Consistency, Replication and CAP theorem

Viet-Trung Tran
is.hust.edu.vn/~trungtv/

Outline

Reasons for Replication

- Reliability
  - Mask failures
  - Mask corrupted data
- Performance
  - Scalability (size and geographical)
- Examples
  - Web caching
  - Horizontal server distribution

Cost of Replication

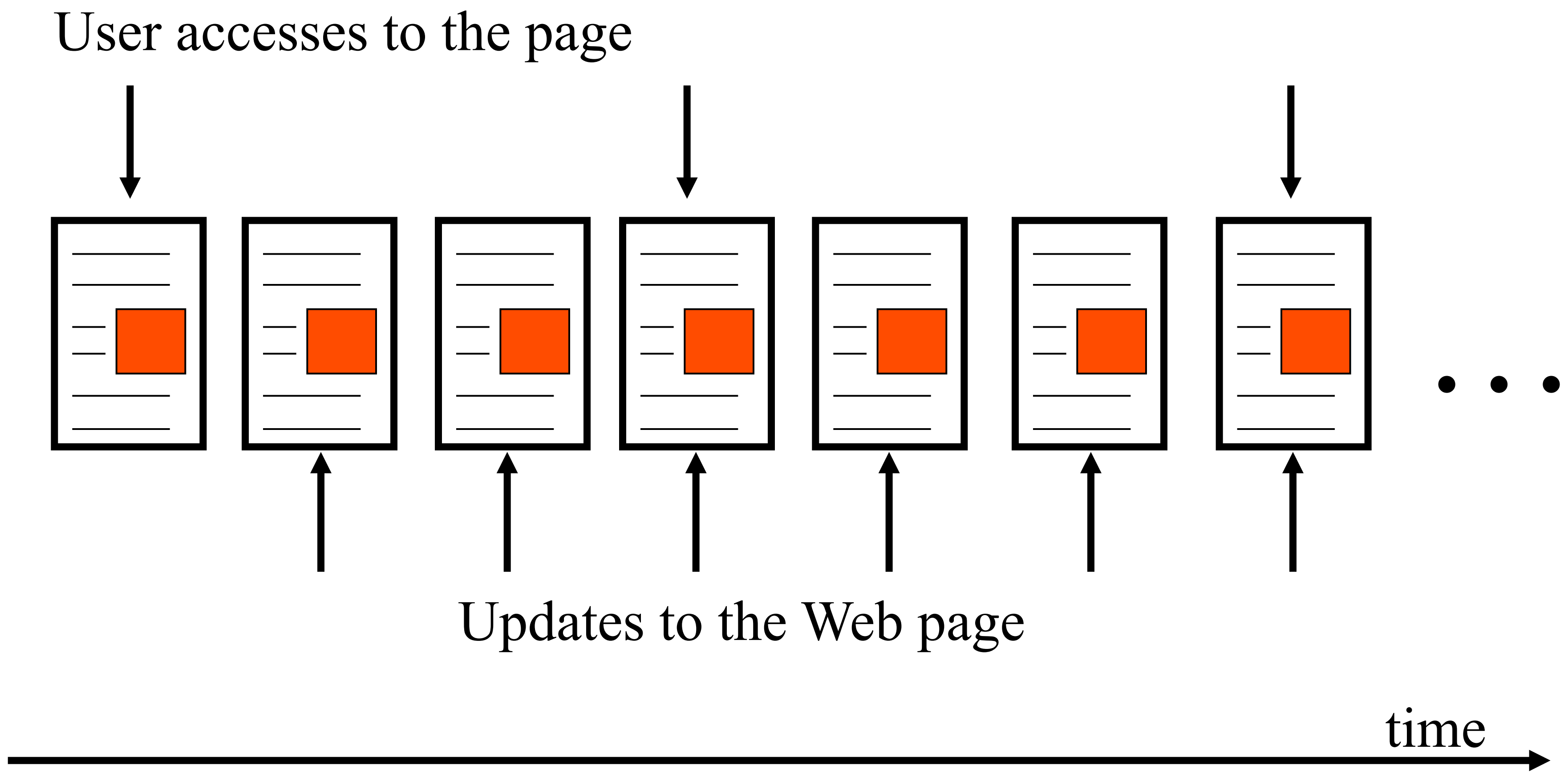
- Replicas must be kept consistent

Dilemma:

1. Replicate data for better performance
2. Modification on one copy triggers modifications on all other replicas
3. Propagating each modification to each replica can degrade performance

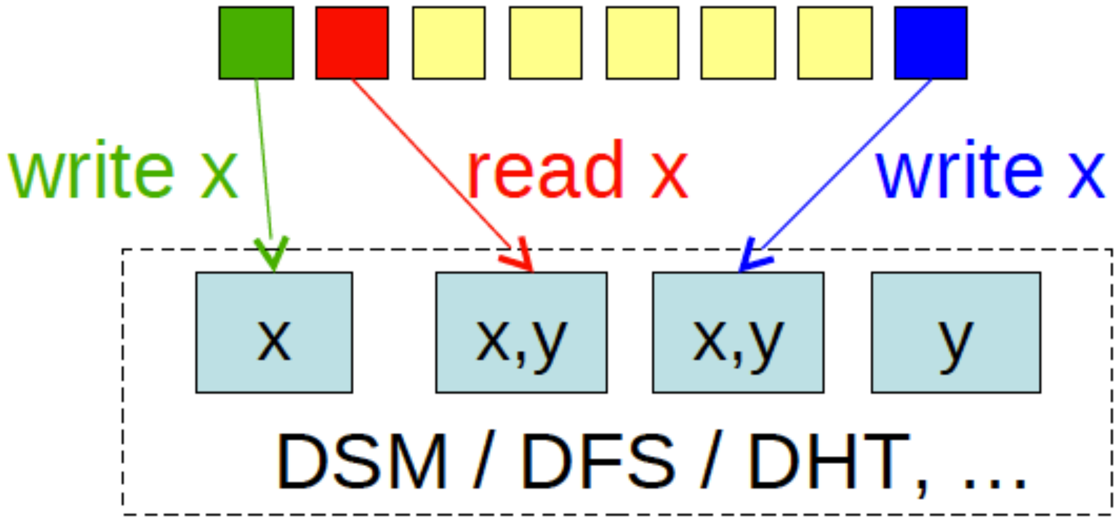
?

Consistency Issues – Access/Update Ratio



Consistency

- What is consistency?
  - What processes can expect when RD/WR shared data concurrently
- When do consistency concerns arise?
  - With replication and caching
- Why are replication and caching needed?
  - For performance, scalability, fault tolerance, disconnection



Consistency model

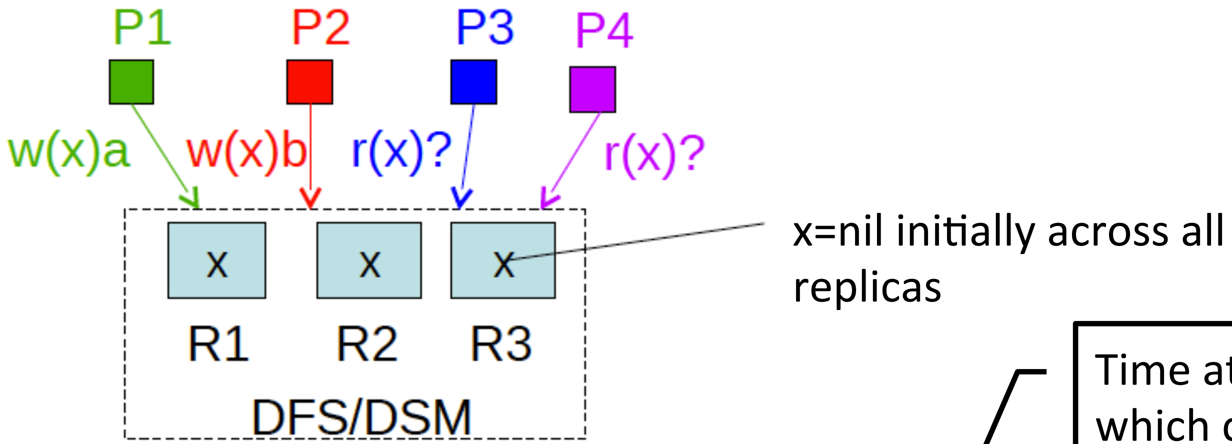
- What is a consistency model?
  - Contract between a distributed data system (e.g., DFS) and processes constituting its applications
  - E.g.: "If a process reads a certain piece of data, I (the DFS) pledge to return the value of the last write"
- What are some consistency models?
  - Strict consistency
  - Sequential consistency
  - Causal consistency
  - Eventual consistency

$\downarrow$

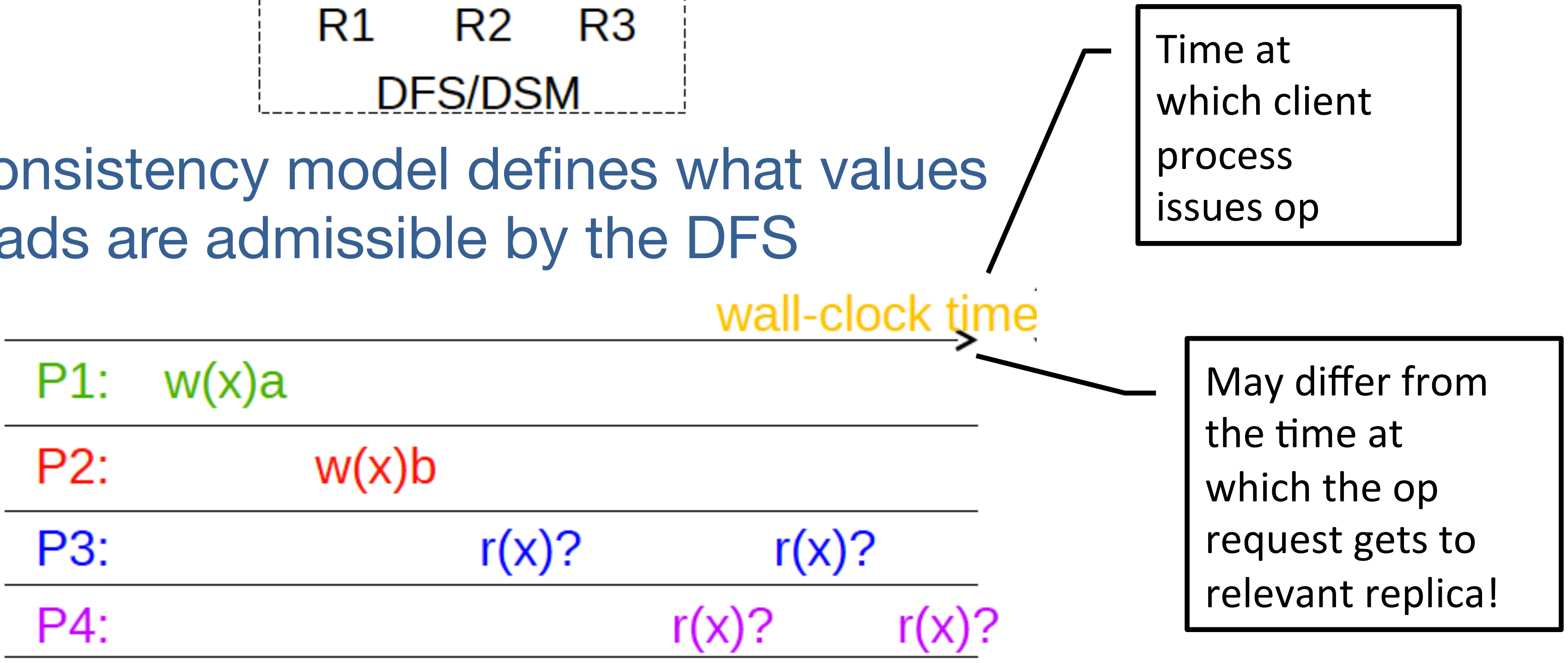
- Less intuitive, harder to program
- More feasible, scalable, efficient (traditionally)

- Variations boil down to:
  - The allowable staleness of reads
  - The ordering of writes across all replicas

Example



- Consistency model defines what values reads are admissible by the DFS


wall-clock time
P1: w(x)a
P2: w(x)b
P3: r(x)? r(x)?
P4: r(x)? r(x)?
Time at which client process issues op
May differ from the time at which the op request gets to relevant replica!

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

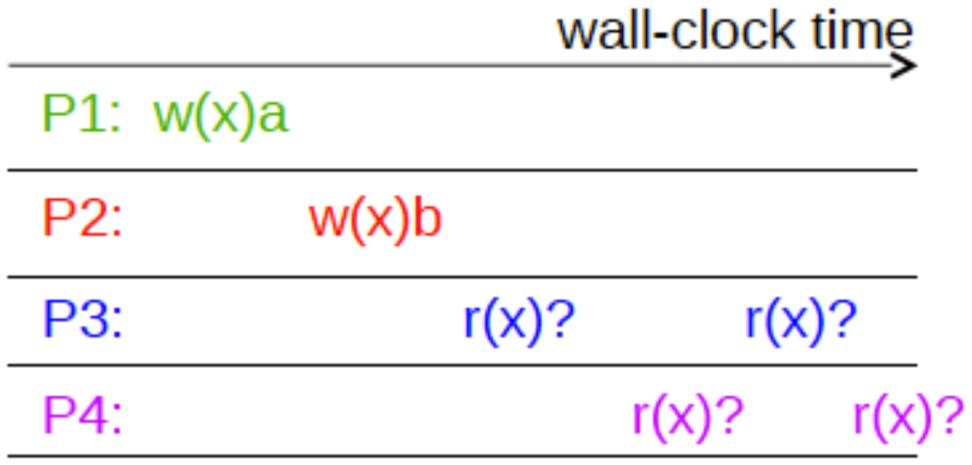
(b)

a) Consistency models not using synchronization operations.

b) Models with synchronization operations.

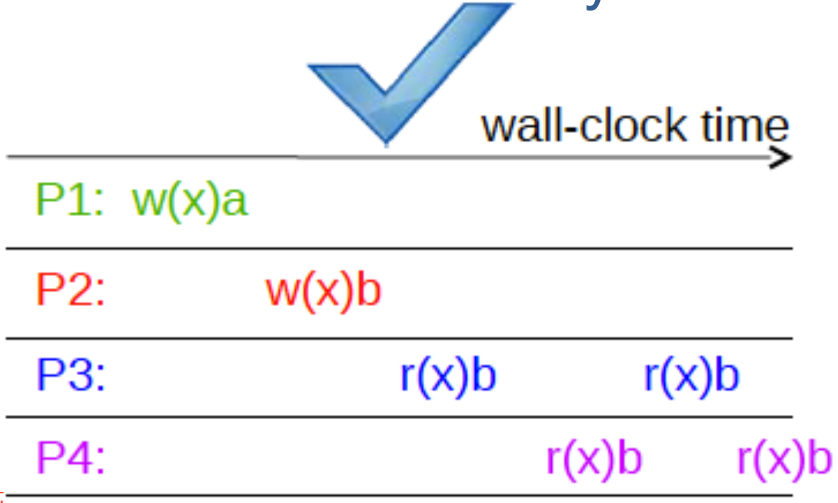
Strict Consistency

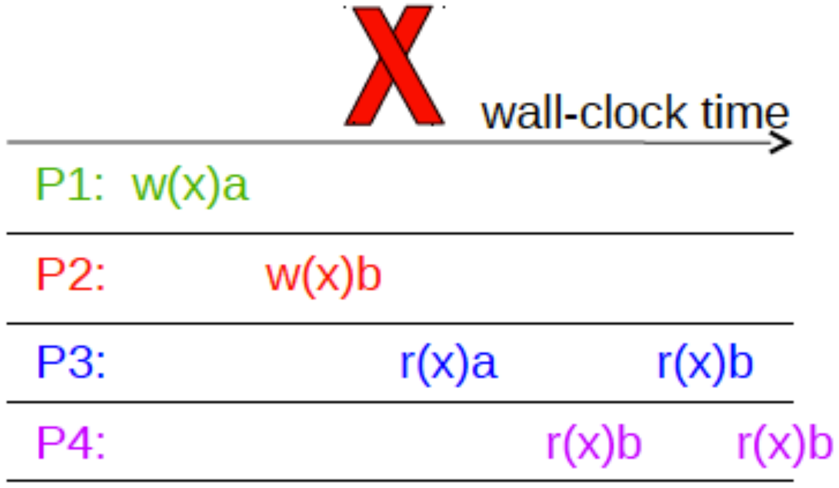
- Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued
- Therefore:
  - Reads are never stale
  - All replicas enforce wall-clock ordering for all writes
- If DFS were strictly consistent, what can these reads return?



Strict Consistency

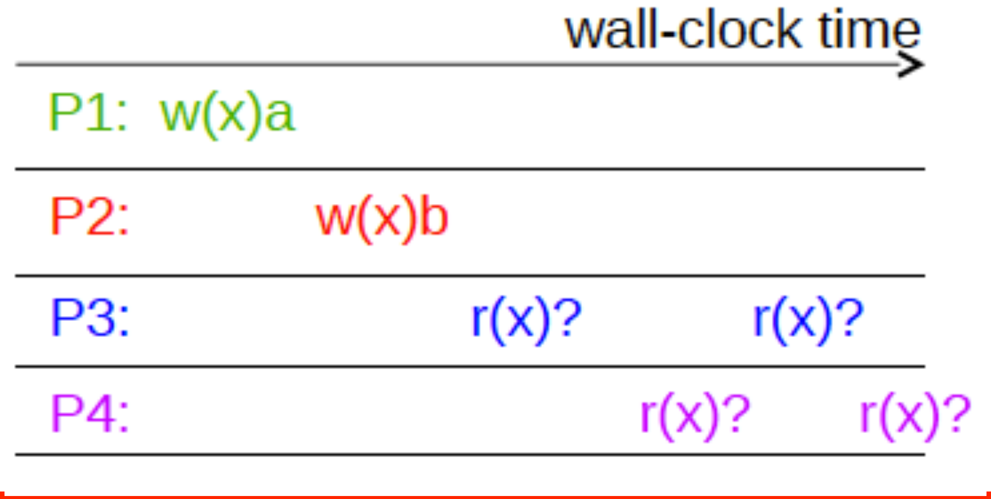
- Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued
- Therefore:
  - Reads are never stale
  - All replicas enforce wall-clock ordering for all writes
- If DFS were strictly consistent, what can these reads return?





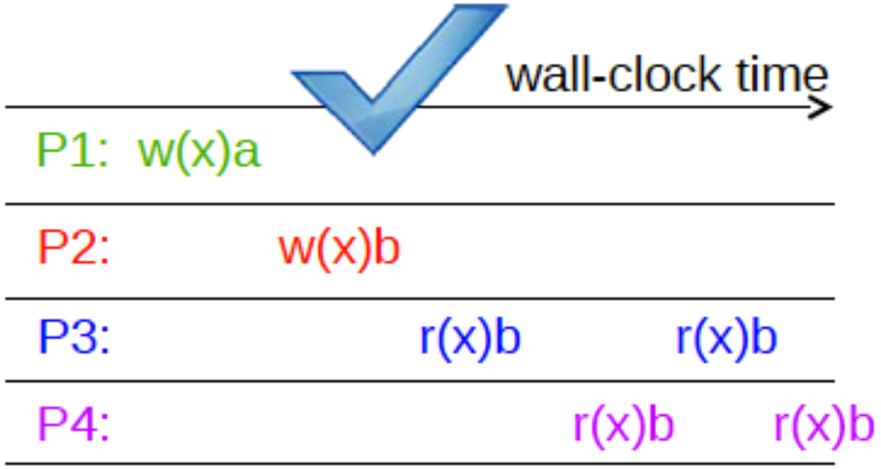
Sequential Consistency

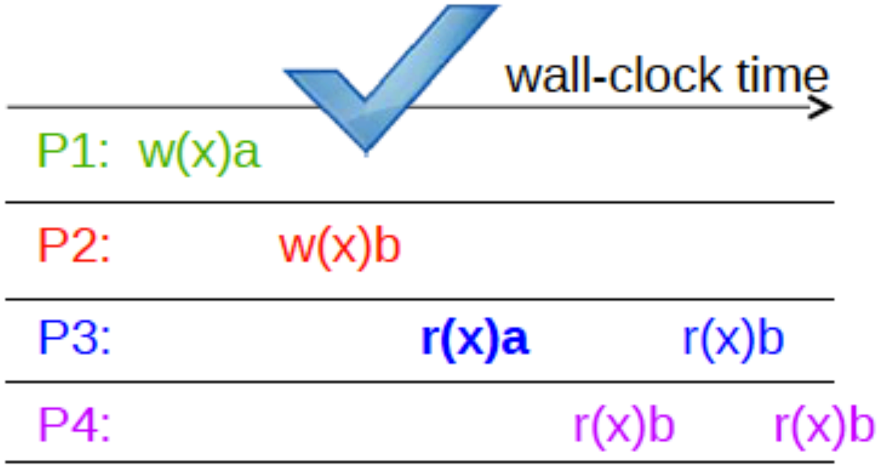
- Any execution is the same as if all read/write ops were executed in some global ordering, and the ops of each client process appear in the order specified by its program
- Therefore:
  - Reads may be stale in terms of real time, but not in logical time
  - Writes are totally ordered according to logical time across all replicas
- If DSM were seq. consistent, what can these reads return?



Sequential Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in some global ordering, and the ops of each client process appear in the order specified by its program





What's a global sequential order that can explain these results?

wall-clock ordering

This is also strictly

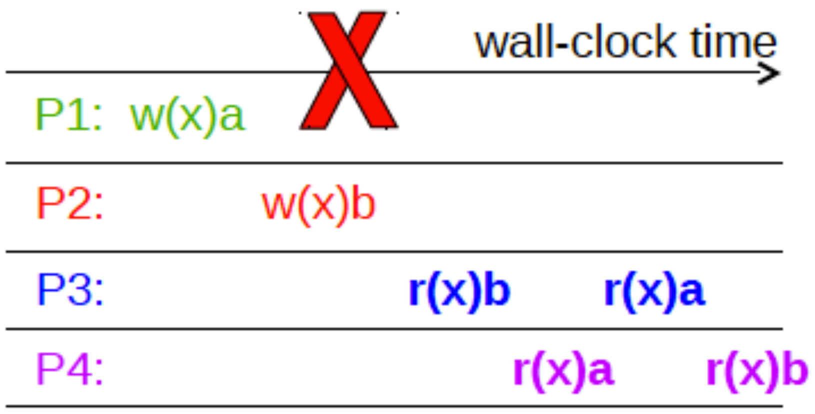
What's a global sequential order that can explain these results?

w(x)a, r(x)a, w(x)b, r(x)b, …

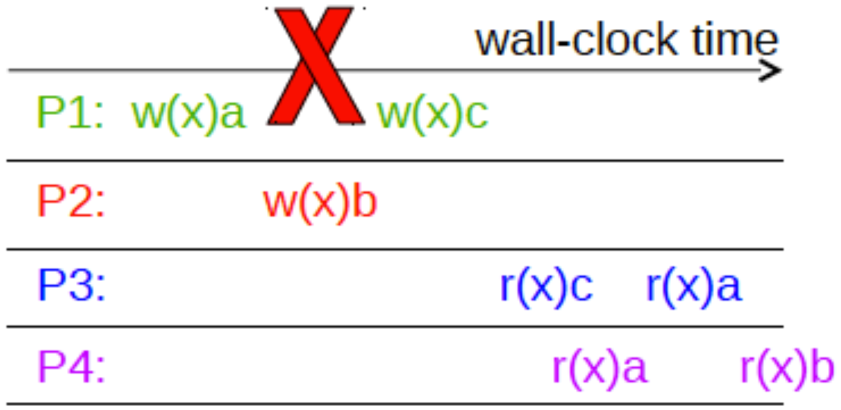
This is not strictly

Sequential Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in some global ordering, and the ops of each client process appear in the order specified by its program



No global ordering can explain these results…
=> not seq. consistent



No global *sequential* global ordering can explain these results…

E.g.: the following global ordering doesn't preserve P1's ordering
w(x)c, r(x)c, w(x)a, r(x)a, w(x)b, …

Weak Consistency

- Consider Critical Section
  - If a process is in a critical section, its intermediate results of operations are not necessarily propagated to others.
- Idea
  - Enforce consistency on a *Group of Operations*
  - Limit the time when consistency holds
  - Let programmer explicitly specify this

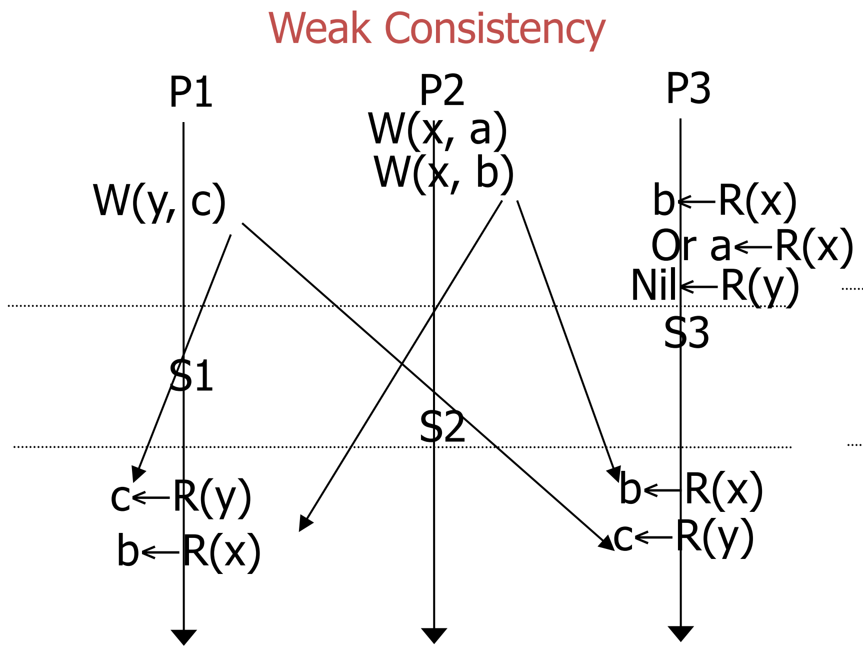
Synchronization Operations

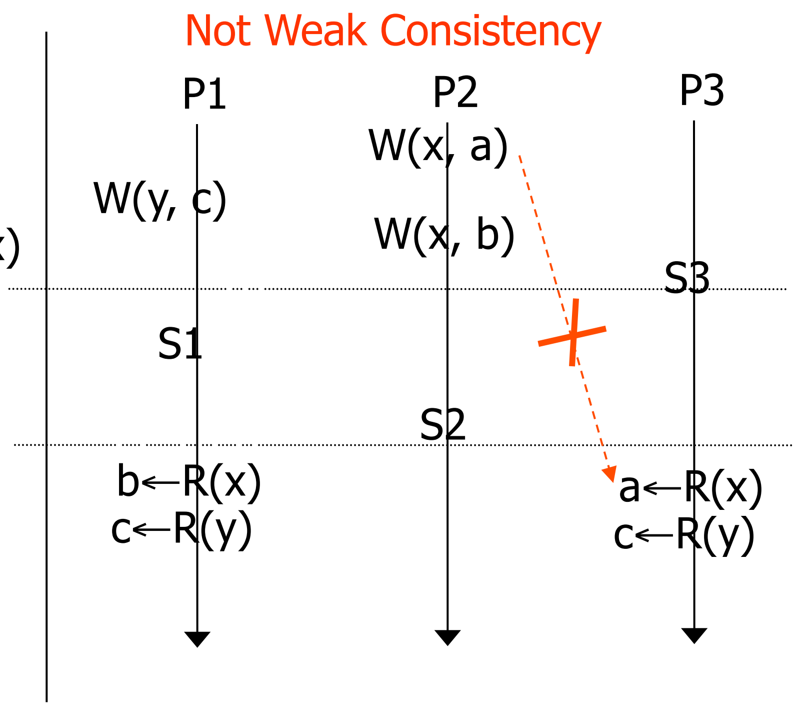
- In addition to reads and writes, introduce *synch$_p$()* operation, which
  - synchronizes all local copies of the data store
    - Propagate local updates
    - Bring in other’s updates

Weak Consistency (cont.)

- Three conditions
    1. No operation on a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
    2. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.
    3. Accesses to synchronization variables associated with a data store, are sequentially consistent.

Weak Consistency (cont.)





No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

No operation on a synchronization variable is allowed to be performed until all previous writes have completed everywhere.

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

a) Consistency models not using synchronization operations.

b) Models with synchronization operations.

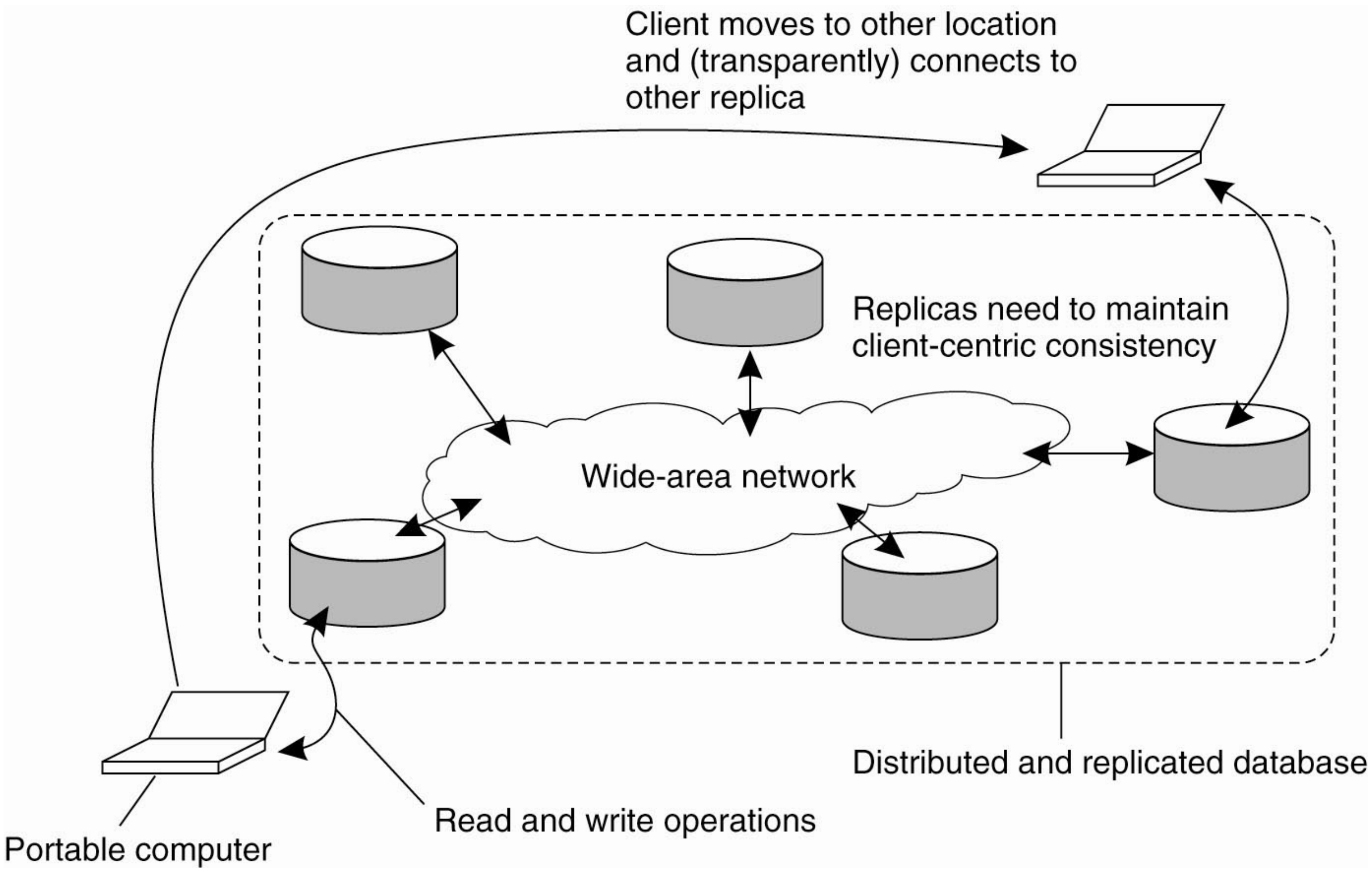
Weaker Models

- Sometimes strong models are needed, if the result of race conditions are very bad.
  - Banks
- Sometimes the result of races are just inefficiency, or inconvenience, etc.
- How strong is SkyScanner.net model?
  - If it shows that a flight ticket with a certain price is available, is it really?
- One kind of weaker model is eventual consistency
  - It eventually becomes consistent

Lazy Consistency Models

- When updates are scarce
- When updates are not conflicting
  - Examples: DNS and WWW
- **Eventual Consistency (EC)**: Lazy propagation of updates to all replicas
  - If no updates take place for a long time, all replicas will become consistent
  - Cheap to implement
  - If a client always accesses the same replica, the same or newer data will be read as time passes. EC works.

Eventual Consistency


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

- How well does EC work for mobile clients?
- Client-centric is for this. Consistent for a single client.

Eventual Consistency

- Allow stale reads, but ensure that reads will eventually reflect previously written values
  - Even after very long times
- Doesn't order concurrent writes as they are executed, which might create conflicts later: which write was first?
- Used in Amazon's Dynamo, a key/value store
  - Plus a lot of academic systems
  - Plus file synchronization

Why Eventual Consistency?

- More concurrency opportunities than strict, sequential, or causal consistency
- Sequential consistency requires highly available connections
  - Lots of chatter between clients/servers
- Sequential consistency many be unsuitable for certain scenarios:
  - Disconnected clients (e.g. your laptop goes offline, but you still want to edit your shared document)
  - Network partitioning across datacenters
  - Apps might prefer potential inconsistency to loss of availability

Session Guarantees

- When client moves around and connects to different replicas, strange things can happen
  - Update you just made was missing
  - Database goes back in time
- Design choice:
  - Insist on stricter consistency
  - Use a client-centric consistency model to enforce some “session” guarantees

- Read-your-writes Consistency
  - A value written by a process on a data item X will be always available to a successive read operation performed by the same process on data item X
- Monotonic Read Consistency
  - If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

- Monotonic Write Consistency
  - A write operation by a process on a data item X is completed before any successive write operation on X by the same process.
- Writes-follows-reads Consistency
  - A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read

REPLICA MANAGEMENT & CONSISTENCY PROTOCOLS

Replication Basics

- Data replication \ maintain multiple copies of data on separate servers/computers
  - Improves availability
  - Improves performance
  - Increased throughput
- Traditional Replication \ single-copy replication
  - Access to replicas are blocked till updates are consistent
- Optimistic Replication \ different approach to concurrency control
  - Allow data access without synchronous replication
  - Internet is slow and unreliable
  - Traditional replication algorithms scale poorly in WAN
  - Cooperative applications benefit from optimistic replication

Replication: Challenges

- Replica Server Placement
  - Selection of sites for replica servers
  - Given replica servers, where to place content
- What are the choices to propagate the changes to the replicas ?
- How to implement changes on replicas to provide consistent view ?

Replica Server Placement

- Optimization problem where K out of N servers are selected as replica site locations (K≤ N)
- Optimization criteria:
  - Distance between client and server
  - Bandwidth of the client-server links
  - Latency between clients and servers
- Dynamic identification of placement region
  - Region: collection of nodes accessing same content and internode latency is low
  - Select a node from a region as replica server

Replica placement

- Permanent Replicas
  - Initial set of replicas that comprise the data store
  - web sites have fixed pages
  - Web site mirroring
  - Databases can be distributed (shared-nothing architecture, where memory and disc are not shared)
- Server Initiated replicas
  - Copies of data store created dynamically to improve performance
    - Web hosting services replicate content near the edges of the network
    - Content distribution networks, like Akamai, maintains edge replicas
  - Server initiated replication can suffice if one can guarantee at least one replica, and not have any permanent replica
- Client initiated replica
  - Client caches

Content Distribution

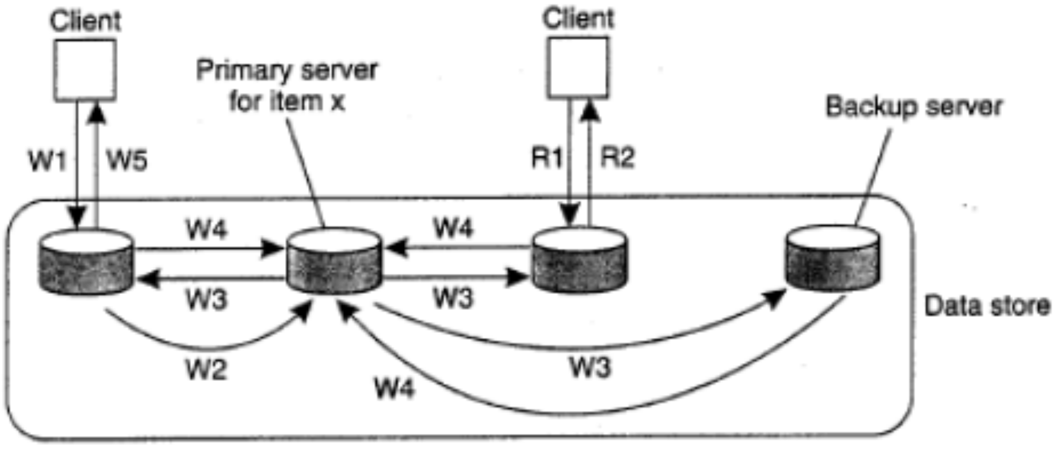
- How to propagate updates to the content to replica servers ?
  - Synchronous replication vs. asynchronous replication
  - Block other operations while replicating
    - Typical in transaction based systems, like Relational DBs
  - Content state vs. operations on content
    - State: transfer data from one copy to another
    - Operations: propagate update operation to other copies, and let the replica execute the operation to generate state
    - Can also propagate an invalidate message to mark a portion of the data that is not up to date
  - Push vs Pull
    - Server pushes the updates
    - clients or other replicas can periodically poll for changes

Implementation Issues

- How to implement a consistent view of the data across the replicas ?
  - Bounded numerical deviation
  - Bounded staleness deviation
  - Bounded ordering deviations
    - Primary based protocols
    - Quorum based protocols

Primary based Protocols

- Remote Write (Single Master):
  - All write operations need to be forwarded to a single master server
  - Read operations can be read locally
  - Easy to order incoming writes in globally consistent order
  - Performance problem \ blocking operation since updates must propagate to all replicas before progress
  - What if we use non-blocking protocol ? Fault tolerance issue

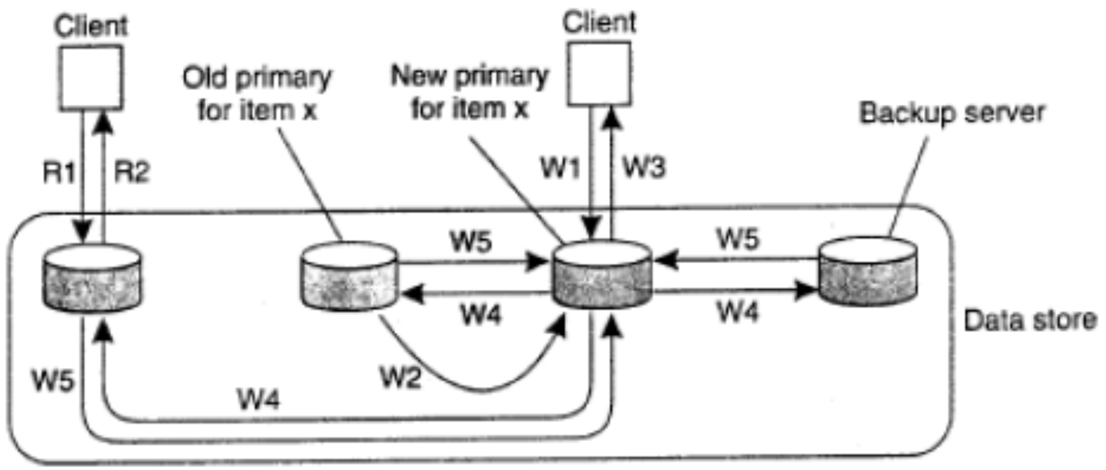


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Primary based Protocols

- Local Write (Single Master but dynamic):
  - Primary copy migrates among replicas where writes are requested
    - Non-blocking protocol followed by update propagation after primary has completed local updates
  - Can be applied to mobile computing setting
    - Note: if primary is disconnected, then other nodes can only read, but cannot write



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

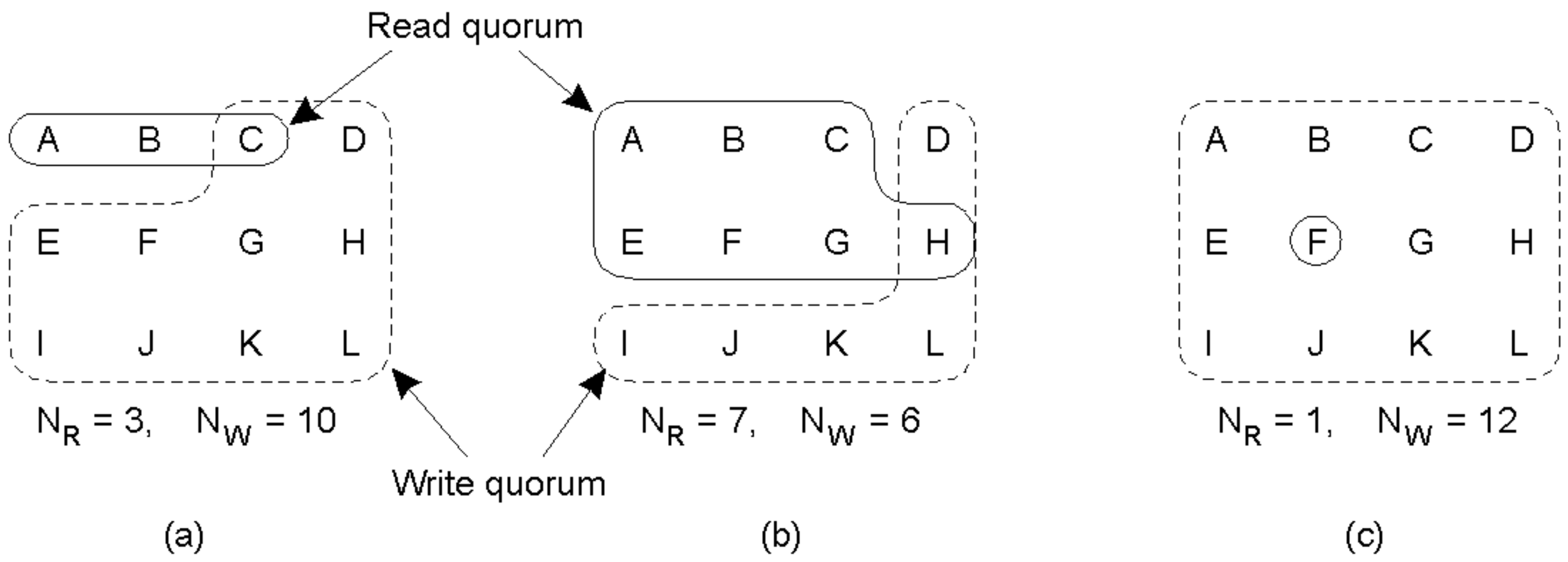
Replicated Write Protocols

- Multiple Masters: write operations can be carried out at multiple replicas \ improved concurrency
- Active Replication
  - Operations are propagated
  - Problem is that all replicas must see the same order of operations \ need a totally ordered multicast
  - Can use a sequencer \ forward all operations to a central process that assigns unique id to an operation
- Quorum-based Protocols
  - Majority Voting based technique
  - clients request and acquire permission from multiple servers before reading or writing a replicated data item

Quorum-based Protocols

- N: Total #Replicas
- N_R: #Replicas in Read Quorum
- N_W: #Replicas in Write Quorum
- Constraints:
  - $N_R + N_W > N$
  - $N_W > N/2$

Quorum-Based Protocols



Three examples of the voting algorithm for N = 12 replicas

(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)

Possible Policies

- ROWAL R=1, W=N
  - Fast reads, slow writes
- Majority: R=W=N/2+1
  - Both moderately slow, but extremely high availability

Implementation: Client-centric consistency

- Note: Conflict detection and conflict resolution is an additional challenge
- Naïve approach:
  - Each write op W is assigned a globally unique id
  - For each client, keep track of two sets of writes
    - Read set: writes relevant to the read operation
    - Write set: writes performed by a client

Implementation: Client-centric consistency

- Monotonic Read consistency
  - On receiving read request, server gets the read set to check all writes have been updated
- Monotonic Write consistency
  - Get the write set; execute all write operations before the write request
- Read your write consistency
  - Ensure that server where read request reaches has seen all preceding write operations (write set)
- Write follows read consistency
  - Bring selected server up to date with write ops in clients read set

Problem: Read and write sets can become large, and transferring across replicas will be costly Solution: Maintain sessions in applications; read and write sets are maintained only within sessions Use efficient ways of representing the set information -> vector timestamps

Distribution Protocols

- How are updates propagated to replicas (independent of the consistency model)?
  - State versus operations
    - Propagate only notification of update, i.e., invalidation
    - Transfer data from one copy to another
    - Propagate the update *operation* to other copies
  - Push versus pull protocols

Pull versus Push Protocols

A comparison between push-based and pull-based protocols in the case of multiple backup replica, single primary replica.

Issue	Push-based	Pull-based
State of primary	List of backups and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at backup	Immediate (or fetch-update time)	Fetch-update time

Leases: a hybrid form of update propagation that dynamically switches between pushing and pulling

- Primary maintains state for a backup for a TTL, i.e., while lease has not expired

Leases

- Combined push and pull
- A primary replica promises
  - push updates for a certain time
  - a lease expires => the backup
    - polls the primary or requests a new lease
  - length of a lease?
  - different types of leases
    - age based: {time to last modification}
    - renewal-frequency based: long-lasting leases to active users
    - state-space overhead: increasing utilization of a primary => lower expiration times of new leases

CAP THEOREM

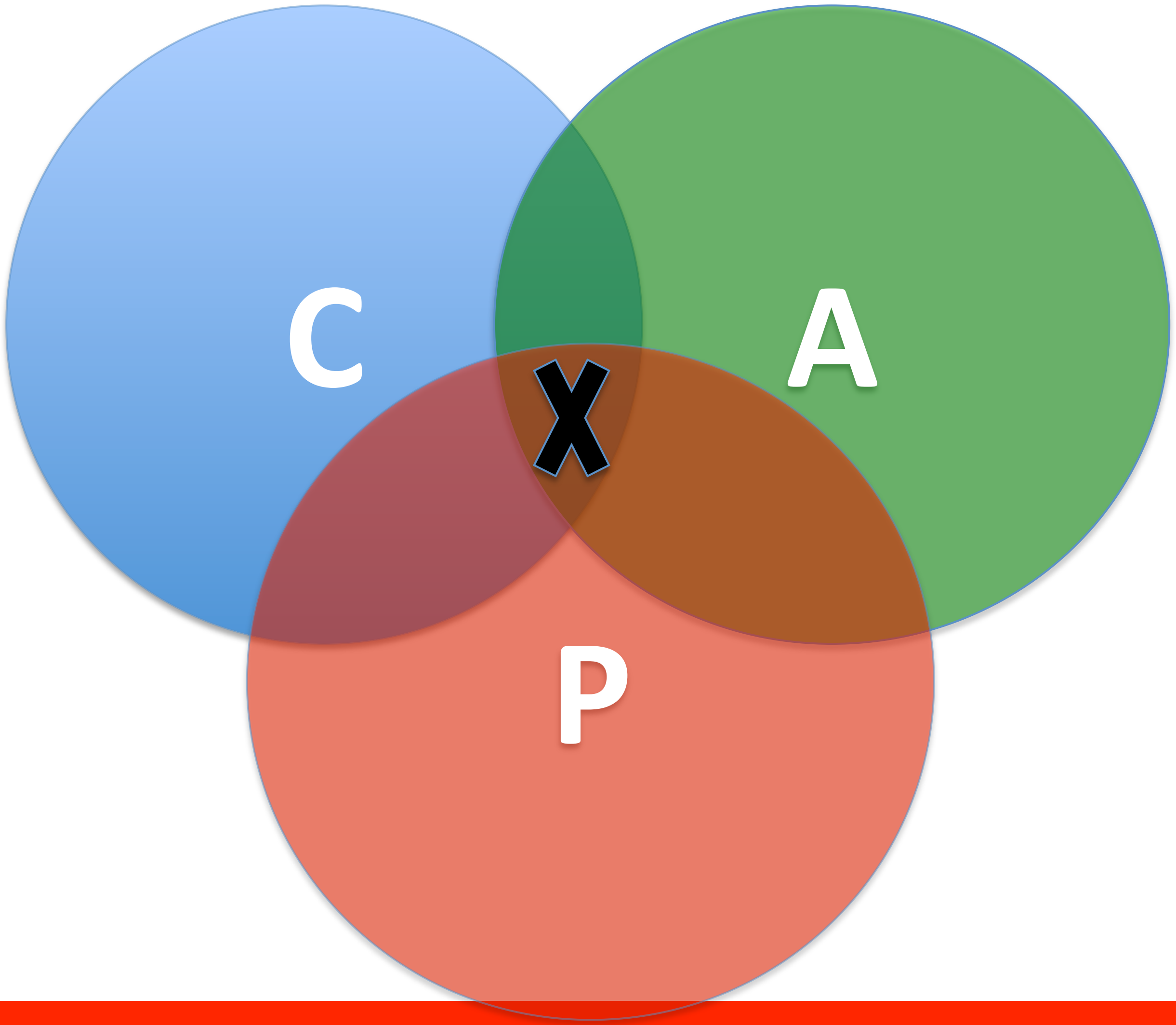
CAP Theorem

- Conjectured by Prof. Eric Brewer at PODC (Principle of Distributed Computing) 2000 keynote talk
- Described the *trade-offs involved in distributed system*
- It is impossible for a web service to provide following *three guarantees at the same time*:
    - **Consistency**
    - **Availability**
    - **Partition-tolerance**

CAP Theorem

- **C**onsistency:
  - All nodes should see the same data at the same time
- **A**vailability:
  - Node failures do not prevent survivors from continuing to operate
- **P**artition-tolerance:
  - The system continues to operate despite network partitions
- A distributed system can satisfy any two of these guarantees at the same time **but not all three**

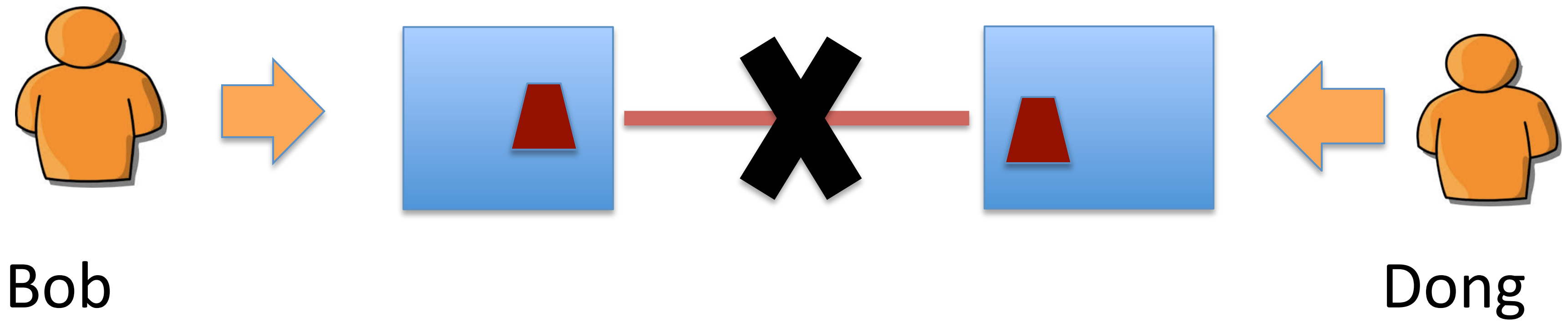
CAP Theorem



CAP Theorem

- A simple example:

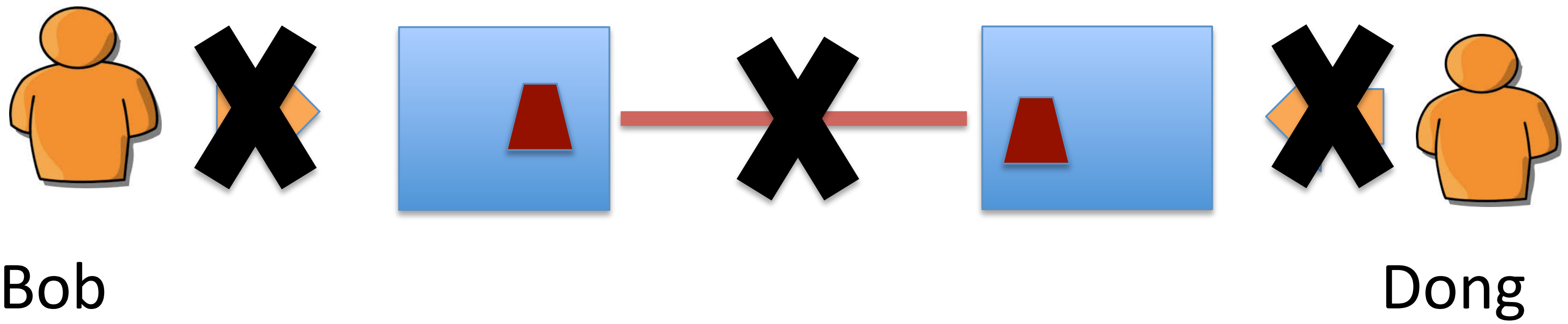
Hotel Booking: are we double-booking the same room?



CAP Theorem

- A simple example:

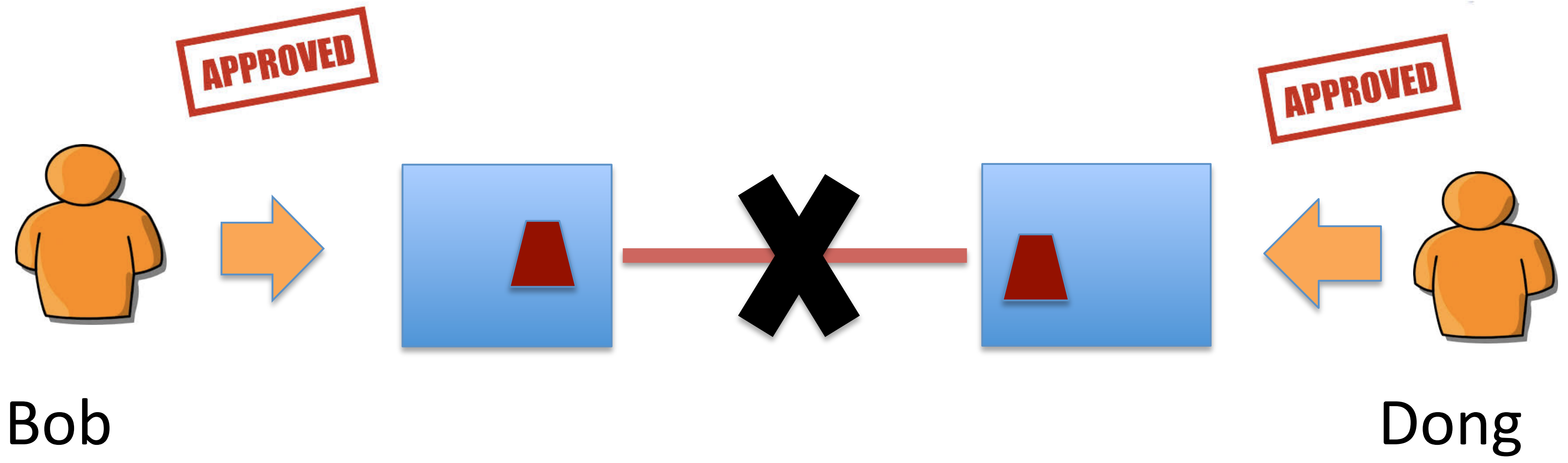
Hotel Booking: are we double-booking the same room?



CAP Theorem

- A simple example:

Hotel Booking: are we double-booking the same room?

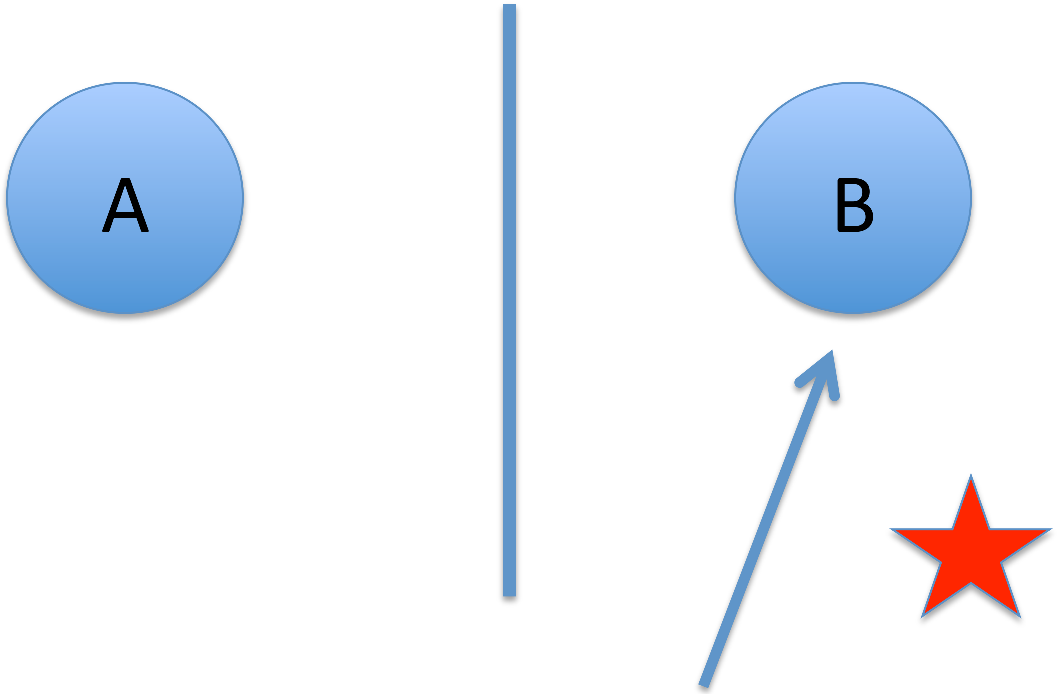


CAP simplified

- The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C.
- Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A.
- Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P.

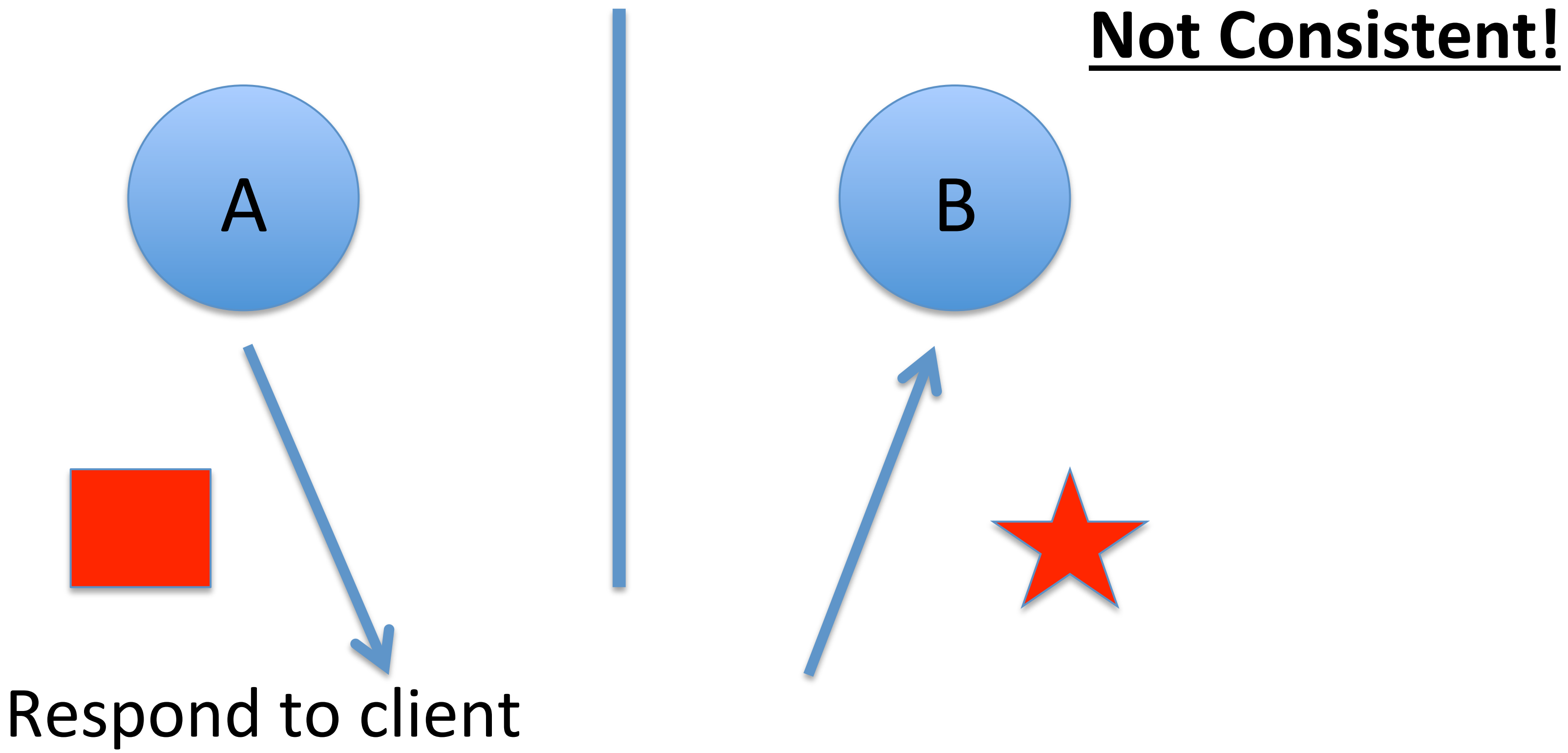
CAP Theorem: Proof

- A simple proof using two nodes:



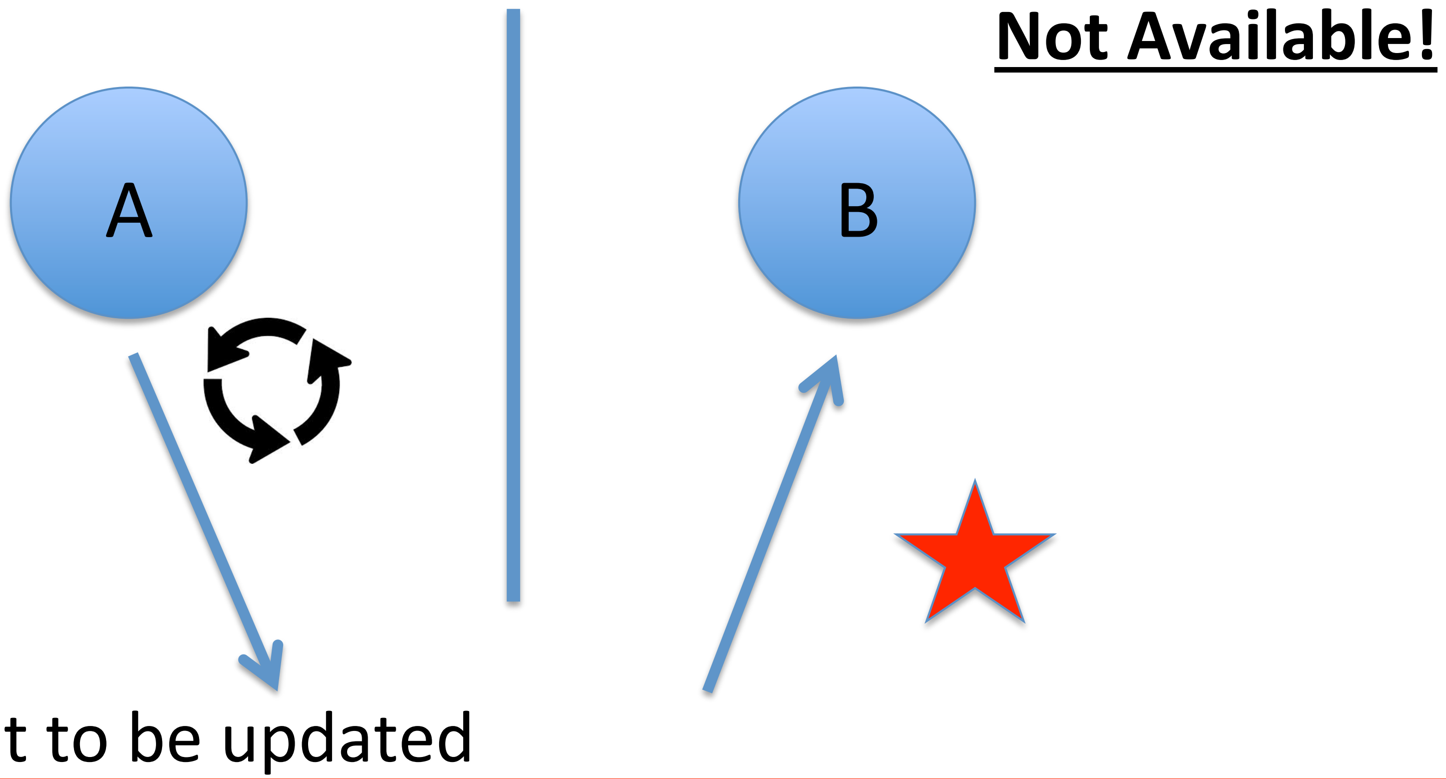
CAP Theorem: Proof

- A simple proof using two nodes:



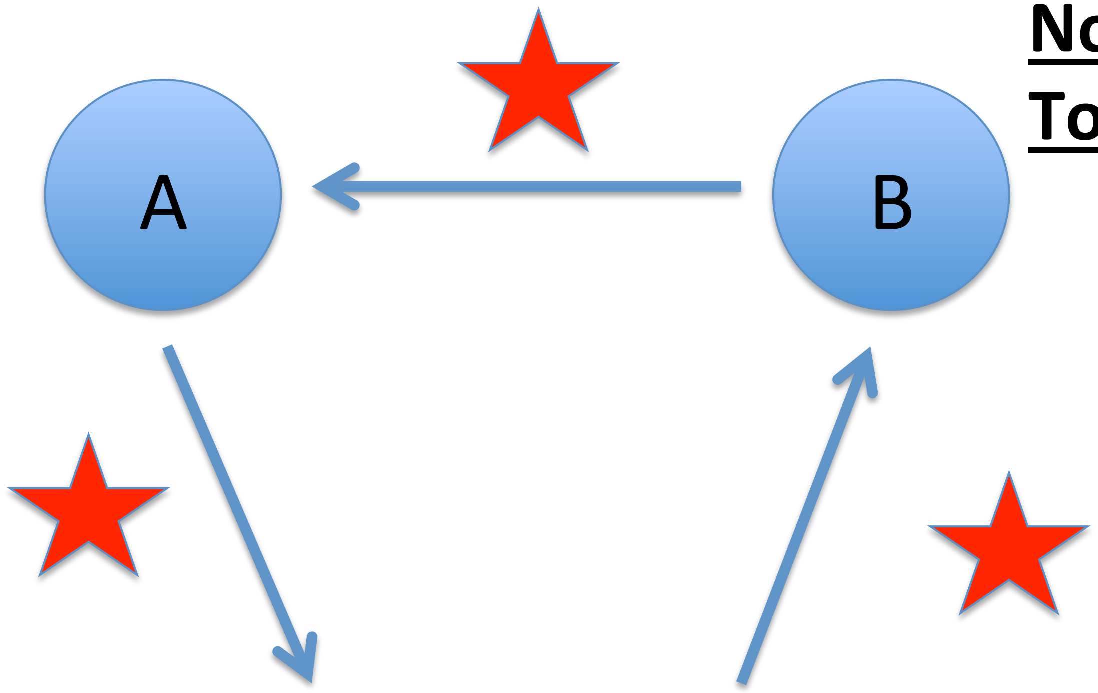
CAP Theorem: Proof

- A simple proof using two nodes:



CAP Theorem: Proof

- A simple proof using two nodes:



Not Partition Tolerant!

A gets updated from B

Why this is important?

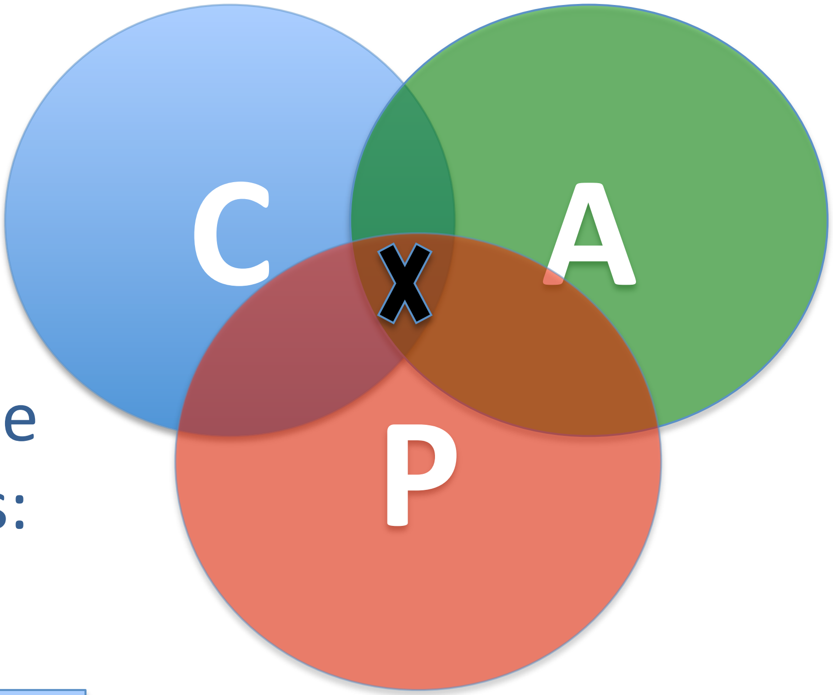
- The future of databases is **distributed** (Big Data Trend, etc.)
- CAP theorem describes the **trade-offs** involved in distributed systems
- A proper understanding of CAP theorem is essential to **making decisions** about the future of distributed database **design**
- Misunderstanding can lead to **erroneous or inappropriate** design choices

Problem for Relational Database to Scale

- The Relational Database is built on the principle of **ACID** (Atomicity, Consistency, Isolation, Durability)
- It implies that a truly distributed relational database should have **availability, consistency and partition tolerance**.
- Which unfortunately is **impossible** …

Revisit CAP Theorem

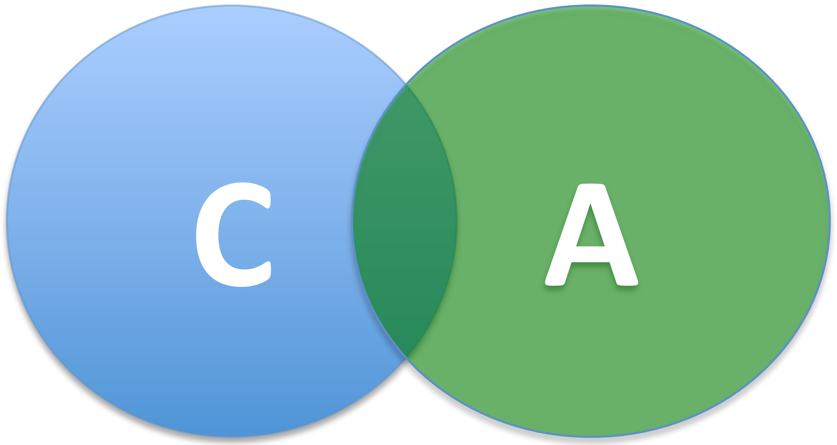
- Of the following three guarantees potentially offered a by distributed systems:
  - Consistency
  - Availability
  - Partition tolerance
- Pick two
- This suggests there are three kinds of distributed systems:
  - CP
  - AP
  - CA



Any problems?

A popular misconception: 2 out 3

- How about CA?
- Can a distributed system (with unreliable network) really be not tolerant of partitions?



A few witnesses

- Coda Hale, Yammer software engineer:
  - "Of the CAP theorem's Consistency, Availability, and Partition Tolerance, **Partition Tolerance is mandatory in distributed systems**. You cannot not choose it."

http://codahale.com/you-cant-sacrifice-partition-tolerance/

A few witnesses

- Werner Vogels, Amazon CTO
  - "An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, **consistency and availability cannot be achieved at the same time.**"

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

A few witnesses

- Daneil Abadi, Co-founder of Hadapt
  - So in reality, there are only two types of systems ... I.e., if there is a partition, **does the system give up availability or consistency?**

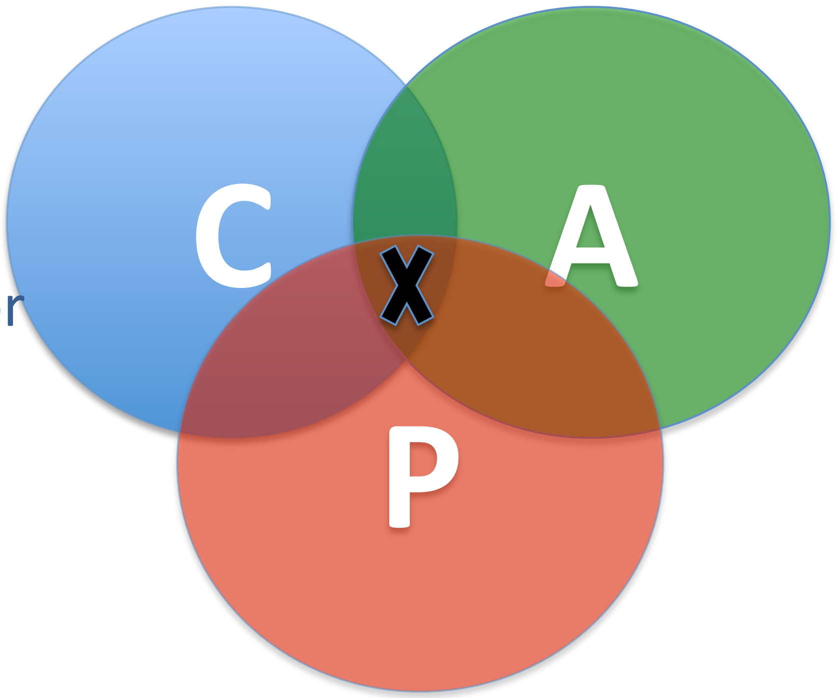
http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

CAP Theorem 12 year later

- Prof. Eric Brewer: father of CAP theorem
  - "The "2 of 3" formulation was always **misleading** because it tended to oversimplify the tensions among properties. ...
  - **CAP prohibits only a tiny part of the design space**: *perfect availability and consistency in the presence of partitions*, which are rare."

Consistency or Availability

- Consistency and Availability is not "binary" decision
- AP systems relax consistency in favor of availability – but are not inconsistent
- CP systems sacrifice availability for consistency- but are not unavailable
- This suggests both AP and CP systems can offer a degree of consistency, and availability, as well as partition tolerance



AP: Best Effort Consistency

- Example:
  - Web Caching
  - DNS
- Trait:
  - Optimistic
  - Expiration/Time-to-live
  - Conflict resolution

CP: Best Effort Availability

- Example:
  - Majority protocols
  - Distributed Locking (Google Chubby Lock service)
- Trait:
  - Pessimistic locking
  - Make minority partition unavailable

Types of Consistency

- Strong Consistency
  - After the update completes, **any subsequent access** will return the **same** updated value.
- Weak Consistency
  - It is **not guaranteed** that subsequent accesses will return the updated value.
- **Eventual Consistency**
  - Specific form of weak consistency
  - It is guaranteed that if **no new updates** are made to object, **eventually** all accesses will return the last updated value (e.g., *propagate updates to replicas in a lazy fashion*)

Eventual Consistency Variations

- Causal consistency
  - Processes that have causal relationship will see consistent data
- Read-your-write consistency
  - A process always accesses the data item after it's update operation and never sees an older value
- Session consistency
  - As long as session exists, system guarantees read-your-write consistency
  - Guarantees do not overlap sessions

Eventual Consistency Variations

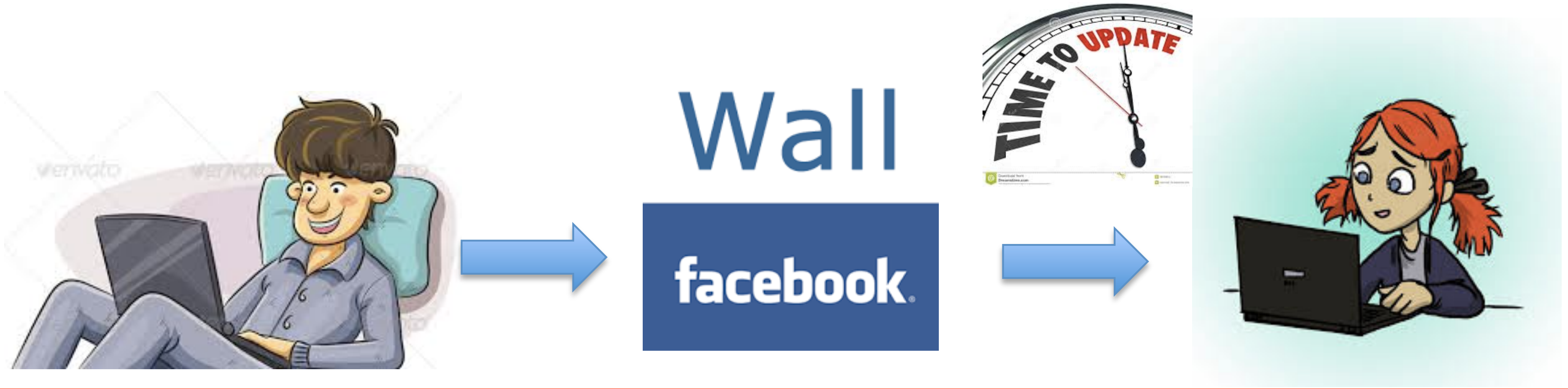
- Monotonic read consistency
  - If a process has seen a particular value of data item, any subsequent processes will never return any previous values
- Monotonic write consistency
  - The system guarantees to serialize the writes by the *same* process
- In practice
  - A number of these properties can be combined
  - Monotonic reads and read-your-writes are most desirable

Eventual Consistency: A Facebook Example

- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but finds:
  - **Nothing is there!**

Eventual Consistency: A Facebook Example

- Bob tells Alice to wait a bit and check out later
- Alice waits for a minute or so and checks back:
  - **She finds the story Bob shared with her!**



Eventual Consistency: A Facebook Example

- Reason: it is possible because Facebook uses an **eventual consistent model**
- Why Facebook chooses eventual consistent model over the strong consistent one?
  - Facebook has more than 1 billion active users
  - It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
  - Eventual consistent model offers the option to **reduce the load and improve availability**

Eventual Consistency: A Dropbox Example

- Dropbox enabled immediate consistency via synchronization in many cases.
- However, what happens in case of a network partition?

Eventual Consistency: A Dropbox Example

- Let's do a simple experiment here:
  - Open a file in your drop box
  - Disable your network connection (e.g., WiFi, 4G)
  - Try to edit the file in the drop box: can you do that?
  - Re-enable your network connection: what happens to your dropbox folder?

Eventual Consistency: A Dropbox Example

- Dropbox embraces eventual consistency:
  - Immediate consistency is impossible in case of a network partition
  - Users will feel bad if their word documents freeze each time they hit Ctrl+S , simply due to the large latency to update all devices across WAN
  - Dropbox is oriented to **personal syncing**, not on collaboration, so it is not a real limitation.

Eventual Consistency: An ATM Example

- In design of automated teller machine (ATM):
  - Strong consistency appear to be a nature choice
  - However, in practice, **A beats C**
  - Higher availability means **higher revenue**
  - ATM will allow you to withdraw money *even if the machine is partitioned from the network*
  - However, it puts **a limit** on the amount of withdraw (e.g., $200)
  - The bank might also charge you a fee when a overdraft happens

Dynamic Tradeoff between **C** and **A**

- An airline reservation system:
  - **When most of seats are available:** it is ok to rely on somewhat out-of-date data, availability is more critical
  - **When the plane is close to be filled:** it needs more accurate data to ensure the plane is not overbooked, consistency is more critical
- Neither strong consistency nor guaranteed availability, but it may significantly increase the tolerance of network disruption

Heterogeneity: Segmenting C and A

- No single uniform requirement
  - Some aspects require strong consistency
  - Others require high availability
- Segment the system into different components
  - Each provides different types of guarantees
- Overall guarantees neither consistency nor availability
  - Each part of the service gets exactly what it needs

- Can be partitioned along different dimensions

Discussion

- In an e-commercial system (e.g., Amazon, e-Bay), what are the trade-offs between consistency and availability you can think of? What is your strategy?
- Hint -> Things you might want to consider:
  - Different types of data (e.g., shopping cart, billing, product, etc.)
  - Different types of operations (e.g., query, purchase, etc.)
  - Different types of services (e.g., distributed lock, DNS, etc.)
  - Different groups of users (e.g., users in different geographic areas, etc.)

Partitioning Examples

- Data Partitioning
- Operational Partitioning
- Functional Partitioning
- User Partitioning
- Hierarchical Partitioning

Data Partitioning

- Different data may require different consistency and availability
- Example:
  - Shopping cart: high availability, responsive, can sometimes suffer anomalies
  - Product information need to be available, slight variation in inventory is sufferable
  - Checkout, billing, shipping records must be consistent

Operational Partitioning

- Each operation may require different balance between consistency and availability
- Example:
  - Reads: high availability; e.g.., “query”
  - Writes: high consistency, lock when writing; e.g., “purchase”

Functional Partitioning

- System consists of sub-services
- Different sub-services provide different balances
- Example: A comprehensive distributed system
  - Distributed lock service (e.g., Chubby) :
    - Strong consistency
  - DNS service:
    - High availability

User Partitioning

- Try to keep related data close together to assure better performance
- Example: Craglist
  - Might want to divide its service into several data centers, e.g., east coast and west coast
    - Users get high performance (e.g., high availability and good consistency) if they query servers closet to them
    - Poorer performance if a New York user query Craglist in San Francisco

Partitioning Examples

Hierarchical Partitioning

- Large global service with local “extensions”
- Different location in hierarchy may use different consistency
- Example:
  - Local servers (better connected) guarantee more consistency and availability
  - Global servers has more partition and relax one of the requirement

What if there are no partitions?

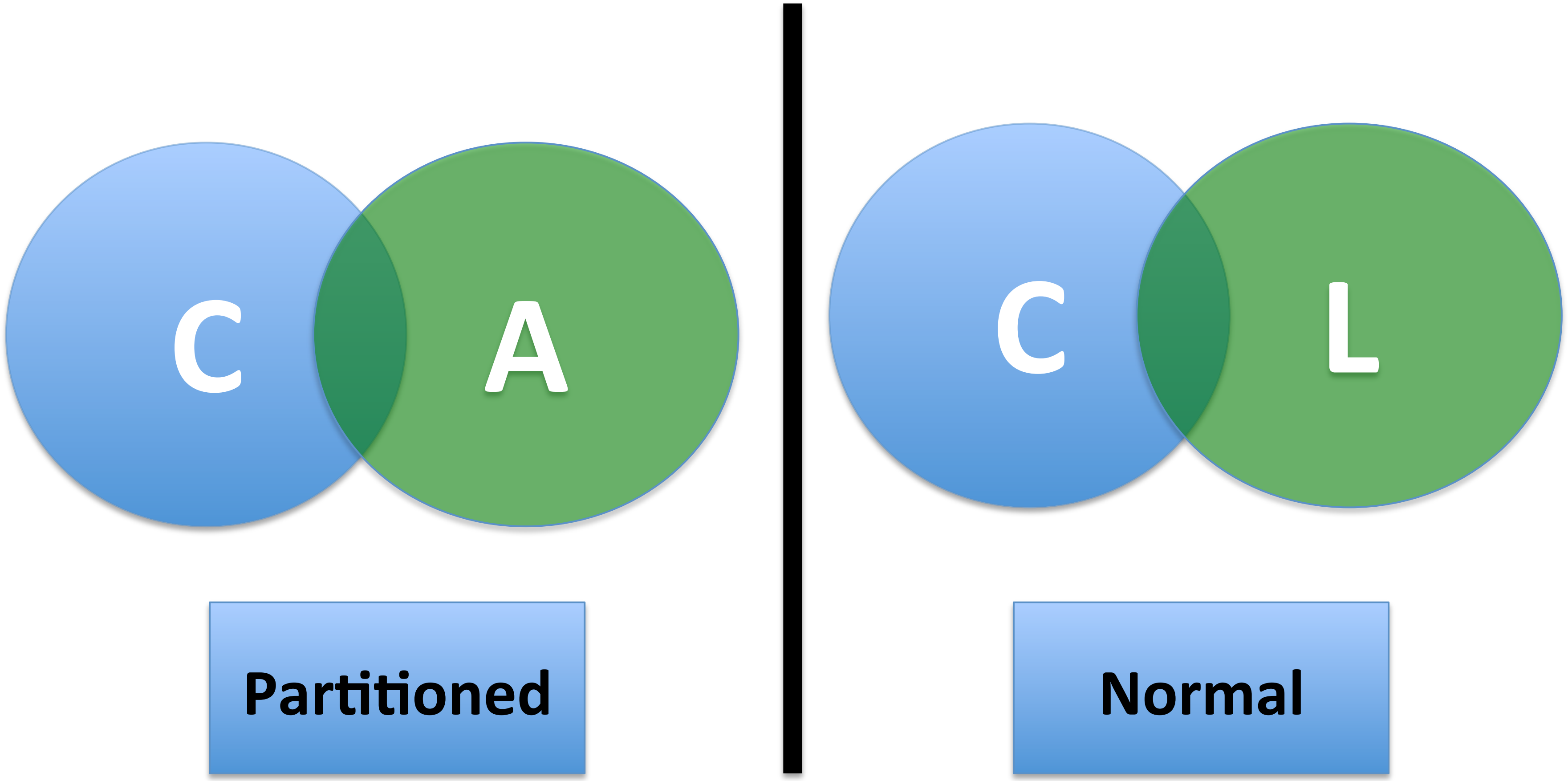
- Tradeoff between **Consistency** and **Latency**:
- Caused by the **possibility of failure** in distributed systems
  - High availability -> replicate data -> consistency problem
- Basic idea:
  - Availability and latency are arguably **the same thing**: unavailable -> extreme high latency
  - Achieving different levels of consistency/ availability takes different amount of time

CAP -> PACELC

- A more complete description of the space of potential tradeoffs for distributed system:
  - If there is a **partition (P)**, how does the system trade off **availability and consistency (A and C)**; **else (E)**, when the system is running normally in the absence of partitions, how does the system trade off **latency (L) and consistency (C)**?

Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." Computer-IEEE Computer Magazine 45.2 (2012): 37.

PACELC



Examples

- **PA/EL Systems:** Give up both Cs for availability and lower latency
  - Dynamo, Cassandra, Riak
- **PC/EC Systems:** Refuse to give up consistency and pay the cost of availability and latency
  - BigTable, Hbase, VoltDB/H-Store
- **PA/EC Systems:** Give up consistency when a partition happens and keep consistency in normal operations
  - MongoDB
- **PC/EL System:** Keep consistency if a partition occurs but gives up consistency for latency in normal operations
  - Yahoo! PNUTS

References

- lecture notes are based on slides by
  - Prof. Jalal Y. Kawash at Univ. of Calgary
  - Prof. Kenneth Chiu at SUNY Binghamton
  - Prof. Sanjeev Setia at George Mason University
  - Prof. Dong Wang at University of Notre Dame

Thanks for your attention!