ECE454 Notes

July 7, 2019

1 Introduction

1.1 Distributed System

- **Distributed System**: A collection of autonomous computing elements that appears to its users as a single coherent system.

1.2 Motivations for Distributed Systems

1. Resource Sharing
2. Simplify Processes by Integrating Multiple Systems
3. Limitations in Centralized Systems: Weak/Unreliable
4. Distributed/Mobile Users

1.3 Goals for Distributed Systems

1. Resource Sharing

   - CPUs, Data, Peripherals, Storage.

2. Transparency

   - Access, Location, Migration, Relocation, Replication, Concurrency, Failure.

3. Open

   - Interoperability, Composability, Extensibility.

4. Scalable

   - Size, Geography, Administration.

1.4 Types of Distributed Systems

- Web Services
- High Performance Computing, Cluster Computing, Cloud Computing, Grid Computing
- Transaction Processing
- Enterprise Application Integration
- Internet of Things, Sensor Networks

1.5 Middleware

- **Middleware**: A layer of software that separates applications from the underlying platforms.

    - Supports Heterogeneous Computers/Networks.
    - *e.g.*: Communication, Transactions, Service Composition, Reliability.
    - **Single-System View**

1.6 Scaling Techniques

1. *Hiding Communication Latencies*: At Server vs. At Client?
2. *Partitioning*
3. *Replication*

1.7 Fallacies of Networked and Distributed Computing

1. Network is reliable.
2. Network is secure.
3. Network is homogeneous.
4. Topology is static.
5. Latency is zero.
6. Bandwidth is infinite.
7. Transport cost is zero.
8. There is only one administrator.

1.8 Shared Memory vs. Message Passing

- **Shared Memory**:

    - Less Scalable
    - Faster
    - CPU-Intensive Problems
    - Parallel Computing

- **Message Passing**:

    - More Scalable
    - Slower
    - Resource Sharing / Coordination Problems
    - Distributed Computing

- Apache Hadoop is an example of a hybrid computing framework that uses message passing at a broad-view and shared memory at a detailed-view.

1.9 Cloud and Grid Computing

- **IaaS**: Infrastructure as a Service

    - VM Computation, Block File Storage

- **PaaS**: Platform as a Service

    - Software Frameworks, Databases

- **SaaS**: Software as a Service

  - Web Services, Business Apps

1.10   Transaction Processing Systems

- **Transaction Processing Monitor**: Coordinates Distributed Transactions

2   Architectures

2.1   Definitions

- **Component**: A modular unit with well-defined interfaces.
- **Connector**: A mechanism that mediates communication, coordination, or cooperation among components.
- **Software Architecture**: Organization of software components.
- **System Architecture**: Instantiation of software architecture in which software components are placed on real machines.
- **Autonomic System**: Adapts to its environment by monitoring its own behavior and reacting accordingly.

2.2   Architectural Styles

- Layered

  - *Note: Assignment Topic*

- Object-Based
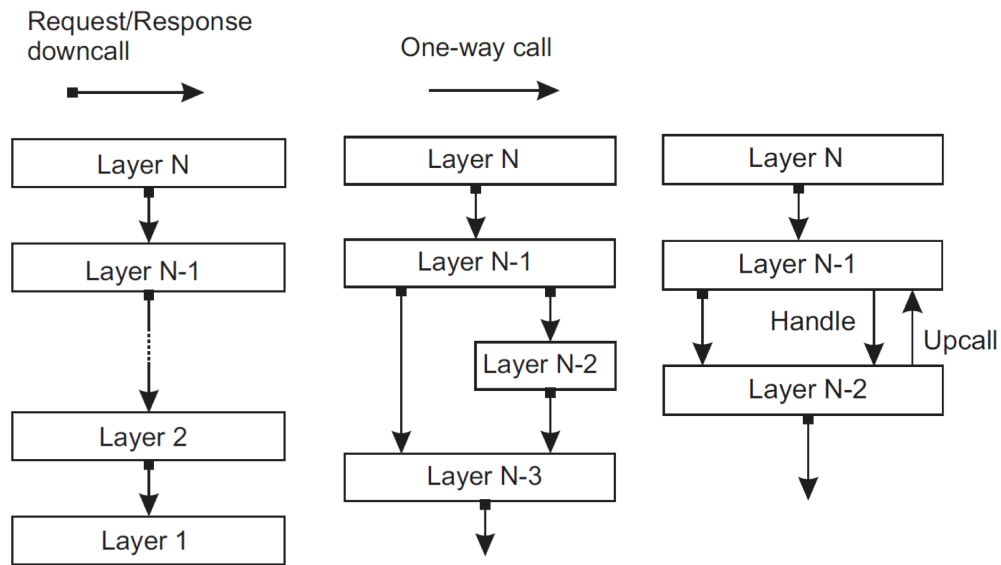- Data-Centered
- Event-Based

2.3 Layered Architecture

Figure 2.1: (a) Pure layered organization. (b) Mixed layered organization. (c) Layered organization with upcalls (adopted from [Krakowiak, 2009]).

Layers

- **Examples**:
    - Database Server, Application Server, Client
    - SSH Server, SSH Client

- Requests Flow Down Stack
- Responses Flow Up Stack
- *Handle-Upcall*: Async Notification

    - Subscribe with Handle
    - Publish with Upcall

2.4 Client-Server Interactions
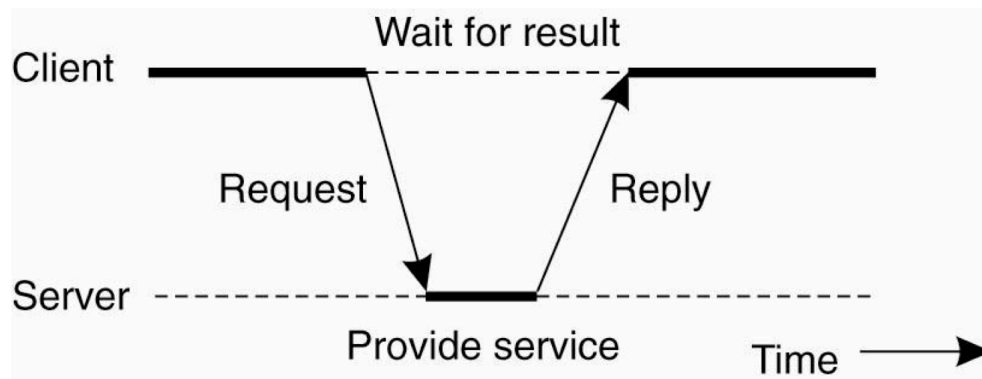
Client-Server Interactions

- *Bolded Lines* = Busy
- *Dashed Lines* = Idle
- **Client**: Initiates with a Request
- **Server**: Follows with a Response
- **Total Round-Trip Time**: $(N - 1) \times t_{\text{Request-Response}}$

  - Layering can reduce the amount of processing time per layer, but the additional communication overhead between the layers introduces diminishing returns.

- An intermediate layer can be both a client and a server to the others.

2.5 Multi-Tiered Architecture

- Logical Software Layers $\mapsto$ Physical Tiers

  - *Trade-Offs*: Ease of Maintenance vs. Reliability

2.6 Horizontal vs. Vertical Distribution

- **Vertical Distribution**: When the logical layers of a system are organized as separate physical tiers.

  - *Performance*: High.
  - *Scalability*: Low.
  - *Dependability*: Low-Medium.

- **Horizontal Distribution**: When one logical layer is split across multiple machines - **sharding**.

  - *Performance*: Low.
  - *Scalability*: High.
  - *Dependability*: Medium-High.

2.7 Object-Based Architecture

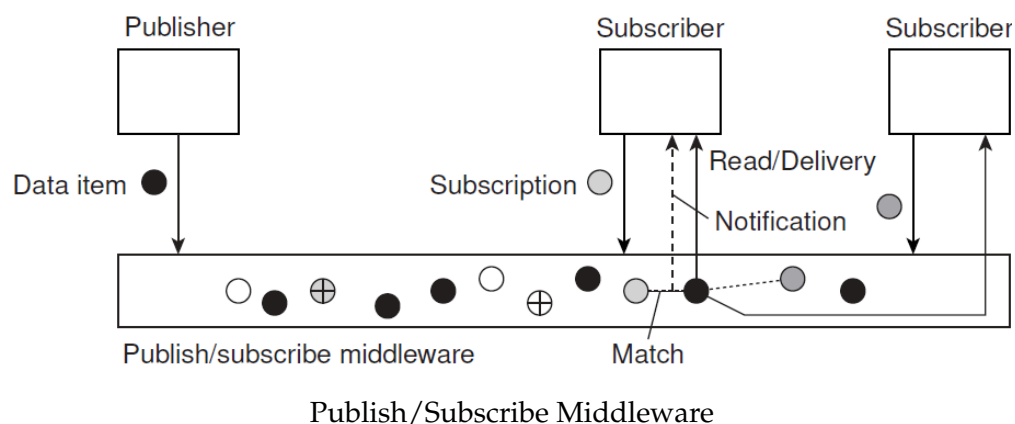- In an object-based architecture, components communicate using remote object references and method calls.

2.8 Problems with Object-Based Architecture

- Complex Communication Interfaces
- Complex Communication Costs
- Not Scalable
- Not Language Agnostic

2.9 Data-Centered Architecture

- In a data-centered architecture, components communicate by accessing a shared data repository.

2.10 Event-Based Architecture

Publish/Subscribe Middleware

- In an event-based architecture, components communicate by propagating events using a publish/subscribe system.

2.11 Handling Asynchronous Delivery Failure

- *At-Least Once Delivery*: Do Retransmit
- *At-Most Once Delivery*: Do Not Retransmit
- *Exactly Once Delivery*: Unknown/Unachievable
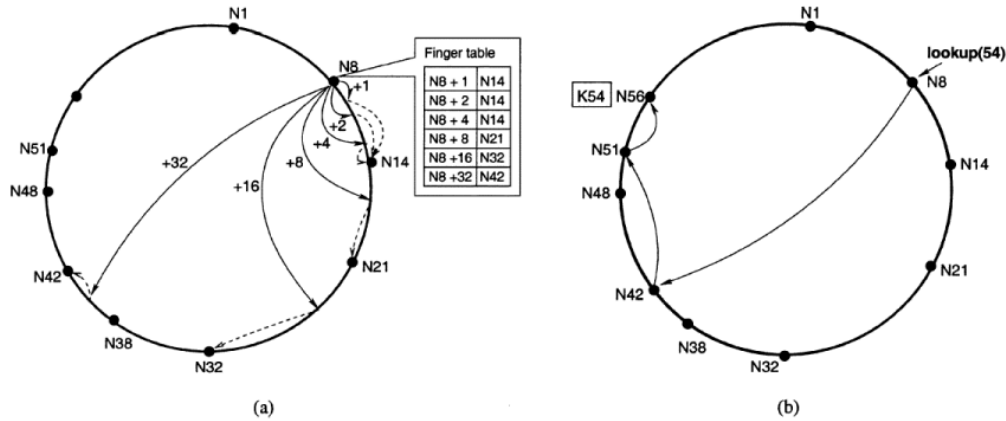
2.12 Peer-to-Peer Systems

Fig. 4. (a) Finger table entries for node 8. (b) Path of a query for key 54 starting at node 8, using the algorithm in Fig. 5.

Chord's Finger Table

- In a peer-to-peer system, decentralized processes are organized in an overlay network that defines a set of communication channels.
- In a peer-to-peer, distributed hash table, a keyspace is represented by a consistent hash ring on top of which nodes partition ranges amongst themselves.
- The mappings of partition ranges to nodes are maintained by a finger table which can be queried in a logarithm process.

2.13 Hybrid Architectures

- BitTorrent is an example of a hybrid architecture combining a client-server architecture and a peer-to-peer architecture.

2.14 Self-Management
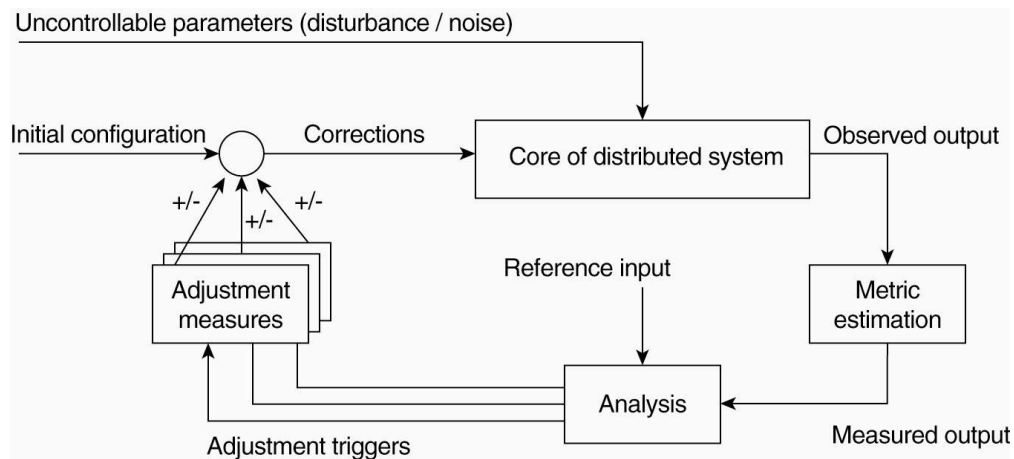
Self-Management Systems

- In self-management, systems use a feedback control loop that monitors system behaviors and adjusts system operations.
- **Assignment Note**: Useful for Unknown Assignment

3 Processes

3.1 IPC

- **Inter-Process Communication (IPC)**: Expensive b/c Context Switching

3.2 Threads

- Typically, an operating system kernel support multi-threading through **lightweight processes (LWP)**.
- **Assignment Note**: Do Not Spawn Too Many Threads
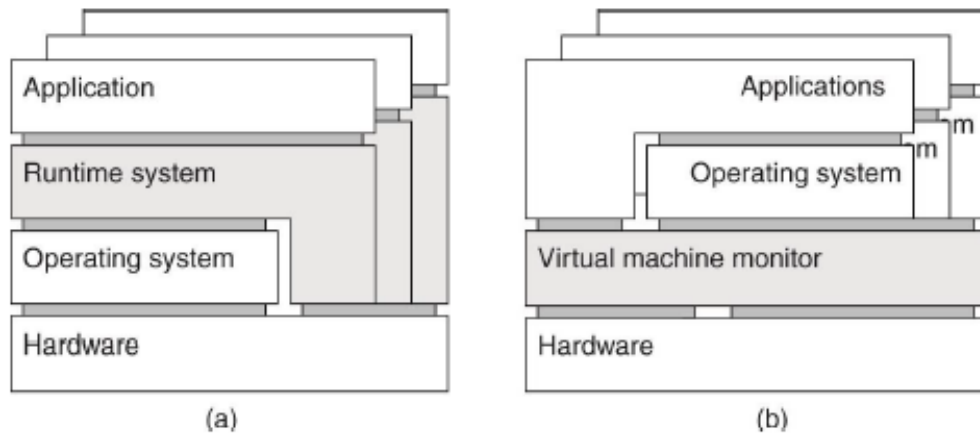
3.3 Multi-Threaded Servers

- **Dispatcher/Worker Design**: A dispatcher thread receives requests from the network and feeds them to a pool of worker threads.
- **Assignment Note**: Useful for Assignment 1 & Partition into Sequential Work and Parallel Work

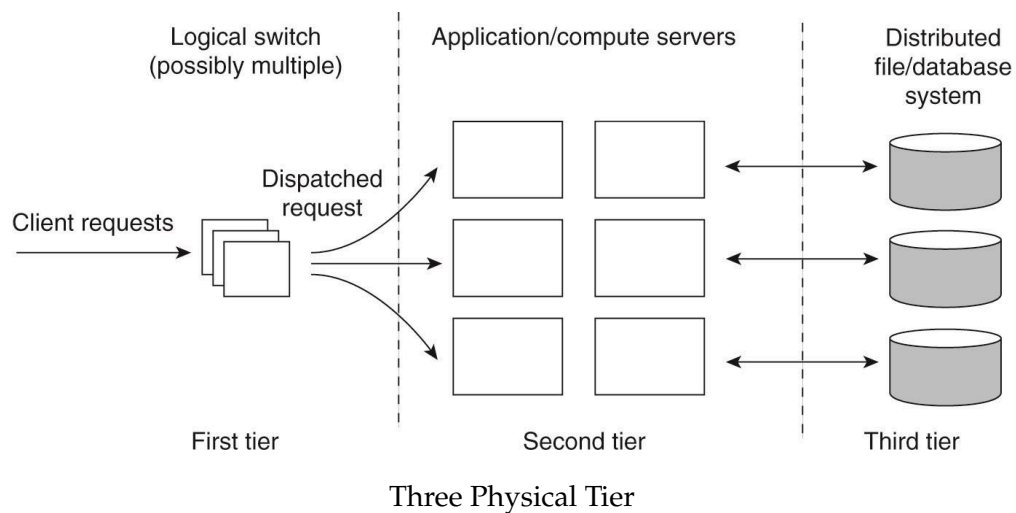## 3.4 Hardware and Software Interfaces
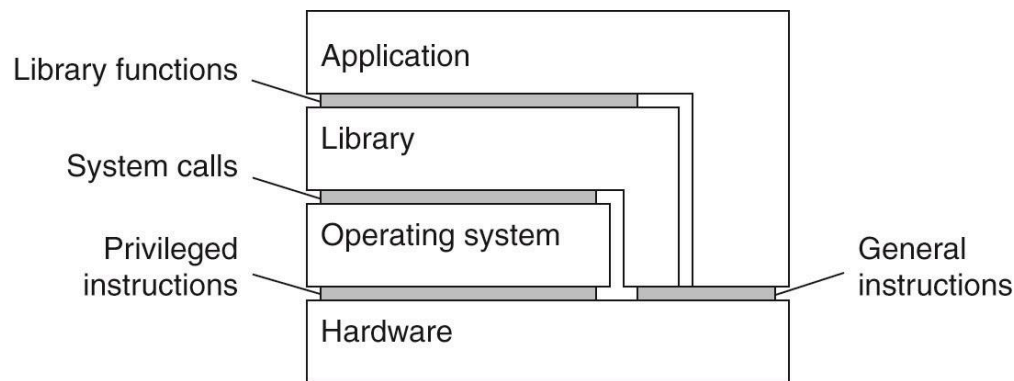
Hardware and Software Interfaces

3.5 Virtualization

(a)   (b)

VMs

- **Advantage**:
  - Portability
  - Live Migration of VMs
  - Replication for Availability/Fault Tolerance

- **Disadvantage**:
  - Performance
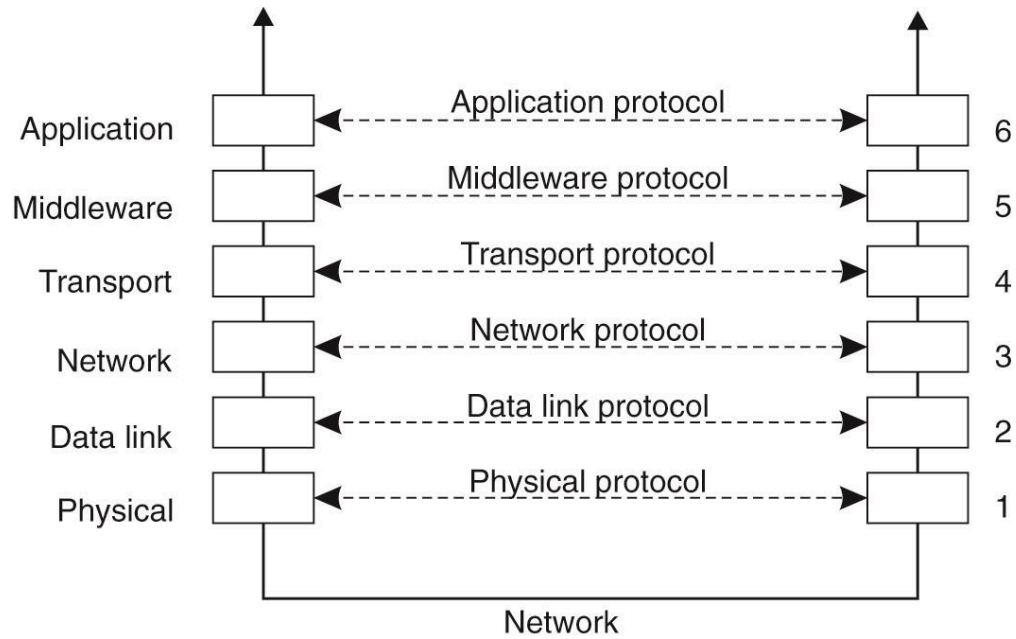
3.6 Server Clusters

Three Physical Tier

- **Assignment Note**: Useful for Assignment 2

4 Communication
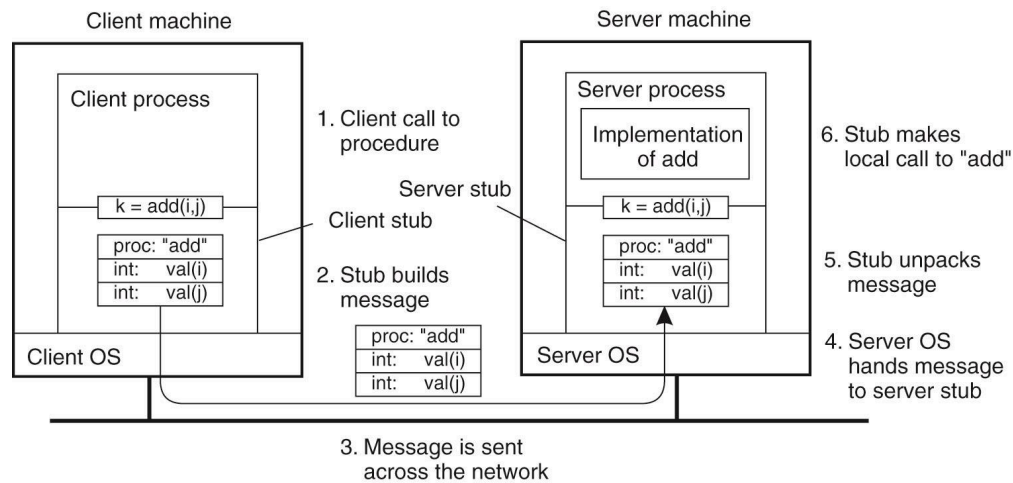
4.1 Layered Network Model

Layered Network Model

4.2 Remote Procedure Calls

- **Remote Procedure Calls**: A transient communication abstraction implemented using a client-server protocol.
- **Client Stub**: Translate a RPC on the client.
- **Server Stub**: Translate a RPC on the server.

4.3 Steps of a RPC

Steps of a RPC

1. The client process invokes the client stub using an ordinary procedure call.
2. The client stub builds a message and passes it to the client's OS.
3. The client's OS sends the message to the server's OS.
4. The server's OS delivers the message to the server stub.
5. The server stub unpacks the parameters and invokes the appropriate service handler in the server process.
6. The **service handler** does the work and returns the result to the server stub.
7. The server stub packs the result into a message and passes it to the server's OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS delivers the message to the client stub.
10. The client stub unpacks the result and returns it to the client process.

- **Parameter Marshalling**: Packing Parameter → Message

  - Processor Architectures, Network Protocols, and VMs ⟹ **Little-Endian** vs. **Big-Endian**

- **Number of System Calls**: 4

  1. Client Process → Client OS Socket
  2. Server OS Socket → Server Process
  3. Server Process → Server OS Socket
  4. Client OS Socket → Client Process

4.4 Defining RPC Interfaces

- **Interface Definition Language (IDL)**: Specify RPC Signatures → Client/Server Stubs

  - High-Level Format
  - Parameter Ordering
  - Byte Sizes

4.5 Synchronous vs. Asynchronous RPCs

- **Synchronous RPC**: The client blocks to wait for the return value.
- **Asynchronous RPC**: The client blocks to wait for the server acknowledgement of the receipt of the request.
- **One-Way RPC**: The client does not block to wait.

4.6 Message Queuing Model

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Message Queue Interface

- **Message Queue**: Alternative to RPCs
- **Persistent Communication**: Loose Coupling between Client/Server

  - *Advantage*: Resilient to Client/Server Hardware Failure
  - *Disadvantage*: Guaranteed Delivery = Impossible

- **Message-Oriented Middleware (MOM)**: Asynchronous Message Passing

4.7 Process Coupling

- **Referential Coupling**: When one process explicitly references another.

  - *Positive Example*: RPC client connects to server using an IP address and a port number
  - *Negative Example*: Publisher inserts a news item into a pub-sub system without knowing which subscriber will read it.

- **Temporal Coupling**: Communicating processes must both be up and running.

  - *Positive Example*: A client cannot execute a RPC if the server is down.
  - *Negative Example*: A producer appends a job to a message queue today, and a consumer extracts the job tomorrow.

4.8 RPC vs. MOM

4.9 RPC

- Used mostly for two-way communication, particularly where the client requires immediate response from the server.
- The middleware is linked into the client and the server processes.
- Tighter coupling means that server failure can prevent client from making progress.

4.10 MOM

- Used mostly for one-way communication where one party does not require an immediate response from another.
- The middleware is a separate component between the sender/publisher/producer and the receiver/subscriber/consumer.
- Looser coupling isolates one process from another which contributes to flexibility and scalability.
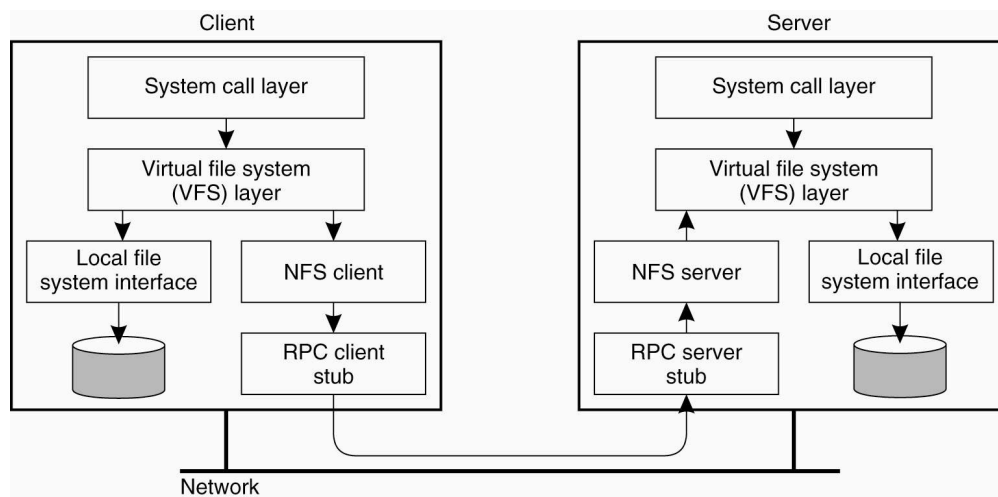
5 Distributed File Systems

5.1 Accessing Remote Files
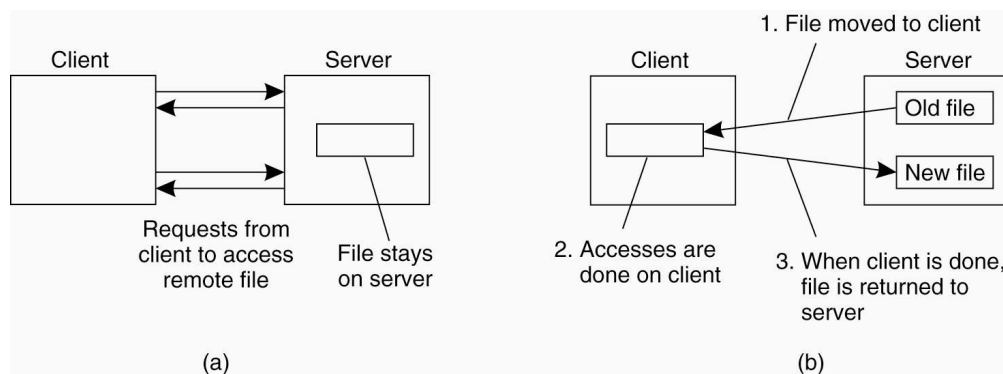
DFS Models

- **Remote Access Model**
- **Upload/Download Model**

5.2 Network File System (NFS)

Overview of NFS

13

- *Supports Client-Side Caching*

    – Modifications are flushed to the server when the client closes the file.
    – Consistency is implementation dependent.

Authority Delegation
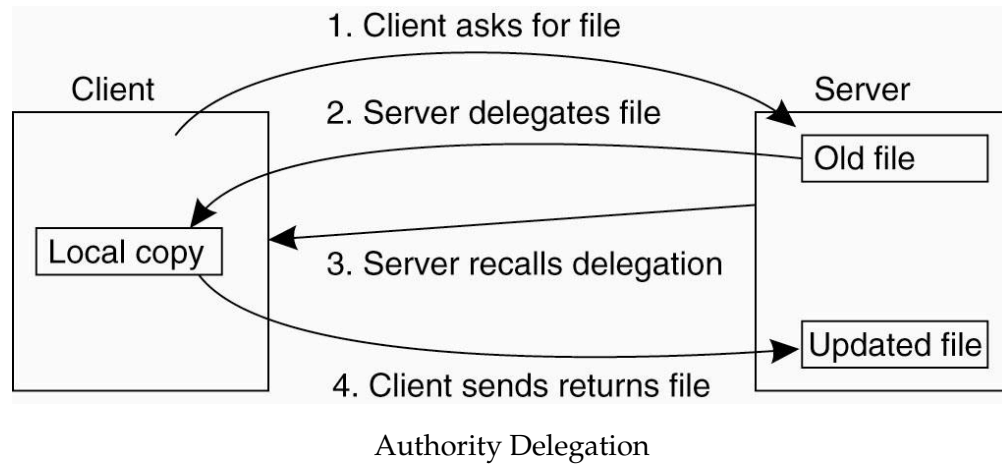
- *Supports Authority Delegation*

    – A server can delegate authority to a client and recall it through a callback mechanism.

Compound Procedure

- *Supports Compound Procedures*
  - Multiple Round Trips to Single Round Trip

Partial Exports

15

- *Supports Partial Exports*

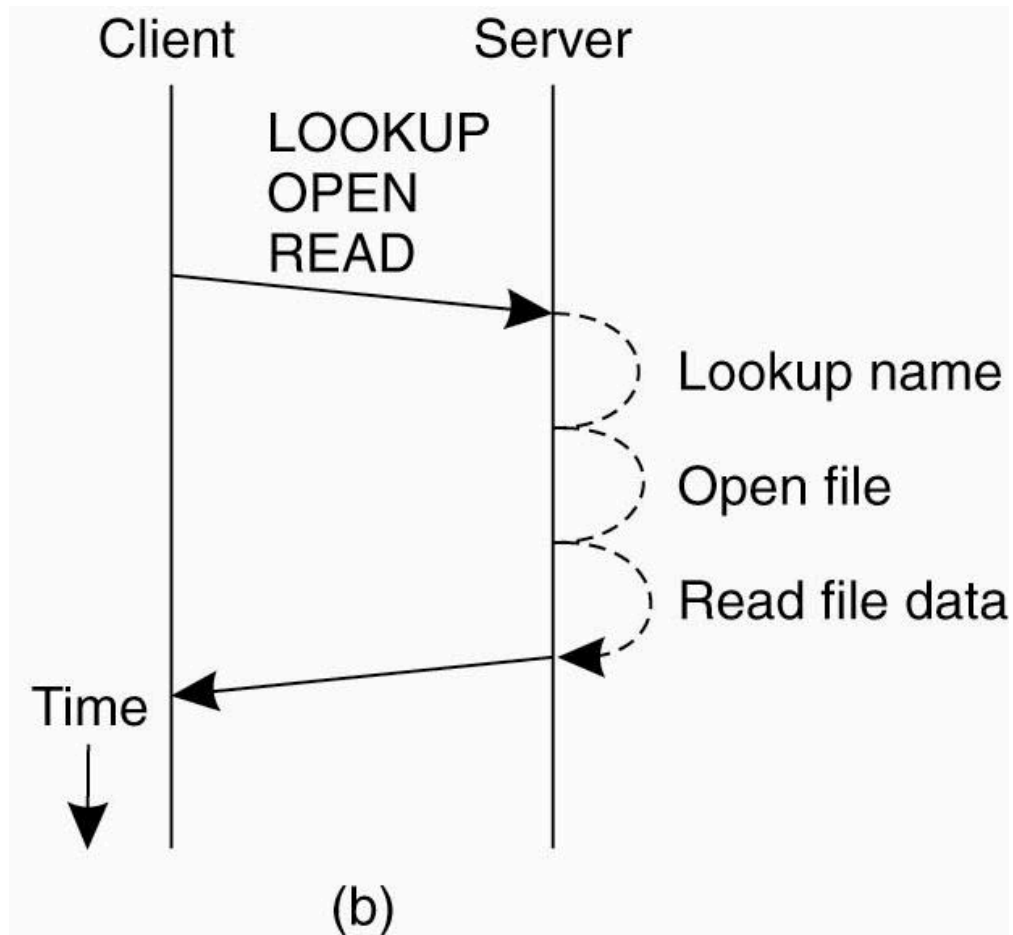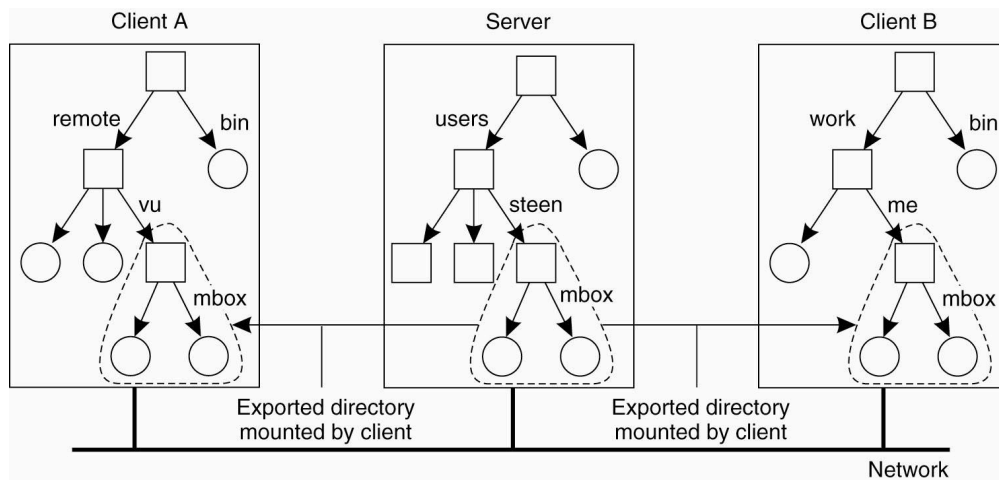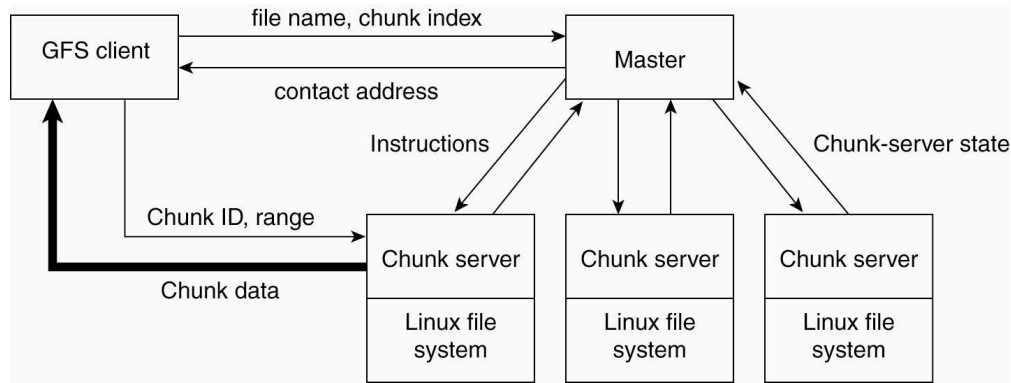## 5.3   Google File System (GFS)

Google File System

- **GFS**: A distributed file system that stripes files across inexpensive commodity servers without RAID.

  - *Layered Above Linux File System*
  - *Fault Tolerance Through Software*

- **GFS Master**: *Stores Metadata About Files/Chunks*

  - *Metadata Cache in Main Memory*
  - *Updated Log in Local Storage*
  - *Periodically Polls Client Servers for Consistency*

5.4   Reading a File

1. A client sends the file name and chunk index to the master.
2. The master responds with a contact address.
3. The client then pulls data directly from a chunk server, bypassing the master.

5.5   Updating a File

1. The client pushes its updates to the nearest chunk server holding the data.
2. The nearest chunk server pushes the update to the next closest chunk server holding the data, and so on.
3. When all replicas have received the data, the primary chunk server assigns a sequence number to the update operation and passes it on to the secondary chunk servers.
4. The primary replica informs the client that the update is complete.

5.6 File Sharing Semantics

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

File Sharing Semantics

6 Apache Hadoop MapReduce

6.1 High-Level Architecture
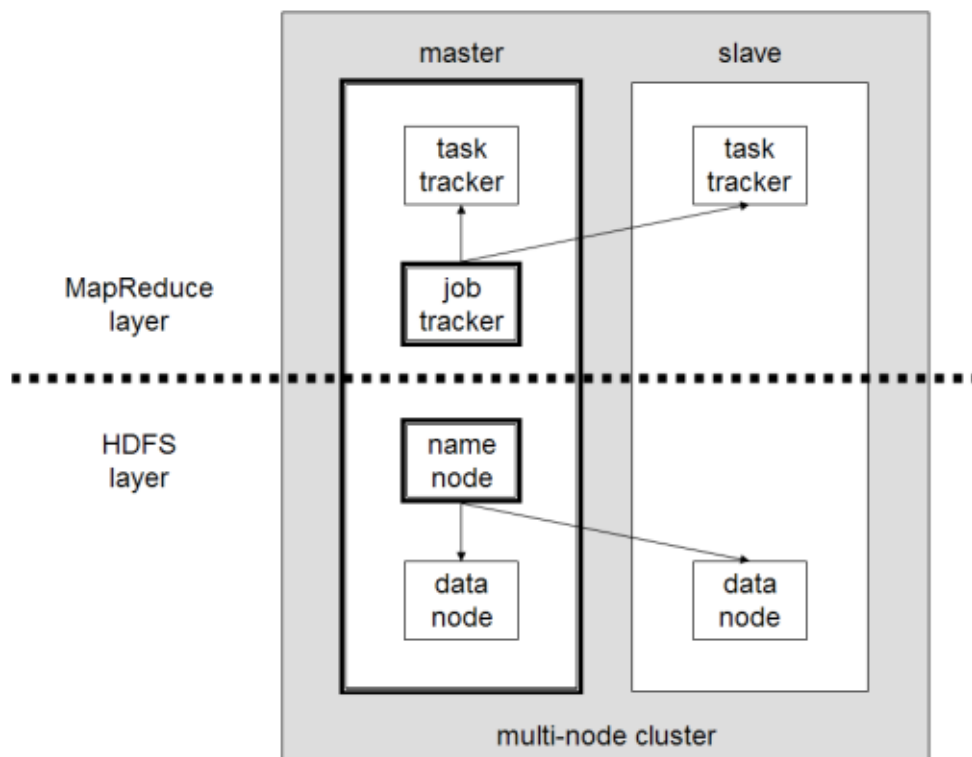
Hadoop High-Level Architecture

- Transform lists of input data elements into lists of output data elements by applying *Mappers* and *Reducers*
    - *Immutable Data*
    - *No Communication*

6.2 Mapper

- A list of input data elements are iterated and individually transformed into zero or more output data elements.

6.3 Reducer

- A list of input data elements are iterated and individually aggregated into a single output data element.

6.4 Combiner

- An optional component that consumes the outputs of a mapper to produce a summary as the inputs for a reducer.

6.5 Terms

- **InputSplit**: A unit of work assigned to one map task.
    - Usually corresponds to a chunk of an input file.
    - Each record in a file belongs to exactly one input split and the framework takes care of dealing with record boundaries.
- **InputFormat**: Determines how the input files are parsed, and defines the input splits.
- **OutputFormat**: Determines how the output files are formatted.
- **RecordReader**: Reads data from an input split and creates key-value pairs for the mapper.
- **RecordWriter**: Writes key-value pairs to output files.
- **Partitioner**: Determines which partition a given key-value pair will go to.

6.6   Data Flow
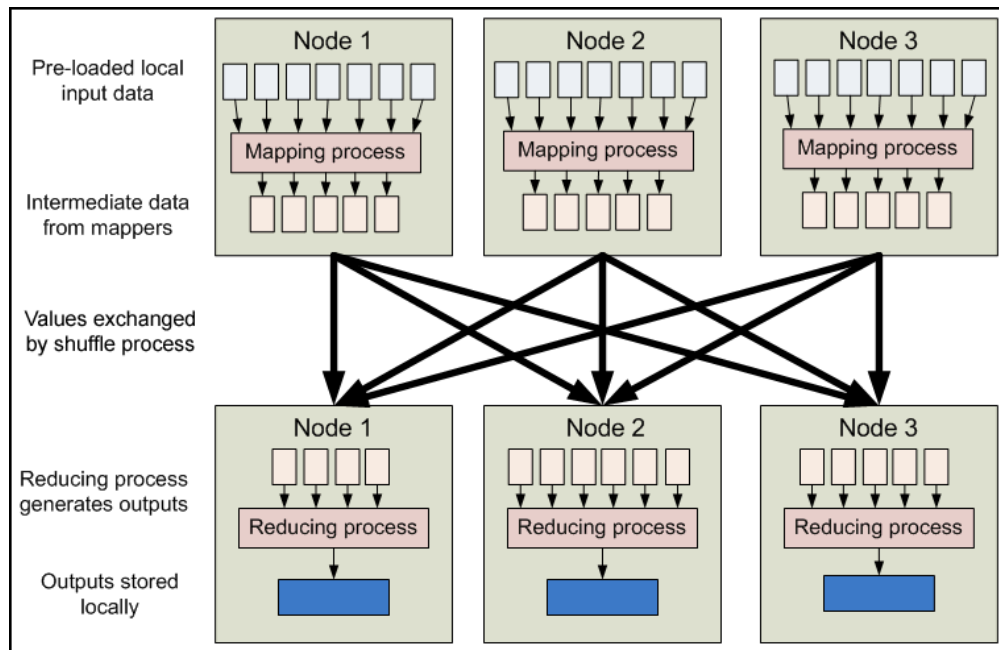
MapReduce Data Flow 1
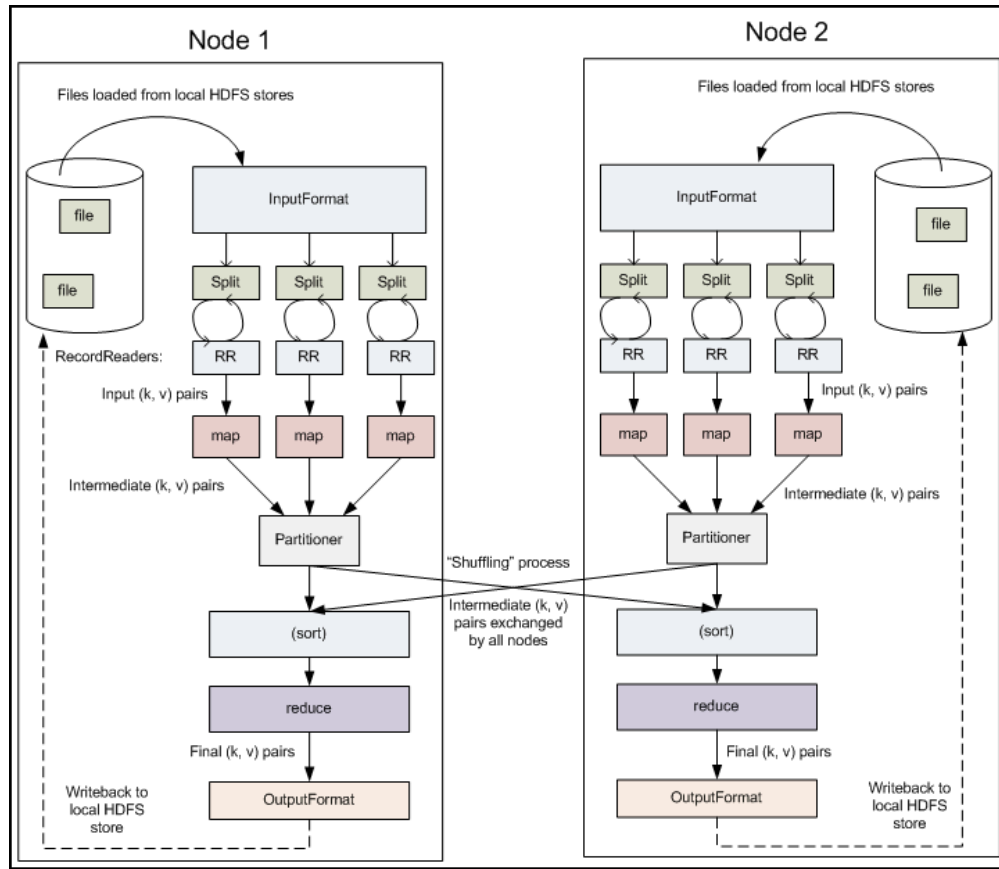
MapReduce Data Flow 2

- **Shuffle**: The process of partitioning by reducer, sorting and copying data partitions from mappers to reducers.

6.7   Fault Tolerance

- *Primarily: Restart Failed Tasks*

  1. Individual *TaskTrackers* periodically emit a heartbeat to the *JobTracker*.
  2. If a *TaskTracker* fails to emit a heartbeat to the *JobTracker*, the *JobTracker* assumes that the *TaskTracker* crashed.
  3. If the failed node was mapping, then other *TaskTrackers* will be asked to re-execute all the map tasks previously run by the failed *TaskTracker*.

  – *Must be Side-Effect Free*

  4. If the failed node was reducing, then other *TaskTrackers* will be asked to re-execute all reduce tasks that were in progress on the failed *TaskTracker*.

  – *Must be Side-Effect Free*

- *Secondarily: Speculative Execution*

  – If some **straggler** nodes rate limit the rest of the program, Hadoop will schedule redundant copies of remaining tasks across several nodes which do not have other work to perform.

6.8 MapReduce Design Patterns

6.9 Counts and Summations

- A mapper can emit a tuple of an element and one for each element.
- A mapper can aggregate the counts for each element and emit a tuple of the element and its count.
- A combiner can aggregate the counts across all the elements processed by a mapper.

6.10 Selection

- A mapper can emit a tuple for each element that satisfies a predicate.

6.11 Projection

- A mapper can emit a tuple whose fields are a subset of each element.
- A reducer can eliminate duplicates.

6.12 Inverted Index

- A mapper can emit a tuple of a value and a key in that specific order.
- A reducer can aggregate all the keys for a distinct value.

6.13 Cross-Correlation

- **Problem**: Given a set of tuples of items, for each possible pair of items, calculate the number of tuples where these items co-occur.

6.13.1 Pairs Approach (Slow)

```
class Mapper
  method Map(void, items [i1, i2, ...])
    for all item i in [i1, i2, ...]
      for all item j in [i1, i2, ...] such that j > i
        Emit(pair [i, j], count 1)

class Reducer
  method Reduce(pair [i, j], counts [c1, c2, ...])
    s = sum([c1, c2, ...])
    Emit(pair [i, j], count s)
```

6.13.2 Stripes Approach (Fast)

```
class Mapper
  method Map(void, items [i1, i2, ...])
    for all item i in [i1, i2, ...]
      H = new AssociativeArray : item -> counter
      for all item j in [i1, i2, ...] such that j > i
        H{j} = H{j} + 1
      Emit(item i, stripe H)
```

```
class Reducer
  method Reduce(item i, stripes [H1, H2, ...])
    H = new AssociativeArray : item -> counter
    H = merge-sum([H1, H2, ...])
    for all item j in H.keys()
      Emit(pair [i, j], H{j})
```

7 Apache Spark

7.1 RDD

- **RDDs** are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

7.2 Lineage
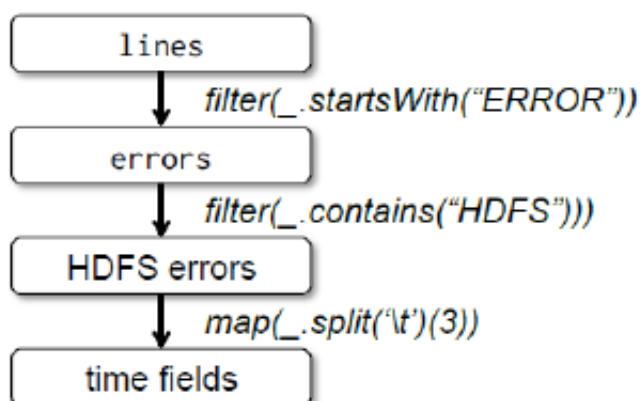
Lineage

- A **lineage** is a directed acyclic graph that expresses the dependencies between RDDs such that a RDD can be rebuilt in the event of a failure.

7.3 Transformation and Actions

- **Transformations**: Operations that convert one RDDs or a pair of RDDs into another RDD.
- **Actions**: Operations that convert a RDD into an output.

7.4 Narrow vs. Wide Dependencies
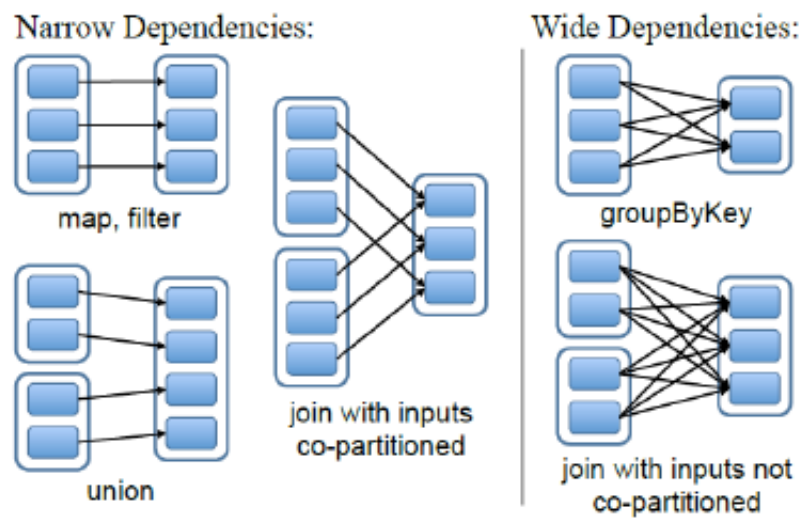
Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

Dependencies
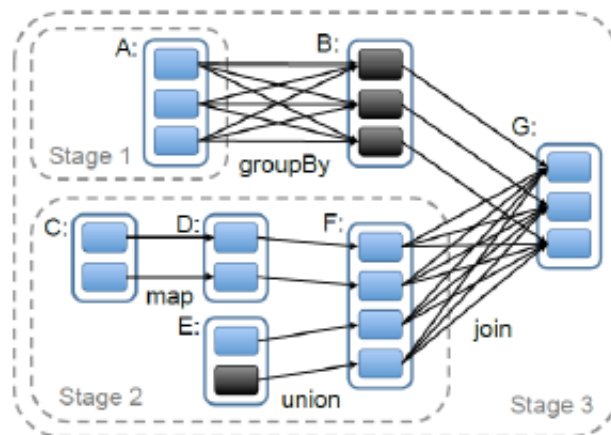
Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Execution

- When an action is invoked on a RDD, the Spark scheduler examines the lineage graph of the RDD and builds a directed acyclic graph of transformations.

23

- The transformations in the DAG are grouped into **stages**.
- A **stage** is a collection of transformations with **narrow dependencies**, meaning that one partition of the output depends on only one partition of each input.
- The boundaries between stages correspond to **wide dependencies**, meaning that one partition of the output depends on multiple partitions of some input, requiring a shuffle.

8   Distributed Graph Processing

8.1   Pregel

- In Pregel, graph processing problems are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology.
- This **vertex-centric approach** is flexible enough to express a broad set of algorithms.

8.2   Architecture

- **Unit of Work**: A *partition* consisting of a set of vertices and their outgoing edges.
- **Coordination**: A single *master* with many *workers*.

1. The master determines the number of partitions and distributes the partitions to each worker.
2. The master assigns a portion of the user's input to each worker.

    1. If the worker is assigned a vertex that belongs to its partition of the graph, then the worker updates the the state of the vertex.
        - *Vertex's Current Value*
        - *Vertex's Outgoing Edges*
        - *Vertex's Activity Flag*
    2. If the worker is assigned a vertex that does not belong to its partition of the graph, then the worker sends a message containing the vertex and its edges to the appropriate remote peer.
    3. All the input vertices are marked as active.

3. The master instructs each worker to perform a **superstep**.

    1. The worker loops to compute through its active vertices.
        1. Asynchronously execute a user-defined function on each vertex.
        2. Receive messages sent in the previous superstep.
        3. Send messages to be received in the next superstep.
        4. Vertices can modify their value.
        5. Vertices can modify the values of their edges.
        6. Vertices can add or remove edges.
        7. Vertices can deactivate themselves.

    2. The worker notifies the master how many vertices will be active in the next superstep.

4. Repeat *Step 3* until all the vertices are inactive.

- **Important Note**: An active vertex is reactivated when it receives a message.

- **Bulk Synchronous Parallel**: Workers compute asynchronously within each superstep, and communicate only between supersteps.

8.3 Combiners and Aggregators

- **Combiners**: An optional component which reduces the amount of data exchanged over the network and the number of messages.

  - i.e., A *commutative* and *associative* user-defined function.

- **Aggregators**: An optional component which computes aggregate statistics from vertex-reported values.

  1. Workers aggregate values from their vertices during each supersep.
  2. At the end of each superstep, the values from the workers are aggregated in a tree structure, and the value from the root of the tree is sent to the master.
  3. The master shares the value with all vertices in the next superstep.

- An aggregator is useful for detecting convergence conditions for vertices to transition to the inactive state.

8.4 Fault Tolerance

- **Key Idea**: *Checkpointing*
- At the beginning of a superstep, the master instructs the workers to save the state of their partitions to persistent storage.

  - Vertex Values.
  - Edge Values.
  - Incoming Messages.

- The master separately saves the aggregator values.
- Worker failures are detected using regular pings from the master to the workers.
- When one or more workers fail, the master reassigns graph partitions to the currently available set of workers, and they all reload their partition state from the most recent available checkpoint.

9 Consistency and Replication

9.1 Motivations for Replication

- *Increased Reliability*
- *Increased Throughput*
- *Decreased Latency*

9.2 Replicated Data Store

- In a **replicated data store**, each data object is replicated at multiple hosts.

  - **Local Replica**: *Same Hosts*
  - **Remote Replica**: *Different Hosts*

9.3 Consistency Models

1. *Sequential Consistency*
2. *Causal Consistency*
3. *Linearizability*
4. *Eventual Consistency*
5. *Session Guarantees*

9.4 Sequential Consistency

- A data store is sequentially consistent when the result of any execution is the same as if the operations by all processes on the data were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

```
P1: W(x)a
P2:          W(x)b
P3:                    R(x)b          R(x)a
P4:                          R(x)b  R(x)a
```

Positive Sequential Consistency Example

```
P1: W(x)a
P2:          W(x)b
P3:                    R(x)b          R(x)a
P4:                          R(x)a  R(x)b
```

Negative Sequential Consistency Example

1. `R(x)a` and `R(x)b` conflict.

9.5 Causal Consistency

- A data store is causally consistent when writes related by the "causally precedes" relation must be seen by all processes in the same oder.
- Concurrent writes may be seen in a different order on different machines.
- **"Causally precedes"** is the transitive closure of two rules.

    1. Operation A causally precedes operation B if A occurs before B in the same process.
    2. Operation A causally precedes operation B if B reads a value written by A.

- If operations A and B are concurrent (no "causally precedes"), then A and B can be read in either order.

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

Positive Causal Consistency Example

1. W(x)a causally precedes R(x)a.
2. R(x)a causally precedes W(x)b.
3. W(x)b and W(x)c are concurrent.
4. Therefore, the reads must occur in the following sequences:

   1. A, B, C, or
   2. A, C, B.

P1:	W(x)a			
P2:		R(x)a	W(x)b	
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

Negative Causal Consistency Example

1. W(x)a causally precedes R(x)a.
2. R(x)a causally precedes W(x)b.
3. However, P3's R(x)b before R(x)a violates causal consistency.

9.6 Linearizability

- A data store is linearizable when the result of any execution is the same as if the operations by all processes on the data store were executed in some sequential order that extends the "happens before" relation.
- If operation A finishes before operation B begins, then A must precede B in the sequential order.

9.7 Eventual Consistency

- If no updates take palce for a long time, all replicas will gradually become consistent.
- Allows different processes to observe write operations taking effect in different orders, even when these write operations are related by "causally precedes" or "happens before".

9.8 Session Guarantees

- **Session Guarantees**: Restrict the behavior of operations applied by a single process in a single session.
- **Monotonic Reads**: If a process reads x, any successive reads on x by that process will always return the same value or a more recent value.

- **Monotonic Writes**: A write by a process on x is completed before any successive write on x by the same process.
- **Read Your Own Writes**: The effect of a write operation by a process on x will always be seen by a successive read on x by the same process.
- **Writes Follow Reads**: A write by a process on x following a previous read on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.

9.9 Primary-Based Replication Protocols

- In a primary-based protocol, each x in the data store has an associated primary, which is responsible for coordinating writes on x.
- If the primary replica fails, then one of the backup replicas may take over as the new primary.

  - *Disadvantage*: If the network is partitioned, the cluster can become **split-brain** such that one replica in each partition believes its the primary replica; hence, a divergence of state.
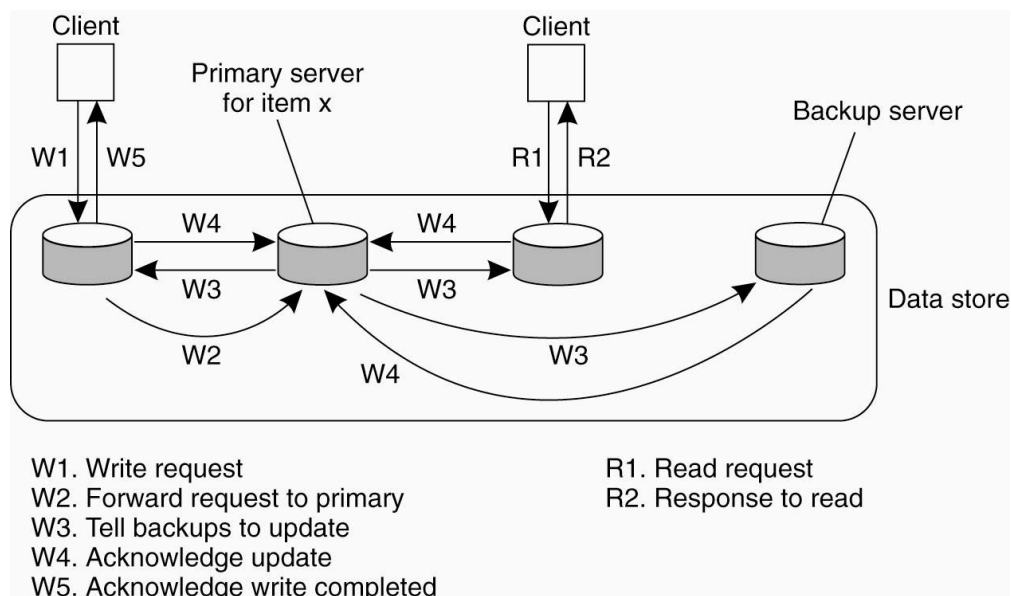
9.10 Advantages and Disadvantages

9.10.1 Advantages

- *Strong Consistency*

9.10.2 Disadvantages

- *Performance Bottlenecks*
- *Loss of Availability*

9.11 Remote-Write Protocol

W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Remote-Write Protocol Example

- **Remote-Write**: The primary replica is generally stationary and therefore must be updated remotely by other servers.

9.12 Local-Write Protocol
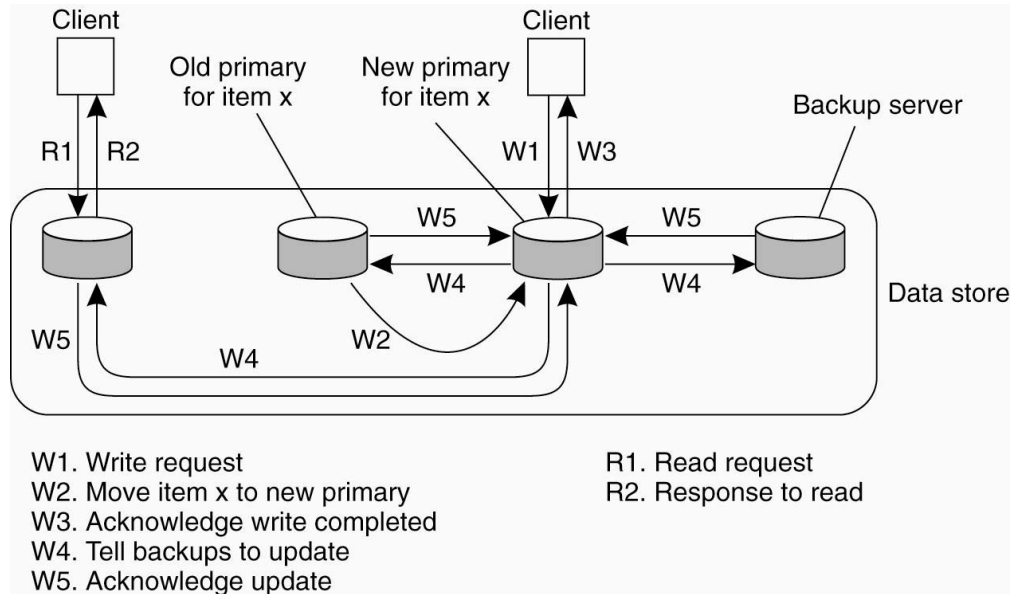
Local-Write Protocol Example

- **Local-Write**: The primary replica migrates from server to server, allowing local updates.

9.13 Quorum-Based Replication Protocols

- In a quorum-based protocol, all replicas are allowed to receive updates and reads, but operations are required to be accepted by a sufficiently large subset of replicas called a **write quorum** or a **read quorum**.
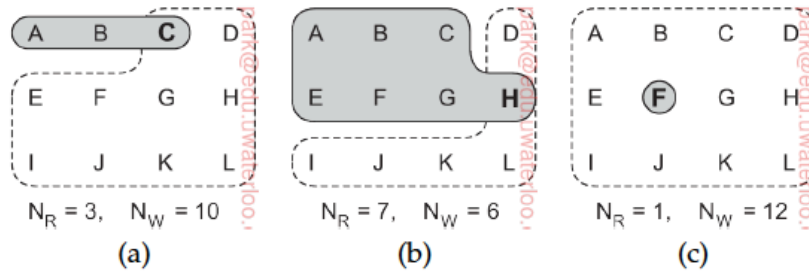
9.14 Requirements of Write and Read Quorums

Figure 7.29: Three examples of the voting algorithm. The gray areas denote a read quorum; the white ones a write quorum. Servers in the intersection are denoted in boldface. (a) A correct choice of read and write set. (b) A choice that may lead to write-write conflicts. (c) A correct choice, known as ROWA (read one, write all).

Write and Read Quorums Examples

- Let N be the total number of replicas.
- Let N_W be the size of the write quorum.
- Let N_R be the size of the read quorum.
- The following two rules must be satisfied:

    1. $N_R + N_E > N$; Read and write quorums must overlap.
    2. $N_W + N_W > N$; Two write quorums must overlap.

- The first rule enables detection of **read-write conflicts**.

    - Read-write conflicts occur when one process wants to update data item while another is concurrently attempting to read that item.

- The second rule enables detection of **write-write conflicts**.

    - Write-write conflicts occur when two processes want to perform an update on the same data.

9.15 Partial Quorums

- Derivatives of Anamzon's Dynamo, allow various degress of consistency by tuning N_R and N_W.

    - **Strong Consistency**: $N_R + N_W > N$
    - **Weak Consistency**: $N_R + N_W \leq N$

- **Important Note 1**: The strong consistency mode does not avoid write-write conflicts.
- **Important Note 2**: The weak consistency mode does not avoid read-write conflicts or write-write conflicts.
- To resolve write-write conflicts, updates are tagged with timestamps, and resolution policies such as **last-write wins** or **vector clocks** are applied.

9.16 Eventually-Consistent Replication

- A server that receives an update replies with an acknowledgement to the client first, and then propagates the update lazily to the remaining replicas.
- If a replica is unreachable, then it can be updated later using an **anti-entropy** mechanism.

    - e.g., Replicas may periodically exchange hashes of data to detect discrepancies using Merkle trees.
    - e.g., Updates can be timestamped to enable determination of the latest version of a data item.