ECE 454 Lecture Notes

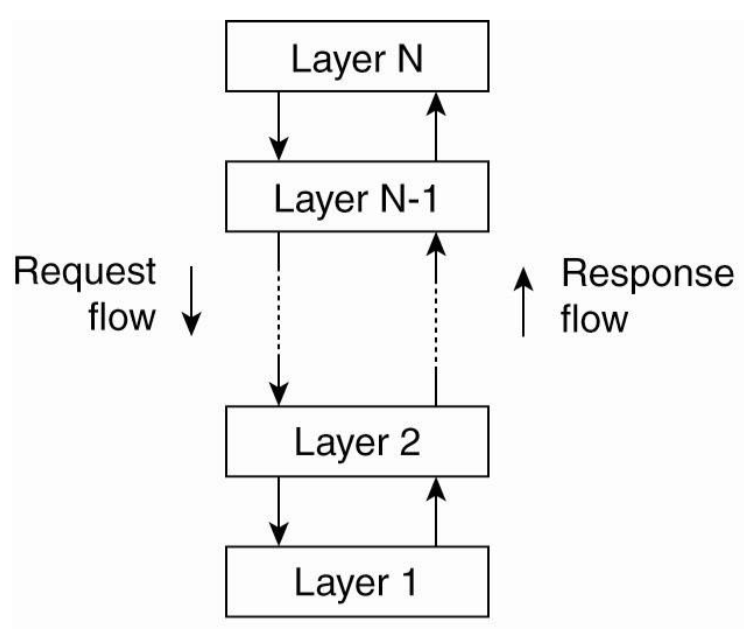
# 2 – Architectures

* **Component**: modular unit with well-defined interfaces
* **Connector**: mechanism that mediates communication/coordination between components
* **Software architecture**: organization of software components
* **System architecture**: instantiation of software architecture, where software components are placed on real machines
* **Autonomic system**: adapts to environment by monitoring its behaviours and reacting accordingly

## Architecture Styles

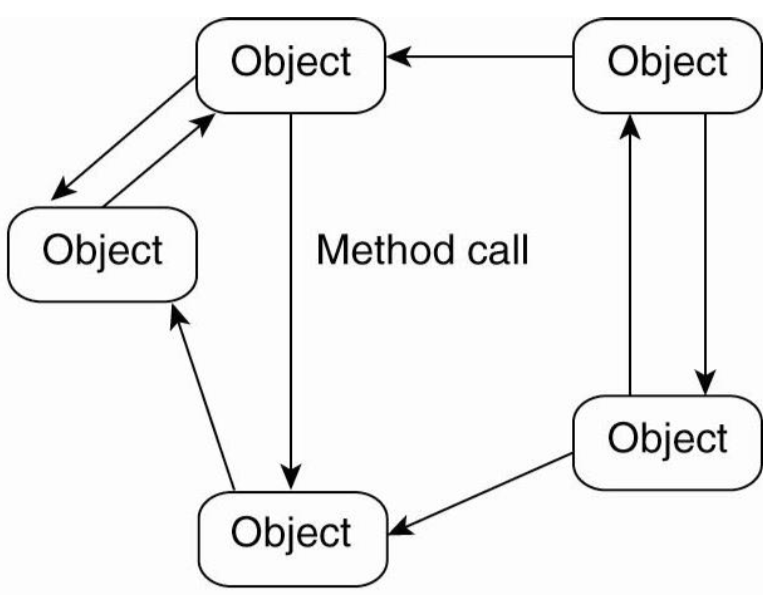
* Layered
* Object-based
* Data-centered
* Event-based

### Layered Architecture



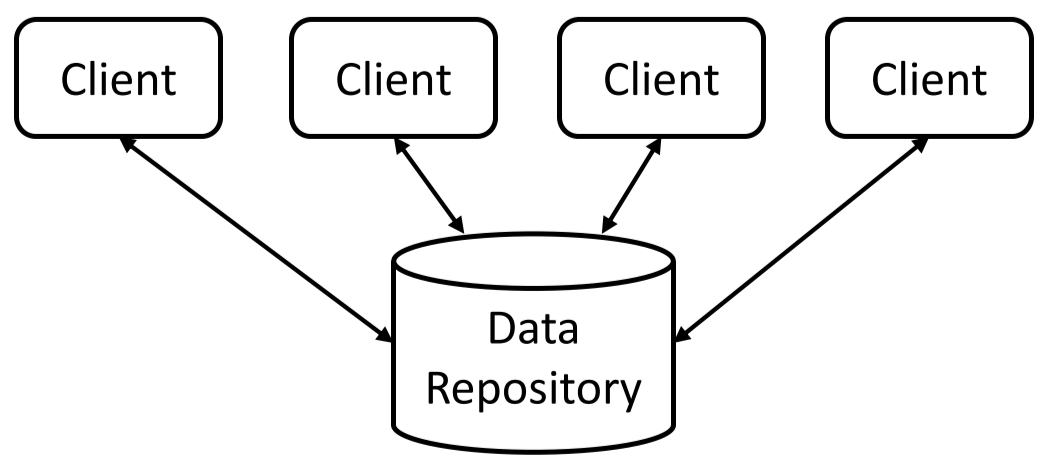
* Control flows from layer to layer
  + Requests flow down, responses flow up the hierarchy
* **Client-server interactions pattern**: client requests service from server, waits for response
* Many systems are organized into three layers:
  + User interface
  + Application server – acts as client and server to other layers
  + Database
* **Multi-tiered architectures**: logical software layers are mapped onto physical tiers
  + E.g. two-tiered architecture comprises of client machines and server machines, leading to different mappings
  + **Vertical distribution**: logical layers are organized as separate physical tiers
    - E.g. use separate machines for application server and database
  + **Horizontal distribution**: one logical layer is spread across multiple machines
    - E.g. Sharding: dataset is hash-partitioned across many database instances running on separate machines

### Object-Based Architecture



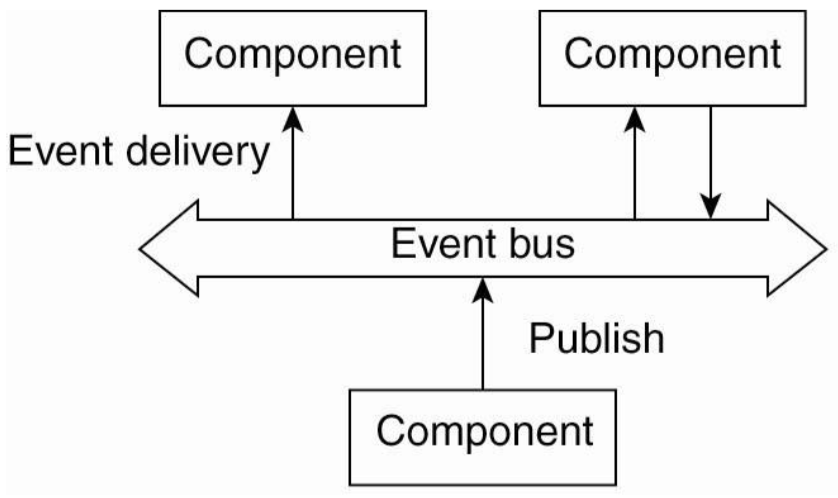
* Components are loosely organized into different objects
* APIs allow for remote object references and method calls

### Data-Centered Architecture



* Components communicate by accessing a shared data repository (e.g. database or storage system)

### Event-Based Architecture



* Components communicate by propagating events
* **Publish/subscribe system**: components subscribe to different events; middleware matches published items to notify subscribers

### Peer-to-Peer (P2P) Systems

* Processes organized in **overlay network** that defines set of communication channels
  + Rely on horizontal distribution
  + Handles churn of machines joining/leaving

### Hybrid Architectures

* BitTorrent combines client-server and P2P architectures
  + Client nodes obtain tracker information from a server, then exchanges data with peer nodes

### Self-Management

* Self-managing systems can be constructed using a **feedback control loop** thatmonitors system behaviours and adjusts system’s internal operations

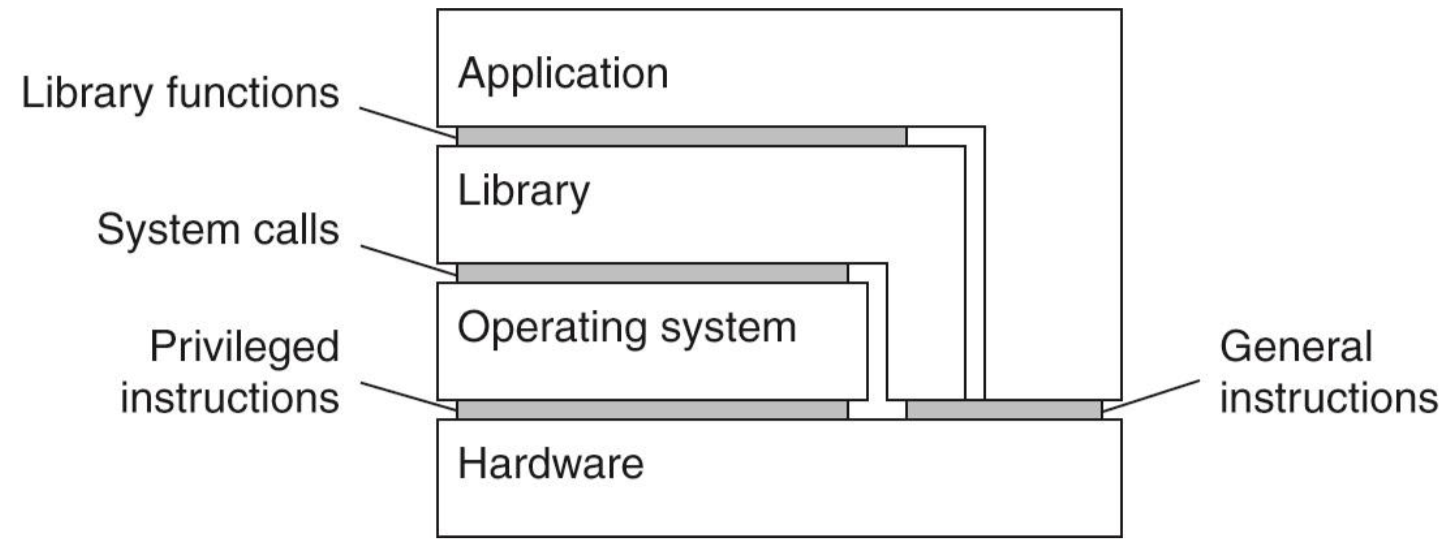
# 3 – Processes

## Inter-Process Communication – Context-Switching

* IPC (e.g. Unix/Linux pipes), is costly because it requires context switching from user space to kernel space, and back

## Threads

* Threads executing in same process can communicate through shared memory
  + **Lightweight processes (LWPs)** – share address space and file descriptors, OS kernel support for multi-threading
* **Dispatcher/worker**: dispatcher receives requests from network, feeds to pool of worker threads
  + Many multi-threaded servers follow this design
* Processes/threads interact with hardware either directly (processor instructions) or indirectly (library functions/OS calls)





## Virtualization

* Distributed systems often run in **virtualized environments** (run on top of host OS)or **virtual machine monitors** (run on top of hardware)
* Advantages:
  + Improves portability – separates applications from underlying hardware/OS
  + Server consolidation – reduces operating costs
  + Load balancing and proactive maintenance, through live migration of VMs
  + VM replication improves availability and fault tolerance

## Networked Systems Interfaces

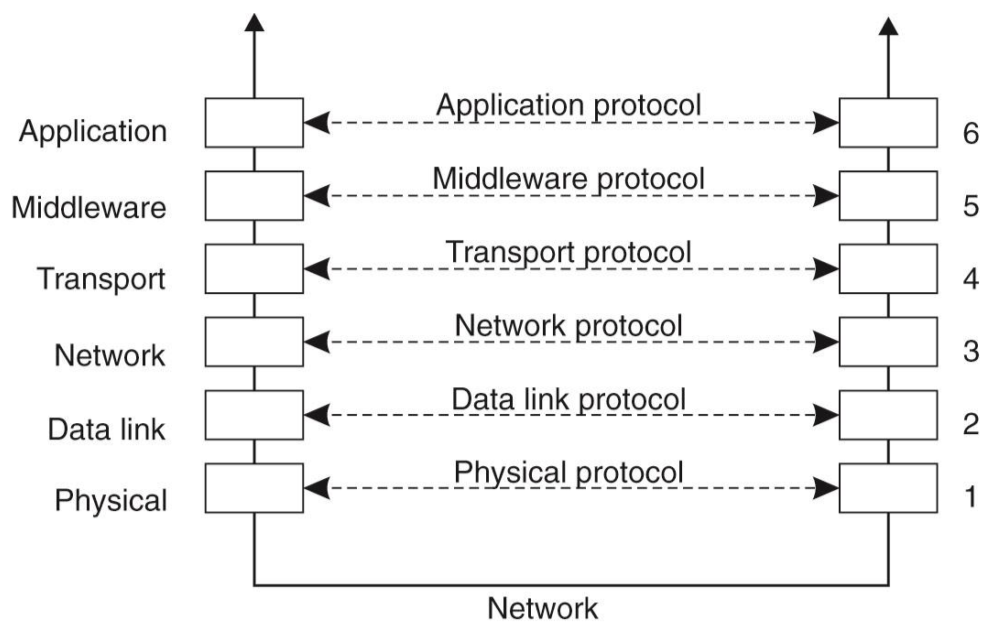
* Networked applications typically communicate by messages, often through middleware layer
* **Message passing protocol**: determines format of messages – application-specific or application-independent

## Server Clusters

* Often organized into three tiers:
  + **First tier**: logical switch (load balancer)
  + **Second tier**: application servers
  + **Third tier**: distributed database/filesystem

# 4 – Communication

## Layered Network Model



* **Middleware** layer isolates application layer from transport layer, to provide access transparency

## Remote Procedure Calls (RPCs)

* **Transient communication** abstraction – behaves similarly to procedure call that passes parameters on the stack
* Implemented using **client-server protocol**
  + Client stub: application calls RPC using this, to translate call into protocol message
  + Server stub: receives and processes protocol message, sends reply that is received by client stub
* **Interface Definition Language (IDL)**: defines signatures of RPCs, compiles into client stub and server stub
  + IDL compiler determines high-level format of messages for a given RPC

### RPC Steps

1. Client involves client stub with procedure call
2. Client stub builds message, sends it to client OS
3. Client OS sends message to server OS
4. Server OS delivers message to server stub
5. Server stub unpacks message, invokes service handler in server process
6. Service handler does work, returns result to server stub
7. Server stub builds response message, sends to server OS
8. Server OS sends message to client OS
9. Client OS delivers message to client stub
10. Client stub unpacks message, returns response to client process

* **Parameter marshalling**: packing parameter values into a message (step 2)
  + Must account for difference in representations of values on different hardware platforms (e.g. big endian, little endian, etc.)

### Synchronous vs. Asynchronous RPCs

* **Synchronous RPC**: client waits for return value while server executes procedure
* **Asynchronous RPC**: client resumes execution as soon as server acknowledges receipt of request; server issues another RPC to return result
  + **One-way RPC**: client doesn’t wait for acknowledgement from server

## Message Queue Model

* Persists sent messages until they are consumed by a receiver
* Enables **persistent communication** that is loosely coupled with time (i.e. can still send messages, even if a sender/receiver server is down)
* Operations:
  + Put: appends message to queue
  + Get: block until queue is non-empty, then remove first message
  + Poll: checks queue for messages, then removes first message
  + Notify: install handler to be notified when a message is put into the queue
* Disadvantage: message delivery depends on receiver, so cannot be guaranteed
* **Message-oriented middleware (MOM)**: asynchronous message-passing
  + E.g. message queueing, publish-subscribe

## Coupling Among Processes

* **Referential coupling**: one process explicitly references another
* **Temporal coupling**: communicating processes must both be up and running

## RPC vs. MOM

* RPCs usually used for two-way communication
  + Middleware linked to client/server processes
  + Tighter coupling – server failure can prevent progress
* MOMs usually used for one-way communication (one party doesn’t require immediate response from the other)
  + Separate middleware component between sender and receiver
  + Loose coupling – isolates one process from the other, so more flexible and scalable

# 5 – Apache Thrift

## Thrift Software Stack

* **Server** – receives incoming connections
* **Processor** – reads/writes I/O streams
* **Protocol** – encodes/decodes data
* **Transport** – reads/writes network

## Thrift IDL

### Syntax

namespace java example1

exception IllegalArgument {

1: string message;

}

service MathService {

i64 sqrt(1:i64 num) throws (1: IllegalArgument ia)

}

* **Base types**: bool, byte, i16, i32, i64, double, binary, string, void
* **Containers**: list<>, set<>, map<>
* **Other types**: const, typedef, enum, struct, exception
* **Field modifiers**: required, optional, default
* **Services/procedures**: service, extends, oneway

### Service Handler Example

package example1;

public class MathServiceHandler implements MathService.Iface {

public MathServiceHandler() { ... }

public long sqrt(long num) throws IllegalArgument { ... }

}

## Thrift Protocols

* **TBinaryProtocol**: encodes numeric values in binary format
* **TCompactProtocol**: variable-length encoding for integers
* **TJSONProtocol**: human-readable JSON format
* Thrift fields can be modified between old and new protocol versions:
  + Never change numeric tags of existing fields
  + New fields can be added as long as they are marked optional and have default values
  + Fields that are no longer needed can be removed, as long as their tags aren’t reused
  + Default values can be changed

## Thrift Clients

### Synchronous RPC Client

public static void main(String[] args) {

try {

TSocket sock = new TSocket(...);

TTransport transport = new TFramedTransport(...);

TProtocol protocol = new TBinaryProtocol(transport);

MathService.Client client = new MathService.Client(protocol);

transport.open();

client.sqrt(...);

transport.close();

}

catch (TException e) {

}

in mainthread:

1. Write socket/send request
2. Read socket/receive response

}

### Asynchronous RPC Client

* **Non-blocking** socket and transport
* Use **TAsyncClientManager** to handle callbacks
* Encapsulate callback method in object that implements **AsyncMethodCallback** interface
* **Synchronization** may be required, through use of Countdown or CountdownLatch
* Note: asynchronous clients can only execute one RPC at a time

public static void main(String[] args) {

create asynchronous client A

create asynchronous client B

send RPC request A

send RPC request B

in **TAsyncClientManager** internal thread:

1. Register interest in writing
2. When socket ready, write socket/send request
3. Register interest in reading
4. When socket ready, read socket/receive response
5. Excecute **onComplete** or **onError** callback

}

## Thrift Servers

### Single-Threaded Server

public static void main(String[] args) {

MathService.Processor processor =

new MathService.Processor(new MathServiceHandler());

TServerSocket socket = new TServerSocket(...);

TSimpleServer.Args sargs = new TSimpleServer.Args(...);

sargs.protocolFactory(...);

sargs.transportFactory(...);

sargs.processorFactory(...);

TServer server = new TSimpleServer(sargs);

server.serve();

**// TSimpleServer: Blocking I/O**

while (not stopped)

accept new connection

while (connection open)

1. Read socket/receive request
2. Execute service handler
3. Write socket/send response

**// TNonblockingServer: Non-blocking I/O**

register interest in accepting connections

for each connection:

while (connection open)

1. Register interest in reading
2. When socket ready, read socket/receive request
3. Execute service handler
4. Register interest in writing
5. When socket ready, write socket/send response

}

### Multi-Threaded Server

* Performance advantages, but require more careful programming
* Servers except TThreadPoolServer, typically uses **non-blocking socket** and **framed transport**
* Other configuration options are specified to control number of threads, etc.

### Java Server Types

* **TSimpleServer**: single thread, blocking I/O
* **TNonblockingServer**: single thread, non-blocking I/O – handles parallel connections, but executes requests serially
* **THsHaServer**: one network I/O thread, pool of worker threads – processes multiple requests in parallel
* **TThreadedSelectorServer**: pool of network I/O threads, pool of worker threads to process requests
* **TThreadPoolServer**: one thread accepts connections, each connection handled by thread from pool of worker threads
  + Often yields best performance, but leads to too many threads created when high number of parallel connections

## Distribution Transparency

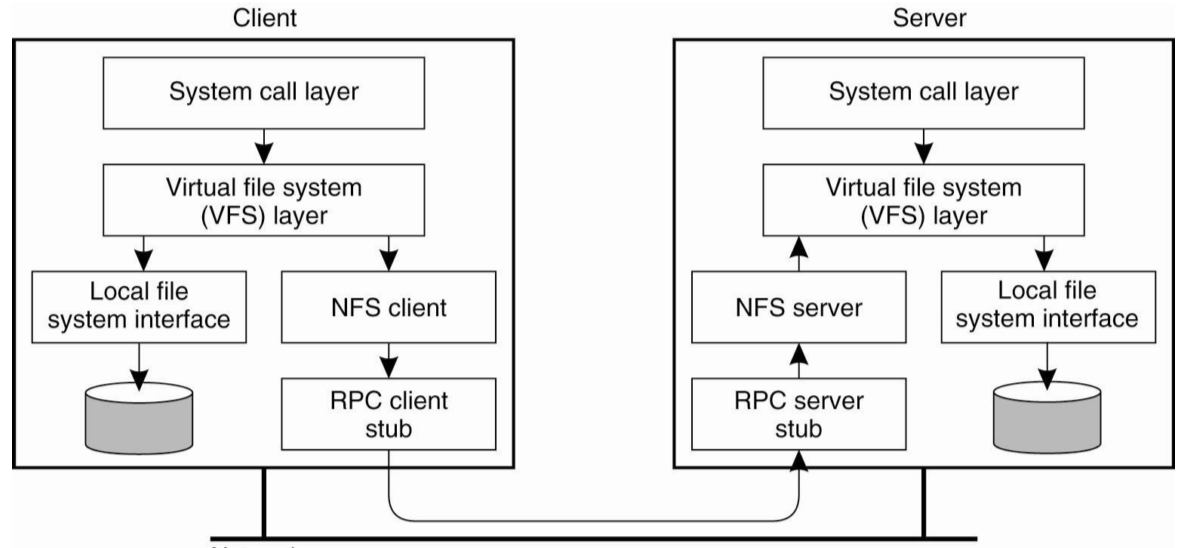
* **Location transparency**: use of names to identify network resources, rather than their location
  + Thrift violates this because client must know host name + port number of services
* **Access transparency**: no apparent difference between local and remote access methods
  + Thrift violates this because Thrift objects may throw a variety of IPC exceptions
    - TTransportException: TCP connection cannot be opened
    - TProtocolException: client and server have mismatched Thrift protocol versions

# 6 – Distributed File Systems

## Accessing Remote Files

* **Remote access model**: file stays on server, client requests to access remote file
* **Upload/download model**:
  + Server moves file to client
  + Access is done on client
  + After client is done, file is returned to server

## Network File System (NFS)



* **Client-side caching** reduces communication between client and server
  + Modifications flushed to server when client closes a file
  + Consistency is handled in implementation-dependent way



* NFSv4 allows servers to **delegate authority** of file to clients, then recall delegation through callback
  + Client requests file
  + Server delegates file – sends copy to client
  + Server recalls delegation
  + Client sends return file – server updates file
* NFS uses internal RPCs
  + NFSv3 executes file operations in multiple round trips (stateless)
  + NFSv4 supports **compound procedures** (batching of operations)
* Each NFS server typically only exports part of its local file system to remote clients
  + Clients can access **nested directories** exported by multiples servers
    - Server A imports directory X from server B, exports directory Y containing subdirectory X
    - A client importing Y from server A needs to explicitly import X from server B

## Distributing Large Files

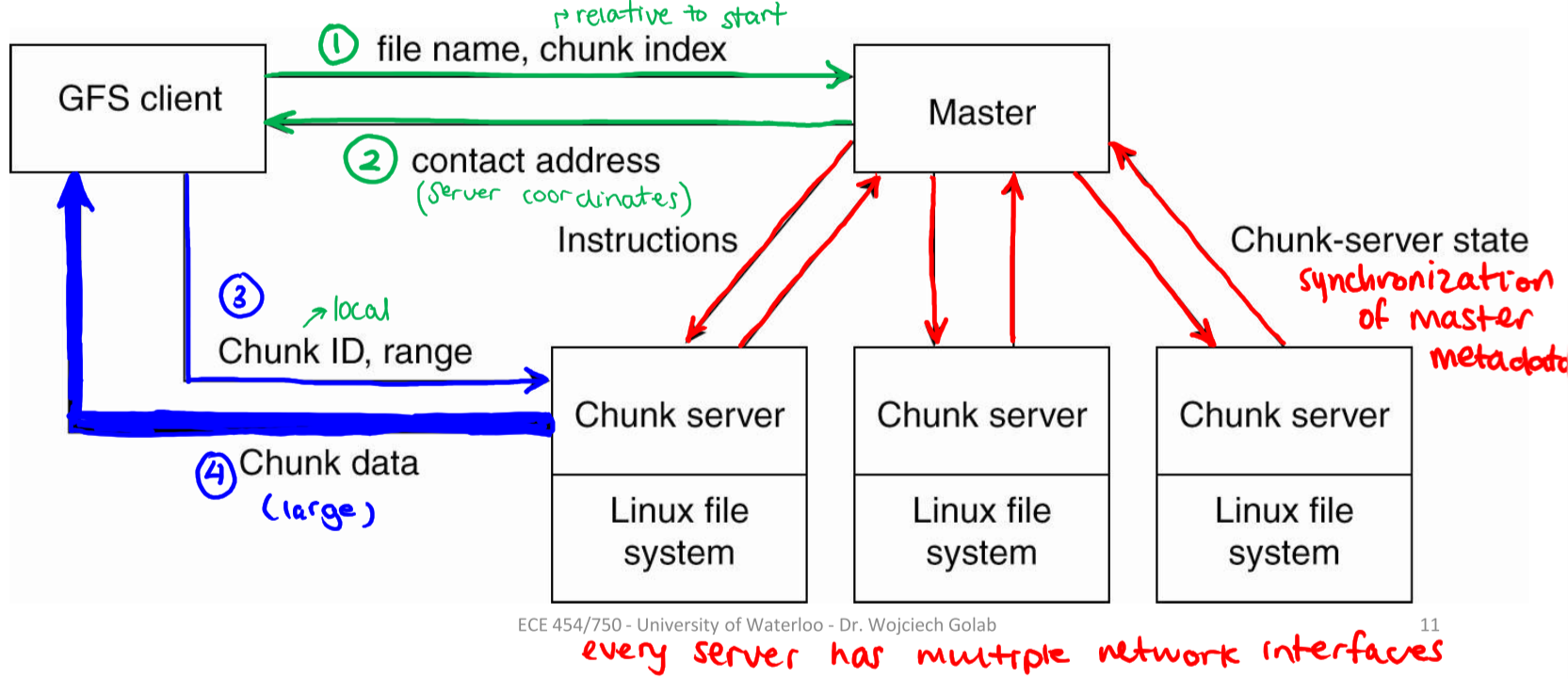
* In NFS, one NFS server shares files with many clients
* Large-scale DFS distribute files across multiple servers
  + Each file resides on one server
  + Files are **striped** – blocks of each file distributed across servers

## Google File System (GFS)

* DFS that stripes files across inexpensive commodity servers without RAID
* Layered on top of Linux file systems to provide fault tolerance
* Hadoop Distributed File System (HDFS) is a simplified open-source implementation of GFS
* **GFS master** stores metadata about files and chunks, and serves it to clients
  + Metadata cached in main memory
  + Updates logged to local storage – checkpoints reduce log length
* GFS master periodically **polls** **chunk servers** to keep metadata consistent
  + Replication allows clients to typically be able to locate at least one copy of a given chunk

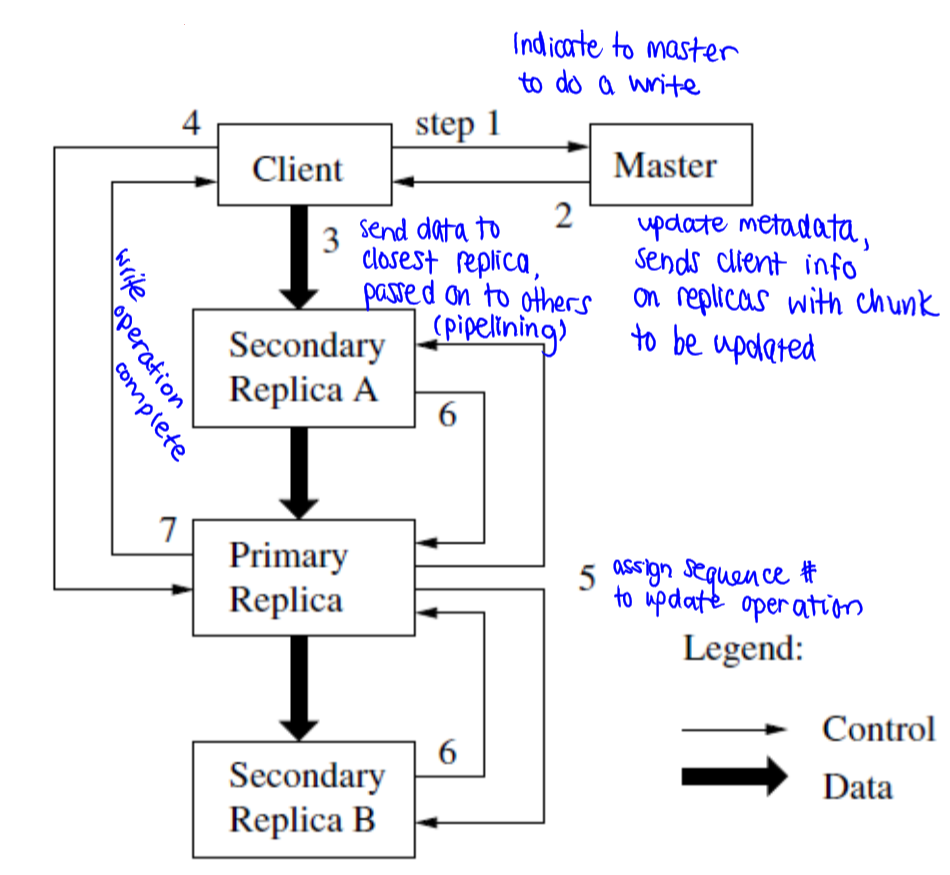
### GFS Read

1. Client sends file name and chunk index to master
2. Master replies with contact address
3. Client pulls data directly from chunk server



### GFS Update

1. Client sends write request to master
2. Master updates metadata and replies with info of replicas with chunk
3. Client contacts nearest chunk server holding data, pushes update to server; chunk server pushes update to next closest chunk server with data, and so on, in pipelined fashion
4. When all replicas have received update, client contacts primary chunk server
5. Primary chunk server assigns sequence number to update operation, passes it to secondary chunk servers
6. Primary replica replies to client with write operation complete



## File Sharing Semantics

### UNIX Semantics

* File system operations strictly ordered in time – ensures application will always read own writes
  + Strong correctness
  + Low performance – no caching

### Session Semantics

* Modifications to a file are only visible to other processes when the file is closed
  + Before then, modifications only visible to process modifying the file
  + Final version determined by last client that closes file
* NFSv4 uses session semantics + **byte range file locking** to avoid consistency anomalies

### Immutable Files

* Files cannot be updated, only deleted and replaced
* HDFS uses immutable files + supports append function
* The more servers, the more common failures are
  + Need master to coordinate sharing copies of files across multiple servers

# 7 – Hadoop MapReduce

* Performs parallel computations on large volumes of data
  + Components aren’t allowed to share data arbitrarily
  + Data elements are immutable – functions generate new outputs to forward
  + List of input elements are transformed into list of output elements
* **Mapper**: transforms each element, individually, to one or more output elements
* **Reducer**: aggregates values together into one output value
* Data flow:
  + Mapper
    - Map
    - Combine values by key (optional – same lambda function as Reduce)
  + Shuffle
  + Reducer
    - Sort values by key
    - Reduce

### Technical Terms

* **InputSplit**: unit of work assigned to one map task – usually a chunk of an input file
* **InputFormat**: determines how input files are parsed, and input splits
* **RecordReader**: loads data from input split, creates key-value pairs for mapper
* **Partitioner**: determines which partition a given key-value pair will go to
* **OutputFormat**: determines how output files are formatted
* **RecordWriter**: records key-value pairs to output files

### Fault Tolerance

* Hadoop achieves fault tolerance by restarting tasks
  + If a TaskTracker fails to communicate with a JobTracker for some time, it will assume the task failed
  + **Mapping phase**: re-execute all map tasks run by failed TaskTracker
  + **Reducing phase**: re-execute all reduce tasks in-progress by failed TaskTracker
* Mappers and reducers must be side-effect free – tasks run in isolation
  + A few slow nodes may rate-limit the rest of the program
  + Same input may be processed multiple times in parallel, since individuals do not know where their inputs came from
    - **Speculative execution**: Hadoop will schedule redundant copies of tasks across nodes that don’t have work

## MapReduce Programming Patterns

### Counting and Summing

* Classic solution:
  + **Mapper**: output (key, 1) for each key
  + **Reducer**: sum tuples for each key
* Alternative solution:
  + **Mapper**: aggregate count for each key in the document
  + Requires additional memory, but potentially faster
* Alternative solution:
  + **Combiner**: aggregate count of each key across all documents processed by map task

### Selection

* Returns subset of input elements that satisfy a predicate
* **Mapper-only**: for each (key, t), output (t, null) for each value that satisfies the predicate

class Mapper

Map(key k, tuple t)

if (predicate satisfied)

emit (tuple t, null)

### Projection

* Returns subset of fields of each input element
* **Mapper**: for each (key, t), extract required fields to tuple g and output (g, null)
* **Reducer**: eliminate duplicates; output (g, null) (one output per value g)

class Mapper

Map(key k, tuple t)

tuple g = project(t)

emit (g, null)

class Reducer

Reduce(tuple g, array n) // n is array of nulls

emit (g, null)

### Inverted Index

* Given a set of text documents with distinct identifies, produce a mapping from term to document ID
* **Mapper**: for each term t in the document, output (t, id)
* **Reducer**: flatten list of document IDs, output (t, docids)

class Mapper

Map(docid id, doc d)

for all term t in d:

emit (t, id)

class Reducer

Reduce(term t, array n) // n is array of array of docids

emit (t, flattened n)

### Cross-Correlation

* For each possible pair of items, calculate number of tuples where items co-occur
* **Pairs approach**: simplest but slowest, since keys are longer, so less likely to combine

class Mapper

Map(null, array items)

for all item i in items:

for all item j in items where j > i:

emit ([i, j], 1)

class Reducer

Reduce(pair [i, j], array counts)

emit ([i, j], sum(counts))

* **Stripes approach**: shorter keys, so more likely to combine

class Mapper

Map(null, array items)

for all item i in items:

H = new HashMap<item, counter>

for all item j in items where j > i:

H{j} = H{j} + 1

Emit (i, H)

class Reducer

Reduce(item i, array stripes)

H = new HashMap<item, counter>

H = merge-sum(stripes)

for all item j in H.keys():

emit ([i, j], H{j})

# 8a – Apache Spark

## Resilient Distributed Datasets

* **Resilient distributed dataset (RDD)**: immutable, partitioned, fault-tolerant, parallel data structures that let users persist intermediate results in memory, control partitioning to optimize data placement, and manipulate them using a rich set of operators
  + RDDs are declarative – framework decides how to optimize

### RDD Example

val lines = spark.textFile(...)

val errors = lines.filter(\_.startsWith(“ERROR”))

errors.persist() // persist near top of memory

val out = errors.count()

out.saveAsTextFile(...)

errors.filter(\_.contains(“MySQL”)).count()

errors.filter(\_.contains(“HDFS”))

.map(\_.split(‘\t’)(3)) // third field (e.g. timestamp, in example)

.collect() // pull output to program variable

### RDD Transformations and Actions

* **Transformations**: convert one RDD or a pair of RDDs to another RDD
* **Actions**: convert an RDD to an output
  + Spark scheduler examines lineage graph, builds directed acyclic graph (DAG) of transformations
  + **Stages – narrow dependencies**: collection of transformations where each output partition only depends on one input partition
    - E.g. map, filter, union, join with inputs co-partitioned by key
  + **Boundaries – wide dependencies**: each output partition depends on multiple input partitions – requires shuffle
    - E.g. groupByKey, reduceByKey, join with inputs not co-partitioned by key

### Word Count Example

val textFile = spark.textFile(“hdfs://inputfilepath”)

val counts = textFile.flatMap(line => line.split(“ ”))

.map(word => (word, 1))

.reduceByKey(\_ + \_) // (x1, x2) => x1 + x2

counts.saveAsTextFile(“hdfs://outputfilepath”)

* **Tuples**: 1-based indexing, use underscore notation

val t = (1, 2, 3)

val sum = t.\_1 + t.\_2 + t.\_3

* **Arrays**: 0-based indexing, use round brackets

val a = Array(1, 2, 3)

val sum = a(0) + a(1) + a(2)

### PageRank Example

val links = spark.textFile(...).map(...).persist()

var ranks = // RDD of (url, rank) pairs – initialize to 1/N

for (i <- 1 to ITERATIONS) {

// get contributions from each page

val contribs = links.join(ranks).flatMap {

(url, (dests, rank)) =>

dests.map(dest => (dest, rank/links.size))

}

// sum contributions, get new ranks

ranks = contribs.reduceByKey(\_ + \_)

.mapValues(sum => a/N + (1-a)\*sum) // normalized +

damping factor

}

## Spark vs. Hadoop MapReduce

* PageRank can be implemented in Hadoop, but with two weaknesses:
  + Intermediate output is dumped to HDFS after each iteration – unnecessary I/O
  + If converge is used to terminate loop, a second MapReduce job is required each iteration to evaluate convergence condition
* Spark caches intermediate data in main memory, if it is given enough resources

# 8b – Distributed Graph Processing

## Google Pregel

* **Master/worker** model, similar to Hadoop and Spark
* Each worker is responsible for a **vertex partition** – disjoint subset of directed graph’s vertices
* Model of computation is **vertex-centric**
* Framework maintains state for each vertex:
  + Problem-specific value
  + List of messages sent to vertex
  + List of outgoing edges
  + Active/inactive state (binary)

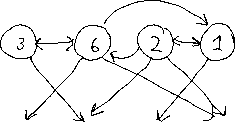
### Bulk Synchronous Parallel (BSP) model

* Computation organized into synchronous **superstep** iterations, driven by the master
* Workers compute asynchronously in each superstep, only communicate between supersteps
* In each superstep:
  + Works asynchronously execute user-defined function on each vertex
  + Vertices can receive messages sent in previous superstep
  + Vertices can send messages to be received in next superstep
  + Modify vertex values and/or edges, and/or add/remove edges
  + Vertex can deactivate itself
  + Inactive vertices reactivate when it receives a message
* Distributed execution stops when all vertices are marked inactive and no more messages

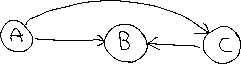
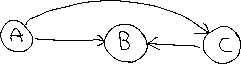
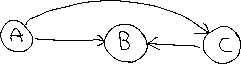
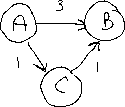
### Initialization

* Master assigns section of input to each worker
* Vertex ownership determined by partitioner – simple hash function on all vertices
  + Ensures fairly even distribution of vertices across workers, but doesn’t always balance computation
* Each worker reads section of input, stores vertices belonging to it, and forward remaining vertices to corresponding workers
* User can override default partitioning scheme

## Max Vertex Label Example



## Single-Source Shortest Paths (SSSP) Example



## Combiners and Aggregators

* **Combiners**: like Hadoop, helps reduce number of messages, trading CPU cycles for network I/O
  + Applicable if function applied at each vertex is commutative and associative
* **Aggregators**: compute aggregate statistics from vertex-reported values
  + Workers aggregate values from vertices during each superstep
  + At end of each superstep, values from each worker are aggregated in tree structure
  + Value from root of tree sent to master, who shares value with all vertices in next superstep

## Fault Tolerance

* Similar to checkpointing in databases – master tells workers to save state (vertex value, edge values, list of incoming messages) in persistent storage at the start of each superstep
* Master saves any aggregator values, if applicable
* When master detects one or more worker failures, it rolls back all workers to most recent checkpoint and computation is repeated
* If only the failed worker should revert to checkpoint, requires **deterministic replay** of messages sent to that worker at each superstep since the checkpoint

# 9a – Consistency and Replication – Part 1

* Replicating data increases **reliability** and **throughput**, and reduces **latency**
* In a replicated data store, each data object is replicated at many hosts
  + Replicas can be **local** to a process (same host) or **remote**

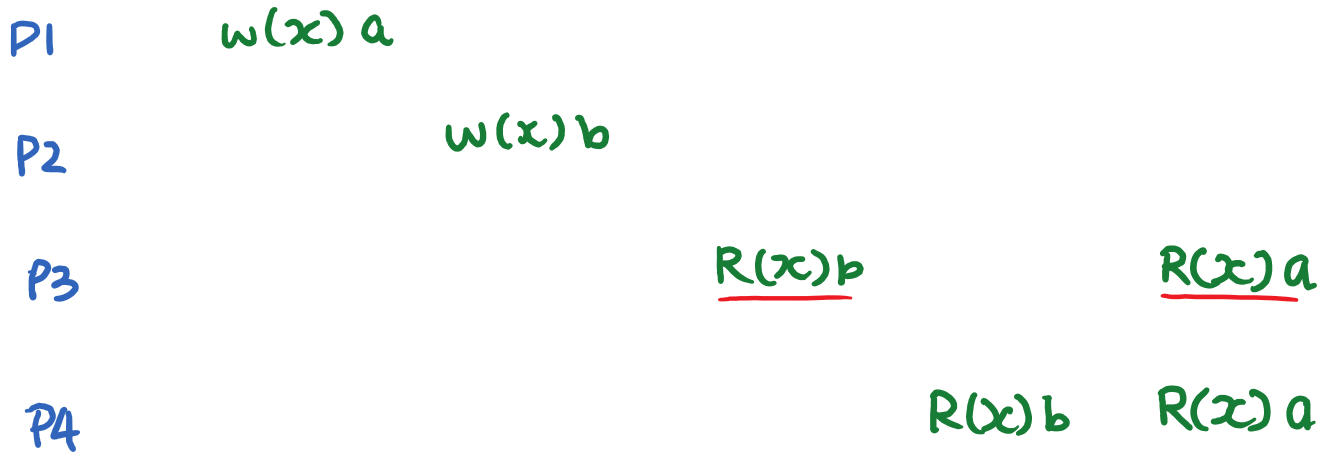
## Consistency Models

* If a data store holds mutable state, then replicas can never be perfectly synchronized, due to network delays
* If a data store holds shared mutable state, then the problem is more complex
* **Consistency models** describe the extent to which replicas can disagree on the state of the data

### Sequential Consistency

* The result of any execution is the same as if all the read and write operations were executed in a **total order**, and operations for each individual process appears in this sequence
  + Preserve program order (per process)
  + All processes agree on order of writes, as reflected in reads

Positive Example

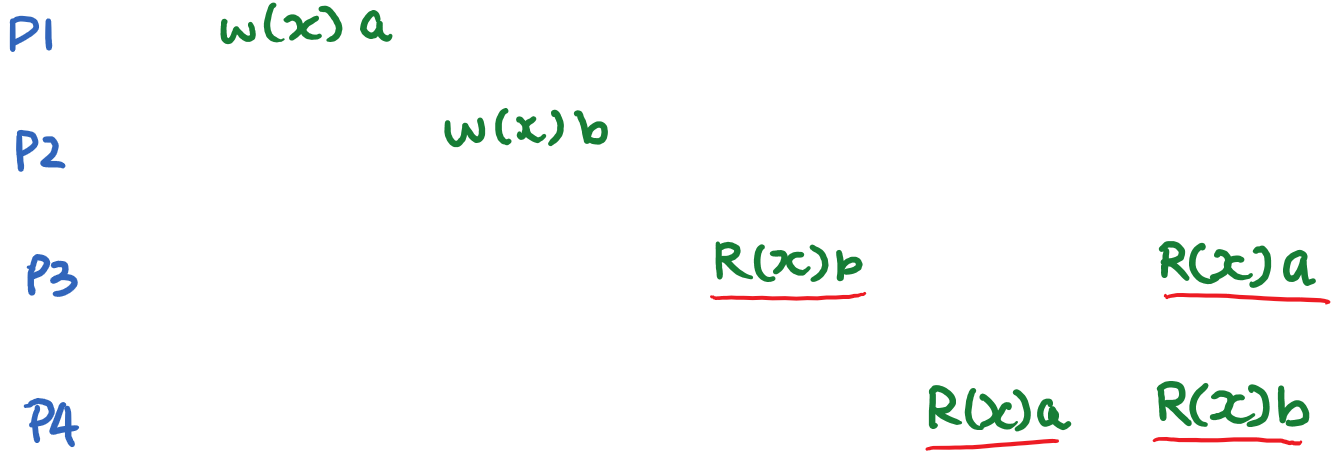




Total order: P2W(x)b, P3R(x)b, P4R(x)b, P1W(x)a, P3R(x)a, P4R(x)a



Negative Example

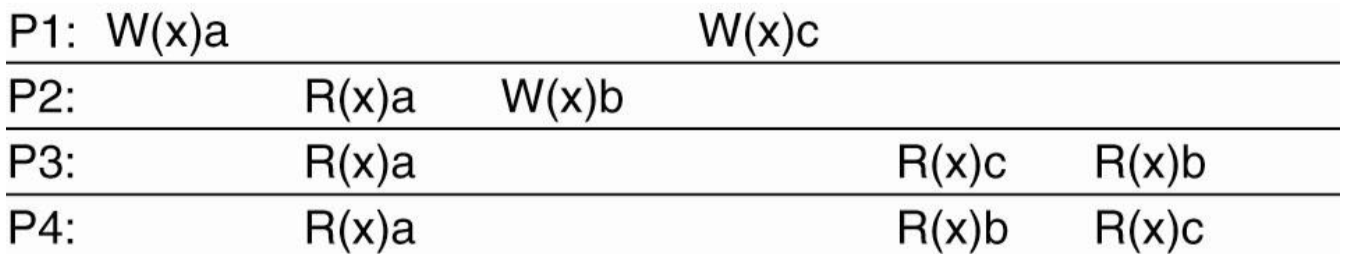


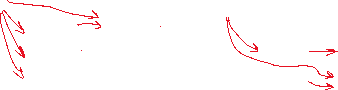
P3 and P4 do not agree on order of writes

### Causal Consistency

* **Safety property**
* Writes related by “causally precedes” relation must be seen by all processes in same order
* **Causally precedes**:
  + Op1 causally precedes Op2 if Op1 occurs before Op2 in same process
  + Op1 causally precedes Op2 if Op2 reads a value written by Op1

Positive Example

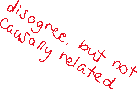




* Program orders:
  + P1: P1W(x)a, P1W(x)c
  + P2: P1W(x)a, P2R(x)a, P2W(x)b
  + P3: P1W(x)a, P3R(x)a, P1W(x)c, P3R(x)c, P2W(x)b, P3R(x)b

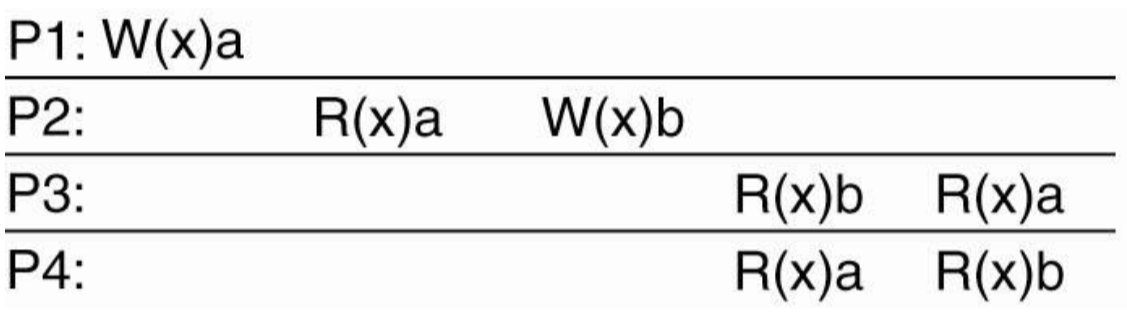


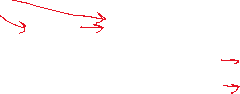
* + P4: P1W(x)a, P4R(x)a, P2W(x)b, P4R(x)b, P1W(x)c, P4R(x)c



* Not sequentially consistent, since P3 and P4 don’t agree on order of writes

Negative Example





* Program orders:
  + P1: P1W(x)a
  + P2: P1W(x)a, P2R(x)a, P2W(x)b



* + P3: P2W(x)b, P3R(x)b, P1W(x)a, P3R(x)a – violation



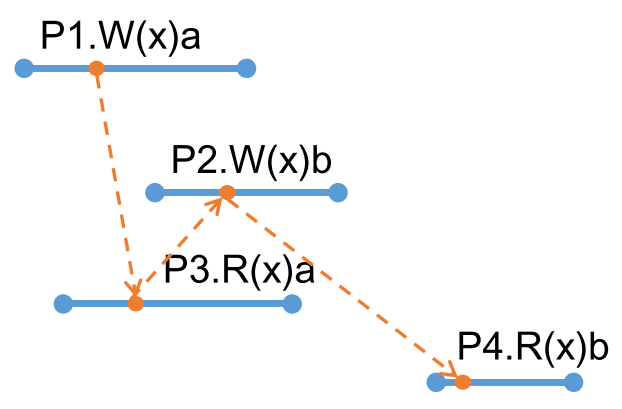
* + P4: P1W(x)a, P4R(x)a, P2W(x)b, P4R(x)b



### Linearizability

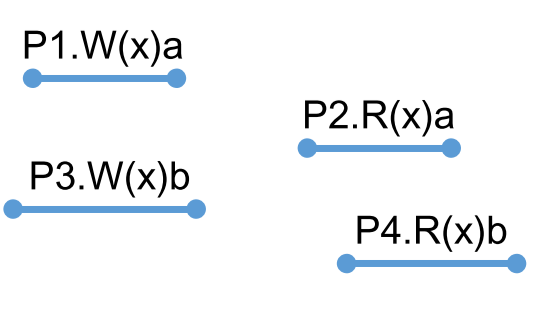
* If Op1 finishes before Op2 begins, then Op1 must precede Op2 in the sequential order

Positive Example



* P1W(x)a must precede P4R(x)b
* Total order: P1W(x)a, P3R(x)a, P2W(x)b, P4(Rx)b

Negative Example

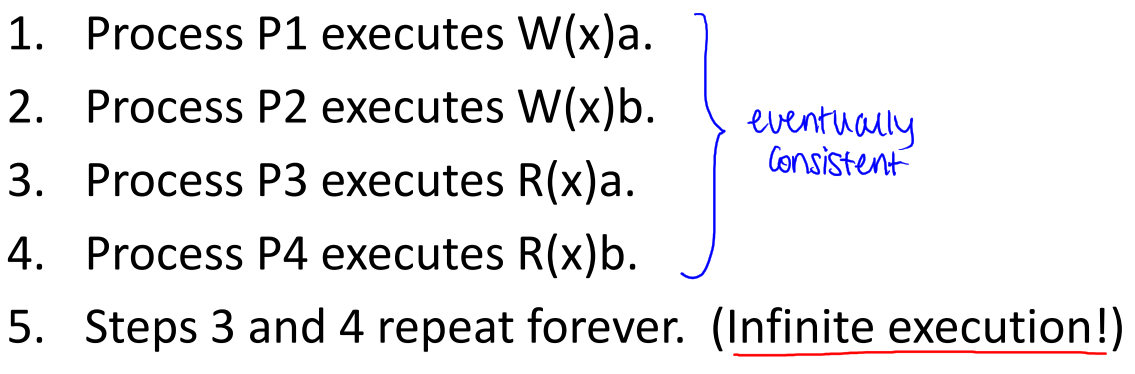


* P1W(x)a must precede P2R(x)a and P4R(x)b
* P3W(x)b must precede P2R(x)a and P4R(x)b
* Total order: cannot write

### Eventual Consistency

* In the absence of new writes, all servers eventually hold the same data – **liveness property**
  + Allows different processes to observe writes in different orders, even when they are related by “causally precedes/happens before”

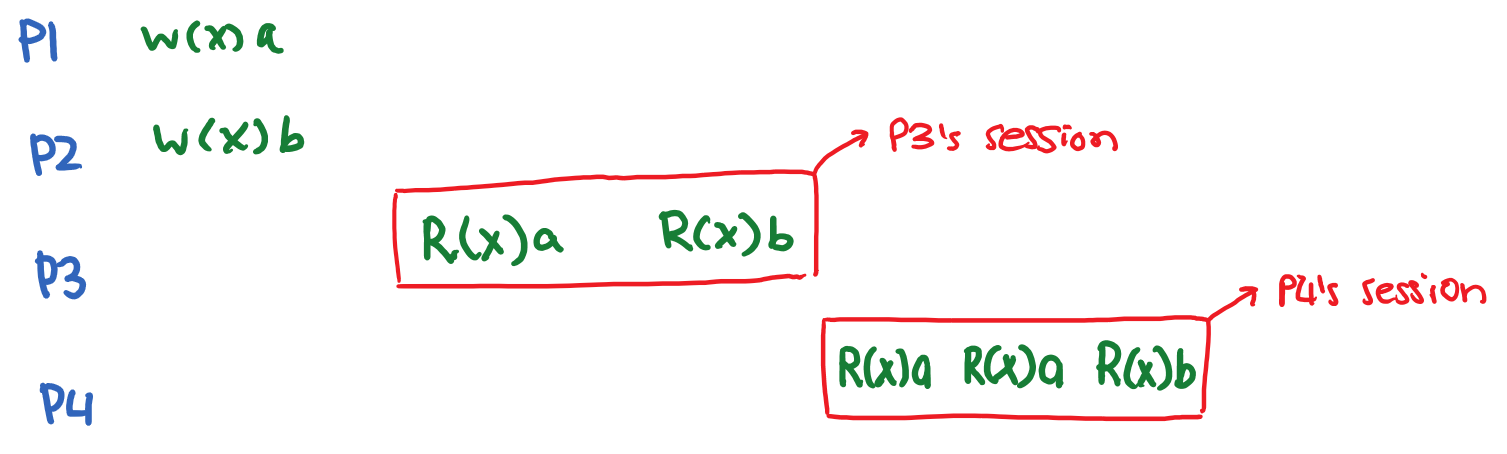
Negative Example



### Session Guarantees

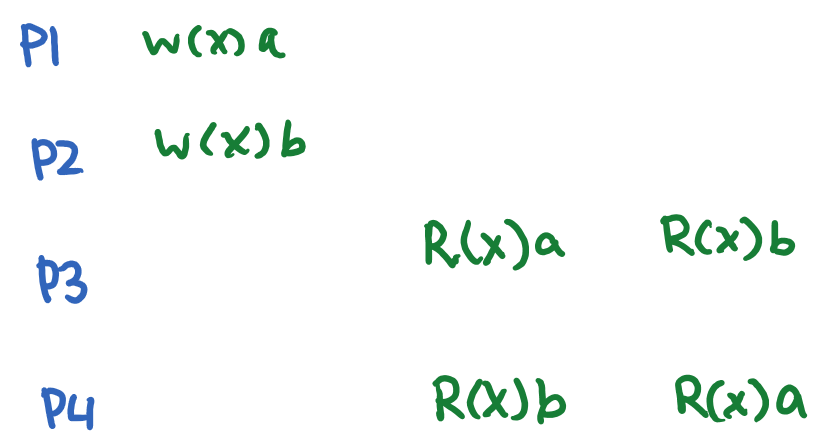
* Restrict behaviours of operations by a single process in a single session
* **Monotonic reads**: if a process reads the value of object , subsequent reads on will return the same value or more recent value

Positive Example





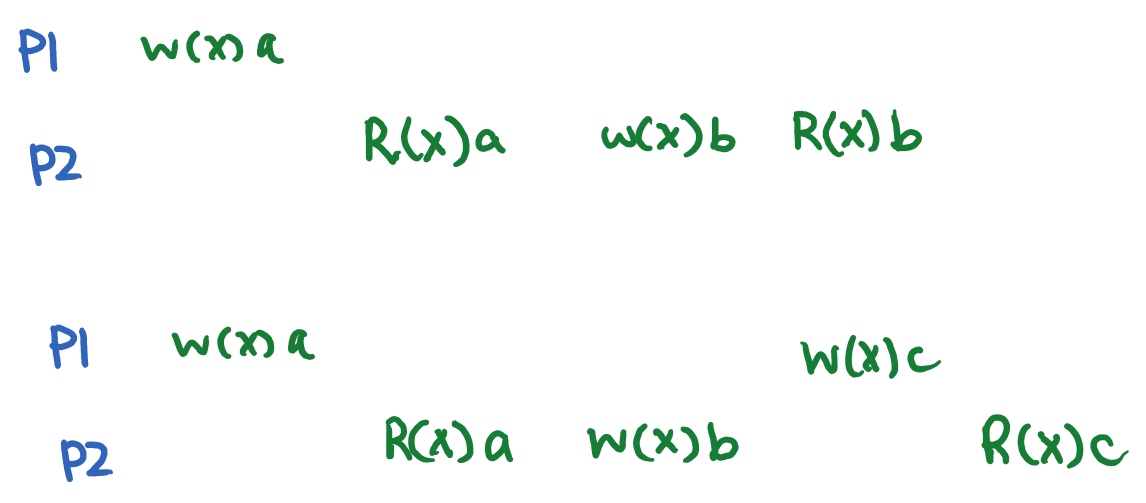
Negative Example





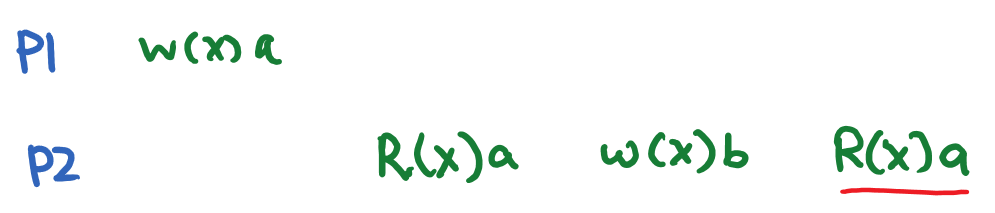
* **Read your own writes**: the effect of a write on will always be seen by a successive read on by the same process

Positive Example





Negative Example





# 9b – Consistency and Replication – Part 2

## Primary-Based Replication Protocols

* Updates are executed by a primary replica, then pushed to backup replicas
* If primary replica fails, one of the backup replicas takes over as new primary
  + Updates cost 2RT time
  + Reads cost 1RT time
  + Linearizable gets/puts

### Remote Write Protocol

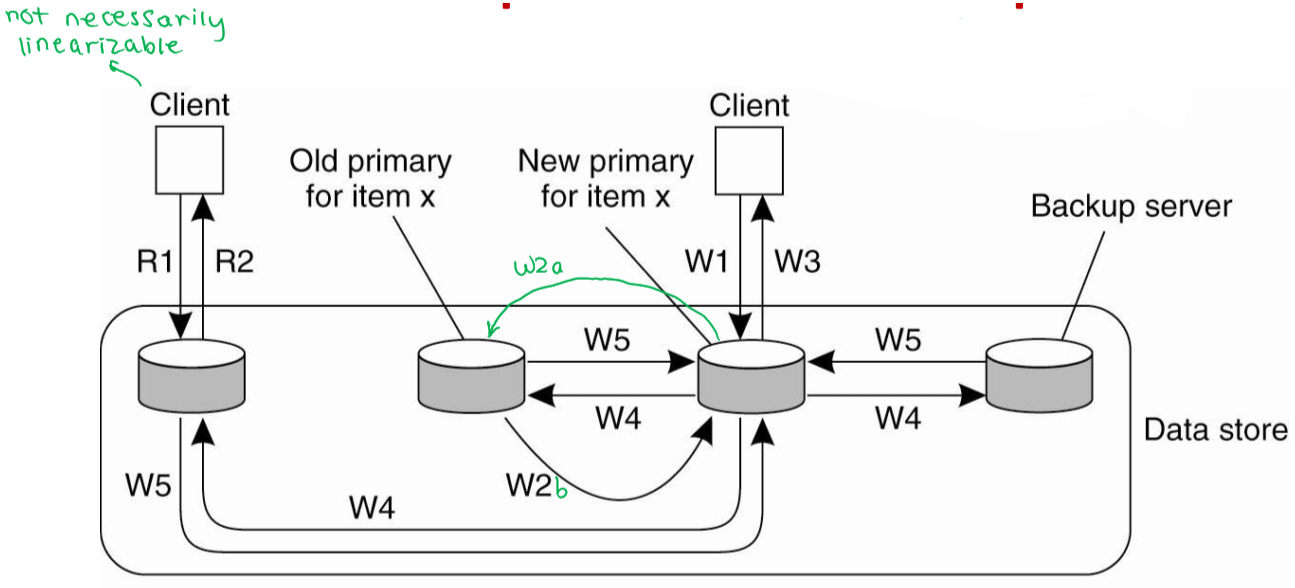
* Primary replica is stationary, must be updated remotely by other servers



* + Write request forwarded to primary (W2), who tells backups to update (W3)
  + Primary waits for all backups to acknowledge update (W4), before returning write complete (W5)

### Local Write Protocol

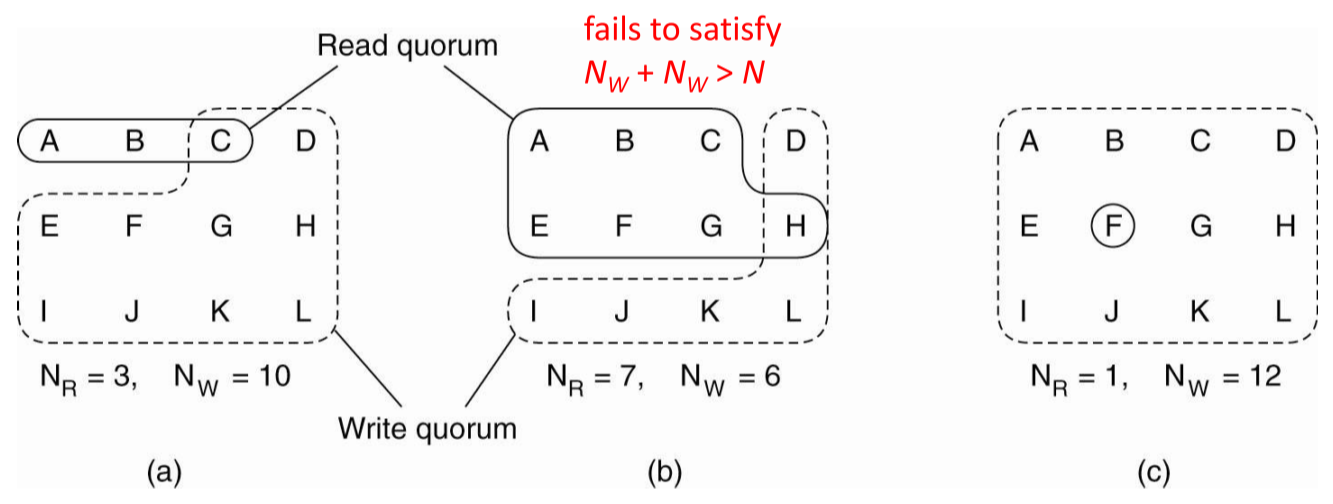
* Primary replica migrates from server to server – allows local updates
* Writes are cheaper, but lower consistency – not necessarily linearizable



* + Primary migrates from old one to new one (W2), and returns write complete (W3)
  + Primary tells backups to update (W4) and waits for all backups to acknowledge update (W5)

## Quorum-Based Protocols

* Allow all replicas to receive updates, but updates must be accepted by a sufficiently large subset of replicas – **write quorum**
* Reads require accessing a sufficiently large subset of replicas – **read quorum**
  + Updates and reads cost 2RT time
  + Not linearizable or sequentially/causally consistent
  + Eventually consistent with anti-entropy
* Two rules:
  + – read and write quorums overlap, allowing detection of read-write conflicts
  + – two write quorums overlap, allowing detection of write-write conflicts
    - total number of replicas, usually odd number
    - size of read quorum
    - size of write quorum



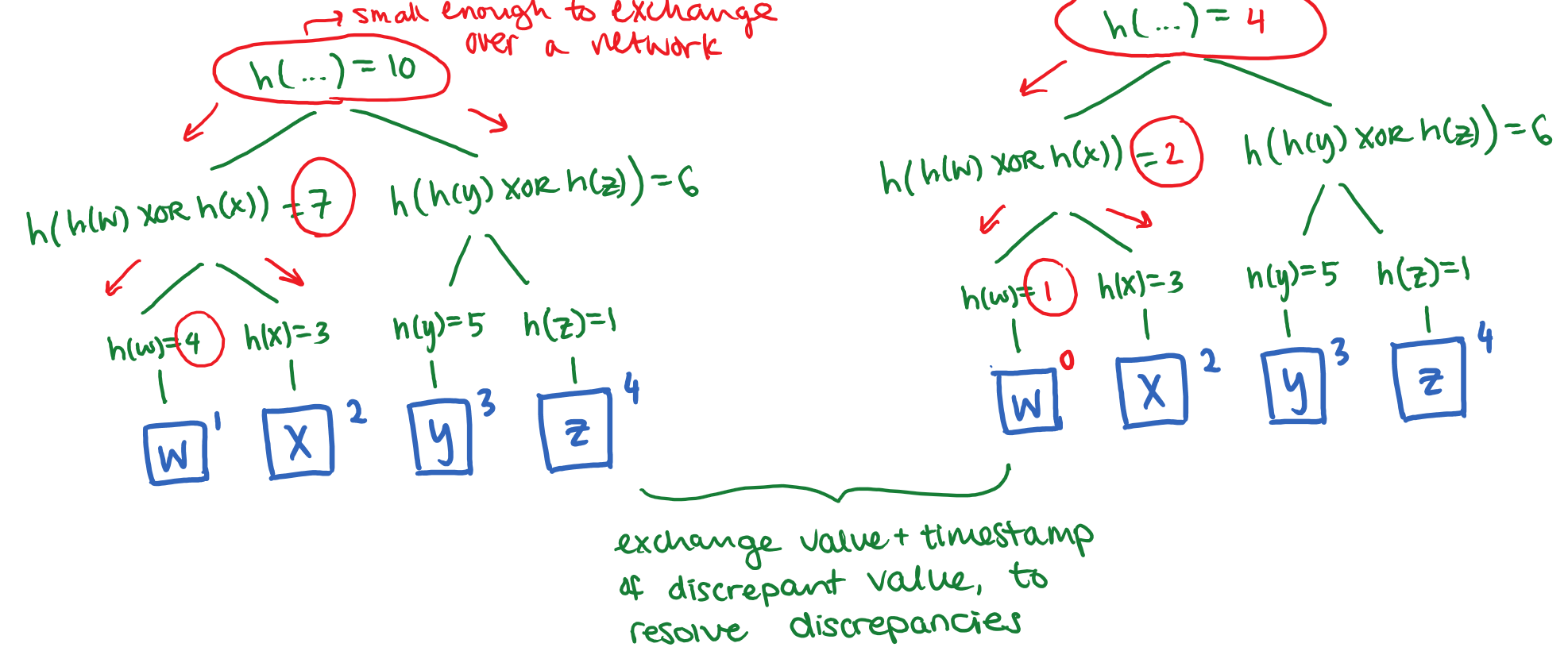
* (c) is example of **read-one-write-all (ROWA)** scheme

### Partial Quorums

* **Strict quorum (strong consistency)**: – do not avoid write-write conflicts
* **Partial quorum (weak consistency)**: – do not avoid read-write or write-write conflicts
* To resolve write-write conflicts, writes are tagged with timestamp, and **last write wins**

## Eventually-Consistent Replication

* In simplest case, server that receives update replies with acknowledgement to client, then lazily updates remaining replicas
* If replica is unreachable, then it can be updated later using anti-entropy
  + Replicas periodically exchange **Merkle/hash trees** to detect discrepancies
  + Timestamped updates allow detection of latest version



* + Replicas compute hash trees, then exchange hash trees using protocol messages
  + Replicas inspect root of hash tree to determine discrepancy
  + Replicas determine w is discrepant, exchange local copy of w with timestamp
  + Replicas update their copy to the latest timestamped one

# 10a – Fault Tolerance

* Fault tolerance is associated with **dependability**, which implies the following requirements:
  + **Availability**: system operates correctly at any given time
  + **Reliability**: system runs continuously without interruption
  + **Safety**: failure of system doesn’t have catastrophic consequences
  + **Maintainability**: failed system is easy to repair
* **Failure**: system cannot fulfill its promises
  + **Crash failure**: server halts
  + **Omission failure**: server fails to receive/respond to incoming messages
  + **Timing failure**: server response outside of specified time interval
  + **Response failure**: server’s response is incorrect
* **Error**: part of system state that may lead to a failure
* **Fault**: cause of an error
  + **Transient**: occurs one and disappears
  + **Intermittent**: occurs and vanishes, then reappears, etc.
  + **Permanent**: continues to occur until repaired
* Masking failure by redundancy: identical processes, organized into flat or hierarchical group

## Consensus Problem

* Each process has procedures propose(val) and decide()
* Each process first proposes a value by calling propose(val) with val determined by initial state of process
* Each process learns about agreed-upon value by calling decide()

**Safety property 1 (agreement)**: two calls to decide() never return different values

**Safety property 2 (validity)**: if a process calls decide() and it returns value x, then some process invoked propose(x)

**Liveness property**: if a process calls propose(val) and decide() and does not crash, the call eventually terminates

### Variations

* Synchronous vs. asynchronous processes
* Communication delays
* Message delivery order
* Unicast vs. multicast messaging

## RPC Semantics Under Failures

* Five classes of failure scenarios:
  1. Client unable to locate server
  2. Request message from client to server is lost
  3. Server crashes after receiving request
  4. Reply message from server to client is lost
  5. Client crashes after sending request

### RPC Server Crashes

* The server can crash before or after executing a request; in both cases, client doesn’t receive reply, so it has no way of knowing when the server crashed
* How to deal with failure scenarios?
  1. **At-least-once semantics** – reissue request
     + This means request may be processed multiple times by service handler
     + Request must be **idempotent** – repeated executions have same effect as one execution
  2. **At-most-once semantics** – give up and report failure
  3. **Exactly-once semantics** – determine whether request was processed, reissue if needed
     + Difficult to implement, as server doesn’t know whether it performed an action or not
  4. **Make no guarantees** – leads to confusion

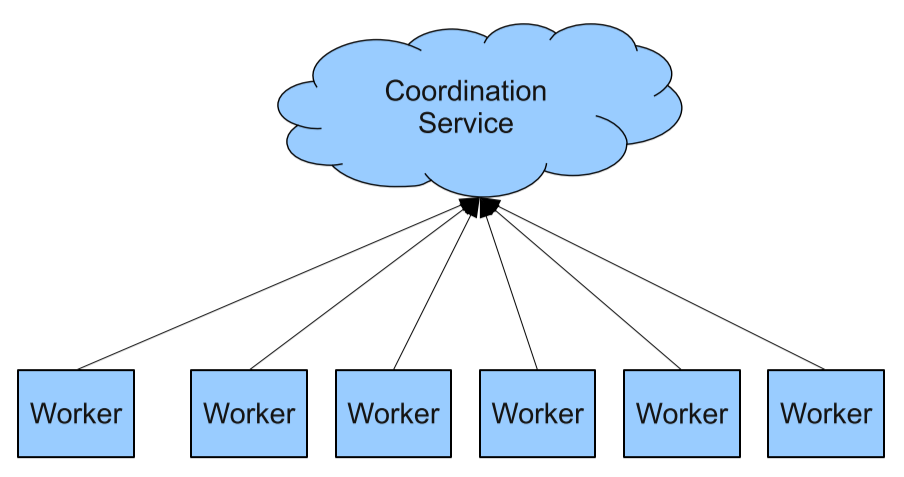
## Actions and Acknowledgements

RPC service that prints text to screen upon receiving client request – can either acknowledge request before or after printing text:

* **M:** server replies to client with acknowledgement
* **P**: server prints text
* **C**: server crashes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Client** | **Server** | | | | | |
| **Reissue strategy** | **M🡪P** | | | **P🡪M** | | |
| **MPC** | **MC(P)** | **C(MP)** | **PMC** | **PC(M)** | **C(PM)** |
| Always | Twice | Once | Once | Twice | Twice | Once |
| Never | Once | None | None | Once | Once | None |
| Only when ACKed | Twice | Once | None | Twice | Once | None |
| Only when not ACKed | Once | None | Once | Once | Twice | Once |

# 10b – Apache ZooKeeper

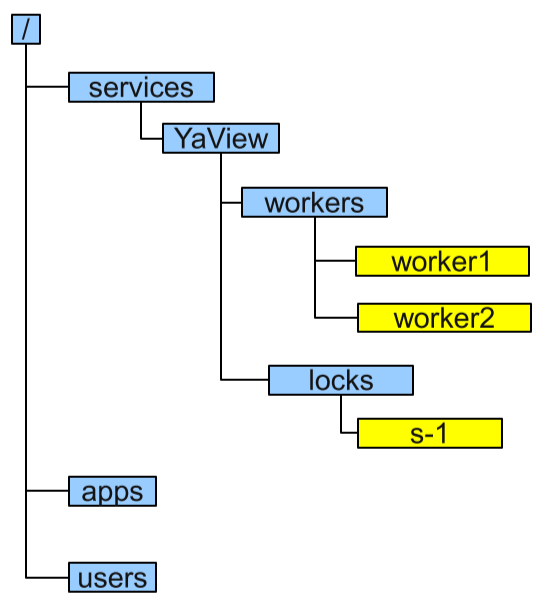


* **Linearizable reads + serializable writes**
* ZooKeeper provides APIs to do the following:
  + Group membership
  + Leader election
  + Dynamic configuration
  + Status monitoring
  + Queuing
  + Barriers
  + Critical sections – transfer lock ownership if lock owner dies
* Pros:
  + Slow processes won’t slow down fast ones
  + No deadlock
  + No blocking in implementations
* Cons:
  + Some coordination primitives are blocking
  + Need to be able to efficiently wait for conditions

### ZooKeeper Servers

* All servers have copy of state in memory
* A leader is elected at start-up – all updates go through leader, others service clients
* Update responses are sent when a majority of servers have persisted change
* Need machines to tolerate failures

## ZooKeeper Data Models

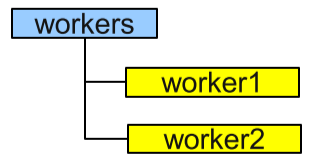


* Hierarchical namespace of **znodes**
  + Each znode has data and children
  + **Ephemeral**: comes and goes – deleted when explicitly deleted or creator fails, good for fault tolerance and failure detection
  + **Sequence**: appends monotonically increasing counter, good for symmetry breaking

### Configuration

* For workers to get configuration:   
  getData(String path, bool shouldSetWatch)
* For admin to change configuration:   
  setData(String path, byte[] data, int version) // -1 if don’t care

## Group Membership

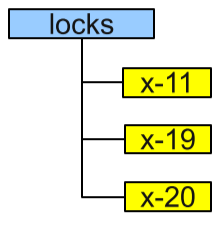
1. Register server name in group:  
   create(“../workers/workerName”, hostInfo, EPHEMERAL)
2. List group members  
   listChildren(“../workers”, true)

## Leader Election

1. getData(“../workers/leader”, true)
2. If successful, follow leader described in data and exit
3. create(“../workers/leader”, hostInfo, EPHEMERAL)
4. If successful, lead and exit
5. Go to step 1

* If watch is triggered, followers will restart leader election process

## Locks

1. id = create(“../locks/x-”, SEQUENCE | EPHEMERAL)
2. getChildren(“../locks”, false)
3. If id is the first child, exit (acquired lock)
4. exists(“last child before id”, true)
5. If does not exist, go to step 2
6. Wait for event from step 4
7. Go to step 2

### Shared Locks

1. id = create(“../locks/s-”, SEQUENCE | EPHEMERAL)
2. getChildren(“../locks”, false)
3. If no children that start with x- before id, exit (acquired lock)
4. exists(“last x-child before id”, true)
5. If does not exist, go to step 2
6. Wait for event from step 4
7. Go to step 2

## Consensus Problem

propose(val) {

1. create(“../consensus/v-”, val, EPHEMERAL | SEQUENCE)

}

decide() {

1. getData(“../consensus/decision”, true)
2. If successful, return data and exit
3. firstVal = getChildren(“../consensus”, false)[0]
4. val = getData(firstVal)
5. If successful, create(“../consensus/ decision”, val, EPHEMERAL)
6. If successful, lead and exit
7. Go to step 1

}

# 10c – Distributed Commits and Checkpoints

## Distributed Commit

* Correctness properties of transactions – ACID:
  + **Atomicity**: all updates take affect, or none of them
  + **Consistency**: constraints (e.g. referential integrity) are preserved
  + **Isolation**: concurrent transactions are unaware of each other
  + **Durability**: updates in a committed transaction aren’t lost during failure
* Distributed commit problem: atomicity in a distributed environment
  + E.g. changing bank account balances in a transfer

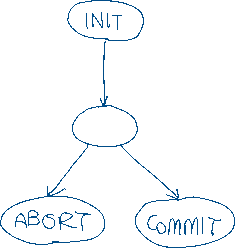
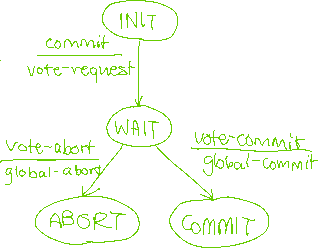
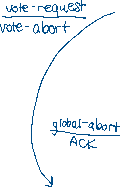
## Two-Phase Commit (2PC)

* Coordinator-based distributed transaction commitment protocol:
  + **Phase 1**: coordinator asks participants if they are ready to commit, participants respond with vote
  + **Phase 2**: coordinator examines vote, determines outcome
    - If all participants vote to commit, success
    - Otherwise, abort
* Assumptions: synchronous processes, bounded delays and crash-failure recoveries, processes have access to stable memory to log recovery information

### Coordinator



### Participant



1. Coordinator logs START\_2PC, sends VOTE-REQUEST to all participants
2. When participant receives VOTE-REQUEST, it either sends VOTE-COMMIT (if prepared to locally commit) or VOTE-ABORT
3. Coordinator collects all votes; if all participants vote to commit, coordinator sends GLOBAL-COMMIT; if one participant votes to abort, or timeout occurs, it sends GLOBAL-ABORT
4. Each participant that voted for commit waits for coordinator response; if it receives GLOBAL-COMMIT, it commits, otherwise, it aborts

### Recovery from Failure

If the coordinator crashes:

* Participant can make progress if it learns about the decision from the coordinator or by contacting another participant
  + If participant P doesn’t hear back from coordinator within a certain time interval, it may try to learn the decision from another participant Q:

|  |  |
| --- | --- |
| **Q’s state** | **Action by P** |
| COMMIT | Make transaction to COMMIT |
| ABORT | Make transaction to ABORT |
| INIT | Make transaction to ABORT |
| READY | Contact another participant |

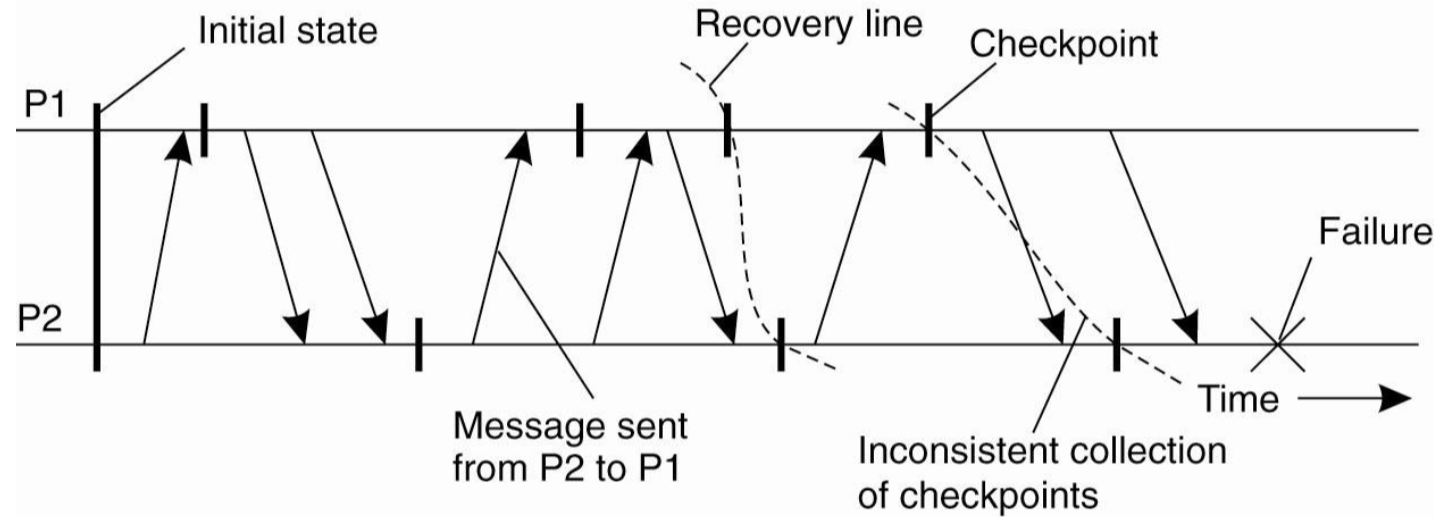
* Transaction is safe to commit if all participants vote to (all in READY or COMMIT state), and safe to abort otherwise
* If all participants in READY state, block until coordinator recovers

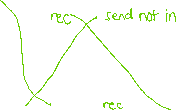
If a participant and coordinator both crash:

* Simultaneous failure of coordinator and a participant makes it harder to determine whether all participants are READY

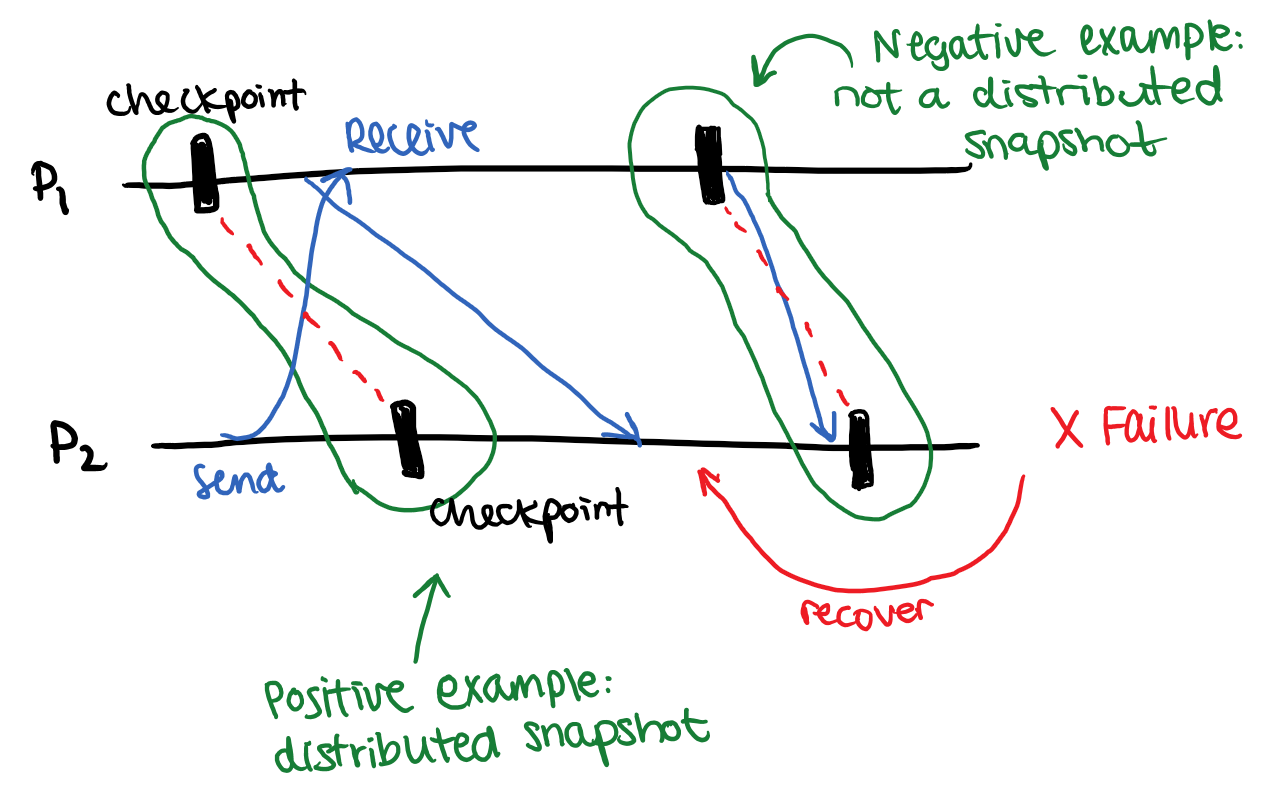
## Distributed Checkpoints

* Recovery is only possible if the collection of checkpoints forms a **distributed snapshot**
  + If receive event is in the snapshot, so is the corresponding send event
* **Recovery line**: most recent distributed snapshot





* + A send can be in the recovery line if **receive event isn’t**



* **Domino effect**: if the most recent checkpoints do not provide a recovery line, then successively earlier checkpoints must be considered

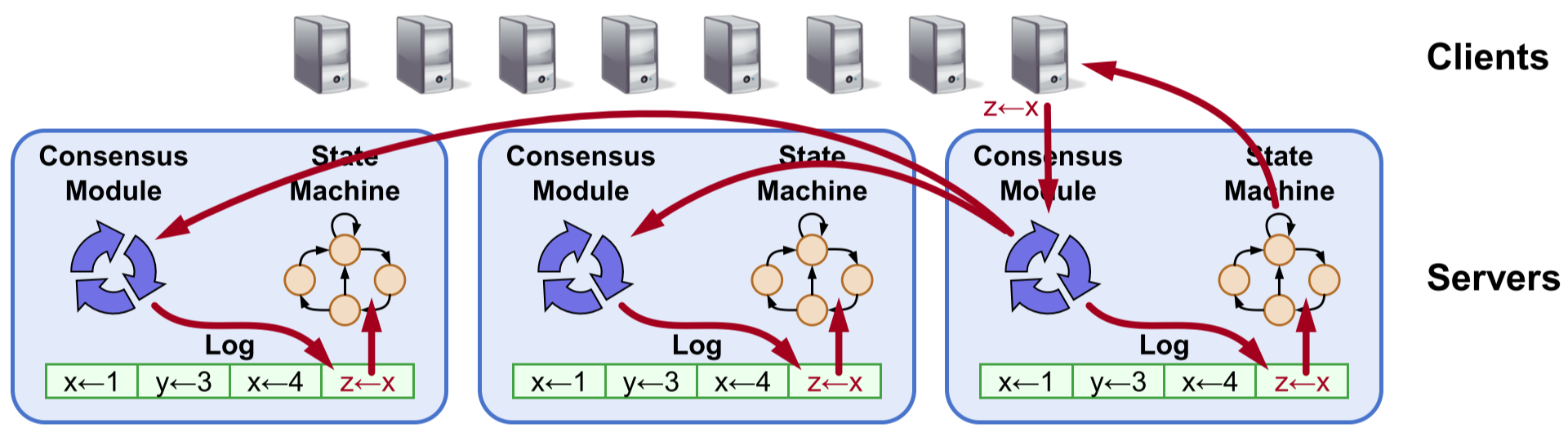
### Coordinated Checkpointing

* Ensures that recovery line is created:
  + **Phase 1**: coordinator sends CHECKPOINT\_REQUEST to all processes
    - Upon receiving message, processes do the following:
      * Pause incoming messages
      * Create local checkpoint
      * Return ACK to coordinator
  + **Phase 2**: after receiving ACK from all processes, coordinator sends CHECKPOINT\_DONE to all processes
    - Incoming messages resume
* Assumption: sending is in response to receiving, so pause receiving 🡪 pause sending

# 10d – Raft (Reliable and Fault Tolerant)

* **Consensus algorithm** that allows continuous service between a collection of machines, even if some machines fail

## Replicated State Machine



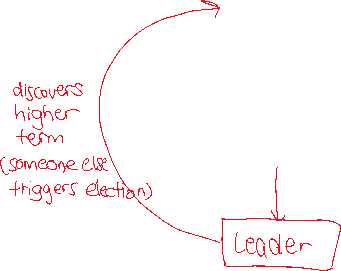
* **Replicated log** ensures that state machines execute commands in same order
  + **Consensus module** ensures proper log replication
  + System makes progress as long as majority of servers are up
  + Failure mode: delayed/lost messages, fail-stop

## Raft Decomposition

1. **Leader election**:
   * Select a server to be leader
   * Detect crashes and choose new leader
2. **Log replication:** 
   * Leader accepts client commands, append own log (never deletes or modifies entries)
   * Leader issues AppendEntries RPC to replicate logs to other servers, overwriting inconsistencies
3. **Safety**:

* **State Machine Safety**: if a server applies a log entry at a given index to its state machine, no other server will apply a different log entry for same index
* **Leader Completeness**: If a long entry is committed, that entry is present in logs of all future leaders
  + Only servers with most updated logs can become leader
    - Candidates include <index, term> of their last log entry
    - Voting server rejects vote if its log is more up-to-date

## Leader/Follower Server States and RPCs

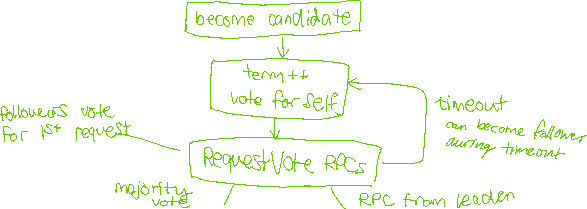


### Terms

* At most one leader per term – some terms have no leader
* Each server maintains **current term** view (no global view)
  + Exchanged in every RPC
  + If discover a higher term, revert to follower
  + If incoming RPC has outdated term, reply with error (containing current term)

### Elections

* **Safety**: at most one leader per term – each server gets one vote, majority required to win
* **Liveness**: some candidate must win eventually
  + Choose random timeout between [T, 2T]
  + One server usually times out and wins before others time out

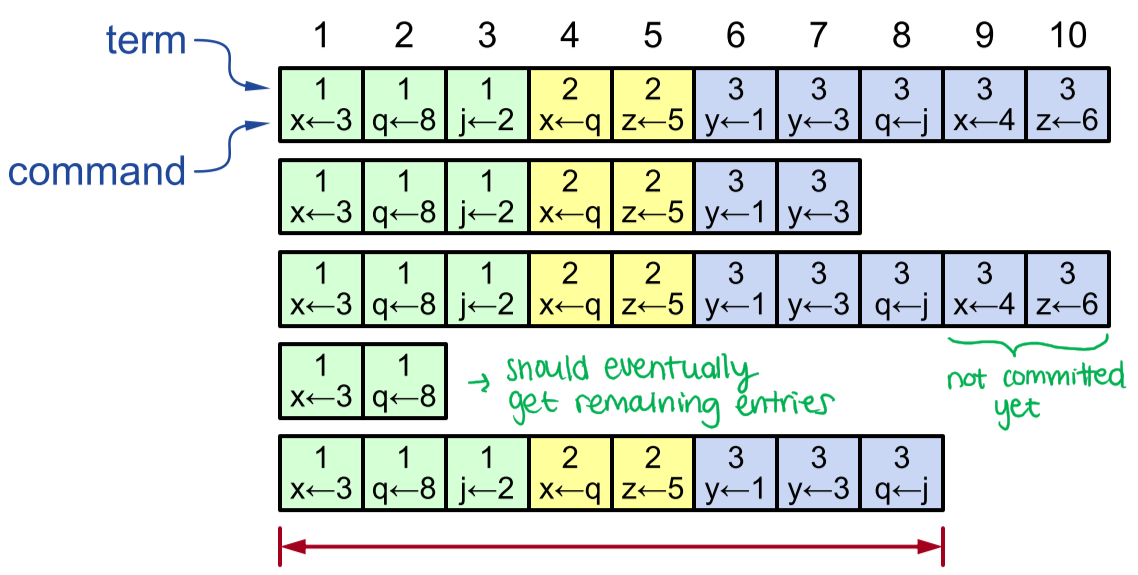


## Normal Operation

1. Client sends command to leader
2. Leader appends command to log
3. Leader sends AppendEntries RPC to all followers
4. When new entry **committed** – replicated on majority of servers:
   1. Leader executes command in state machine, returns result to client
   2. Leader notifies followers of committed entries via AppendEntries RPCs
   3. Followers execute committed commands in state machines

* Leader retries AppendEntries RPCs to crashed/slow followers until success
* Optimal performance when RPC successful to any majority of servers

## Log Structure



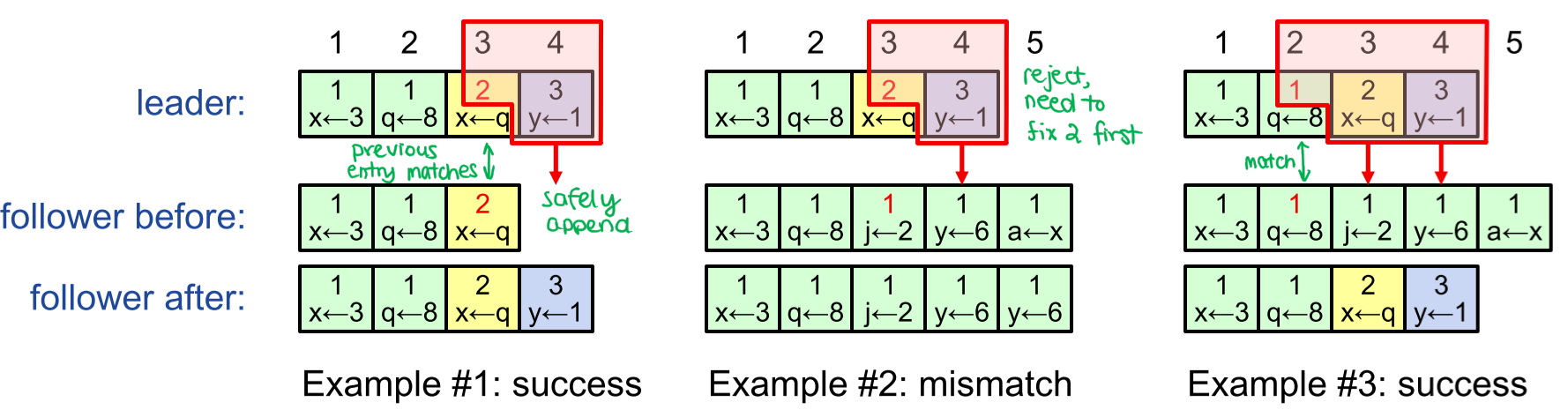


* Leader always assumes it has correct log
* Normal operations will repair inconsistencies

### Log Matching Property

* If two logs contain an entry with **same index and term**, then they store the same command, and all previous entries are identical
* If any given entry is committed, so are all preceding entries
  + Need to fix entries in increasing order

### AppendEntries Consistency Check



* Each AppendEntries RPC contains **<index, term>** of preceding entry to new one
* Follower must contain matching preceding entry – otherwise, rejects request
  + Then, leader retries with lower log index

# 11 – Apache Kafka

* Message-oriented communication – publish/subscribe
* Real-time stream processing
* Distributed and replicated storage of messages and streams
* **Topic**: stream of records, stored as partitioned log
* For each consumer, Kafka stores position of next record to be read (offset)

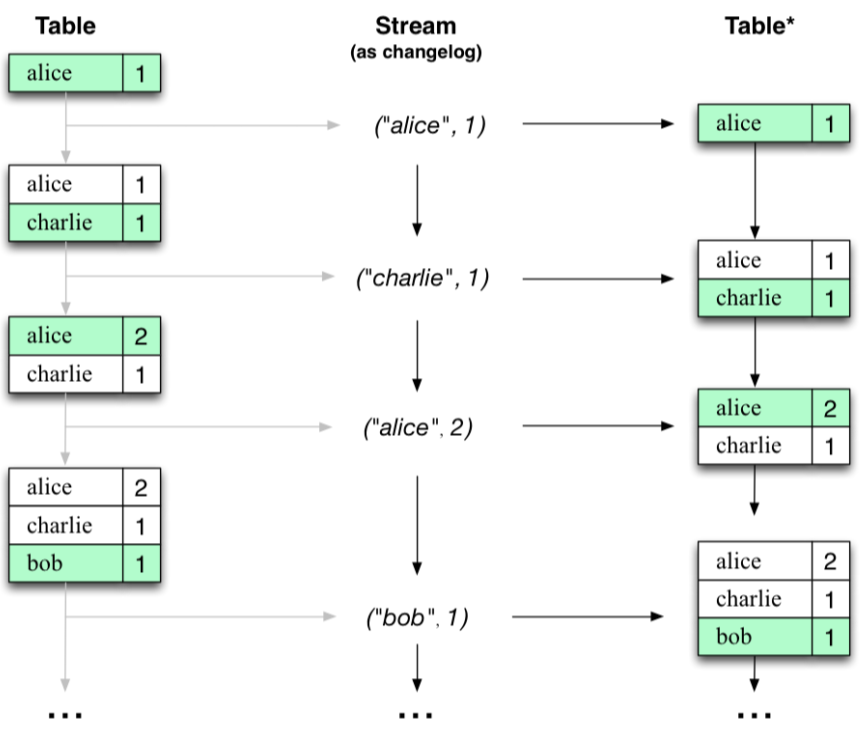
### Producers

* Pushes records to Kafka broker, chooses partition to contact for given topic
* Batch records and send asynchronously
* Idempotent delivery, to avoid duplicates

### Consumers

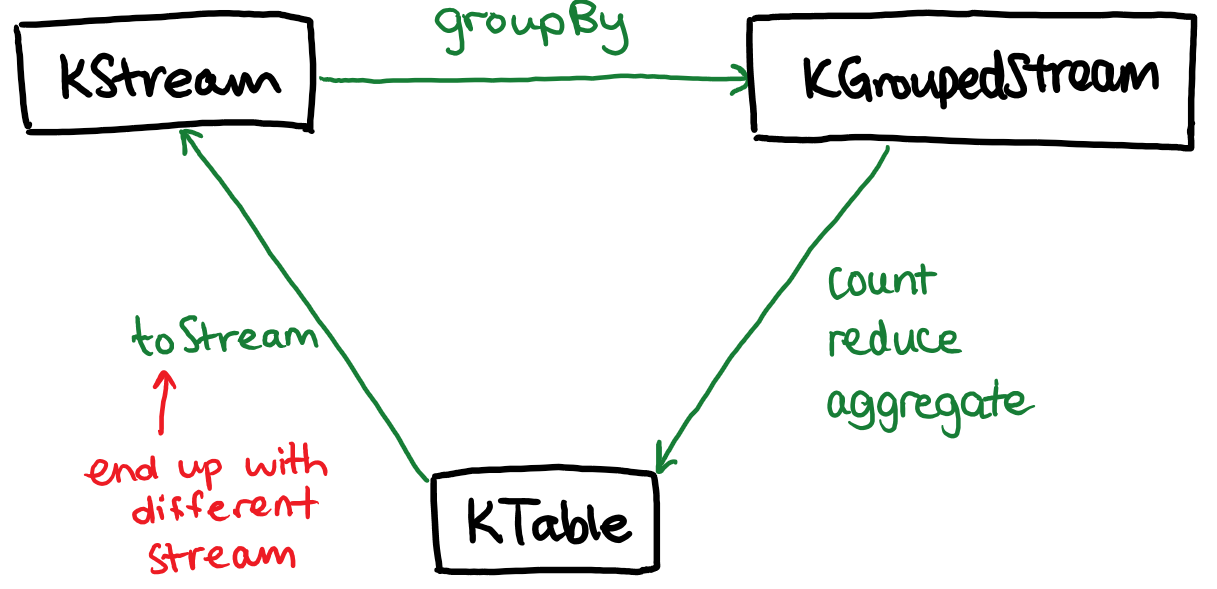
* Pulls records in batches from Kafka broker, who advances offset
* Exactly once semantics when client consumes one topic and produces to another

## Record Stream vs. Changelog Stream



* **Record stream (KStream)**: each record represents state transition
  + List of value changes for each key
* **Changelog stream (KTable)**: each record represents a state
  + Snapshot of latest value for each key

### Converting between KStream and KTable



## Windowed Streams

* **Hopping time windows**: defined by size and advance interval
  + E.g. hop = 10 seconds, size = 60 seconds of data
  + May overlap or have gaps
* **Tumbling time windows**: special case of ^, where windows size = advance interval
  + This makes them non-overlapping and gapless
* **Sliding windows**: slide continuously over time axis
  + Only used for joins
* **Session windows**: aggregate data by period of activity
  + New session created after inactivity timeout

# 12 – Clocks

## Calendars

* **Roman Calendar**: lunar calendar, 10 🡪 12 months, 29 to 30-day months
* **Julian Calendar**: solar calendar, leap years 3 🡪 4 years, not aligned with solar events (equinoxes/solstices)
* **Gregorian Calendar**: calculated leap year more carefully

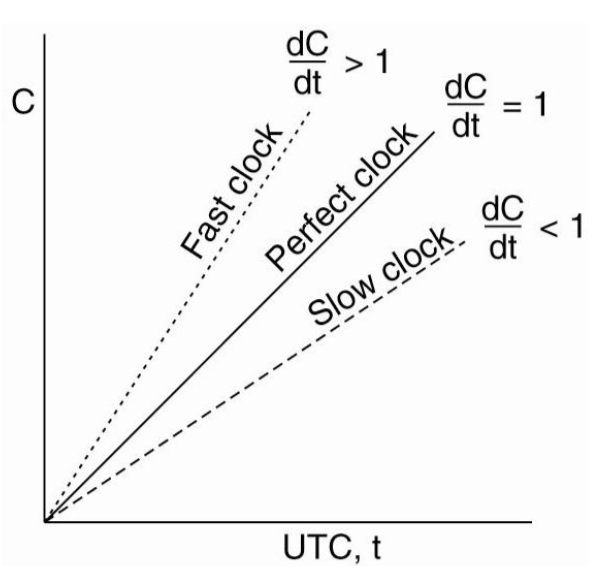
## Timekeeping Standards

* **Solar transit**: occurs when Sun reaches highest point in the day
  + **Solar day**: time between two consecutive transits of sun
  + **Temps Atomique International (TAI)**: international timescale, based on average of multiple Cesium-133 atomic clocks
  + **Universal Coordinated Time (UTC)**: based on TAI, adjusted using leap seconds when discrepancy exceeds 800ms; synchronized with Earth’s rotation

### Hafele-Keating Experiment

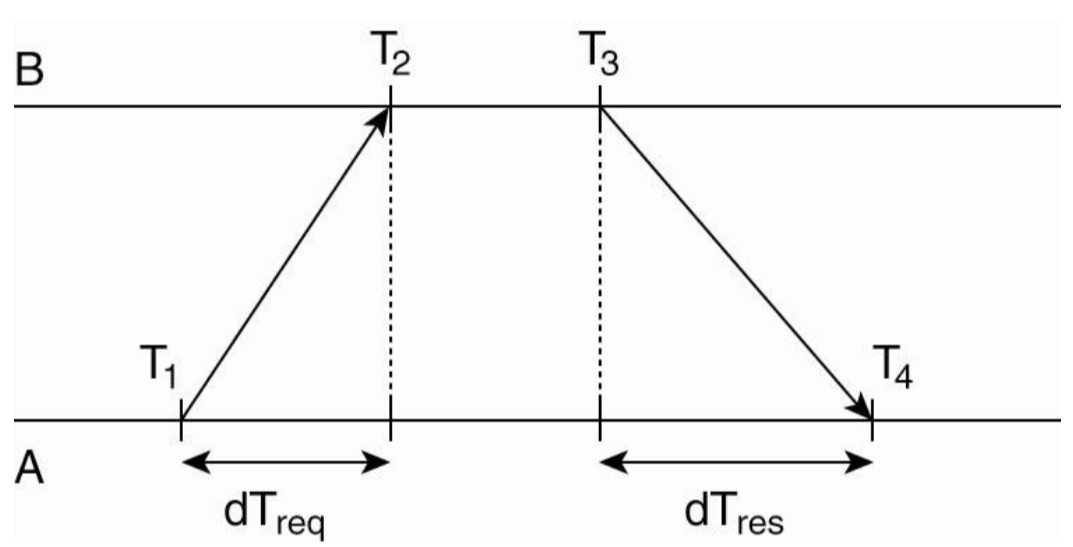
* Four atomic clocks flown around the world in opposite directions, then compared to atomic clocks on the ground:
  + **Gravitational time dilation**: both gained time due to altitude
  + **Kinematic time dilation**: west = gained time, east = lost time

### Fast and Slow Clocks



* **Clock skew**:
* **Offset**:
* **Maximum drift rate**: constant such that

## Network Time Protocol (NTP)





Suppose

Then:

Solving for :

Solving for :

Example: let for all , and .

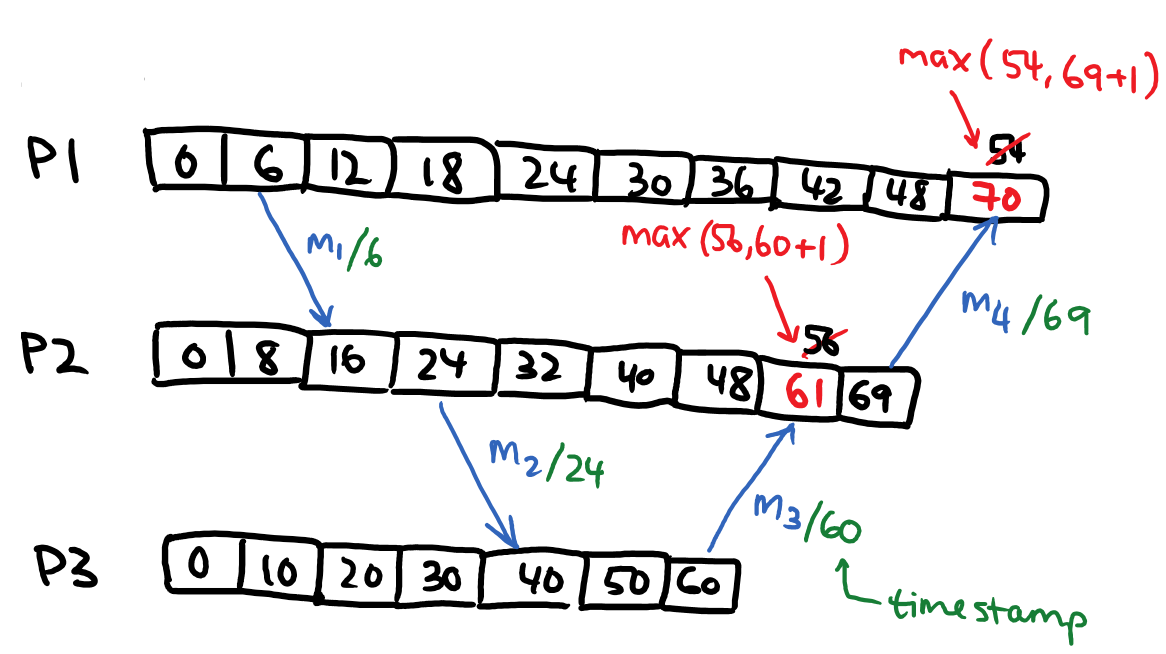
### NTP Practical Considerations

* Stratum levels:
  + **Stratum 0**: atomic clock
  + **Stratum 1**: server with atomic clock
  + **Stratum N**: server synchronized with a stratum N–1 time source
* When host A contacts host B, it will only adjust time if it has higher stratum level
  + Then, stratum level becomes B’s level + 1
* **Precision Time Protocol**: leverages hardware timestamping to achieve much better accuracy

## Lamport Clocks

* **Happens before** relation ():
  + If and are events in same process and occurs before ,   
     is true
  + If is event [process sends message] and is event [message received by another process],   
     is true
  + If neither nor , then and are **concurrent**

### Updating Local Clocks



* Every process tags message with timestamp of their local time
* When message received, process updates local time =
  + This ensures that if ,
  + However, does NOT imply
    - If this is the case, implies **either or and are concurrent**

## Vector Clocks

* Vectors , one entry per process:
  + is the number of events that have occurred so far at process
  + If , then knows that events have occurred at
* Updating vector clocks:
  + Before sending message, increments own clock
  + Upon receiving message, process takes for each vector entry
* if **all elements** of corresponding element of and **at least one** element of corresponding element of
* **Complete causality**: if then and if then

# 13 – CAP Principle

## Common Myths

1. Conventional DBs do not scale
   * Scale up: add memory/storage/cores
   * Scale out: add replicas
2. Transactions do not scale
   * **Multi-master replication**
   * **Partitioning/sharing**
3. Scalability implies high latency

## Brewer’s Conjecture

* It is impossible to obtain all 3 simultaneously in a distributed system:
  + **Consistency**: clients agree on latest state of data
  + **Availability**: clients can execute reads and writes on data
  + **Partition tolerance**: system functions when networks fail, and nodes form disjoint sets
* More precisely:
  + In the event of **partition**, system must choose **consistency** or **availability**
  + During failure-free operation, system can be both

### Consistency Models

* **CP system**: e.g. ACID database
  + Serializability
  + Linearizability
  + Sequential consistency
* **AP system**: e.g. eventually-consistent system with hinted handoff
  + Eventual consistency
  + Causal consistency

## PACELC

if there is a network **Partition**, then choose between  
 **Availability** and **Consistency**  
else, choose between  
 **Latency** and **Consistency**

### Tunable Consistency

* **Strong consistency**: if clients read and write overlapping replicas, every read is guaranteed to observe the effects of every write finished before read started
* **Tunable consistency**: values for partial read/write quorums determined on per-request basis, based on client-side consistency settings
  + E.g. Apache Cassandra: ONE, QUORUM, ALL for reads and writes

### Client-Side Consistency Settings vs. CAP

* Strong consistency () consider C in CAP
* In general, read ONE / write ONE for is NOT considered AP
  + **Sloppy quorums**: change partial quorums dynamically
  + **Hinted handoff (WRITE ANY)**: in Cassandra, allows arbitrary node to accept an update for a given key; surrogate node holds value until a replica is available

## Cassandra

* Quorum-replicated key-value store supporting tunable consistency and optional full write availability
* Schema:
  + **Keyspace** – namespace for column families
  + **Column family** – each column has name/value/timestamp
  + **Row key** uniquely identifies each row
  + **Sparse-column storage**: only columns present are stored
  + **NO JOINS or FOREIGN KEYS**

### Consistency

* Wait times ():
* **ONE/TWO/THREE**:
* **ANY**: writes only, like ONE but uses hinted handoff if needed
* **QUORUM**:
* **ALL**:
* **LOCAL\_ONE/LOCAL\_QUORUM**: local data center only
* **EACH\_QUORUM**: writes to each data center

### Put

* Executed by **coordinator** on behalf of client
* Coordinator sends update to all replicas of a row – wait for quorum # replies

### Get

* Executed by **coordinator** on behalf of client
* Contacts all replicas of a row, using two types of requests:
  + **Direct read request** – closest replica
  + **Digest request** – hash of data from remaining replicas (waits for to respond)
  + **Background read repair request** – discrepancy discovered among hashes from different replicas – tells replicas to obtain latest value