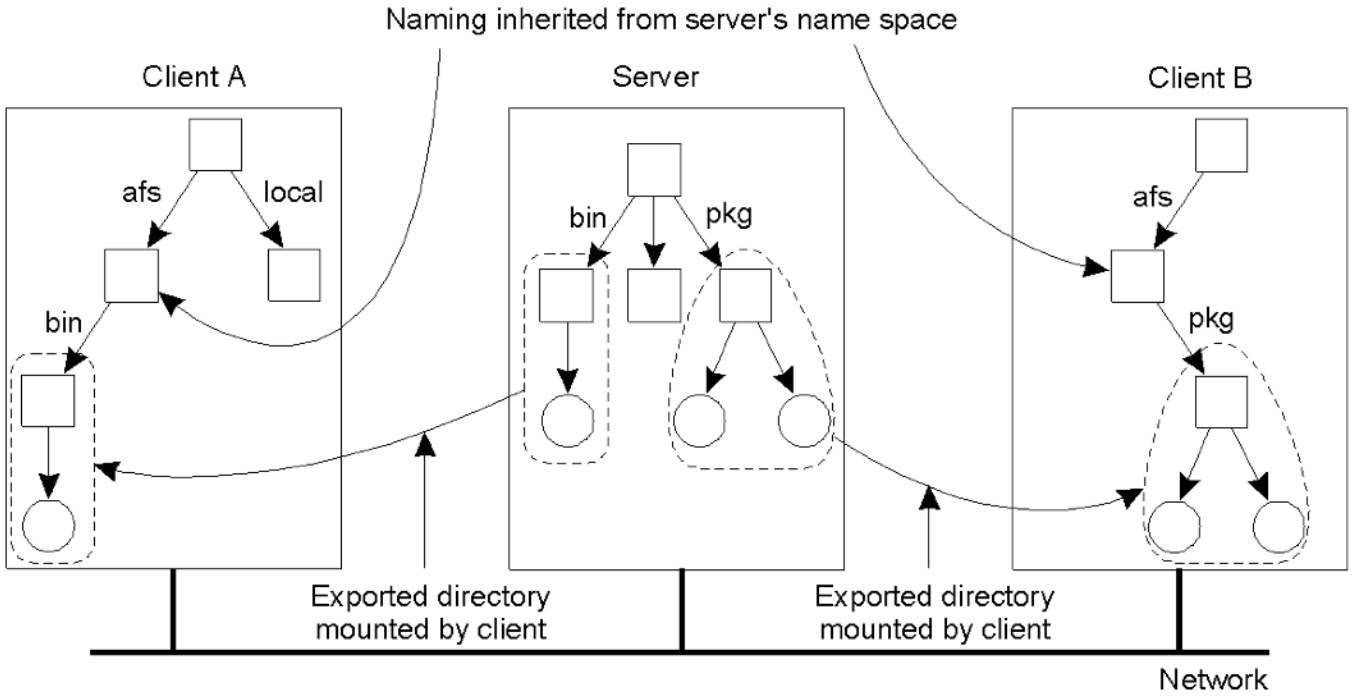
(b)

a) Sending an invalidation message one at a time using traditional RPC.
b) Sending invalidation messages in parallel using RPC2.

Client/Venus/Vice in Coda

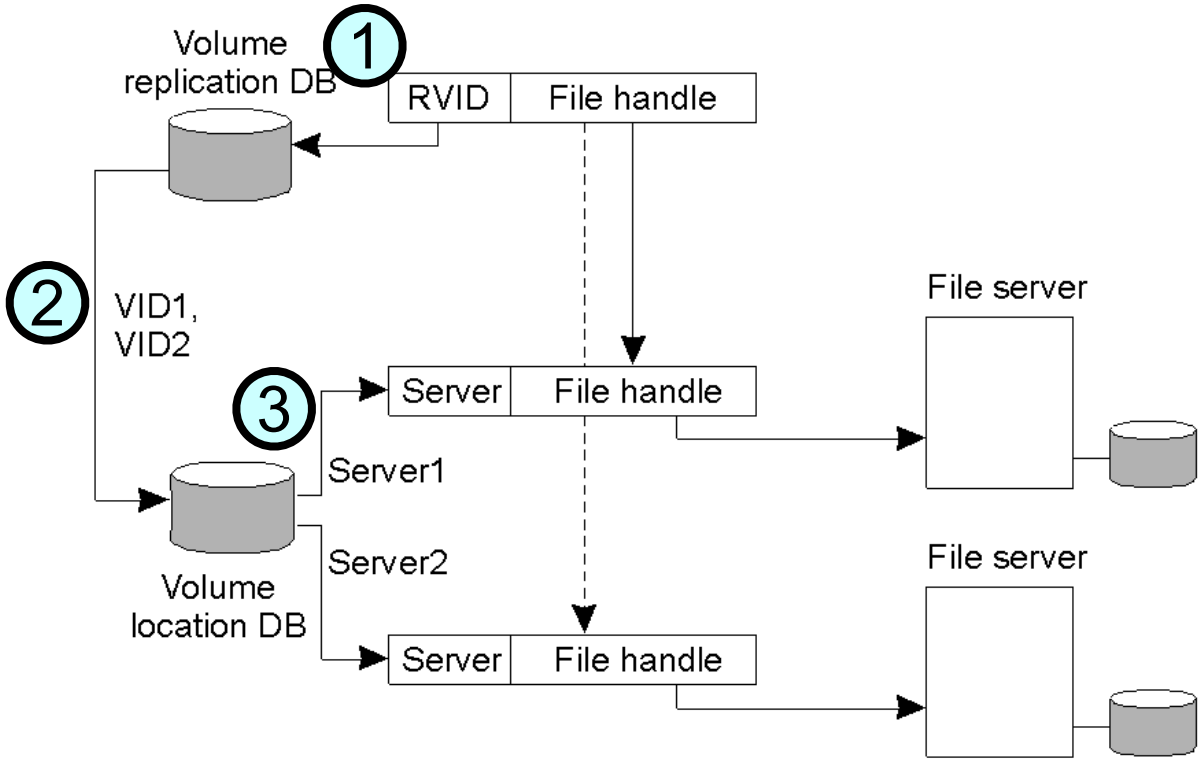


Naming



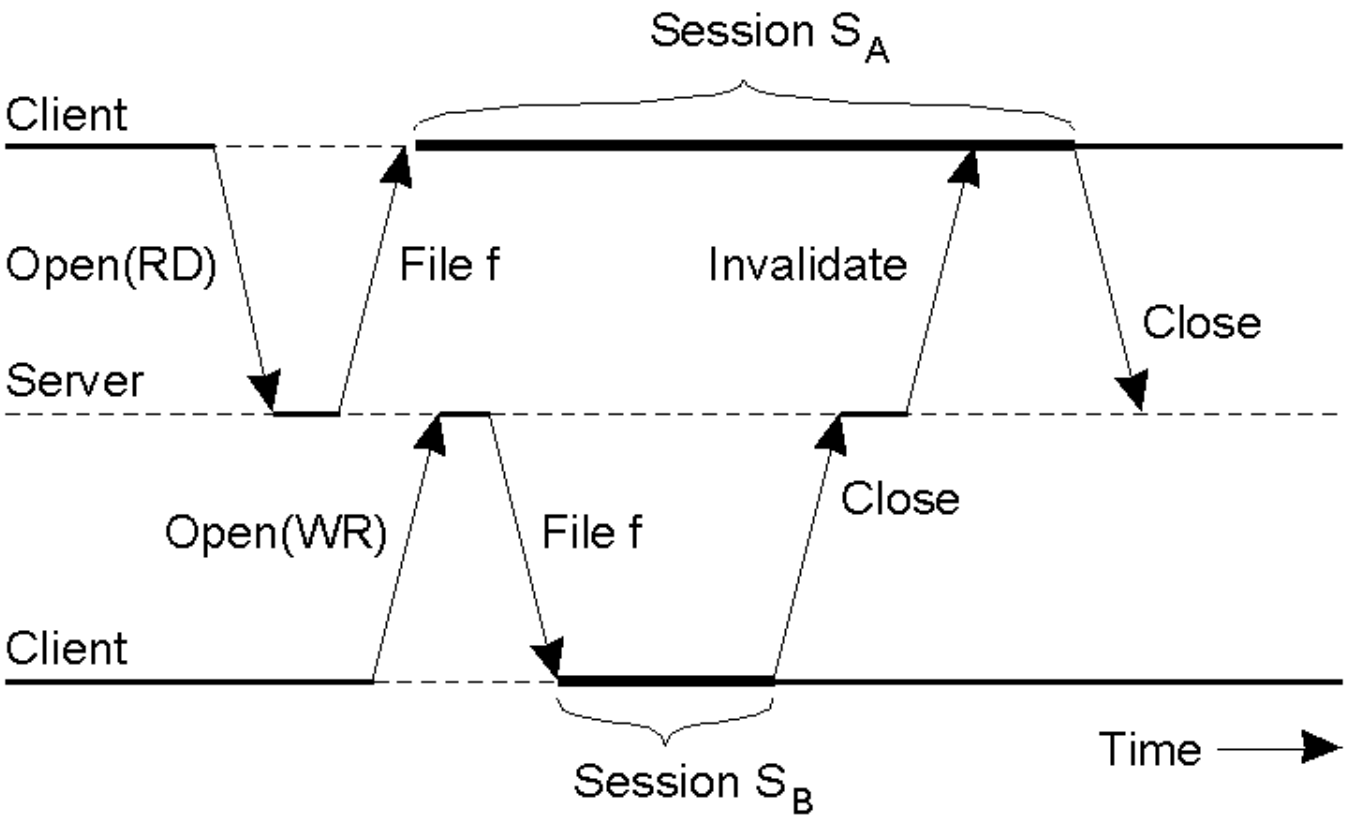
- Clients in Coda have access to a single shared name space
- Files are grouped into *volumes* [partial sub-tree in the directory structure]
  - Volume is the basic unit of mounting
  - Namespace: /afs/filesrv.csie.ntu.edu.tw [same namespace on all clients]
  - Name lookup can cross mount points: support for detecting crossing and automounts
- Volumes is the unit for server-side replication.

File Identifiers



- Each file in Coda belongs to exactly one volume
  - Volume may be replicated across several servers
  - Multiple logical (replicated) volumes map to the same physical volume
  - 96 bit file identifier = 32 bit RVID + 64 bit file handle

Sharing Files in Coda



- Transactional behavior for sharing files: similar to share reservations in NFS
  - File open: transfer entire file to client machine [similar to delegation]
  - Uses session semantics: each session is like a transaction
    - Updates are sent back to the server only when the file is closed

Transactional Semantics

- Network partition: part of network isolated from rest
  - Allow conflicting operations on replicas across file partitions
  - Reconcile upon reconnection
  - Transactional semantics => operations must be serializable
  - Ensure that operations were serializable *after they have executed*
  - Conflict => force manual reconciliation
- Knowing the metadata for each session makes it easier for CODA to recognize the conflicts.

File-associated data	Read?	Modified?
File identifier	**Yes**	**No**
Access rights	**Yes**	**No**
Last modification time	**Yes**	**Yes**
File length	**Yes**	**Yes**
File contents	**Yes**	**Yes**

Metadata for *store* session in Coda

Serializability

- A schedule is serial if the actions of the different transactions are not interleaved; they are executed one after another
- A schedule is serializable if its effect is the same as that of some serial schedule

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

Transactions T1, T2, and T3

Schedule 1	**x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3**	?
Schedule 2	**x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;**	?
Schedule 3	**x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;**	?

Possible schedules

Conflicts Across Partitions

- Two important observations:
  - A Venus process knows which data to fetch from the server at the start of a session. So, it can acquire the necessary locks at the start of a session.
  - Two-phase locking (2PL) is used. So, all result schedules are serializable.

Two-Phase Locking

If anything will go wrong, it will.

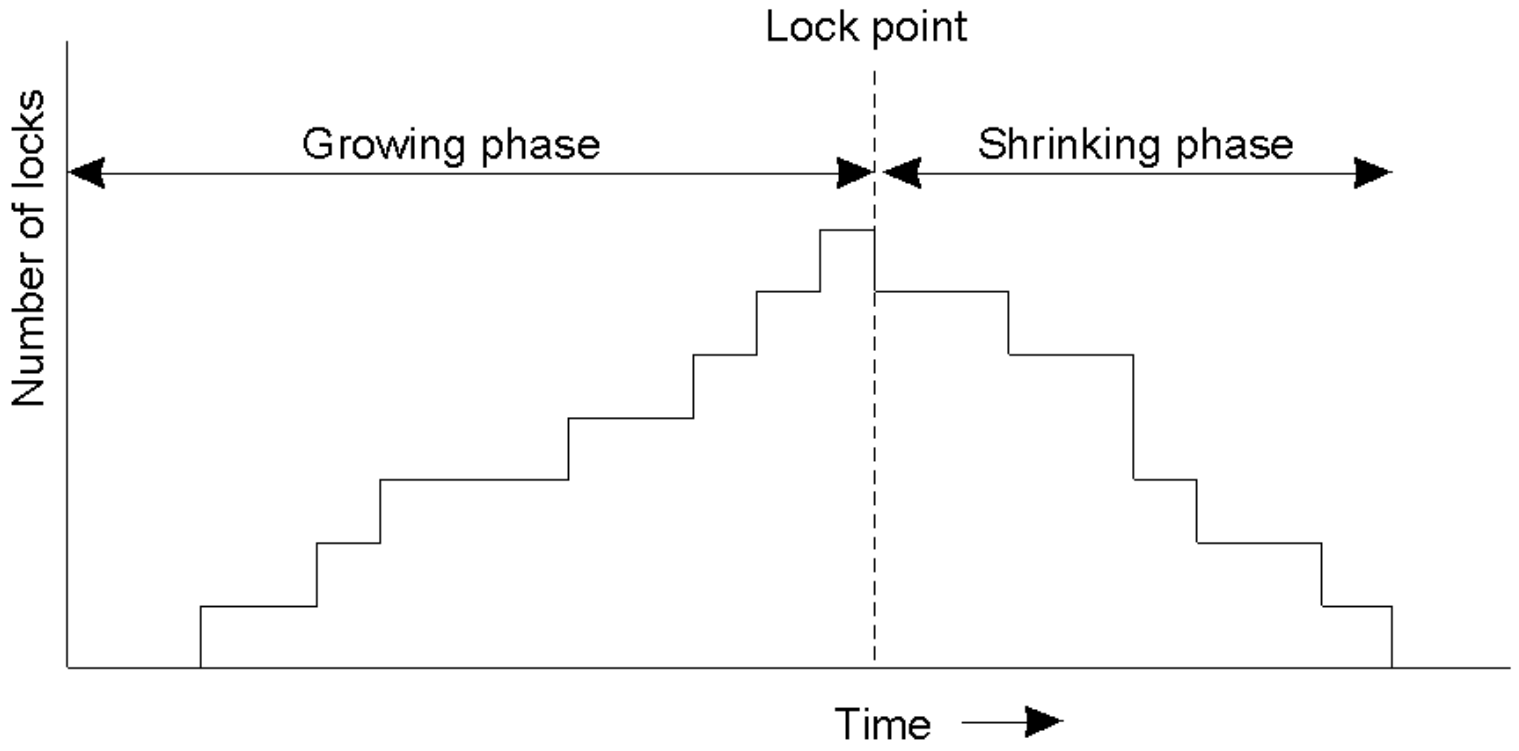
-- Murphy's Laws

In nature, nothing is ever right. Therefore, if everything is going right ... something is wrong.

- Lock the data when read/write.
- (Non-Strict) Two-Phase Locking:
  - If a transaction T wants to read/write an object, it must request a shared/exclusive lock on the object.
  - A transaction cannot request additional locks on an object once it releases any lock, and it can release locks at any time.

Two-Phase Locking

- Two-phase locking.



Testing for Serializability: Serialization Graphs

- Input: Schedule *S* for set of transactions $T_1, T_2, \ldots, T_k$.
- Output: Determination whether *S* is serializable.
- Method:
  - Create *serialization graph G*:
    - Nodes: correspond to transactions
    - Arcs: G has an arc from T_i to T_j if there is a T_i:*UNLOCK*(A_m) operation followed by a T_j:*LOCK*(A_m) operation in the schedule.
  - Perform topological sorting of the graph.
    - If graph has cycles, then *S* is not serializable.
    - If graph has no cycles, then topological order is a serial order for transactions.

Testing for Serializability: Serialization Graphs

- Input: Schedule *S* for se
- Output: Determination
- Method:
  - Create *serialization grapl*
    - Nodes: correspond to trans
    - Arcs: G has an arc from T_i t
      followed by a T_j:*LOCK*(A_m)
  - Perform topological sorting of the graph.
    - If graph has cycles, then *S* is not serializable.
    - If graph has no cycles, then topological order is a serial order for transactions.

A **topological sort** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

Serializability of Two-Phase Locking

- Theorem: If *S* is any schedule of two-phase transactions, then *S* is serializable.
- Proof:
  - Suppose not. Then the serialization graph *G* for *S* has a cycle, $T_{i1} \rightarrow T_{i2} \rightarrow \ldots \rightarrow T_{ip} \rightarrow T_{i1}$
  - Therefore, a lock by T_{i1} follows an unlock by T_{i1}, contradicting the assumption that T_{i1} is two-phase.

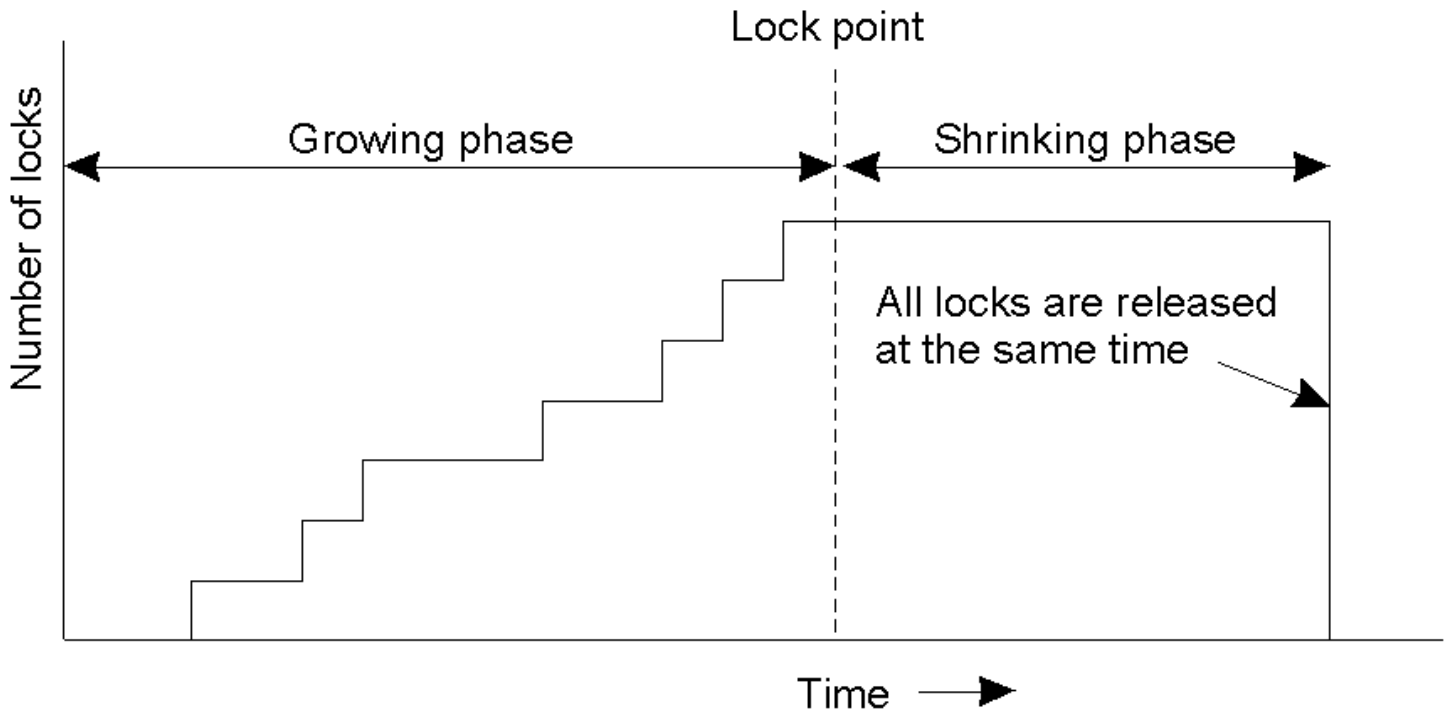
Transactions that read 'dirty ' data

	T_1	T_2
(1)	LOCK A	
(2)	READ A	
(3)	A:=A-1	
(4)	WRITE A	
(5)	LOCK B	
(6)	UNLOCK A	
(7)		LOCK A
(8)		READ A
(9)		A:=A*2
(10)	READ B	
(11)		WRITE A
(12)		COMMIT
(13)		UNLOCK A
(14)	B:=B/A	

- Assume that *T1* fails after (13).
  - *T1* still holds lock on B.
  - Value read by *T2* at step (8) is wrong.
- *T2* must be rolled back and restarted.
  - Some transaction *T3* may have read value of A between steps (13) and (14)

Strong Strict Two-Phase Locking

- Transaction may be aborted because of conflicts during shrinking phase
- Strong Strict two-phase locking: locks are released after the transactions are *committed*.



Prove serializability with SS2PL

- A serializable schedule contain no read/write and write/write conflicts in the schedule.
- 2PL assures that at most one transaction can write on one data item.
- Proof by contradiction:
  - Suppose there is a unserializable schedule following SS2PL.
  - There must be a read/write or write/write conflict in the schedule.
  - However, SS2PS assures that there is at most one transaction to read or write a data item.
  - The assumption is contradicted.

Conflicts Across Partitions

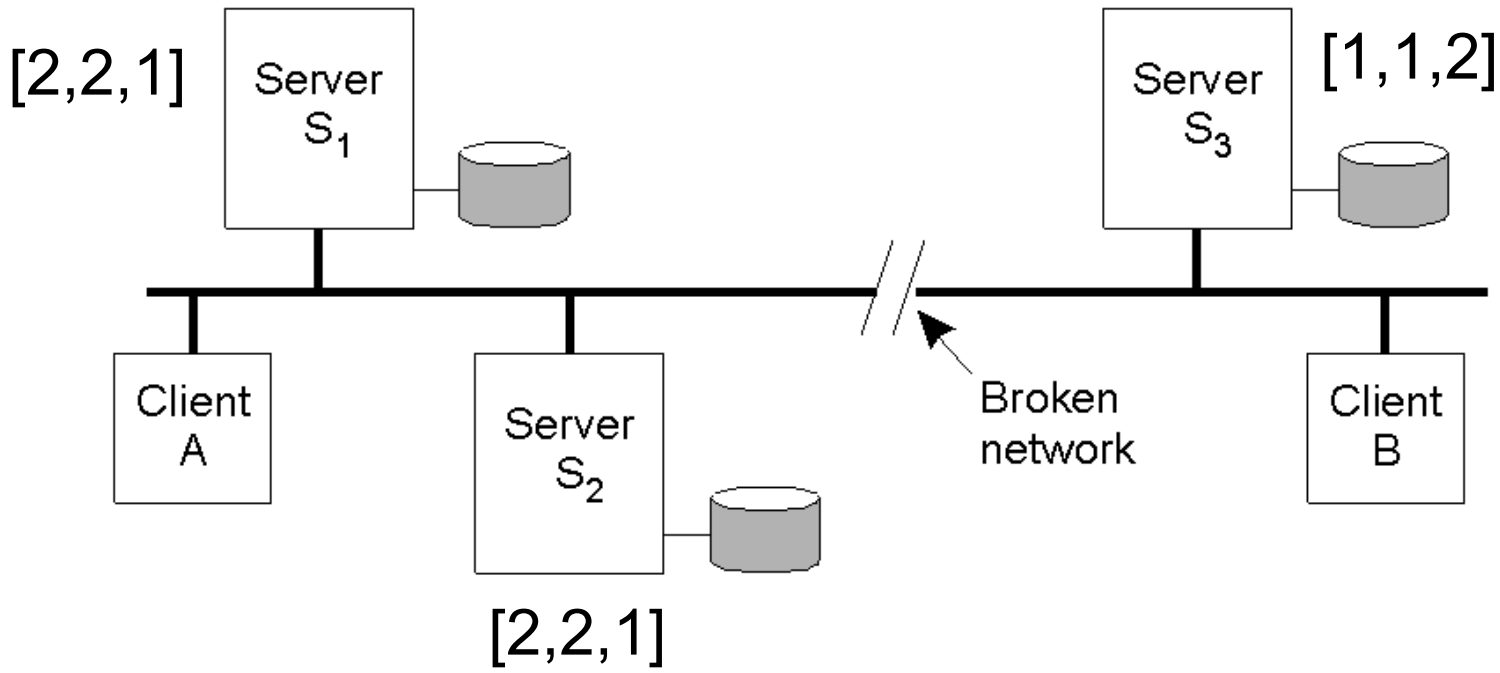
- A file version system is used to solve conflicts.
  - In a partition, the operations are executed as if nothing happened.
  - When reconnected, updates are transferred to the server in the same order as they took place at the client.
  - Version number is used to solve the conflicts.
  - A update is accepted if and only if

    Current version number + 1 = Last version number on the client + numbers of successful update during the session on the server.

Discussion: Replica Control

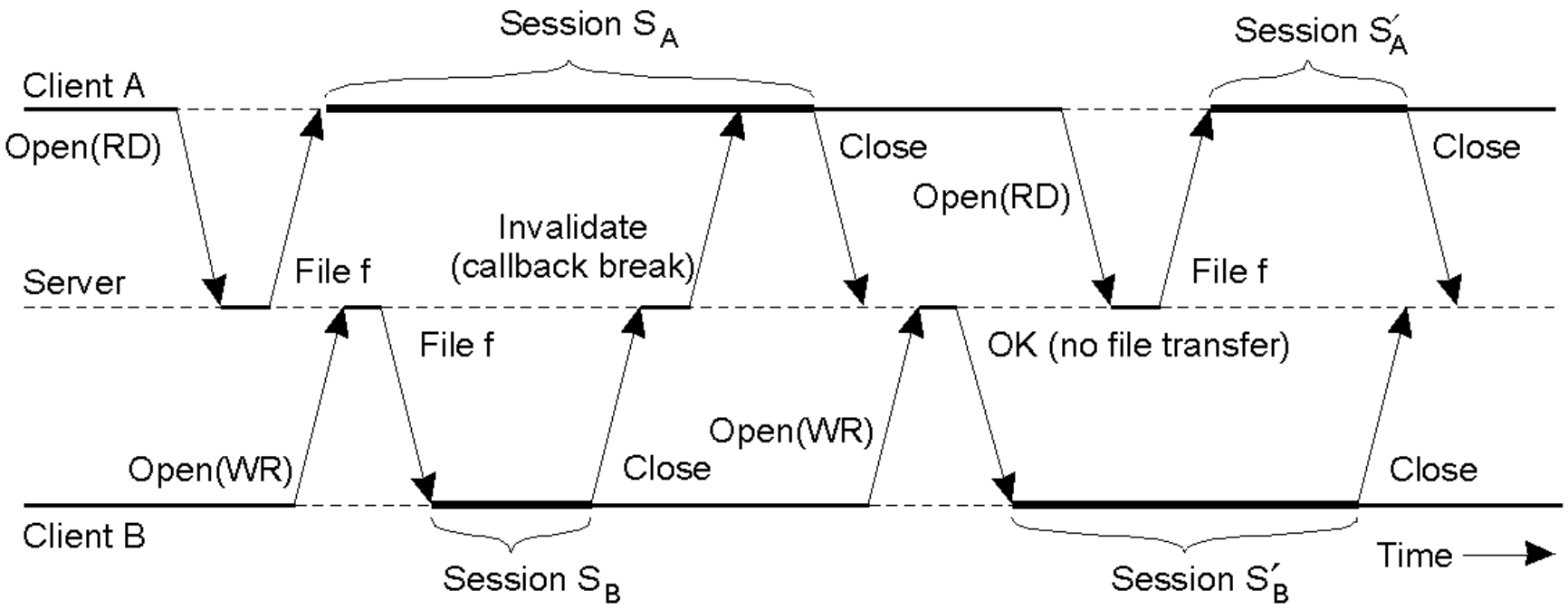
- Replica of file servers are kept to improve the availability of the system. However, the network may be partitioned into sub-networks from time to time.
- Pessimistic vs. Optimistic replica control
  - Pessimistic: make sure no one is accessing the file.
  - Optimistic: read/write the file and solve the conflict later.
- Which strategy is better for CODA?
  - Think about the assumption for CODA.

Server Replication



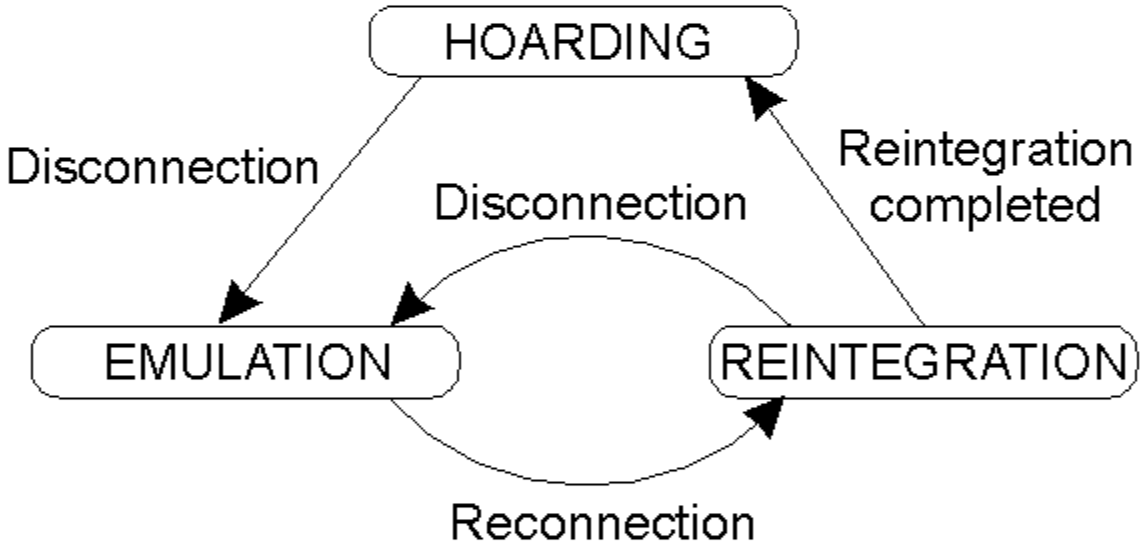
- Use replicated writes: read-once write-all
  - Writes are sent to all AVSG (all accessible replicas)
- How to handle network partitions?
  - Use optimistic strategy for replication
  - Detect conflicts using a Coda version vector
  - Example: [2,2,1] and [1,1,2] is a conflict => manual reconciliation

Client Caching



- Cache consistency maintained using callbacks
  - Server tracks all clients that have a copy of the file [provide *callback promise*]
  - Upon modification: send invalidate to clients
  - No file transfer is need when callback promise holds.

Disconnected Operation



- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection
  - Prefetch all files that may be accessed and cache (hoard) locally
  - If AVSG=0, go to emulation mode and reintegrate upon reconnection

Caching Management

- Cache space is finite and CODA caches entire file not parts of the files.
  - Caching parts of the files is difficult to support disconnected services.
- Prioritized cache management:
  - The users specify the priority of the files and directories to construct hoard database (HDB).
  - Priority also changes from time to time based on use history.
  - Less critical files are removed.
  - *Hierarchical cache management* allows the system to resolve the pathname of a cached object while disconnected.
- Hoarding walk:
  - A cache is in equilibrium when no un-cached object has a higher priority than cached objects.
  - Venus periodically restore equilibrium by performing hoard walk.

BigTable and Google File Systems (GFS)

BigTable and Google File Systems (GFS)

- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages.
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (SOSP '03). ACM, New York, NY, USA, 29-43.
- Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (January 2008), 107-113.

Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands of q/sec
  - 100TB+ of satellite image data

How About Commercial DB?

- Scale is too large for most commercial databases
- Even if it weren't, cost would be very high
  - Building internally means system can be applied across many projects for low incremental cost
- Low-level storage optimizations help performance significantly
  - Much harder to do when running on top of a database layer

Goals

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time
- Need to support:
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets
- Often want to examine data changes over time
  - E.g. Contents of a web page over multiple crawls

BigTable: a distributed storage system

- BigTable is a distributed storage system for managing (semi-)structured data.
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Flexible, high-performance solution for all of Google's products

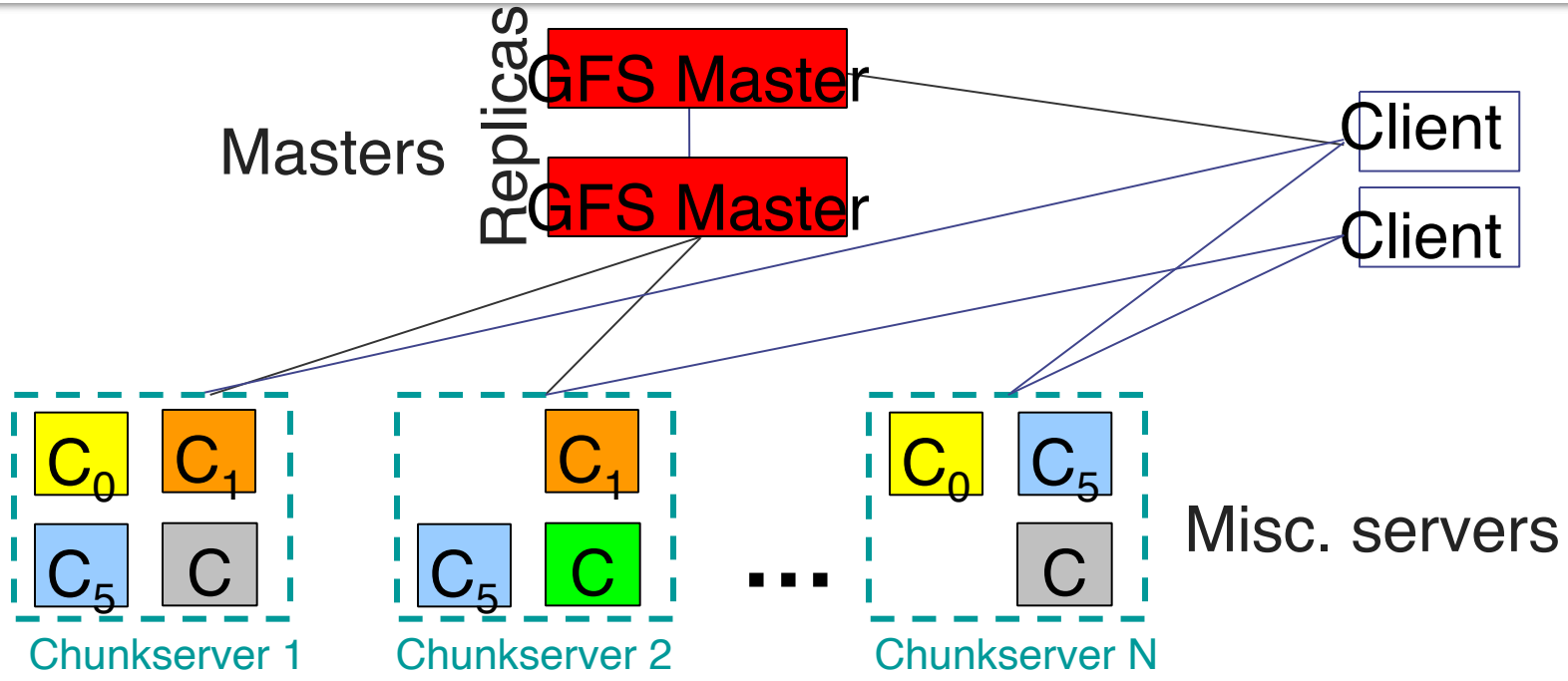
Features of BigTable

- Distributed multi-level map for MapReduce applications
- Fault-tolerant and persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

Building Blocks

- Building blocks:
  - GFS: raw storage to store persistent data (SSTable file format for storage of data)
  - Scheduler: schedules jobs on machines involved in BigTable serving
  - Chubby Lock service: master election, location bootstrapping, which is a distributed lock manager.
  - Map Reduce: simplified large-scale data processing and often used to read/write BigTable data

Google File System (GFS)



- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety

Google File System

- Large-scale distributed “file system”
- **Master**: responsible for metadata
- **Chunk servers**: responsible for reading and writing large chunks of data
  - Chunks replicated on 3 machines, master responsible for ensuring replicas exist
  - Chubby:
    - {lock/file/name} service
    - Coarse-grained locks, can store small amount of data in a lock.

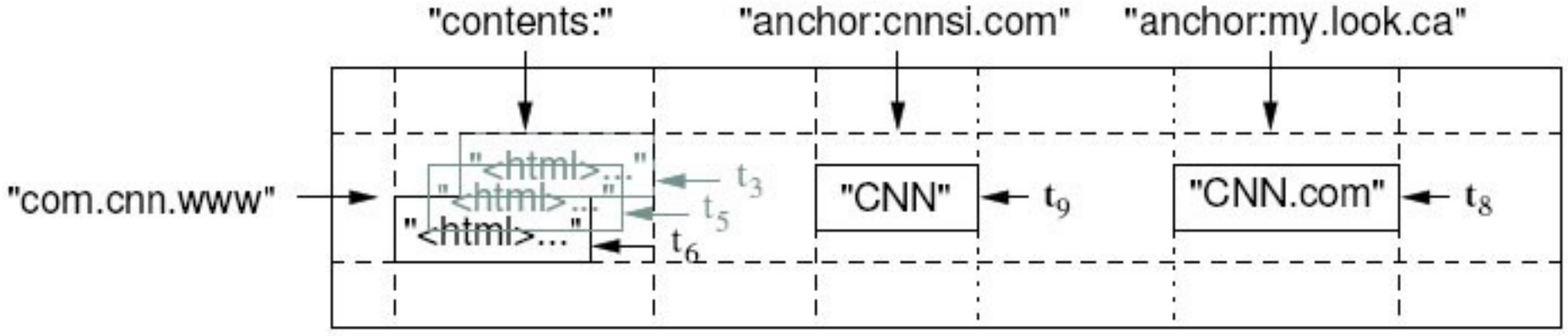
Chubby - Lock Services in GFS

- When there are number of replicas located at various locations, lock and consensus are mandatory.
  - 3 to 5 replicas, need a majority vote to be active
  - OSDI '06 Paper
- Link

Basic Data Model

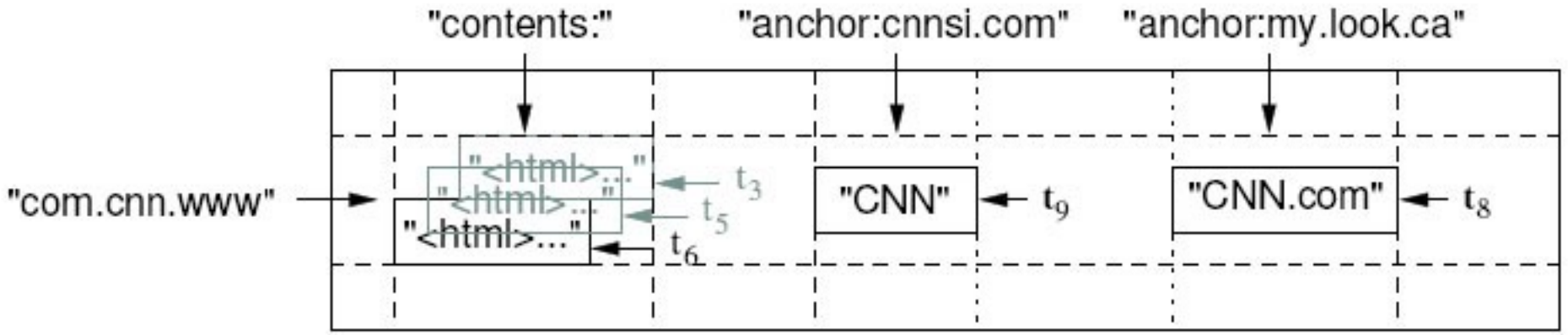
- A BigTable is a sparse, distributed persistent multi-dimensional sorted map

  (row, column, timestamp) -> cell contents



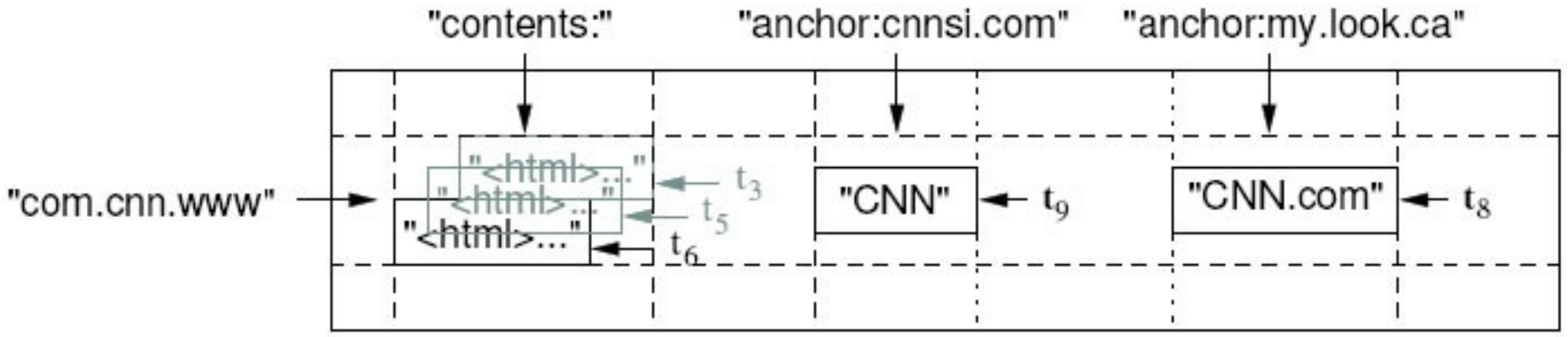
- Good match for most Google applications

WebTable Example



- Want to keep copy of a large collection of web pages and related information
- Use URLs as row keys
- Various aspects of web page as column names
- Store contents of web pages in the `contents:` column under the timestamps when they were fetched.

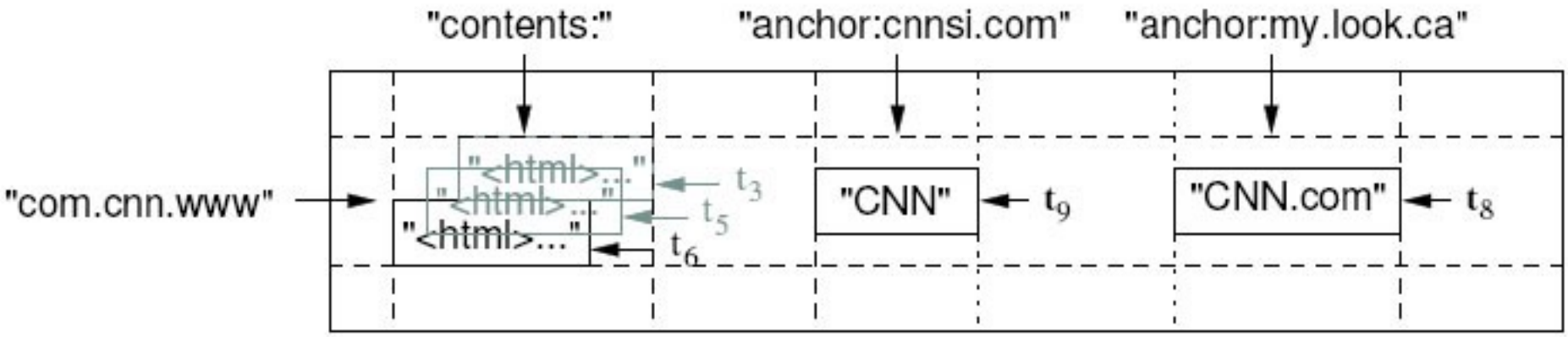
Rows



- Name is an arbitrary string
  - Access to data in a row is atomic
  - Row creation is implicit upon storing data
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines
- Example:

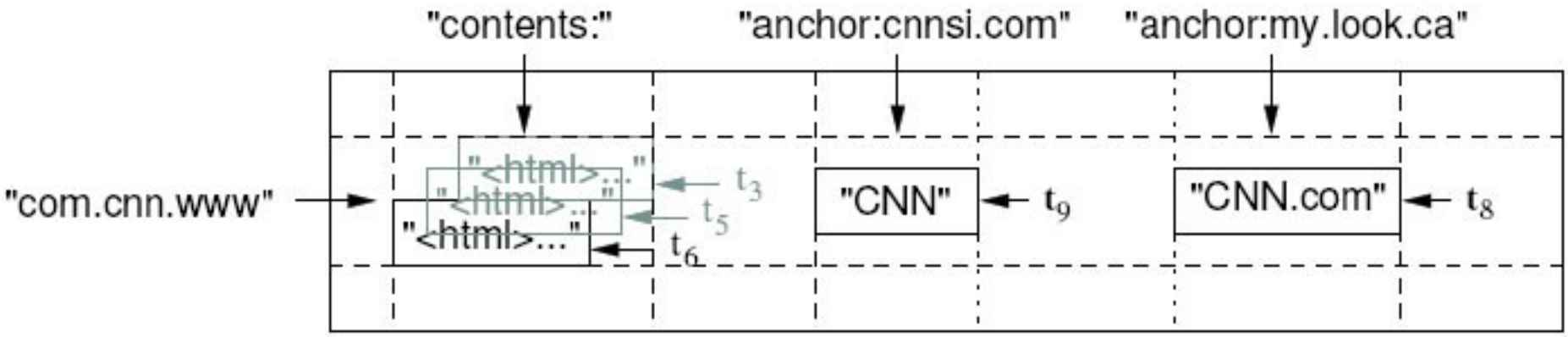
```
math.gatech.edu, math.uga.edu, phys.gatech.edu, phys.uga.edu
VS
edu.gatech.math, edu.gatech.phys, edu.uga.math, edu.uga.phys
```

Columns



- Columns have two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional levels of indexing, if desired

Timestamps



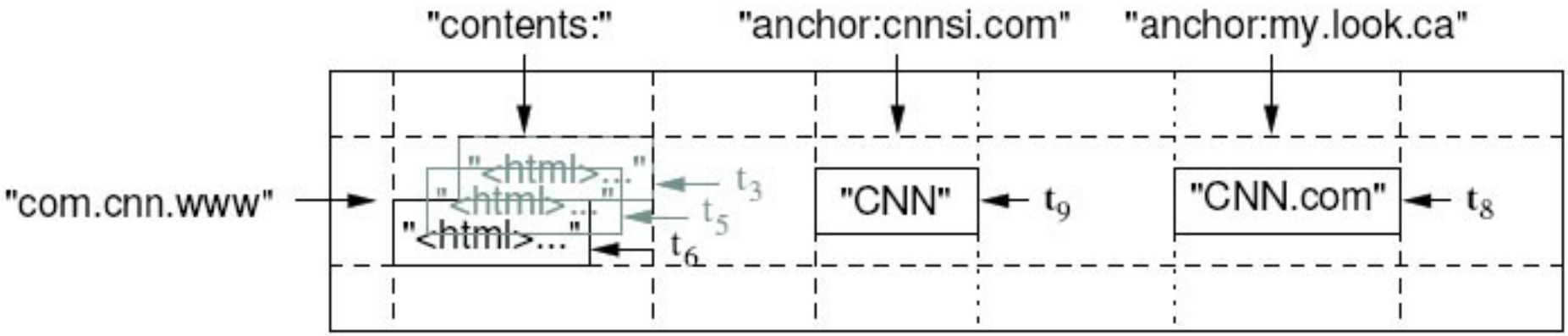
- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - *"Return most recent K values"*
  - *"Return all values in timestamp range (or all values)"*
- Column families can be marked w/ attributes:
  - *"Only retain most recent K values in a cell"*
  - *"Keep values until they are older than K seconds"*

Data Structure for BigTable

- BigTable is a large *map* that is indexed by a row key, column key, and a timestamp.
- A map is an associative array;
  - a data structure that allows one to look up a value to a corresponding key quickly.
  - BigTable is a collection of (key, value) pairs where the key identifies a row and the value is the set of columns.
  - Each value within the map is an array of bytes that is interpreted by the application

Multidimensional Table

- A table is indexed by rows.
- Each row contains one or more named column families.
- Within a column family, one may have one or more named columns.
- Time is another dimension in BigTable data. Every column family may keep multiple versions of column family data.




```
com.cnn.www" : {                    // row
        ”anchor" : {                                    // column family
        ”cnnsi.com": ”CNN",                             // column
        ”my.look.ca": ”CNN.com",                        // column
}
”contents" : {                                          // another column family
        "" : “<html>...</html>"                         // column (null name)
}
}
```

Adding Timestamp

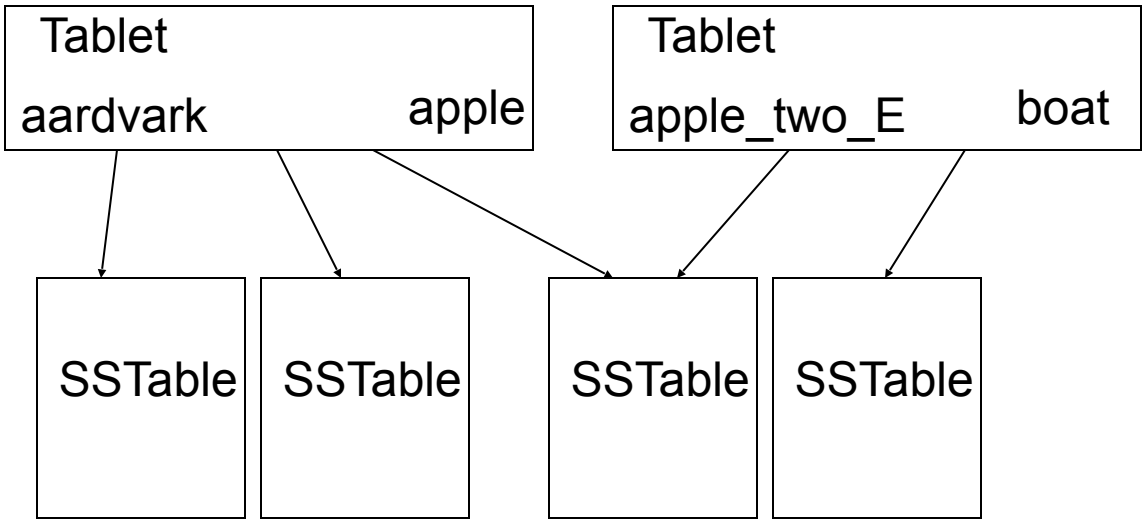
```
com.cnn.www" : {                    // row
        "anchor" : {                                    // column family
        "cnnsi.com": {
                "2010/01/01:00:00:00": "CNN",                   //
column
                "2012/04/01:00:00:00": "CNNSI",                 //
column
        "my.look.ca": "CNN.com",                        // column
    }
    "contents" : {
        "2010/01/01:00:00:00":{                         // another column family
        "" : "<html>…</html>"                           // column (null name)
        }
        "2012/04/01:00:00:00":{                         // another column family
        "" : "<html>…</html>"                           // column (null name)
        }
    }
}
```

Relational database vs. BigTable

Relational DB	BigTable
Database	BigTable
Table	Column Family
Primary Key	Row
B-Tree Node	Tablet
Transaction	Atomic Row Update
Schema	Column Family Schema

Table

- Multiple tablets make up the table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap

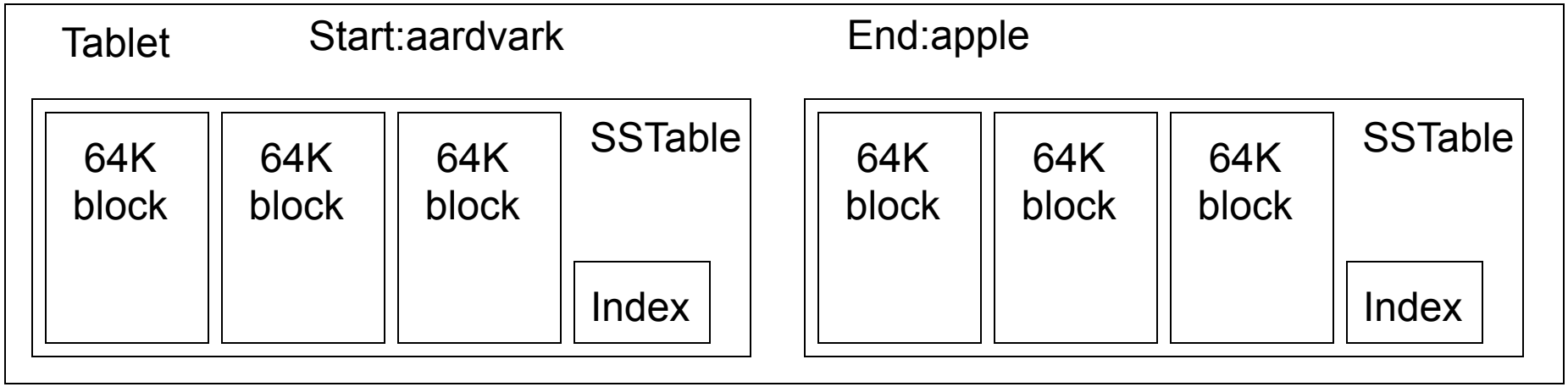


Tablets

- Large tables broken into tablets at row boundaries
  - Tablet holds contiguous range of rows
    - Clients can often choose row keys to achieve locality
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet for failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions

SSTables

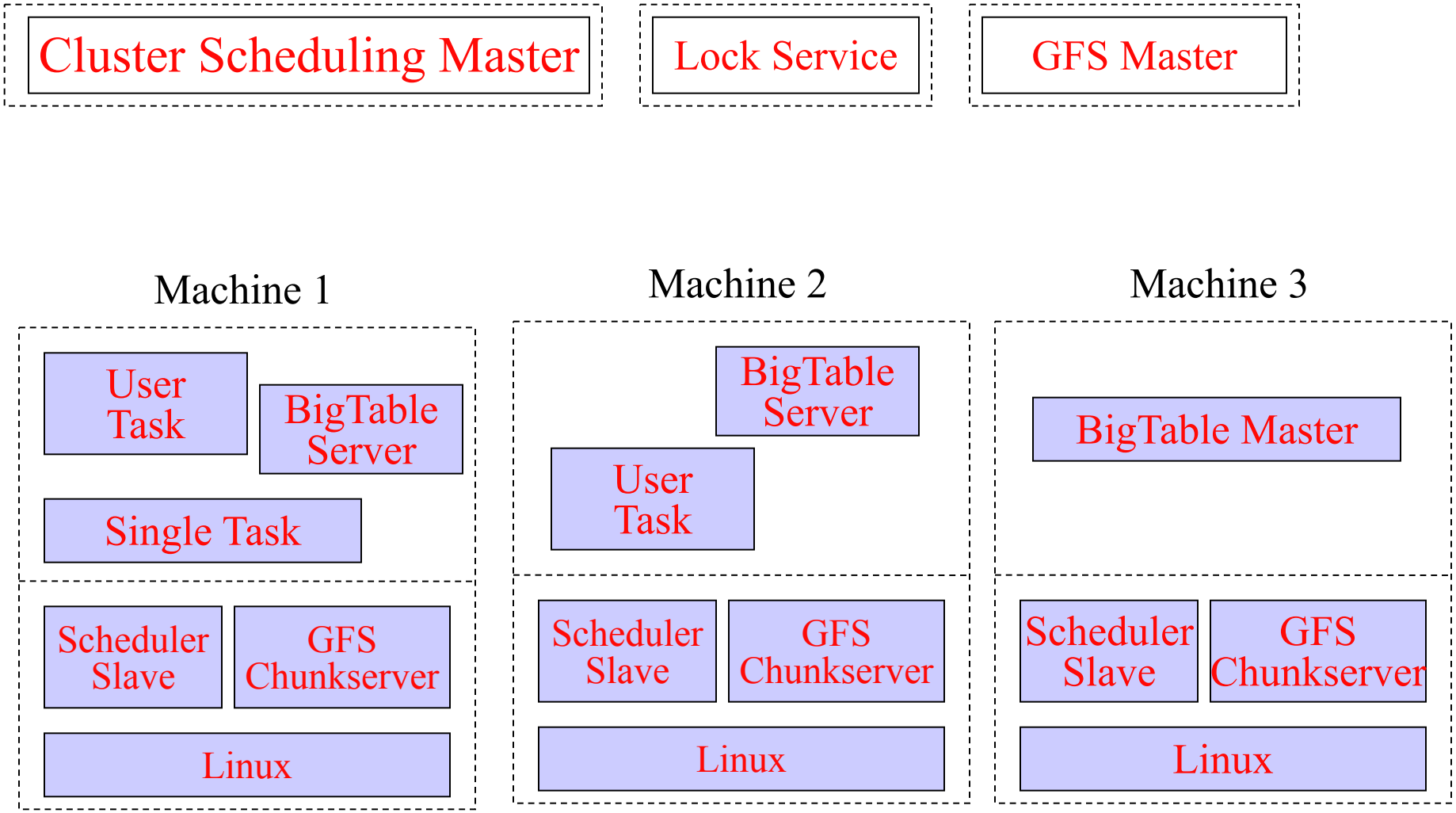
- Tablets are built out of multiple SSTables
- SSTable:
  - Immutable, sorted file of key-value pairs
  - Chunks of data plus an index, which is of block ranges, not values



Implementation – Three Major Components

- Library linked into every client
- One master server
  - Responsible for:
    - Assigning tablets to tablet servers
    - Detecting addition and expiration of tablet servers
    - Balancing tablet-server load
    - Garbage collection
- Many tablet servers
  - Tablet servers handle read and write requests to its table
  - Splits tablets that have grown too large

Typical Cluster

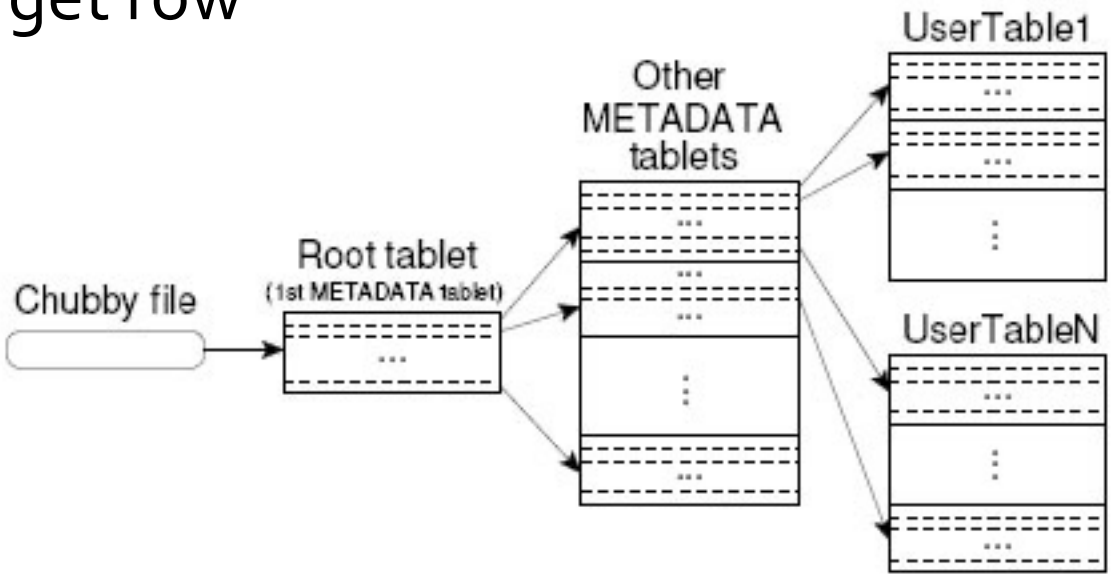


Implementation (cont.)

- Client data doesn't move through master server. Clients communicate directly with tablet servers for reads and writes.
- Most clients never communicate with the master server, leaving it lightly loaded in practice.

Tablet Location

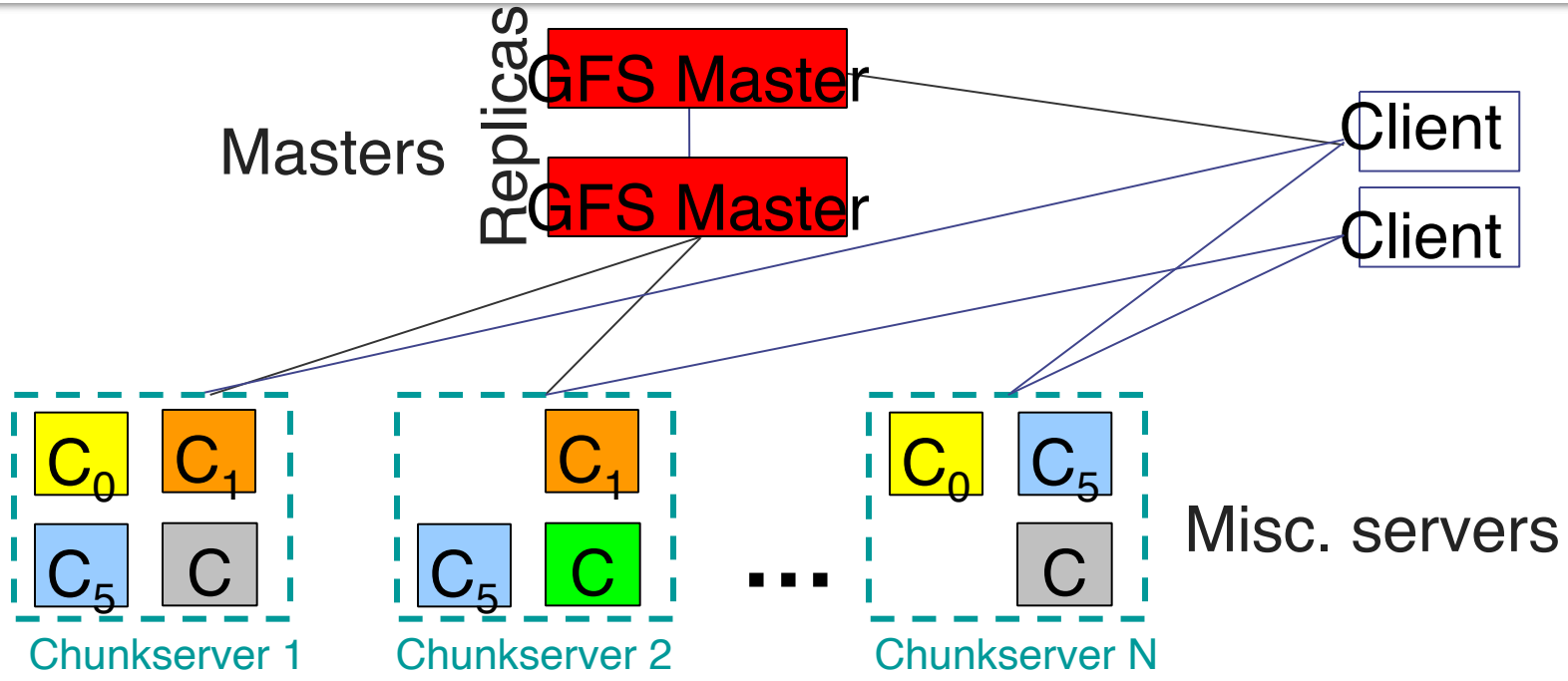
- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row



Tablet Assignment

- Each tablet is assigned to one tablet server at a time.
- Master server keeps track of the set of live tablet servers and current assignments of tablets to servers. Also keeps track of unassigned tablets.
- When a tablet is unassigned, master assigns the tablet to an tablet server with sufficient room.

Google File System (GFS)



- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety

Chubby Lock service

- The lock service must be
  - Available: the show must go on even if one lock server is down
  - Consistent: multiple lock servers must have the same data.
  - Fault-Tolerant: disk crash, message loss/delay, servers goes down/up
- Chubby is Google's distributed lock service using the Paxos consensus algorithm.

Consensus Problem

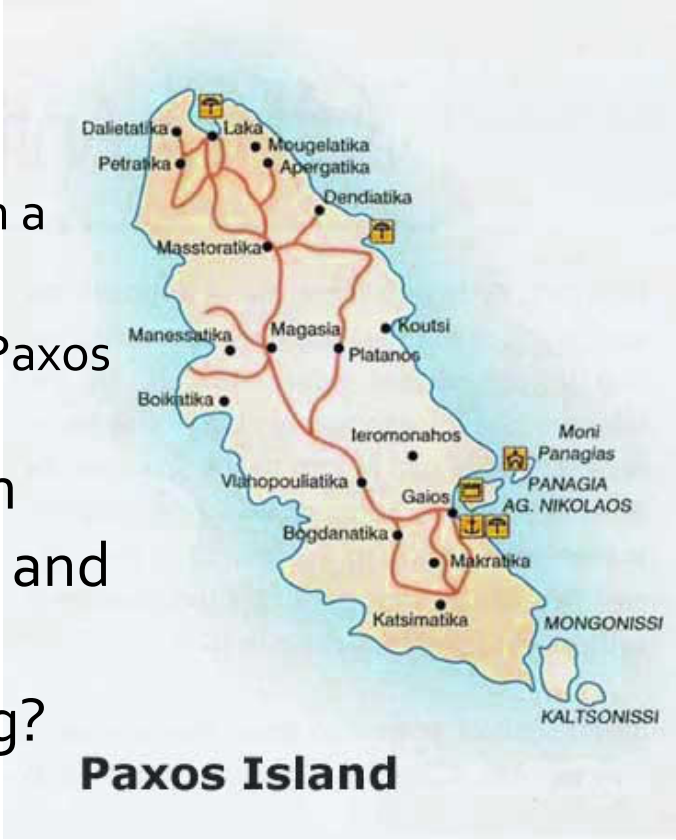
- Consensus problem: reaching agreement among a collection of N processes that can propose values
  - Each process can have a value V[i], i=1, …, M
  - At the end of the consensus algorithm, every process has chosen the same value V[k] for some k in 1, …, M.
  - Cannot choose more than 1 value or non-proposed value.
  - Cannot learn that a value is chosen until it is actually chosen.
- A hard problem in the presence of failures
  - Crash failure: processes may crash and recover, disks may get corrupted.
  - Omission failures: messages may be lost
  - Byzantine failures: processes may lie

Terminology

- Value: any computer datum (e.g., integer, string)
- Process: a program that participates in the algorithm.
- Consensus: a value agreed by a collection of processes.
- Quorum: absolute majority of processes agreeing on a value
- Consensus algorithm: an algorithm that produces a consensus from a collection of processes.
- Safety: the algorithm never goes into an illegal state (e.g., data consistency is maintained across processes.)
- Liviness: algorithm makes progress.

Paxos Parliament: Part-time Parliament

- Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.
- Wealth led to political sophistication
  - the Paxons replaced their ancient theocracy with a parliamentary form of government.
  - But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament.
- The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber
- How did the Parliament decide on anything?



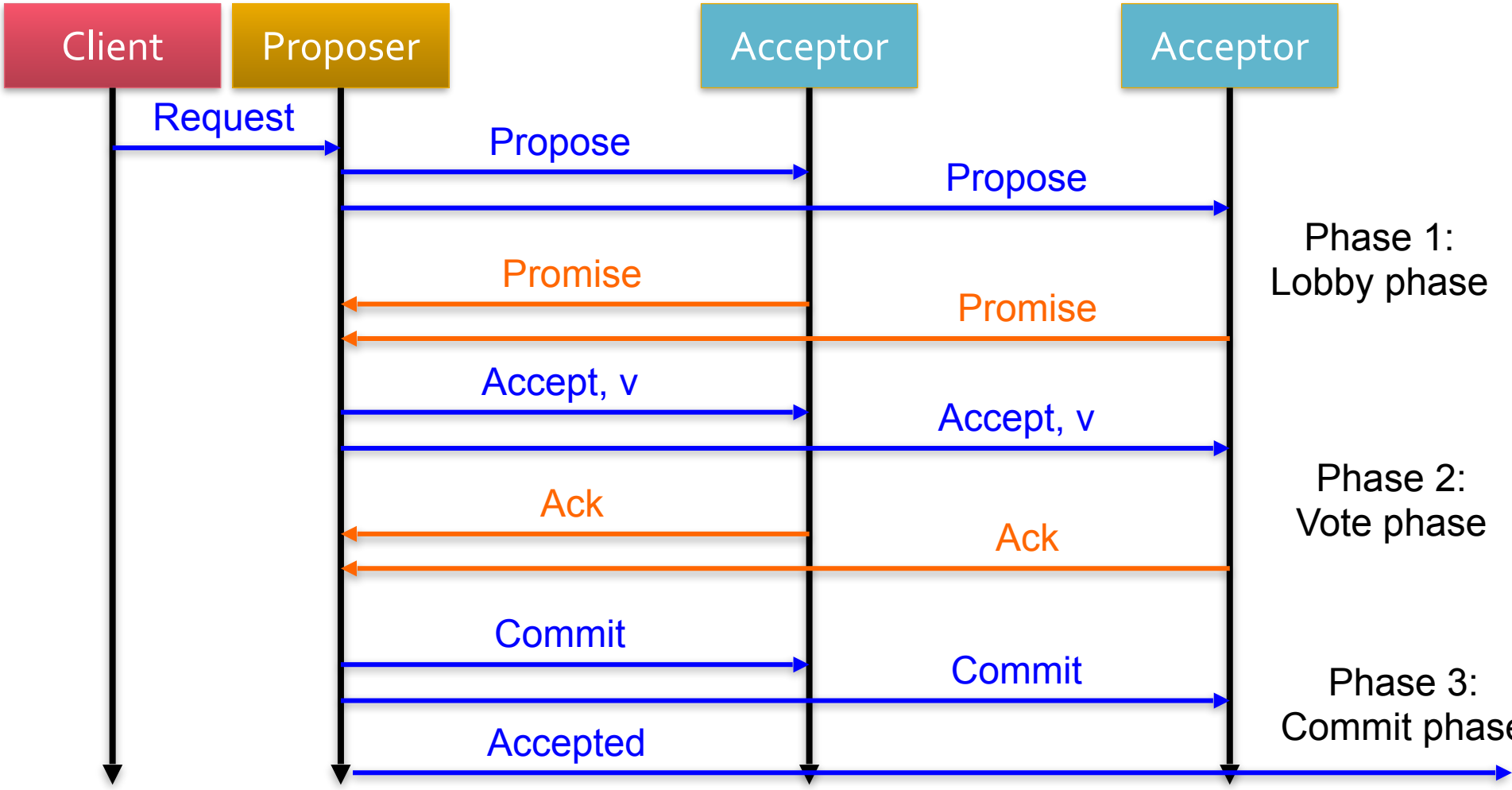
Features of Paxos Algorithm

- Invented by Lesie Lamport in 1998.
- Solves the distributed consensus problem with the following features:
  - Asynchronous: agents operate at arbitrary speed, messages may take arbitrarily long to deliver.
  - Fault-tolerant:
    - Agents may stop and may restart.
    - Messages can be duplicated or lost, but never corrupted.
    - Doesn't handle Byzantine failure: agents must recover its state after a restart.
- Properties:
  - **Non-triviality**: Only proposed values can be learned.
  - **Safety**: At most one value can be learned (i.e., two different learners cannot learn different values).
  - **Liviness**(C;L): If value C has been proposed, then eventually learner L will learn some value (if sufficient processes remain non-faulty).

Roles in Paxos Algorithm

- **Client**: The Client issues a *request* to the distributed system, and waits for a *response*.
- **Proposers**: A Proposer advocates a client request, attempting to convince the Acceptors to agree on it, and acting as a coordinator to move the protocol forward when conflicts occur.
- **Acceptors** (**voters**): acceptors vote on the proposal sent by Proposers.
- **Learners**: Learners act as the replication factor for the protocol. Once a Client request has been agreed on by the Acceptors, the Learner may take action (i.e.: execute the request and send a response to the client).

Paxos: the simplest scenario



Ballot Numbers

- To which proposer should an acceptor promise?
- Solution: use ballot numbers to identify proposals
  - Each proposer has its own ballot numbers
  - No two proposers can use the same ballot number
  - Usually: proposer k uses ballot numbers B that B%N=k
- How to reach a consensus?
  - If a proposal with value V is chosen, all higher numbered ballots must propose the same value v.

Acceptors in the Lobby Phase (Phase 1)

- An acceptor does not have to respond to any message
  - No response has the same effect as lost message.
  - But it should try to respond in order to make progress.
    - Denial message can speed up the process if desired.
- Responding a "promise" to a lobby means the acceptors won't help proposals with smaller ballot numbers.
- When receiving a proposal with ballot number B:
  - If the acceptor has promised to a higher numbered ballot ignore ballot B.
  - Otherwise, it promises to help ballot B, with optional data:
    - If the acceptor has voted for value V from ballot X, optional data=(V,X)
    - Otherwise, the optional data is empty.

Vote Phase (Phase 2)

- If the proposer receives promises from a quorum, it starts the vote phase
  - If no promise contains accepted values, send its own value.
  - If some promise contain accepted values, send the value from the largest ballot number.
- If it does not receive enough promises
  - Give up, accept (later), or
  - Propose a higher numbered ballot, probably after a back-off time is passed.
- An acceptor accepts value from ballot B that it promised to help, unless it has promised to help a higher numbered ballot.

Commit Phase (Phase 3)

- If proposer receives ack messages from majority of acceptors
  - Send commit message to ALL participants.
- If acceptors receives commit
  - Done=true
  - Agreement reached; agreed-on value is v.
- Accepted values are sent to learners and clients.

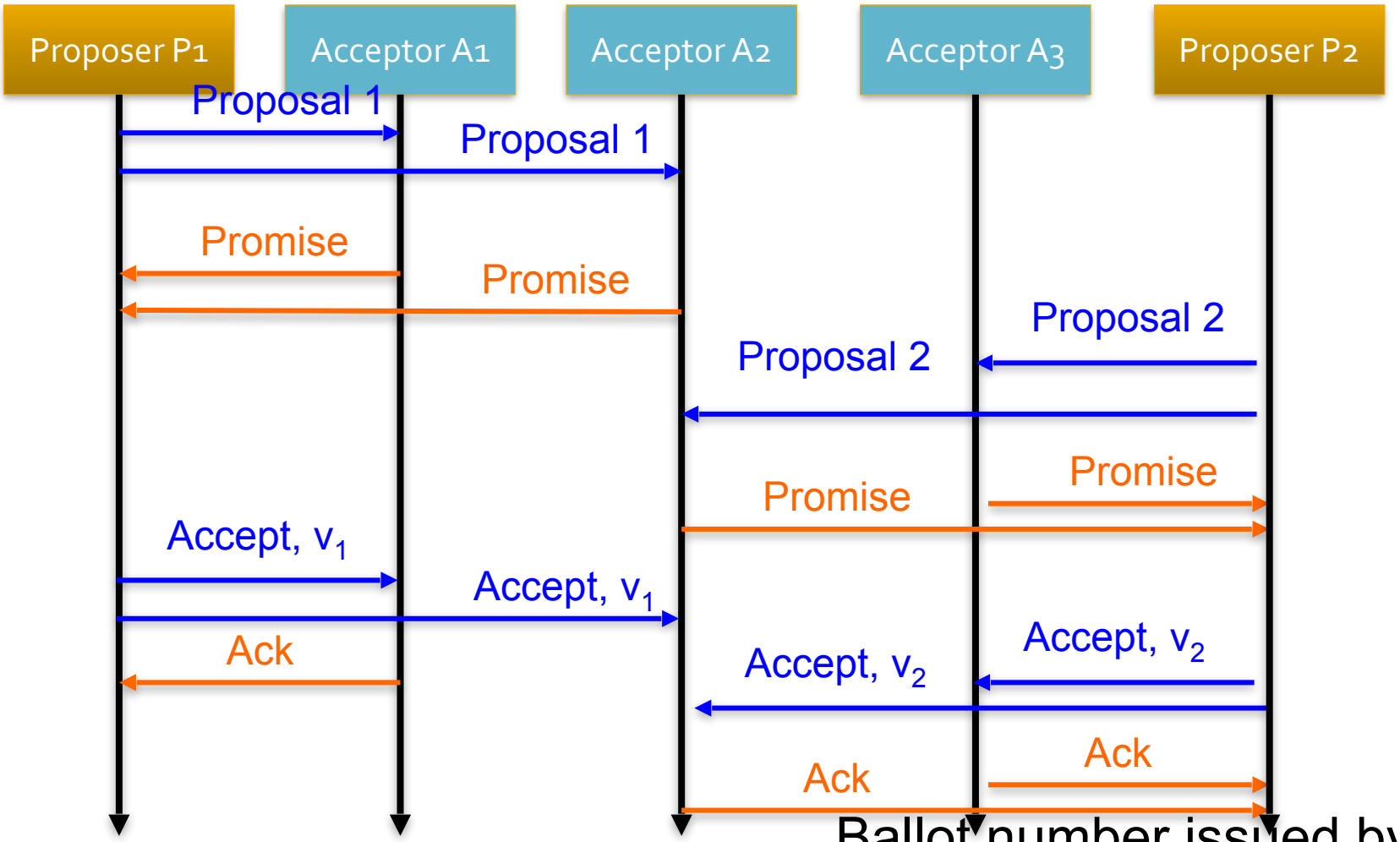
To be discussed

- Multiple proposers
- Live lock
- Timeout for voted value:
  - Is it true that one a value is voted, all the process will eventually agree on this value?
    - As long as the majority requirement is met, only one value will be chosen.
  - Is it possible to vote for different value?
    - One start more than one Paxos algorithm to vote on different issue.

Multiple Proposers

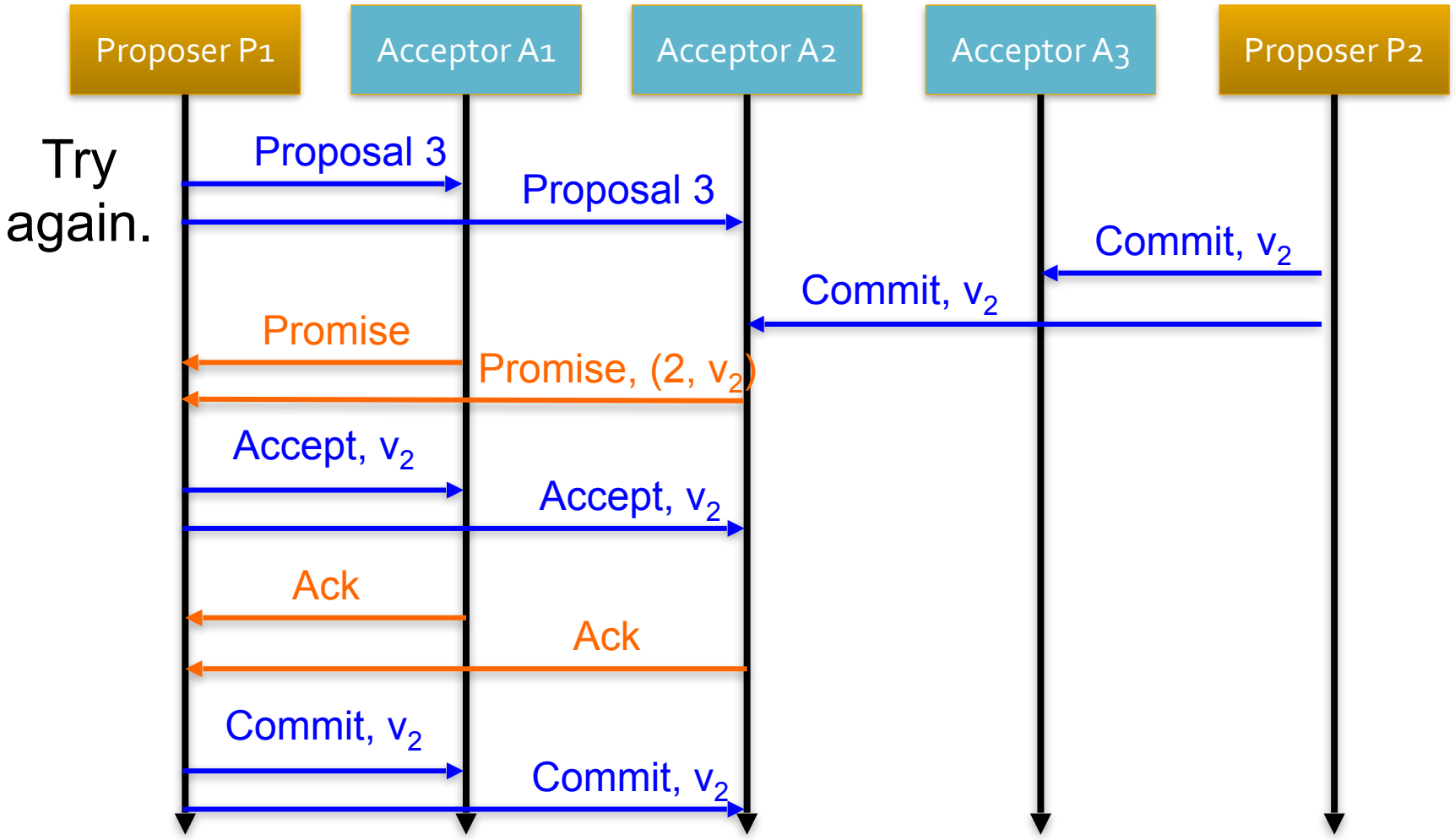
- There may be more than one proposer in the system, each proposing a value for same request
  - Single proposer can fail
  - Every node must be willing to become a proposer
- All nodes wait a maximum period (timeout) for messages they expect
  - Upon timeout, a node declares itself a proposer and initiates a new Phase 1 of algorithm.

Multiple Proposers



Ballot number issued by P_2 is greater than that of P_1.

Multiple Proposers



Progress isn't guaranteed

- If there are multiple proposers, progress is not guaranteed.
  - Proposers may repeatedly propose conflicting values.
- One distinguished proposer can be elected to propose.
  - All the proposals are sent to the distinguish proposer first.
- When the distinguish proposer fails, a leader election algorithm is executed first to elect the new distinguish proposer.

Spanner: Globally-Distributed Database Systems for Google

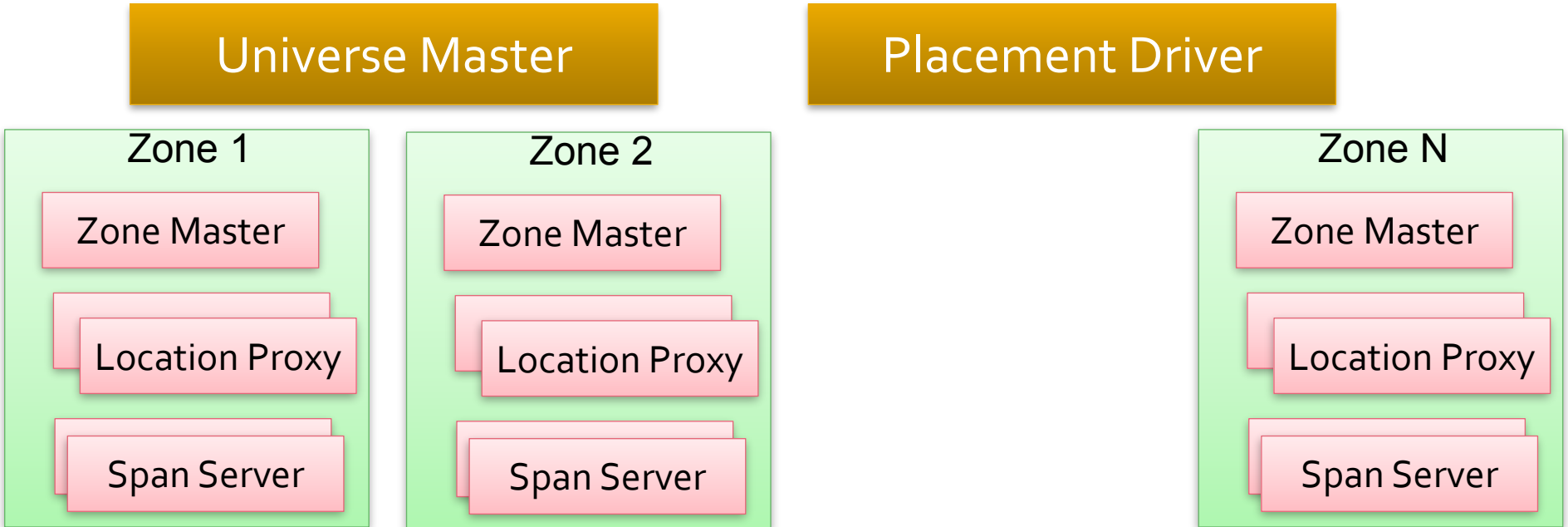
- Goals:
  - Managing cross-data center replicated data
  - Supporting interactive applications for low latency data processing (Not batch applications)
  - SQL-like programming language.

New mechanisms in Spanner

- Client-driven replication on database level:
  - In BigTable, the replications are managed by GFS to share the load on server.
  - In Spanner, the replications are driven by application workloads and managed by Spanner, not GFS.
- Time with uncertainty
  - Time is given as an interval to contain *true time* of the event.
  - Time is synchronized via GPS and atomic clocks around the world.
    - These two source of time have different failure models.

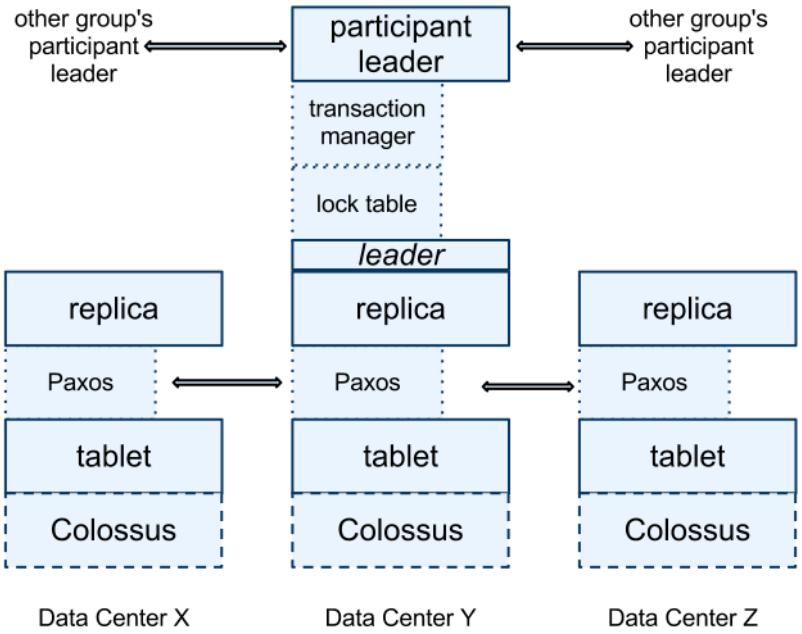
Server Organization

- A Spanner deployment is called a universe.
- Spanner is organized as a set of zones,
  - Each zone is the rough analog of a deployment of Bigtable server.
  - Zones are the unit of administrative deployment.
  - The set of zones is also the set of locations across which data can be replicated.



Span Servers

- Each span server is responsible for between 100 and 1000 instances of a data structure called a tablet.
- Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multi-version database than a key-value store.



Distributed File Systems for Hadoop

Assumption on Targeted Systems and Workloads

- **Hardware Failure**: Hardware failure is the norm rather than the exception.
- **Streaming Data Access**: Applications that run on HDFS need streaming access to their data sets.
- **Large Data Sets**: Applications that run on HDFS have large data sets.
- **Simple Coherency Model**: HDFS applications need a write-once-read-many access model for files.
- **Moving Computation is Cheaper than Moving Data**: A computation requested by an application is much more efficient if it is executed near the data it operates on.
- **Portability Across Heterogeneous Hardware and Software Platforms**

Hadoop Distributed File Systems (HDFS)

- HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. (In other words, not reliable and can be replaced at any time.)
- HDFS provides high throughput access to application data and is suitable for applications that have large data sets.
- HDFS relaxes a few POSIX requirements to enable streaming access to file system data.
- HDFS was originally built as infrastructure for the Apache Nutch web search engine project.

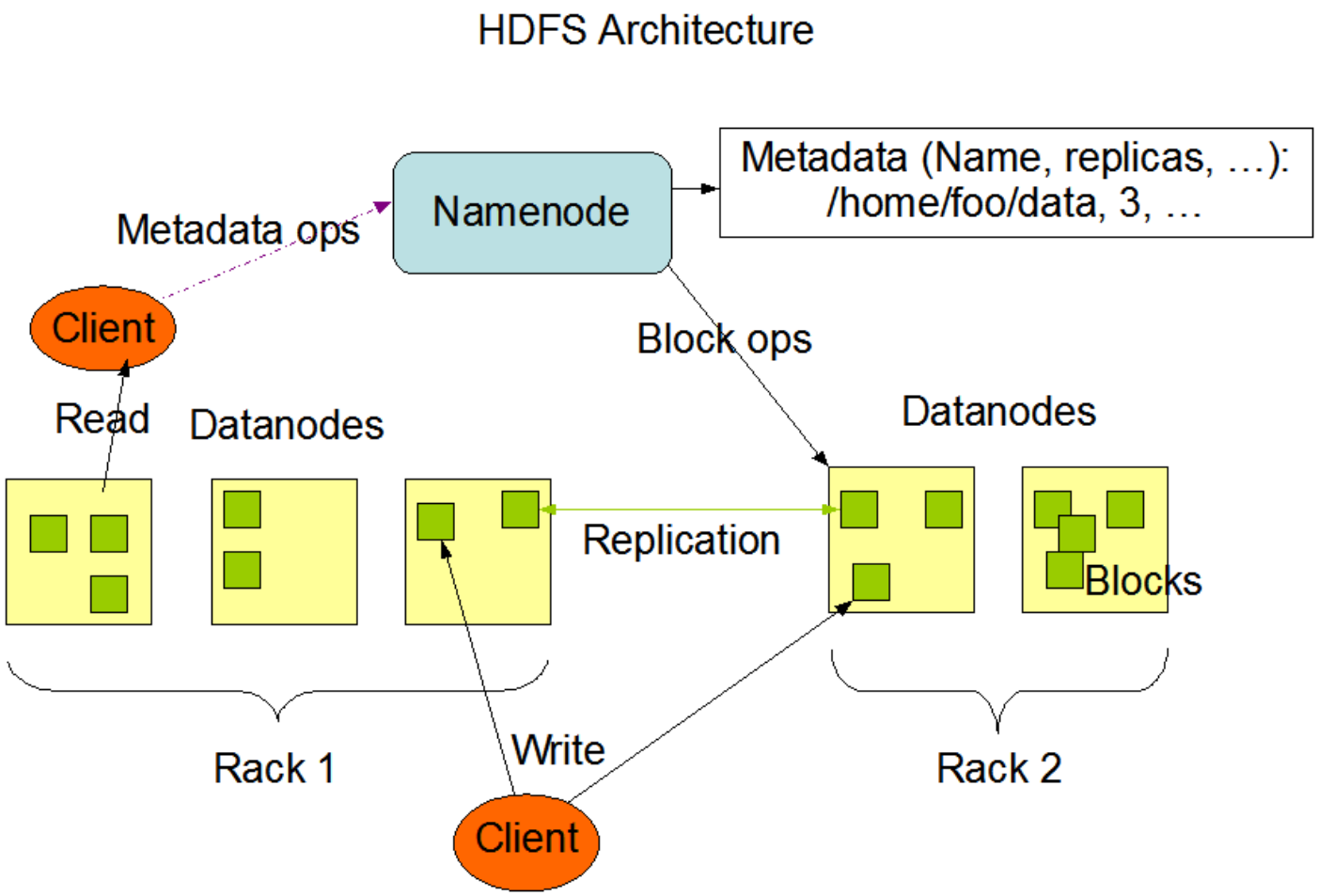
Hbase in Hadoop

- HBase is the Hadoop database.
  - Use it when you need random, realtime read/write access to your Big Data.
  - This project's goal is the hosting of very large tables -- billions of rows X millions of columns -- atop clusters of commodity hardware.
- HBase is an open-source, distributed, versioned, column-oriented store modeled after Google' Bigtable.
- HBase provides Bigtable-like capabilities on top of Hadoop.

HBase

- HBase provides:
  - Convenient base classes for backing Hadoop MapReduce jobs with HBase tables
  - Query predicate push down via server side scan and get filters
  - Optimizations for real time queries
  - A high performance Thrift gateway
  - A REST-ful Web service gateway that supports XML, Protobuf, and binary data encoding options
  - Cascading, hive, and pig source and sink modules
  - Extensible jruby-based (JIRB) shell
  - Support for exporting metrics via the Hadoop metrics subsystem to files or Ganglia; or via JMX
- HBase 0.20 (0.95 is the latest version) has greatly improved on its predecessors:
  - No HBase single point of failure
  - Rolling restart for configuration changes and minor upgrades
  - Random access performance on par with open source relational databases such as MySQL.

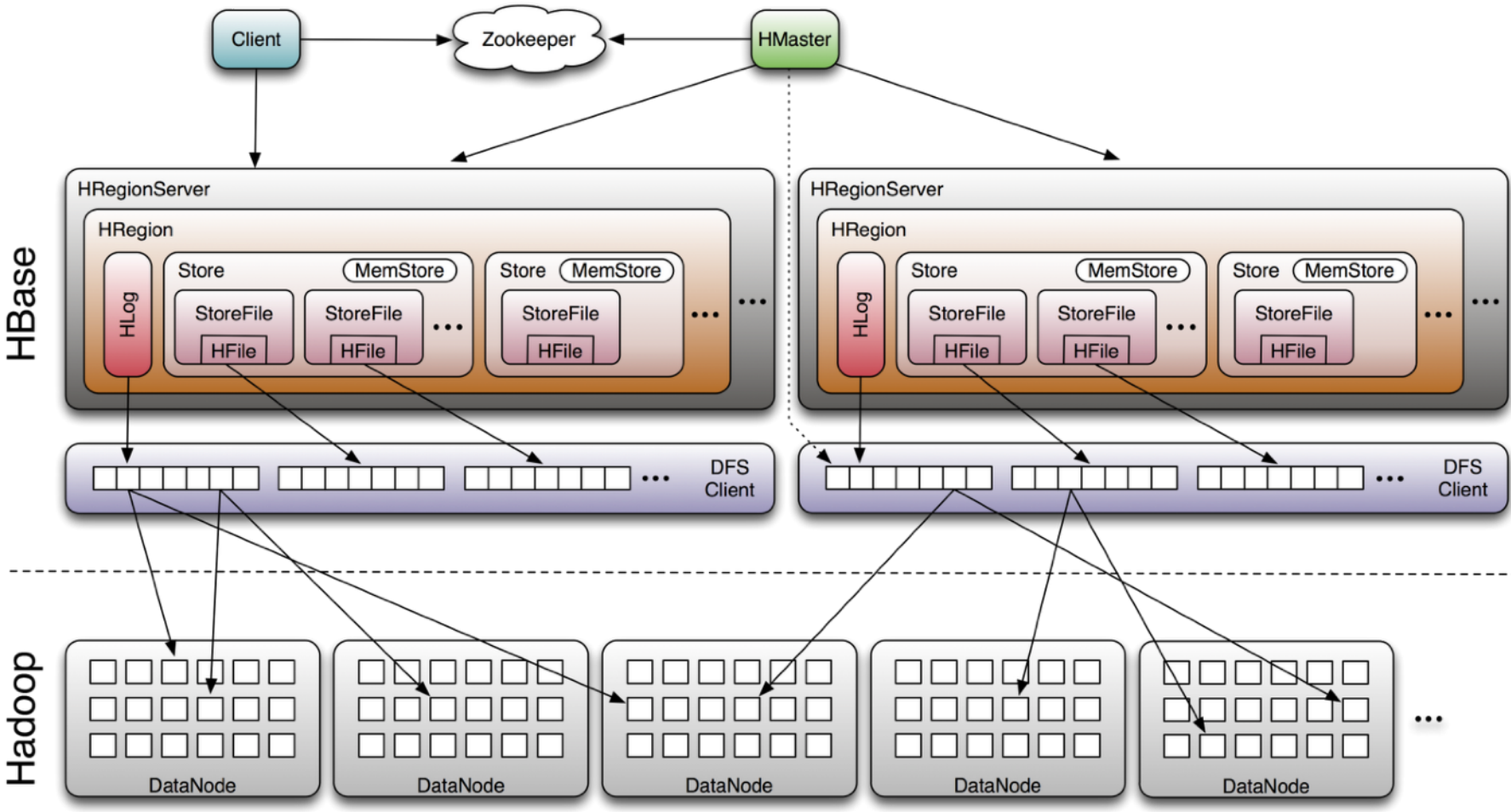
HDFS Architecture



HDFS Architecture

- NameNode:
  - An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients.
  - The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.
- DataNodes:
  - The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.
  - There are several data nodes in each cluster.
  - It manages storage attached to the nodes that they run on.
  - A file is split into one or more blocks and these blocks are stored in a set of DataNodes.

Big Picture for HBase

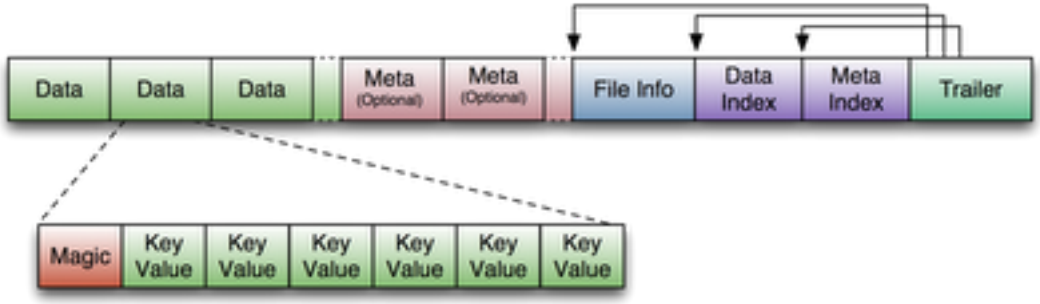


Big Picture for HBase

- HRegionServer: used for the write-ahead log
- HMaster will have to perform low-level file operations
  - The HMaster is responsible to assign the regions to each HRegionServer when you start HBase.

HFile

- Hfiles are the actual storage files, specifically created to serve one purpose: store HBase's data fast and efficiently.
- They are apparently based on Hadoop's Tfile and mimic the SSTable format used in Googles BigTable architecture.
- The files have a variable length, the only fixed blocks are the FileInfo and Trailer block.



File Access Flow

- The general flow is that
  - a new client contacts the Zookeeper quorum (a separate cluster of Zookeeper nodes) first to find a particular row key.
  - It does so by retrieving the server name (i.e. host name) that hosts the -ROOT- region from Zookeeper.
  - With that information it can query that server to get the server that hosts the .META. table. Both of these two details are cached and only looked up once.
  - Lastly it can query the .META. server and retrieve the server that has the row the client is looking for.
  - Once it has been told where the row resides, i.e. in what region, it caches this information as well and contacts the HRegionServer hosting that region directly. So over time the client has a pretty complete picture of where to get rows from without needing to query the .META. server again.

Discussion: GFS vs. Coda

- What's the difference on targeted system architecture?
  - Connectivity between client and servers.
- What's the challenges of interested?
  - Performance
  - Scalability
  - Availability
  - Robustness

Summary

- We have learned
  - the features and services DFS should provide,
  - the issues of designing DFSs,
  - File access models,
  - File-caching models,
  - File-replication, and
  - Fault-tolerance.
- We also studied CODA, and BigTable distributed file system.

Reference

- Hbase Architecture 101 – Storage, http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html