Cloud Computing
CS 15-319

Distributed File Systems and Cloud Storage – Part III

Lecture 14, Feb 29, 2012

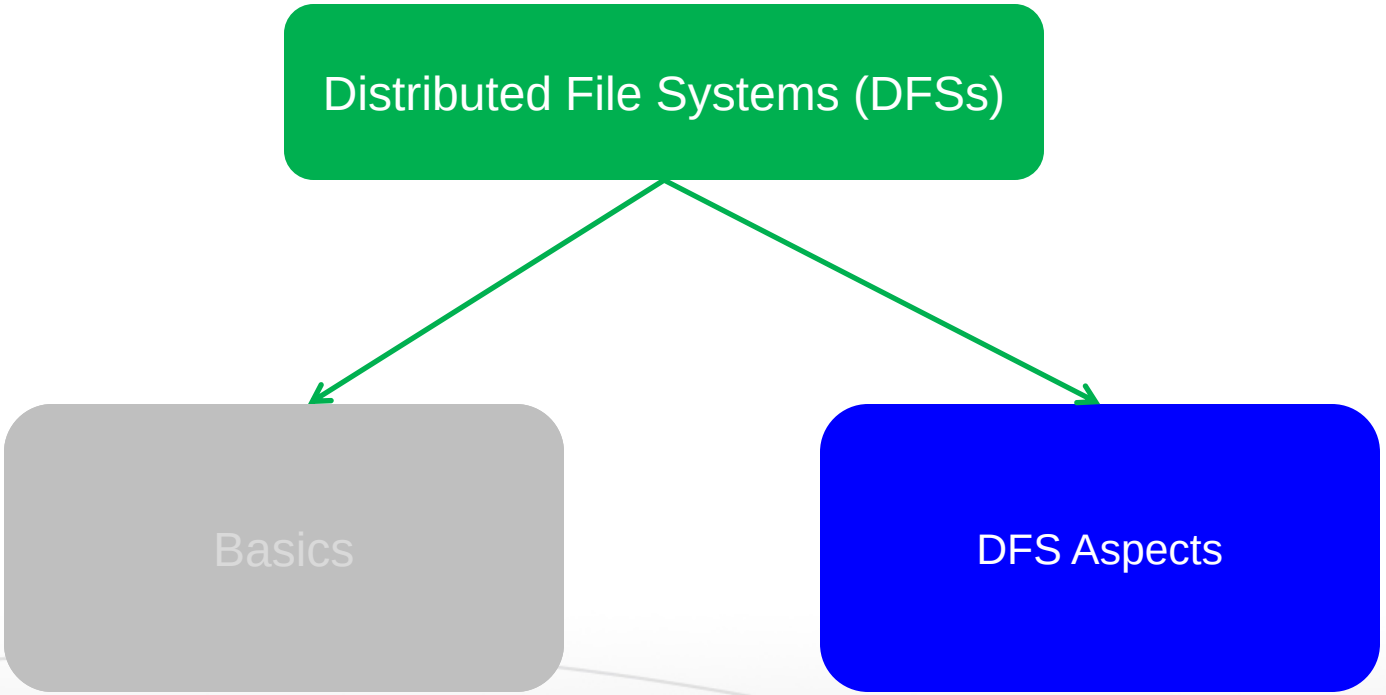
Majd F. Sakr, Mohammad Hammoud and
Suhail Rehman

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Today…

- Last session
  - Distributed File Systems and Cloud Storage- Part II
- Today's session
  - Distributed File Systems and Cloud Storage- Part III
- Announcements:
  - Project update is due today
  - Spring break: March 3-7, classes resume March 10

Discussion on Distributed File Systems



DFS Aspects

Aspect	Description
Architecture	How are DFSs generally organized?
Processes	• Who are the cooperating processes? • Are processes *stateful* or *stateless*?
Communication	• What is the typical communication paradigm followed by DFSs? • How do processes in DFSs communicate?
Naming	How is naming often handled in DFSs?
Synchronization	What are the file sharing semantics adopted by DFSs?
Consistency and Replication	What are the various features of client-side caching as well as server-side replication?
Fault Tolerance	How is fault tolerance handled in DFSs?

DFS Aspects

Aspect	Description
Architecture	How are DFSs generally organized?
Processes	• Who are the cooperating processes? • Are processes *stateful* or *stateless*?
Communication	• What is the typical communication paradigm followed by DFSs? • How do processes in DFSs communicate?
Naming	How is naming often handled in DFSs?
Synchronization	What are the file sharing semantics adopted by DFSs?

Synchronization In DFSs

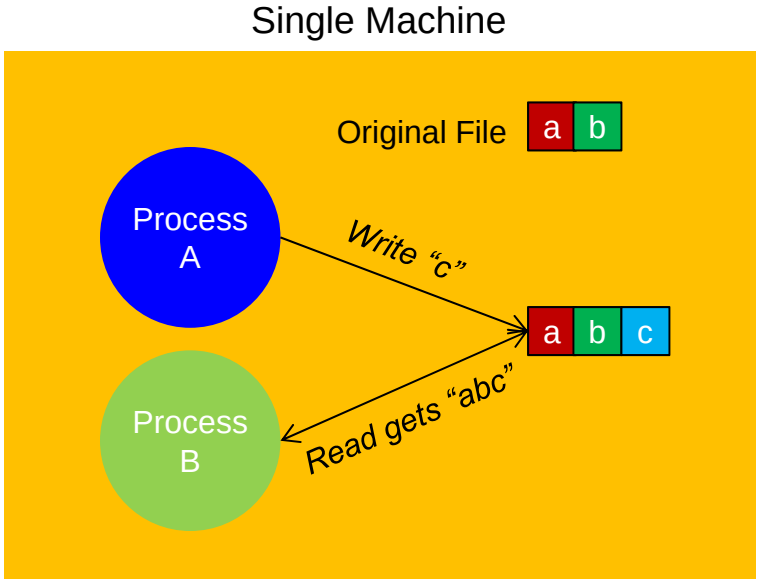
- File Sharing Semantics
- Lock Management

Synchronization In DFSs

- File Sharing Semantics
- Lock Management

Unix Semantics In Single Processor Systems

- Synchronization for file systems would not be an issue if files were not shared
- When two or more users share the same file at the same time, it is necessary to define the *semantics* of reading and writing
- In single processor systems, a read operation after a write will return the value just written
- Such a model is referred to as Unix Semantics



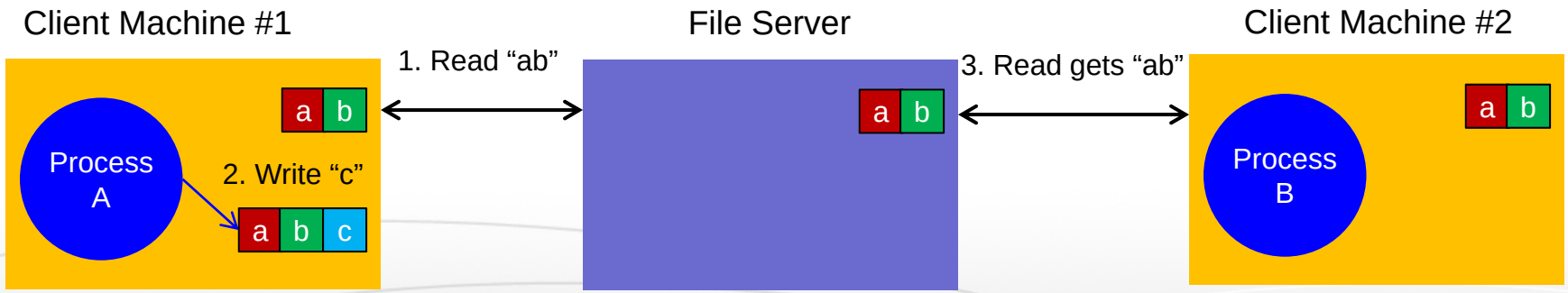
Unix Semantics In DFSs

- In a DFS, Unix semantics can be achieved *easily* if there is only one file server and clients do not cache files
- Hence, all reads and writes go directly to the file server, which processes them *strictly sequentially*
- This approach provides UNIX semantics, however, performance might degrade as all file requests must go to a single server

Caching and Unix Semantics

- The performance of a DFS with one single file server and Unix semantics can be improved by caching

- If a client, however, locally modifies a cache file and shortly another client reads the file from the server, it will get an *obsolete* file



Session Semantics (1)

- One way out of getting an obsolete file is to propagate all changes to cached files back to the server *immediately*
- Implementing such an approach is very difficult
- An alternative solution is to *relax* the semantics of file sharing

Changes to an open file are initially visible only to the process that modified the file. Only when the file is closed, the changes are made visible to other processes.

Session Semantics

Session Semantics (2)

- Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file
- One solution is to say that as each file is closed in turn, its value is sent back to the server
  - The final result depends on whose close request is most recently processed by the server
- A less pleasant solution, but easier to implement, is to say that the final result is one of the candidates and leave the choice of the candidate unspecified

Immutable Semantics (1)

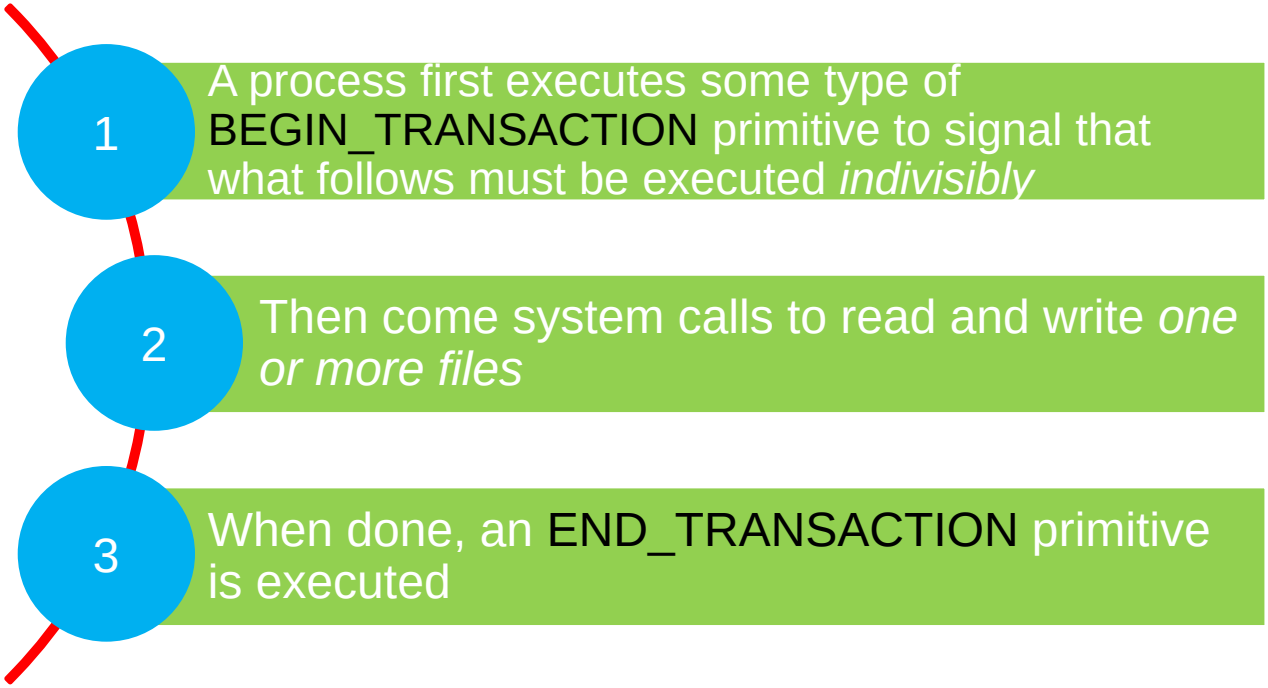
- A different approach to the semantics of file sharing in DFSs is to make all files immutable
- With immutable semantics there is no way to open a file for writing
- What is possible is to create an entirely new file
- Hence, the problem of how to deal with two processes, one writing and the other reading, just disappears

Immutable Semantics (2)

- However, what happens if two processes try to replace the same file?
  - Allow one of the new files to replace the old one (either the last one or non-deterministically)
- What to do if a file is replaced while another process is busy reading it?
  - Solution 1: Arrange for the reader to continue using the old file
  - Solution 2: Detect that the file has changed and make subsequent attempts to read from it fail

Atomic Transactions

- A different approach to the semantics of file sharing in DFSs is to use atomic transactions where all changes occur atomically



- A key property is that all calls contained in a transaction will be carried out *in-order*

Semantics of File Sharing: Summary

- There are four ways of dealing with the shared files in a DFS:

Method	Comment
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
Transactions	All changes occur atomically

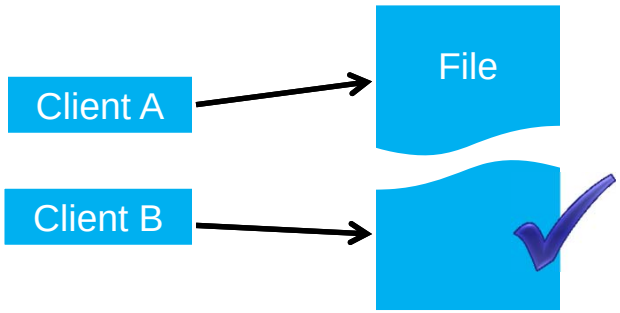
HDFS Semantics

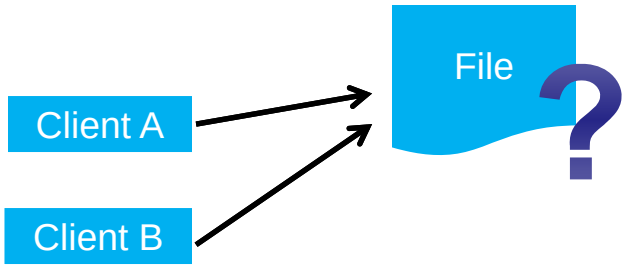
- HDFS follows immutable semantics
  - HDFS does not allow clients to modify an existing file
  - All fresh write operations should be made to new files

- Designed for MapReduce
  - Restricted computational model with predefined stages
  - Each reducer in a MapReduce job writes an independent file for its portion of the output

PVFS semantics

- PVFS follows Unix semantics with restrictions
  - Sequential consistency guaranteed for non-conflicting writes (write operations to different regions of a file)
  - If two clients write to the same region of the file, the behavior is undefined





- Research prototypes of PVFS have write-anywhere support
  - This is achieved by implementing an atomic transaction model

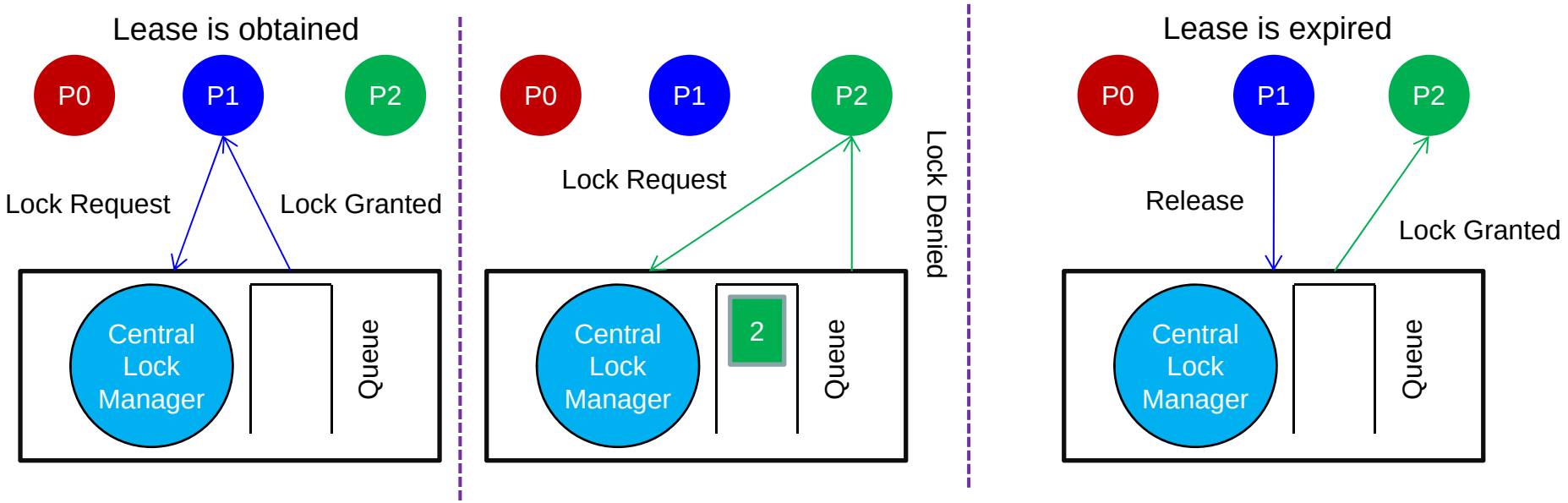
Synchronization In DFSs

- File Sharing Semantics
- Lock Management

Central Lock Manager

- In client-server architectures (*especially* with stateless servers), additional facilities for synchronizing accesses to shared files are required

- A central lock manager can be deployed where accesses to a shared resource are synchronized by granting and denying access permissions

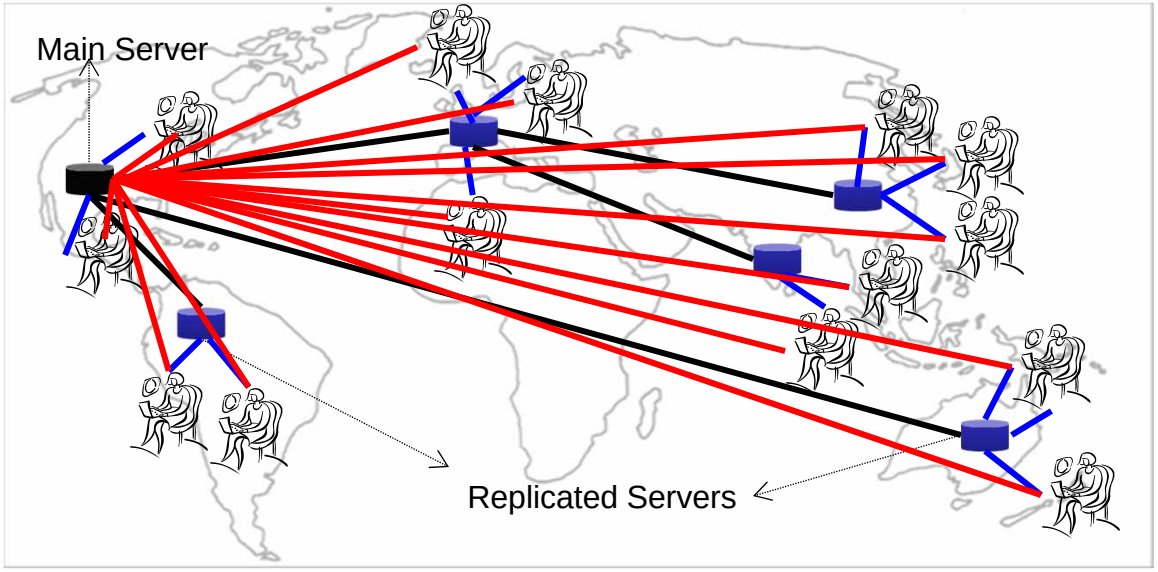


DFS Aspects

Aspect	Description
Architecture	How are DFSs generally organized?
Processes	• Who are the cooperating processes? • Are processes *stateful* or *stateless*?
Communication	• What is the typical communication paradigm followed by DFSs? • How do processes in DFSs communicate?
Naming	How is naming often handled in DFSs?
Synchronization	What are the file sharing semantics adopted by DFSs?
Consistency and Replication	What are the various features of client-side caching as well as server-side replication?

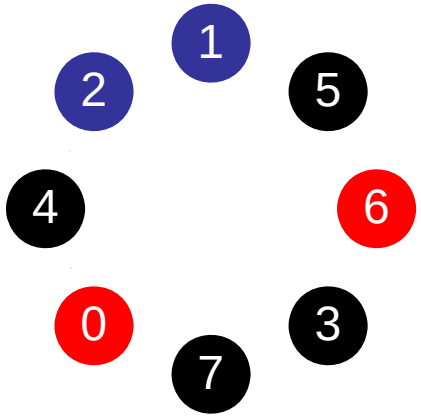
Why Replication?

- Replication is the process of maintaining the data at multiple computers
- Replication is necessary for:
  1. Improving performance
     - A client can access the replicated copy of the data that is near to its location
  2. Enhancing the scalability of the system
     - Requests to the data can be distributed to many servers which contain replicated copies of the data



Why Replication?

- Replication is necessary for (*Cont'd*):
  3. Increasing the availability of services
     - Replication can mask failures such as server crashes and network disconnection
  4. Securing against malicious attacks
     - Even if some replicas are malicious, secure data can be guaranteed to the client by relying on the replicated copies at the non-compromised servers
       - If a minority of the servers that hold the data are malicious, the non-malicious servers can outvote the malicious servers, thus providing security



Number of servers with correct data outvote the faulty servers

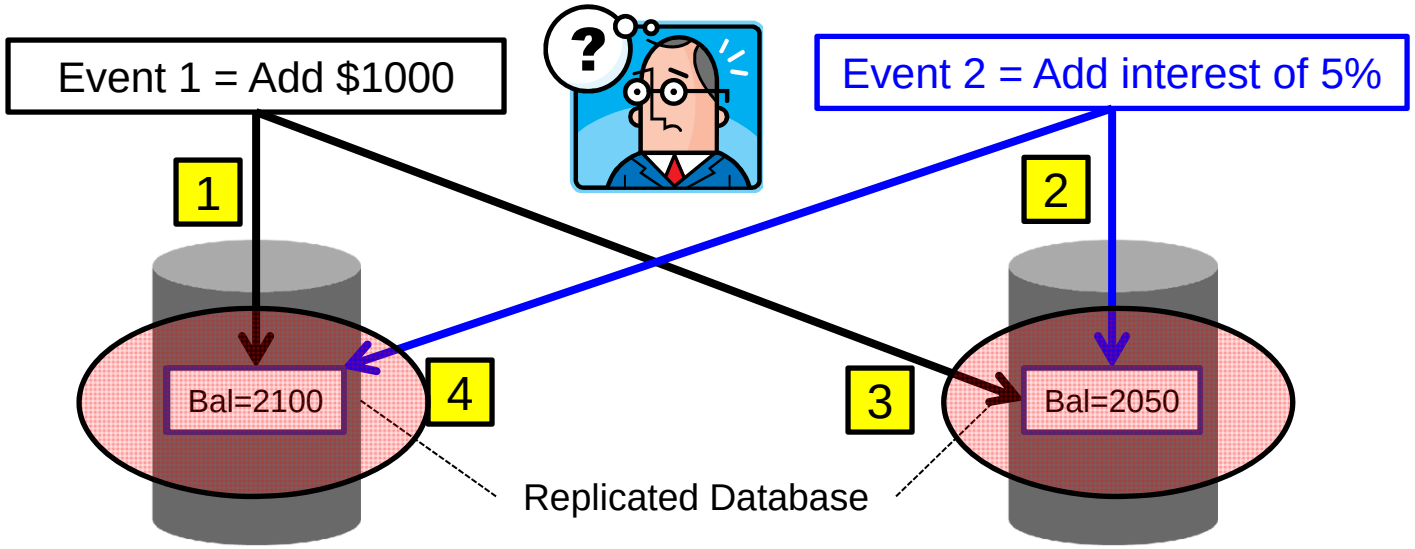
n = Servers that do not have the requested data

n = Servers with correct data

n = Servers with faulty data

Why Consistency?

- In a DS with replicated data, one of the main problems is keeping the data consistent
- An example:
  - In an e-commerce application, the bank database has been replicated across two servers
  - Maintaining consistency of replicated data is a challenge



Client-Side Caching In HDFS

- Clients in HDFS cache the DataNode layout information received from the NameNode
  - This allows the clients to interact repeatedly and directly with DataNodes
- HDFS stores lists of chunks for each file
- During reads, HDFS can fetch and cache the chunk list of a file because HDFS files are immutable
- During writes, HDFS must contact the NameNode for each new chunk allocation (as the file grows)
- HDFS does not cache any file data at the client machines

Client-Side Caching In PVFS

- Clients in PVFS cache the I/O server (iod) layout information received from the MGR
  - This allows the clients to interact repeatedly and directly with iods
- PVFS is able to easily cache a file's layout because a stripe unit's location in an object is algorithmically derived from its offset in the file
- PVFS does not need to contact the MGR for each new stripe unit allocation
- PVFS does not cache any file data at the client machines

Consistency models in HDFS and PVFS

- In large distributed systems with many concurrent writes, consistency is typically a potential source of bottleneck

- HDFS follows write-once-read-many semantics
  - Only one writer per file and no appending to a file by another client
  - Simplified model does not require any consistency model

- PVFS allows concurrent writes
  - Only for non-overlapping segments of a file, sequential consistency guaranteed in this case
  - If two clients write to same file segment, the result is undefined
  - Appending to a file also not supported

DFS Aspects

Aspect	Description
Architecture	How are DFSs generally organized?
Processes	• Who are the cooperating processes? • Are processes *stateful* or *stateless*?
Communication	• What is the typical communication paradigm followed by DFSs? • How do processes in DFSs communicate?
Naming	How is naming often handled in DFSs?
Synchronization	What are the file sharing semantics adopted by DFSs?
Consistency and Replication	What are the various features of client-side caching as well as server-side replication?
Fault Tolerance	How is fault tolerance handled in DFSs?

Failures, Due to What?

- Failures can happen due to a variety of reasons:
  - Hardware faults
  - Software bugs
  - Network errors/outages
- A system is said to fail when it cannot meet its promises

Failures in Distributed Systems

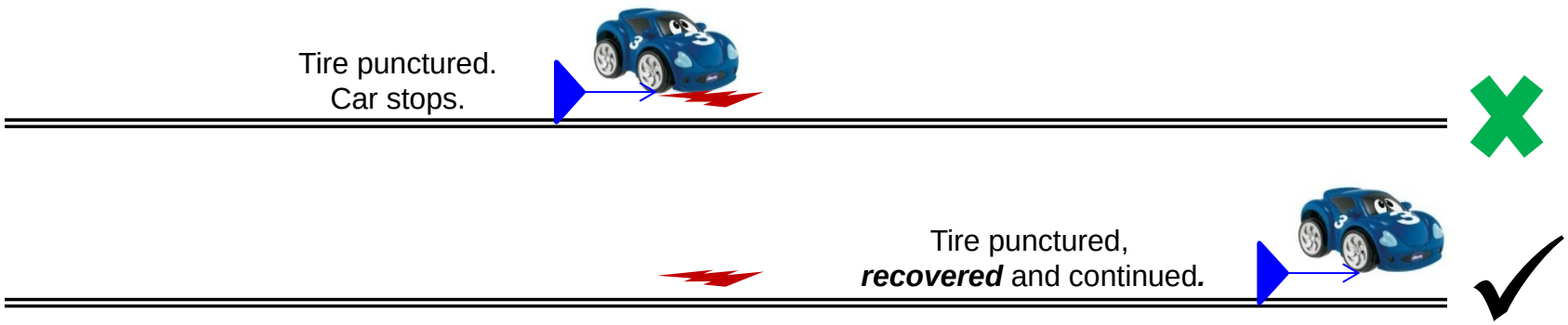
- A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of a partial failure

- A partial failure may happen when a component in a distributed system fails

  - This failure may affect the proper operation of other components, while at the same time leaving yet other components unaffected

Goal and Fault-Tolerance

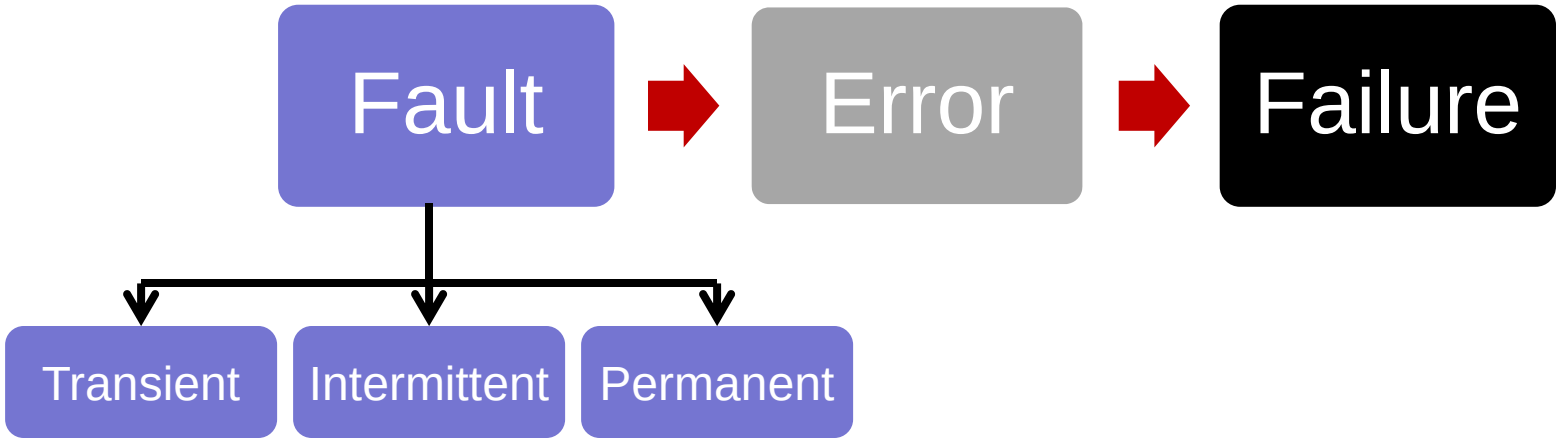
- An overall goal in distributed systems is to construct the system in such a way that it can automatically recover from partial failures



- Fault-tolerance is the property that enables a system to continue operating properly in the event of failures

- For example, TCP is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communication links which are imperfect or overloaded

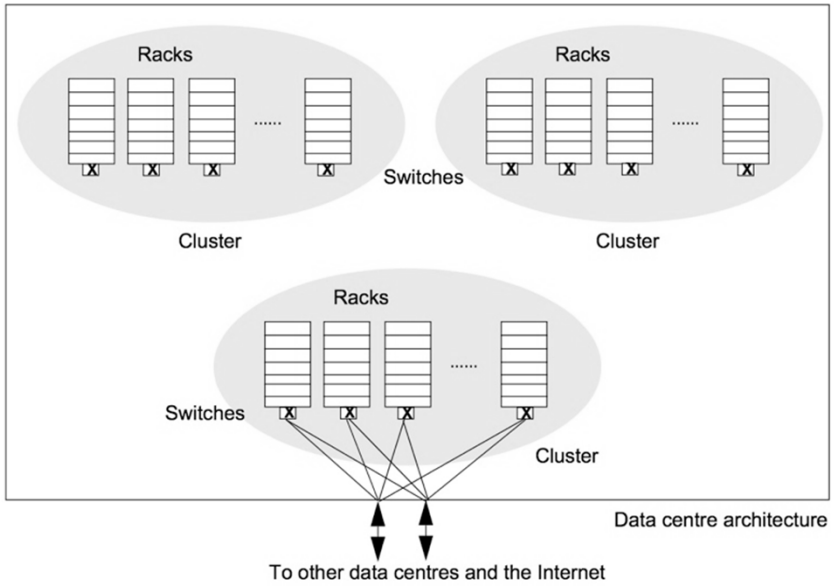
Faults, Errors and Failures

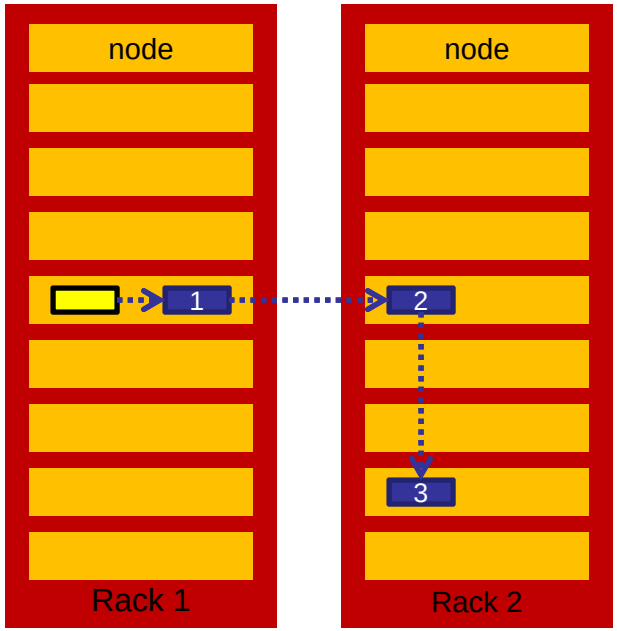


A system is said to be *fault tolerant* if it can provide its services even in the presence of *faults*

Fault Tolerance in HDFS

- Faults are handled in HDFS though replication
  - Default replication factor is 3
- Replica placement in HDFS tries to protect against node and rack failures





DataNode Failures

- DataNode failures are detected using a heartbeat mechanism
  - DataNodes periodically send heartbeat messages to the NameNode to indicate that they are *alive*
  - A network disconnection can cause a subset of DataNodes to lose connectivity with the NameNode
  - The NameNode detects this condition by the absence of heartbeat messages
  - The NameNode marks DataNodes without recent heartbeats as *dead* and does not forward any new IO requests to them
  - DataNode death may cause the replication factor of some blocks to fall below their specified value
  - The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary

NameNode Failure

- NameNode failure brings HDFS completely down as files cannot be mapped to the respective blocks on DataNodes
  - HDFS uses a checkpointing mechanism denoted as the Secondary NameNode
  - The secondary NameNode is not a backup NameNode (hence, it cannot take over the primary NameNode's function upon a NameNode failure)
  - The secondary NameNode holds an out-of-date copy of the primary's persistent state, which, in extremis, can be used to recover HDFS's metadata state
  - There is ongoing work to create a true backup NameNode

Fault Tolerance in PVFS

- PVFS originally did not address fault-tolerance at the file system level
  - Hardware reliability systems such as RAID were used to handle disk failures
  - An I/O server failure would lead to loss of data
- PVFS2 introduced “high-availability” configuration for better fault tolerance
  - Optional configuration scheme for PVFS2
  - Similar to HDFS
    - Heartbeat mechanism detects storage node failures
    - Spare nodes can be configured as well to write replicas in parallel

HDFS Versus PVFS

	Hadoop Distributed File System (HDFS)	Parallel Virtual File System (PVFS)
Architecture	Co-locate storage and compute (beneficial to Hadoop model where computation is moved closer to the data)	Separate Compute and Storage Nodes
Metadata Location	Central Metadata on NameNode	Central Metadata on MGR
Data Layout	File blocks randomly placed; block location exposed to Hadoop applications	Round-robin striping of data blocks; location not exposed to applications
Synchronization and Sharing Semantics	Immutable Semantics – Only one concurrent writer per file	Unix Semantics - POSIX sequential consistency guaranteed for non-conflicting writes
Naming	Central Namespace maintained by NameNode, accessed through API calls	FS mounted simultaneously to all nodes, metadata handled by Master node
Client-Side Caching	Only for Metadata	Only for Metadata
Fault Tolerance	Heartbeat mechanism and rack-aware replication	No file system level support; relies on RAID on/across storage devices; PVFS2 introduces HA mode similar to HDFS
Designed for	Data Intensive Computing, Cloud Computing	High Performance Computing

Next Class

BigTable