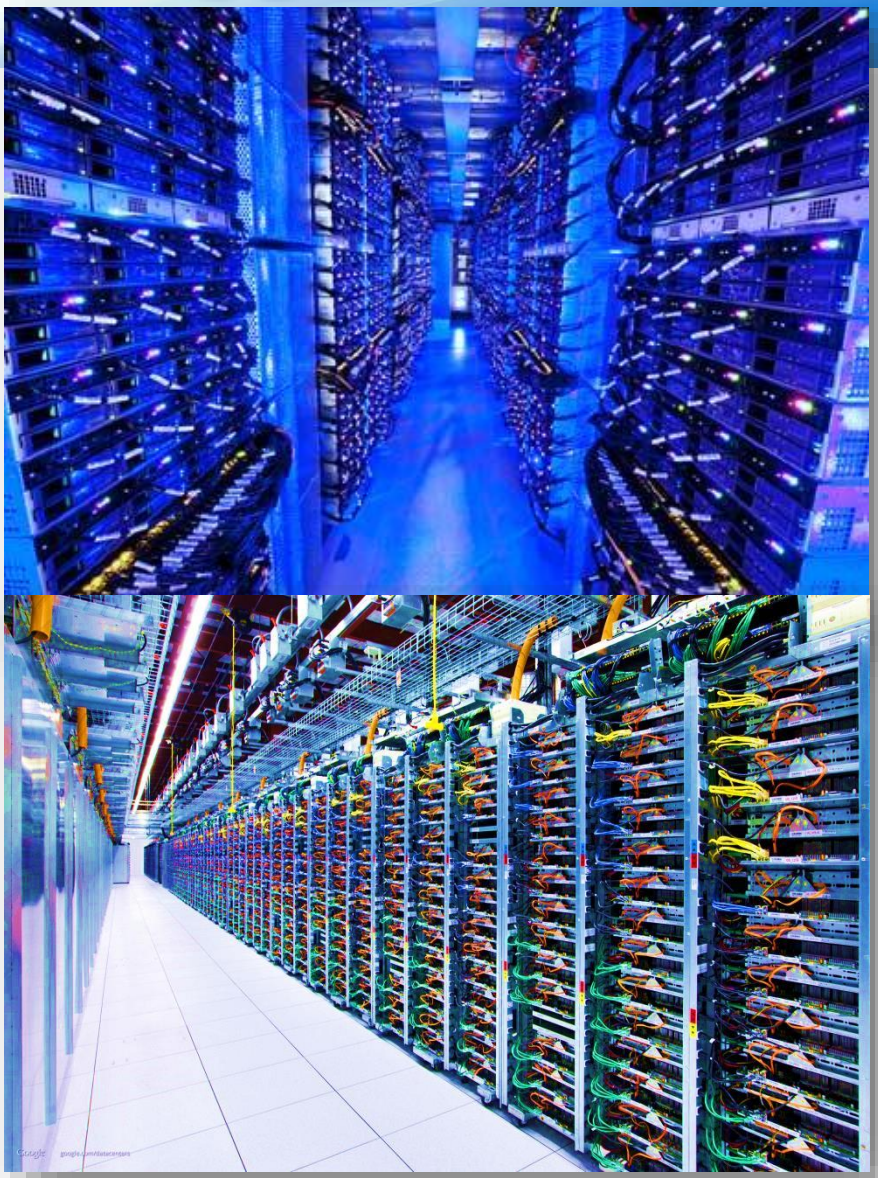
Distributed Computer Systems Engineering

CIS 508: Lecture 4
Fault Tolerance

Lecturer: Sid C.K. Chau
Email: ckchau@masdar.ac.ae

Applications considering Failures



- Google datacenters comprise of many commodity computers (cheap low-end computers), each which can fail any time
- Communication is not reliable (e.g. noise, limited capacity, equipment failure)
- Devices and processors in critical environments (hot temperature) can easily fail

What are Failures

- A system is said to “fail” when it cannot meet its promises
- A failure is brought about by the existence of “errors” in the system
- The cause of an error is a “fault”

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure Receive omission Send omission	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure Value failure State transition failure	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Fault Tolerance

Fault Tolerance is related to the notion of *dependability*:

- ***Availability***:
  - Likelihood that the system can operate correctly at any time
- ***Reliability***:
  - Ability of the system to run correctly for a long interval of time
- ***Safety***:
  - Ability of the system to prevent failures; incorrect operations may lead to catastrophic failures
- ***Maintainability***:
  - Ability that the system to be repaired in the presence of failures

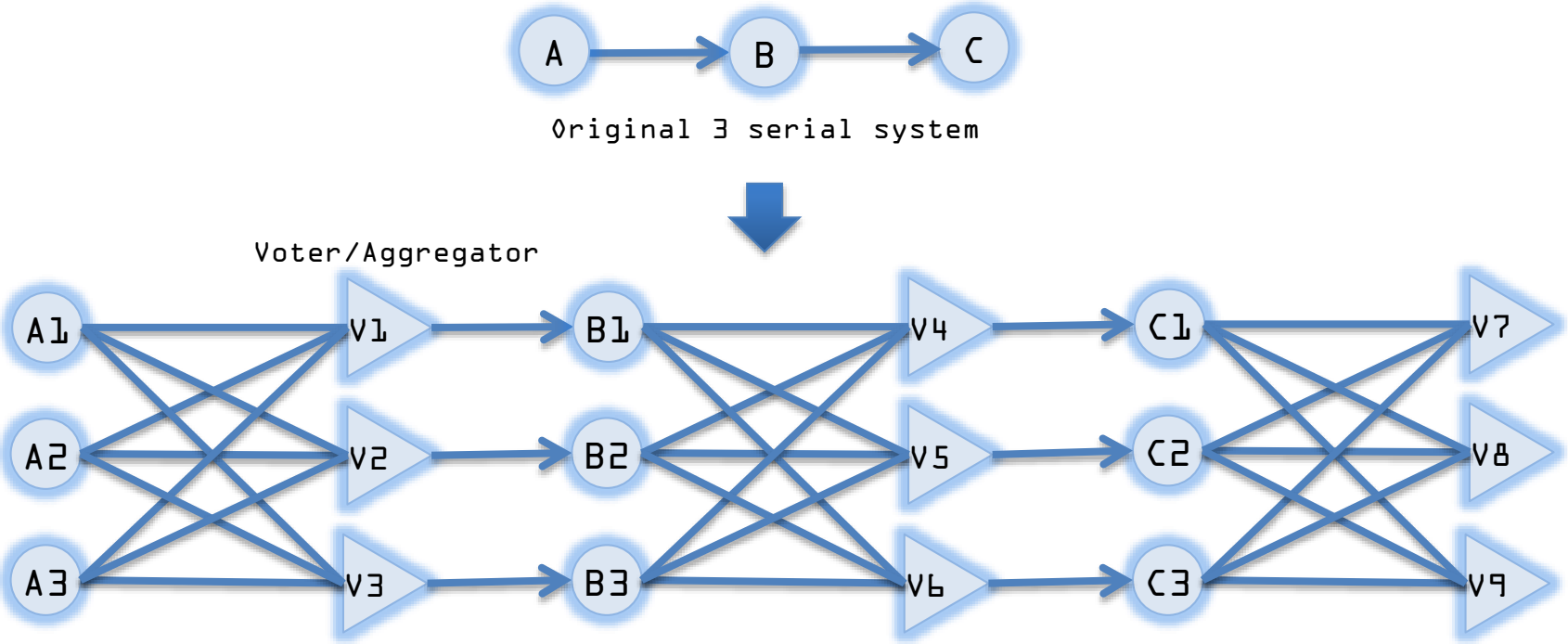
Solution: Redundancy

Hiding failure from other processes using redundancy:

- ***Time Redundancy***
  - Performing operations and, if needed, perform them again (e.g. transactions: `BEGIN/END/COMMIT/ABORT`)
- ***Physical Redundancy***
  - Adding extra hardware and/or software to duplicate the system
  - Parallel duplicate operations are in place
- ***Information Redundancy***
  - Adding extra bits for error detection/recovery
  - Backing up in log files, and roll-back to logged states

Physical Redundancy: Example

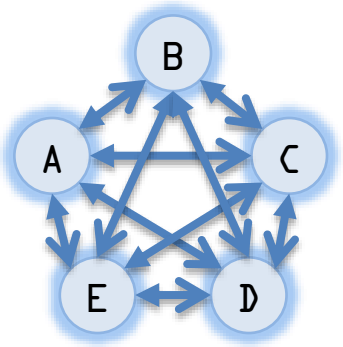
- ***Triple modular redundancy***:
  - Each process is replicated three times
  - If 2-3 inputs of a *voter* are the same, the output is equal to that input
  - A voter itself might be faulty, hence, a separate voter at each stage



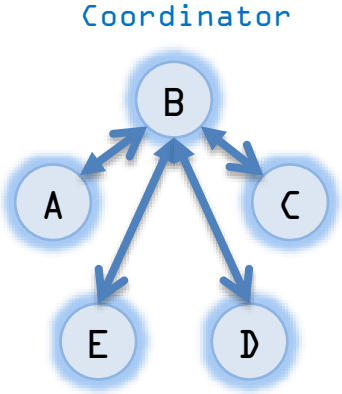
Duplicate Processes, One Operation

- Processes can be made fault tolerant by arranging to have a group of duplicate processes
  - Each member of the group being *identical*
- A message sent to the group is delivered to all of the “copies” of the process
  - Then ***only one*** of them performs the required service
- If one of the processes fails, it is assumed that one of the others will still be able to function (and service any pending request or operation)
  - Assuming that other processes (or an external process) can monitor the state of the operating process

Flat Groups vs. Hierarchical Groups



- **Communication in a flat group** – all the processes are identical; decisions are made collectively
  - No single point-of-failure; but decision making is complicated and requires consensus



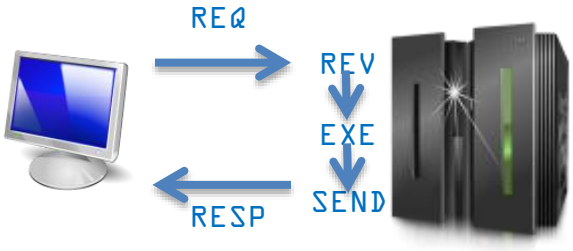
- **Communication in a simple hierarchical group** - one of the processes is elected as coordinator, which selects another process (a subordinate) to perform the operation
  - Single point-of failure; but decisions are efficient without having to get consensus

Client-Server Reliability

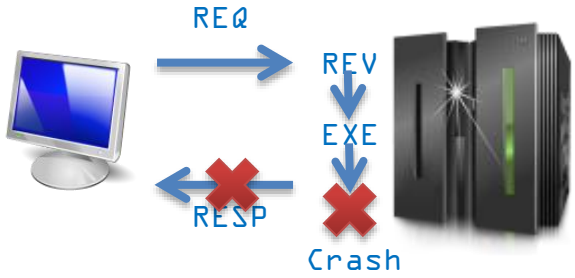
- Common causes of client-server failures:
  - The client is unable to locate the server
  - The request message from the client to server is lost
  - The reply message from the client is lost
  - The server crashes after receiving a request
  - The client crashes after sending a request
- Besides of process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures
  - For example: point-to-point TCP masks omission failures by guarding against lost messages using ACKs and retransmissions

Server Crashes

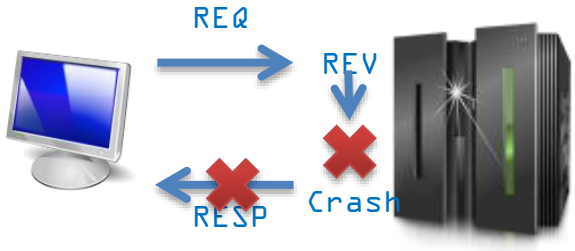
- Crash after execution
- Crash before execution
- Server's behavior
  - *Stateful*: If a client-server communication protocol cannot proceed unless the server fully recovers information it had on clients
  - *Stateless*: If the server can continue working even after losing this information



Normal operation



Crash after execution



Crash before execution

Lost Reply Message

- Lost request message
  - Client keeps trying until it gets a reply
  - Client gives up on failure of locating server
- Lost reply message is difficult to deal with
  - No reply: Is the server *dead*, *slow*, or did the reply just go *missing*?
- A request that can be repeated any number of times without any nasty side-effects is said to be *idempotent*
  - For example: a read of a static web-page is said to be idempotent
- *Nonidempotent* requests (e.g. electronic transfer of funds) are harder to deal with
  - A common solution is to employ *unique sequence numbers*

Client Crashes

- When a client crashes, and 'old' reply arrives, such a reply aka an *orphan*
  - Orphans can waste CPU cycles
  - Cause unintended data locking
- Four orphan solutions:
  - *Extermination*:
    - Client keeps a log, reads it when reboots, and kills the orphan
    - Disadvantage: high overhead to maintain the log
  - *Reincarnation*:
    - Divide times in epochs
    - Client broadcasts epoch when reboots
    - Upon hearing a new epoch servers kills the orphans
    - Disadvantage: cannot solve problem when network disconnected

Client Crashes

- Orphan solutions
  - *Expiration*:
    - Each client-server operation is given a lease *T* to finish computation
    - If it does not finish, it needs to ask server for another lease
    - If client reboots after *T* sec all orphans are gone
    - Problem: difficult to set a good value of *T*
  - *Gentle reincarnation*:
    - When a new epoch is identified, an attempt is made to locate a requests owner
    - Otherwise the orphan is killed

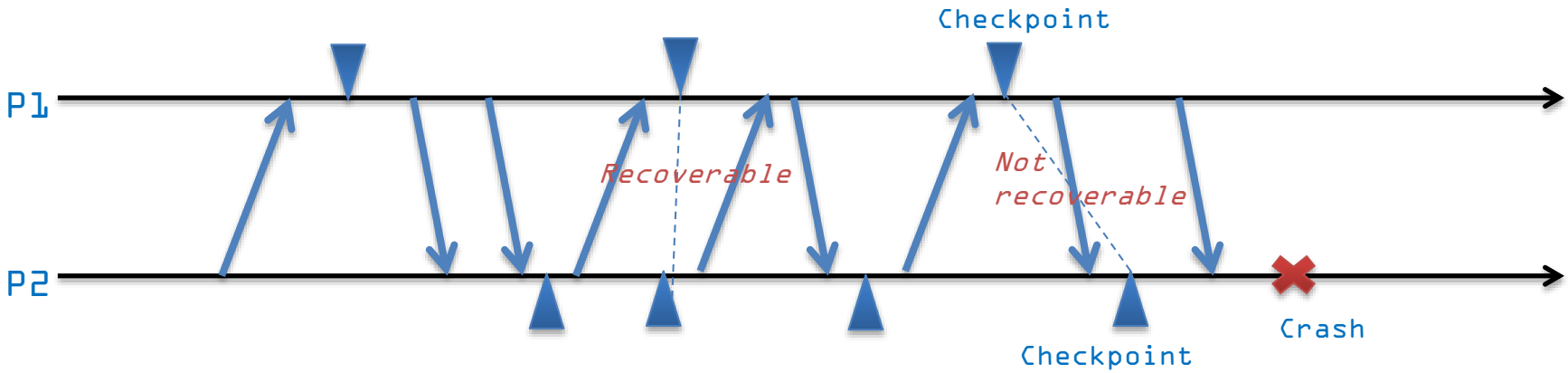
Recovery

- Recovery tries to correct errors due to faults (e.g. crashes, message lost)
- *Backward recovery*
  - Recovery to the previous state before faults occur
  - Examples: checkpointing, retransmit lost packets
  - Advantages: simple techniques
  - Disadvantages: inefficiency, ignoring partial progress during faults
- *Forward recovery*
  - Recovery to the corrected state after faults occur
  - Example: error-correction codes
  - Advantages: efficiency
  - Disadvantage: more complicated techniques, not always work

Backward Recovery: Checkpointing

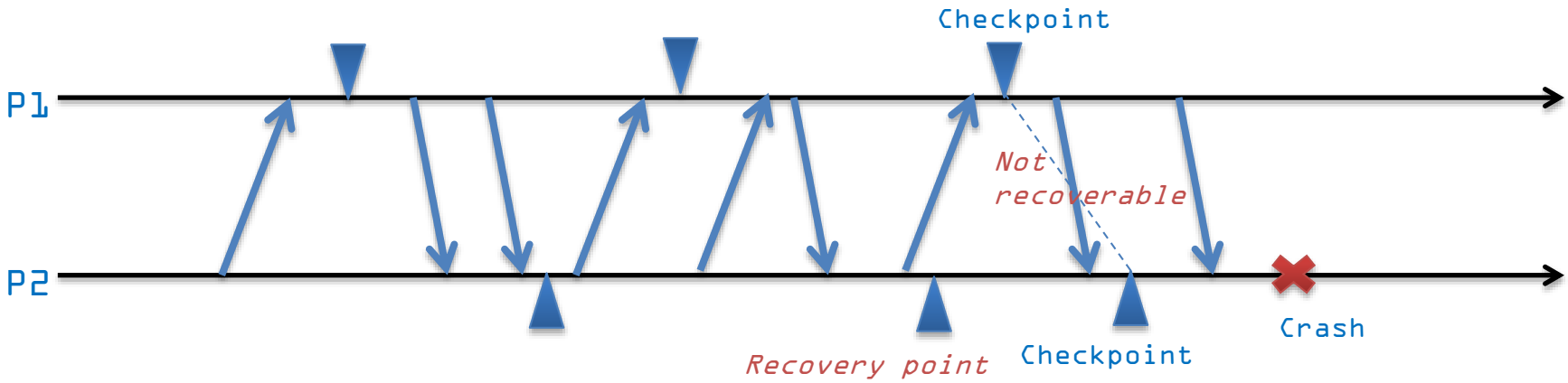
- Checkpointing is a useful approach for recovery
  - Consistent global state (distributed snapshot)
  - If *P* recorded receipt of a message from *Q*
  - Then *Q* also has a record that *Q* has sent a message to *P*
- Rolling back to consistent global state
  - When a process crashes and restarts
  - All processes roll back to the most recent consistent global state

Backward Recovery: Checkpointing



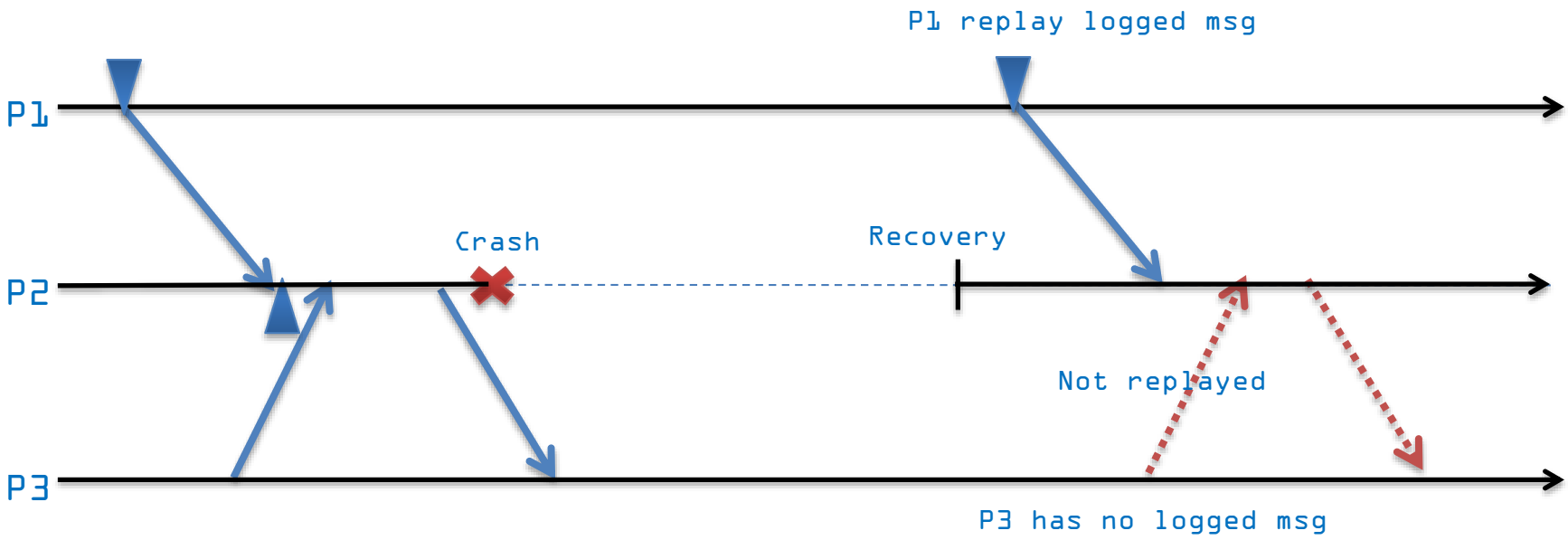
- Rolling back to consistent global state
  - When a process crashes and restarts
  - All processes roll back to the most recent consistent global state
- When two checkpoints give inconsistent global state
  - Checkpoints cannot recover the faults after them

Backward Recovery: Checkpointing



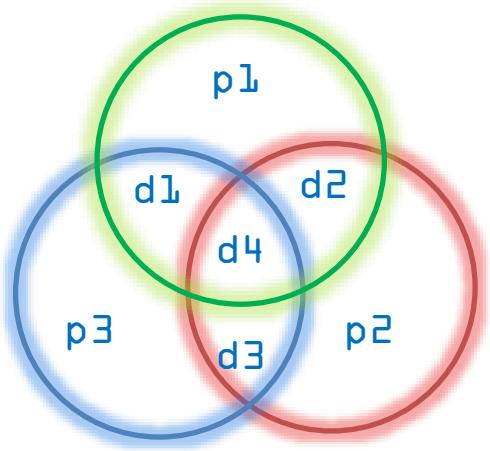
- Iterative rolling-back is required
  - Keep rolling-back until the checkpoints give consistent global state
- The default consistent global state will be the initial state
  - No process has sent out messages

Backward Recovery: Checkpointing



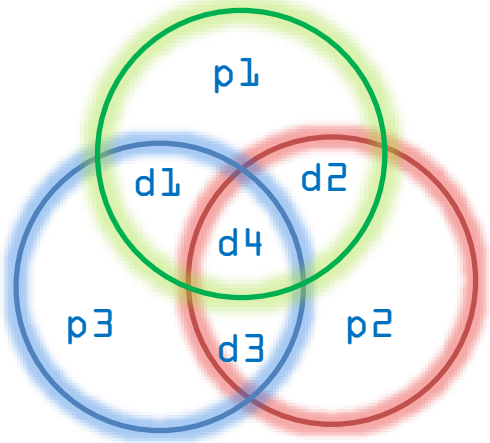
- If messages are dependent on each other
- Simple rolling-back is not sufficient
- Incorrect replay of messages after recovery, leading to an orphan process

Forward Recovery: Hamming Code



- Error correction code
  - Use information redundancy to include more information for error correction and detection
- Hamming (7,4) code
  - 4 bits for data
  - 3 bits for redundancy (parity bits)
  - Can detect two bits of error
  - Can correct one bit of error

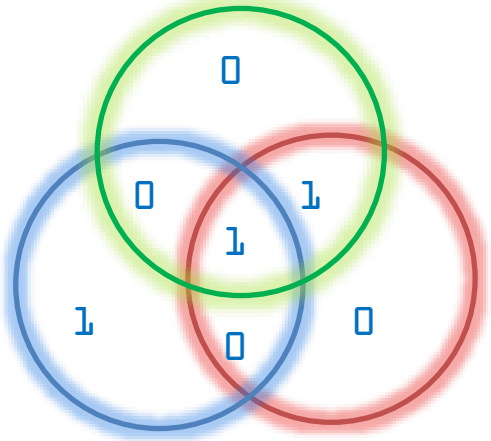
Forward Recovery: Hamming Code



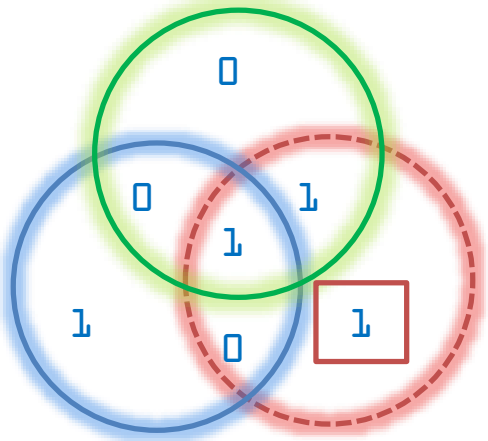
- XOR operator: ⊕
  - 1 ⊕ 0= 1
  - 0 ⊕ 1= 1
  - 0 ⊕ 0= 0
  - 1 ⊕ 1= 0
- d1, d2, d3 , d4 are data bits
- p1, p2, p3 are parity bits
  - p1 = d1 ⊕ d2 ⊕ d4
  - p2 = d2 ⊕ d3 ⊕ d4
  - p3 = d1 ⊕ d3 ⊕ d4

- Correction
  - One parity error
    - Parity bit is incorrect
  - Two/Three parity errors
    - The common data bit is incorrect

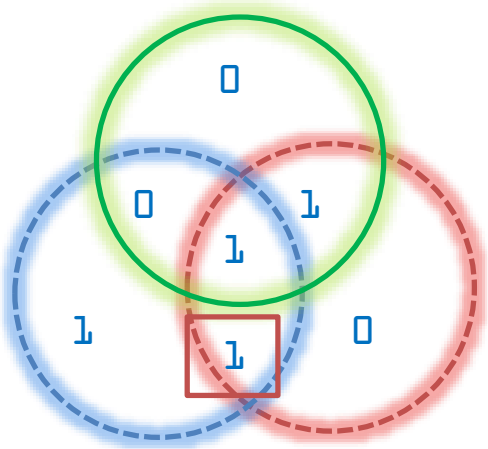
Forward Recovery: Hamming Code



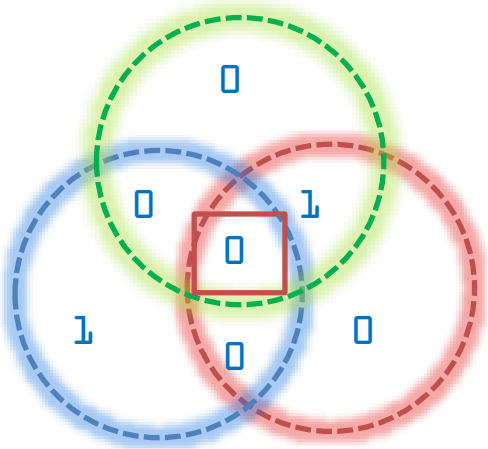
No error



1 parity error: incorrect parity bit



2 parity errors: incorrect data bit

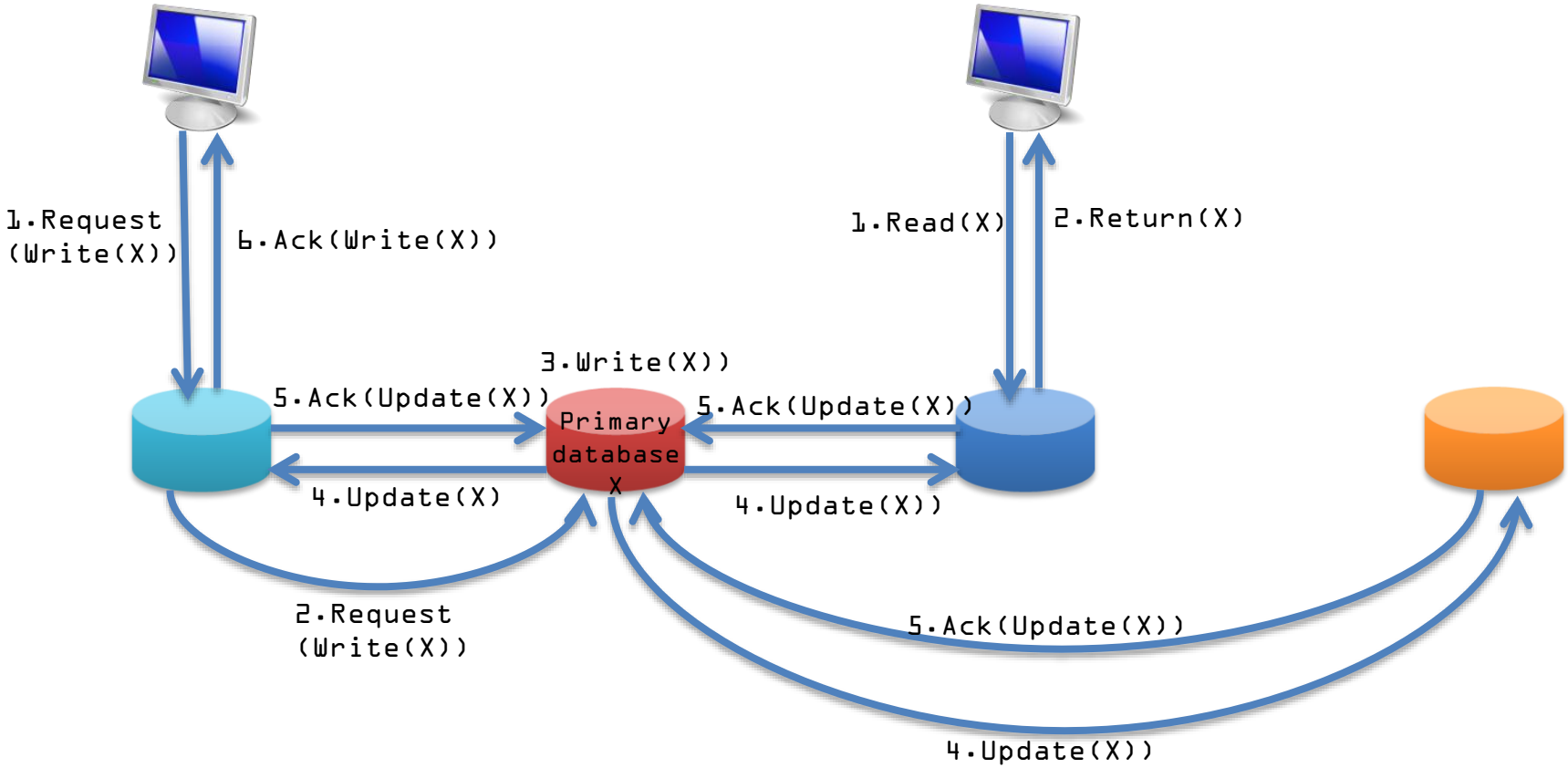


3 parity errors: incorrect data bit

Failure Masking by Replication

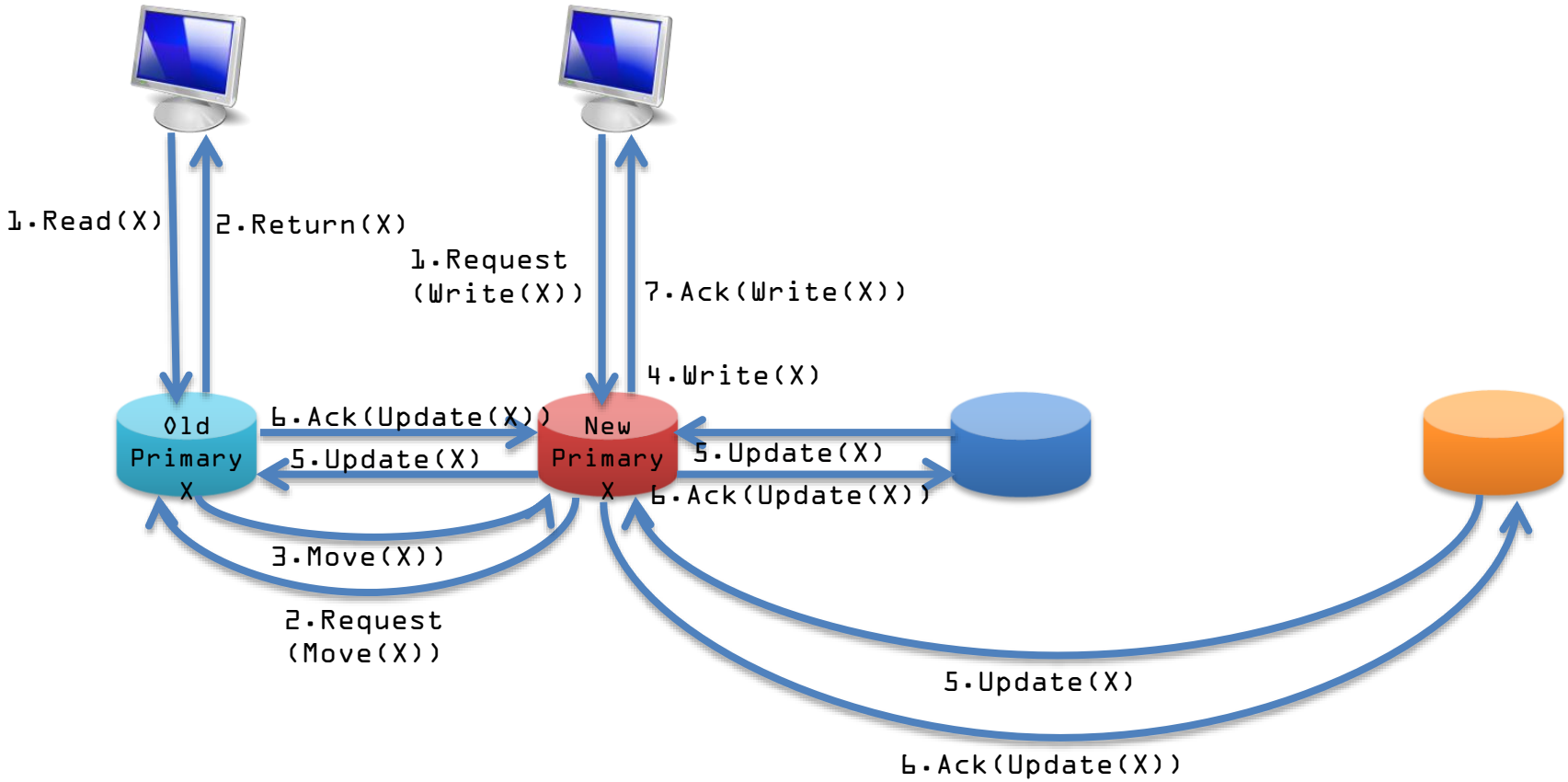
- By organizing a fault tolerant group of replicate data server, we can protect a single vulnerable data server
- Two approaches to arranging the replication of the group:
  - Primary (backup) Protocols
    - A primary server coordinate all write operations on all backup servers
  - Replicated-Write Protocols
    - Group of identical servers; quorum-based protocols

Primary (Backup) Remote Read/Write



- A primary process controls all write operations to other backups
- Read can be done to local (nearest) process

Primary (Backup) Local Read/Write

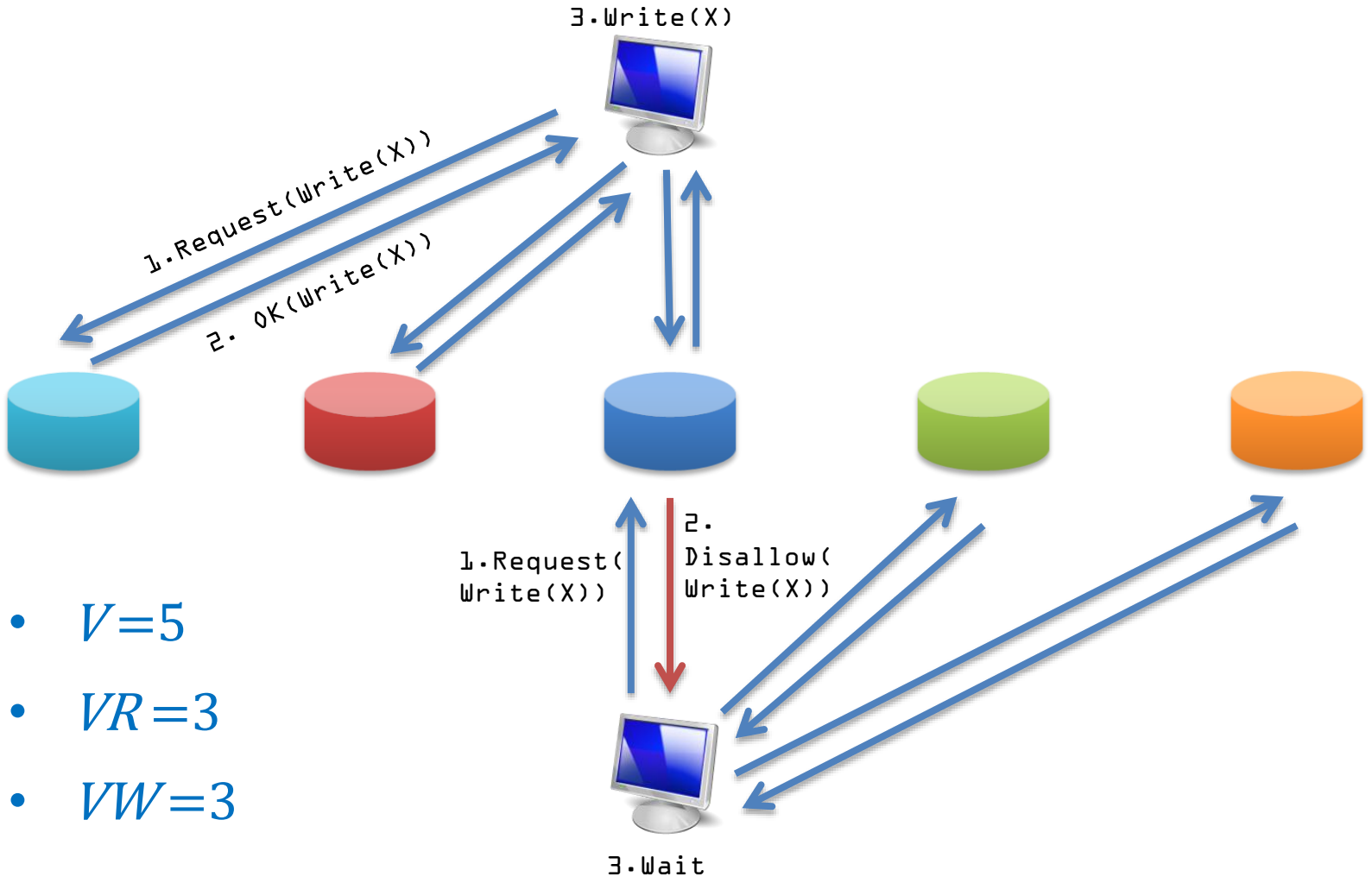


- A primary process will migrates to the nearest process to clients
- Optimize performance and maintain consistency for infrequent write operations

Quorum-based Protocols

- No primary copy! Distributed Writing
- Assign a number of votes $V(I)$ to each replica I
- Let V be the total number of votes
- Let VR = read quorum, VW = write quorum
- Example Conditions
  - $VR + VW > V$
  - $VW > V/2$
- Only one writer at a time can achieve write quorum
- Every reader sees at least one copy of the most recent read (takes one with most recent version number)

Quorum-based Protocols



- $V=5$
- $VR=3$
- $VW=3$

Possible Quorum Policies

- ROWA (Read-Once-Write-All):
  - $VR = 1,\ VW = N$
  - Fast reads, slow writes (and easily blocked)
- RAWO (Read-All-Write-Once):
  - $VR = N,\ VW = 1$
  - Fast writes, slow reads (and easily blocked)
- Majority:
  - $VR = VW = N/2 + 1$
  - Both moderately slow, but extremely high availability

Google's Solution

- Google use many inexpensive machines running on Linux for its mega file system
- System requirements
  - Component failures are common
    - 1000s of components fail per cluster per year
    - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
    - Monitoring, error detection, fault tolerance, automatic recovery
  - Files are huge by traditional standards
    - Multi-GB files are common
    - Billions of objects

Google's Solution

- System requirements
  - Most modifications are appends
    - Random writes are practically nonexistent
    - Many files are written once, and read sequentially
  - Two types of reads
    - Large streaming reads
    - Small random reads (in the forward direction)
  - Sustained bandwidth more important than latency
  - File system APIs are open to changes

Scaling Out

- The scaling “up" approach
  - A small number of high-end servers
  - Require symmetric multi-processing (SMP) machines with a large number of processor sockets and a large amount of shared memory
  - Not cost effective, since the costs of such machines do not scale linearly (i.e., a machine with twice more processors is often significantly more than twice as expensive)
- The scaling “out" approach
  - A large number of commodity low-end servers
  - Low-end server market overlaps with the high-volume desktop computing market, which has the effect of keeping prices low due to competition, interchangeable components, and economies of scale

Google File System

- Google designed its own mega file system Google File System (GFS)
  - Capitalizes on the strengths of off-the-shelf servers
  - Cope with for any hardware weaknesses
  - Organize and manipulate huge files and to allow application developers the research and development resources they require
- GFS is unique to Google and is not for sale
- Detailed implementation of GFS remains a secret
- Some papers discuss the high-level architecture of GFS

Google File System

- Multiple GFS clusters are currently deployed
- The largest ones have
  - 1000+ storage nodes
  - 300+ TeraBytes of disk storage
  - Heavily accessed by hundreds of clients on distinct machines
- Share many same goals as previous distributed file systems
  - Performance, scalability, reliability, etc
  - GFS design has been driven by four key observation of Google application workloads and technological environment

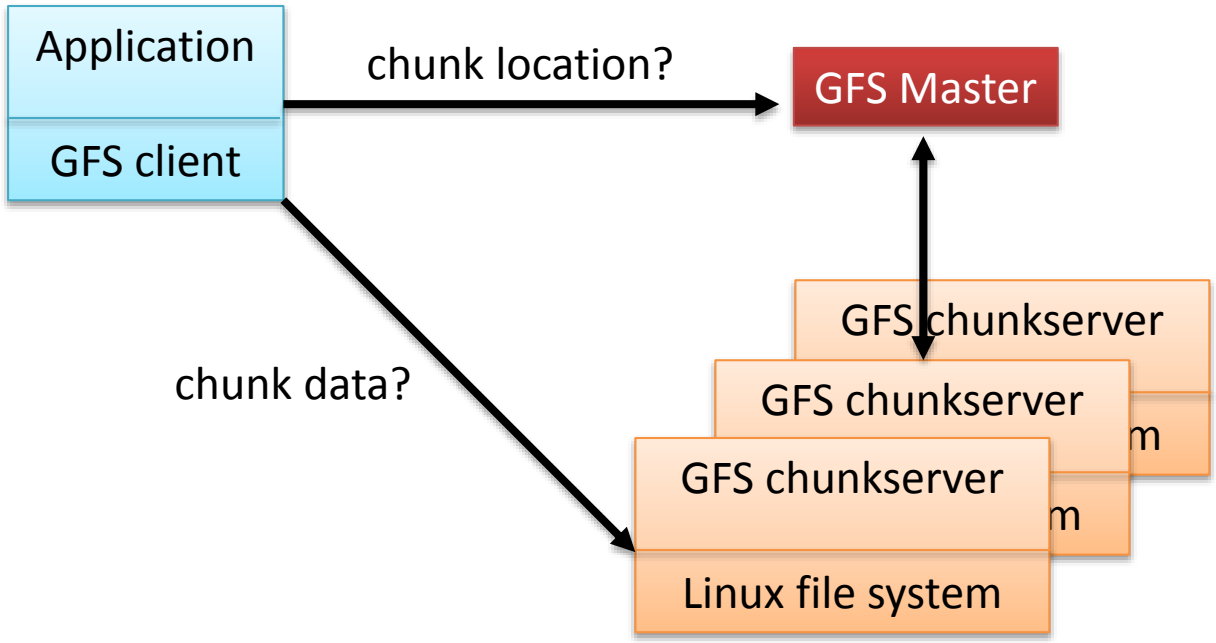
Google Characteristics

- Component failures are the norm
  - constant monitoring, error detection, fault tolerance and automatic recovery are integral to the system
- Huge files (by traditional standards)
  - Multi GB files are common
  - I/O operations and blocks sizes must be revisited
- Most files are mutated by appending new data
  - This is the focus of performance optimization and atomicity guarantees
- Co-designing the applications and APIs benefits overall system by increasing flexibility

GFS: Architectural Design

- A GFS cluster
  - A single *master*
  - Multiple *chunkservers* per master
    - Accessed by multiple *clients*
  - Running on commodity Linux machines
- A file
  - Represented as fixed-sized *chunks*
    - Labeled with 64-bit unique global IDs
    - Stored at chunkservers
    - 3-way mirrored across chunkservers

GFS: Architectural Design

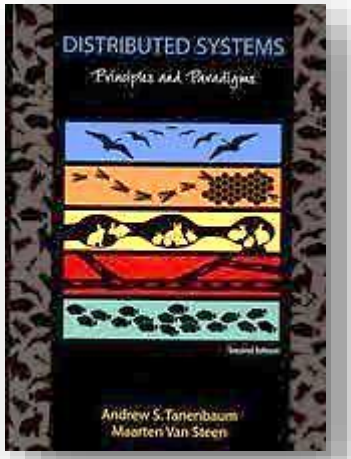


GFS: Architectural Design

- Single-Master Design
  - Master answers only chunk locations
  - A client typically asks for multiple chunk locations in a single request
  - The master also predicatively provide chunk locations immediately following those requested
- Chunkservers
  - Files are broken into chunks
  - Each chunk has a immutable globally unique 64-bit chunk-handle (Handle is assigned by the master at chunk creation)
  - Chunk size is 64 MB
    - Fewer chunk location requests to the master
    - Reduced overhead to access a chunk
    - Fewer metadata entries (Kept in memory)
    - Some potential problems with fragmentation
  - Each chunk is replicated on 3 (default) servers

References

- Textbooks
  - Distributed Systems: Principles and Paradigms
    Andrew Tanenbaum and Maarten van Steen; Prentice Hall



- "The Google File System", Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung; ACM Symposium on Operating Systems Principles, 2003