CAP Theorem, Technologies for Scalable Distribution, AWS Dynamo

CS4230

Jay Urbain, Ph.D.

Credits:

"**Dynamo: Amazon's Highly Available Key-value Store**,"

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels.

Willet

"Of course I'm insecure. I store vital computer information in a cloud!"

CAP Theorem

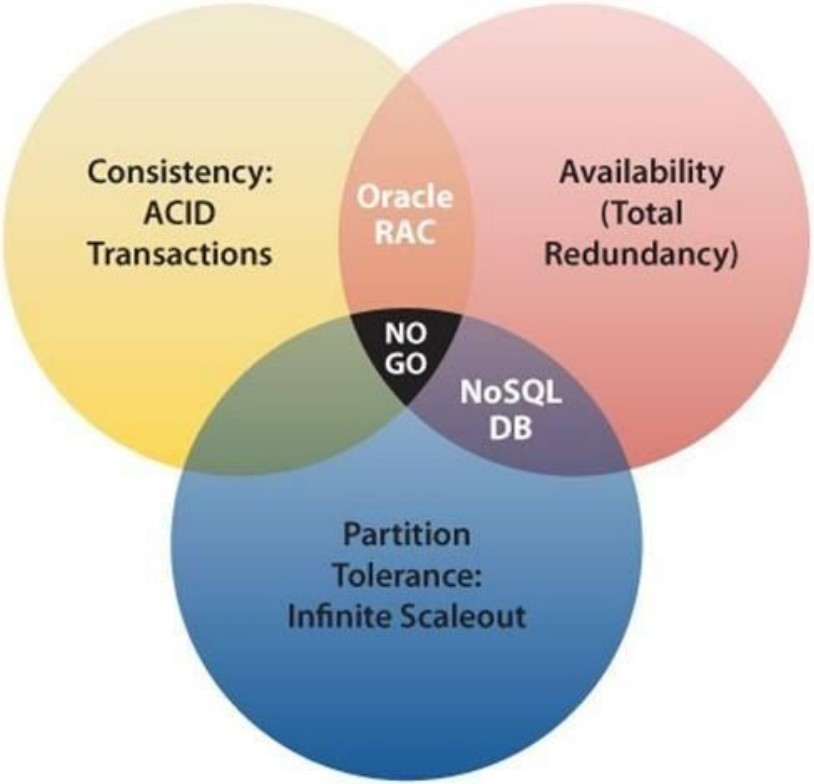
- In 2000, Eric Brewer outlined the CAP (Brewer's) Theorem.
- The CAP theorem states that in a distributed database system, you can only have *at most two of what three characteristics*?

CAP Theorem

- In 2000, Eric Brewer outlined the CAP (Brewer's) Theorem.
- The CAP theorem states that in a distributed database system, you can only have *at most two of the following three characteristics*:
  - Consistency: All nodes in the (computing) cluster see exactly the same data at any point in time.
  - Availability: Failure of a node does not render the database inoperative.
  - Partition tolerance: Nodes can still function when communication with other groups of nodes is lost.

CAP Theorem, NoSQL

- Interpretation and implementations of CAP theorem vary, but most of the NoSQL database systems favor *partition tolerance and availability over strong consistency.*



NoSQL???

SQL vs NoSQL Database Comparison

	Relational database	NoSQL database
Data model		
ACID properties		

SQL vs NoSQL Database Comparison

	Relational database	NoSQL database
Data model	The relational model normalizes data into tabular structures known as tables, which consist of rows and columns. A schema strictly defines the tables, columns, indexes, relationships between tables, and other database elements.	Non-relational (NoSQL) databases typically do not enforce a schema. A partition key is generally used to retrieve values, column sets, or semi-structured JSON, XML, or other documents containing related item attributes.
ACID properties	Traditional relational database management systems (RDBMS) support a set of properties defined by the acronym ACID: Atomicity, Consistency, Isolation, and Durability. **Atomicity** means "all or nothing" – a transaction executes completely or not at all. **Consistency** means once a transaction has been committed, the data must conform to the database schema. **Isolation** requires that concurrent transactions execute separately from one another. **Durability** is the ability to recover from an unexpected system failure or power outage to the last known state.	NoSQL databases often trade some ACID properties of traditional relational database management systems (RDBMS) for a more flexible data model that scales horizontally. These characteristics make NoSQL databases an excellent choice in situations where traditional RDBMS encounter architectural challenges to overcome some combination of performance bottlenecks, scalability, operational complexity, and increasing administration and support costs.

SQL vs NoSQL Database Comparison

	Relational database	NoSQL database
Performance	Performance is generally dependent on the disk subsystem. Optimization of queries, indexes, and table structure is required to achieve peak performance.	Performance is generally a function of the underlying hardware cluster size, network latency, and the calling application.
Scale	Easiest to scale "up" with faster hardware. Additional investments are required for relational tables to span a distributed system.	Designed to scale "out" using distributed clusters of low-cost hardware to increase throughput without increasing latency.
APIs	Requests to store and retrieve data are communicated using queries which conform to a structured query language (SQL). These queries are parsed and executed by relational database management systems (RDBMS).	Object-based APIs allow app developers to easily store and retrieve in-memory data structures. Partition keys let apps look up key-value pairs, column sets, or semi-structured documents containing serialized app objects and attributes.
Tools	SQL databases generally offer a rich set of tools for simplifying the development of database-driven applications.	NoSQL databases generally offer tools to manage clusters and scaling. Applications are the primary interface to the underlying data.

https://aws.amazon.com/nosql/?sc_channel=PS&sc_campaign=pac_ps_q4&sc_publisher=google&sc_medium=dynamodb_hv_b_pac_q42017&sc_content=sitelink&sc_detail=dynamodb&sc_category=dynamodb&sc_segment=webp&sc_matchtype=e&sc_country=US&sc_geo=namer&sc_outcome=pac&s_kwcid=AL!4422!3!224596745488!e!!g!!dynamodb&ef_id=WM1YAAAAAHwO1Q7Z:20171022182636:s

NoSQL Summary

- Use to mean “not SQL.”
- Now it means “not only SQL” or “non-relational” database.
- Use non-tabular approach for storing/retrieving data.
- Utilize a variety of data models, including document, graph, key-value, and columnar.
- NoSQL databases can store and retrieve data in various ways such as key-value, document-based, column-family, and graph databases.
- Designed to handle large volumes of unstructured or semi-structured data that may not fit well into a traditional relational database schema.
- Often used in modern web applications, big data analytics, and other use cases where scalability, performance, and flexibility are important.
- Some examples of popular NoSQL databases include MongoDB, Cassandra, Couchbase, and Redis.

SQL vs NoSQL Terminology

SQL	MongoDB	DynamoDB	Cassandra	Couchbase
Table	Collection	Table	Table	Data Bucket
Row	Document	Item	Row	Document
Column	Field	Attribute	Column	Field
Primary Key	ObjectId	Primary Key	Primary Key	Document ID
Index	Index	Secondary Index	Index	Index
View	View	Global Secondary Index	Materialized View	View
Nested Table or Object	Embedded Document	Map	Map	Map
Array	Array	List	List	List

Introduction - Amazon

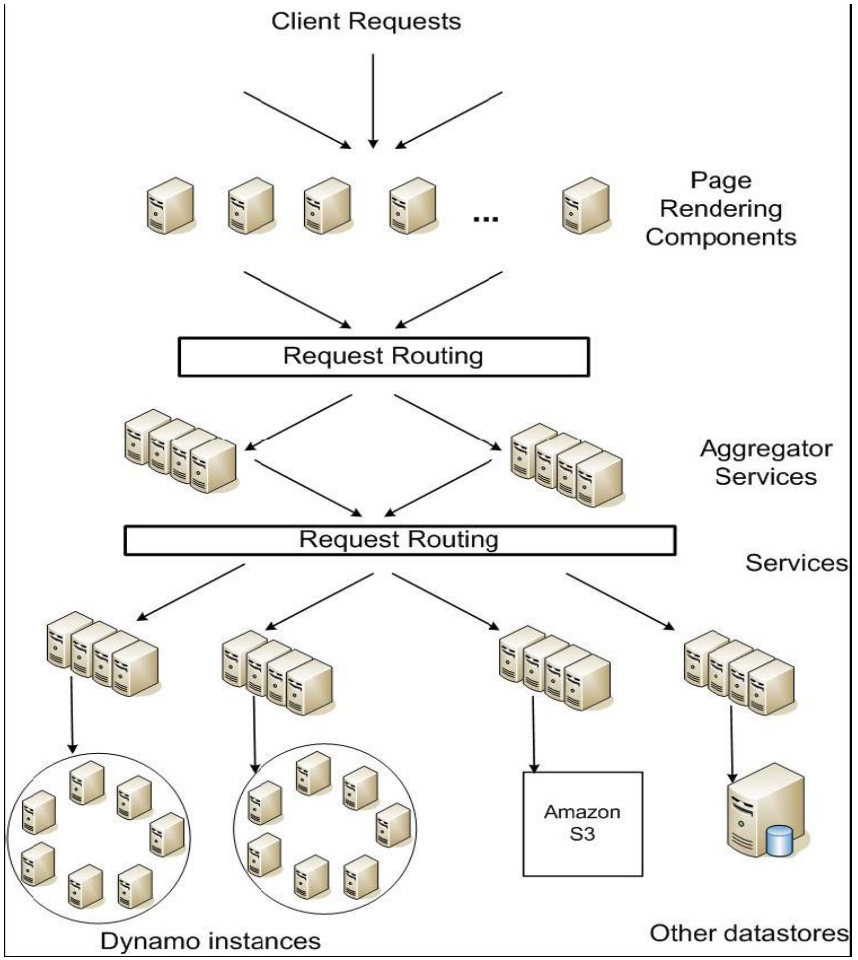
- Amazon runs a world-wide e-commerce platform that serves *hundreds of millions of customers* at peak times using *hundreds of thousands of servers* located in *many data centers around the world* (availability zones!).
- Operational requirements ???

Introduction - Amazon

- Amazon runs a world-wide e-commerce platform that serves *hundreds of millions of customers* at peak times using *hundreds of thousands of servers* located in *many data centers around the world* (availability zones!).
- Operational requirements:
  - Performance
  - Reliability
  - Efficiency
  - Scalability for continuous growth: horizontal scalability

Services provided through loosely coupled distributed architecture

- Application can deliver its functionality in bounded time *if*:
  - Every dependency in the platform delivers its functionality with even tighter bounds.
  - Like factory cycle time.
- Example:
  - Service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second. (Tail Latency).



Service-oriented architecture of Amazon's platform

System Architecture

In addition to the actual data persistence component, the system needs scalable and robust solutions for:

- Load balancing
- Failure recovery
- Replica synchronization
- Overload handling, auto-scaling
- State transfer
- Concurrency and job scheduling
- Request marshalling
- Request routing
- System monitor and alarming
- Configuration management

Meeting reliability & scalability needs

- Need to **provide control over tradeoffs** for diverse set of applications using multiple storage technologies:
  - Availability
  - Consistency
  - Partition tolerance
  - Cost-effectiveness
  - Performance
- To meet the *reliability* and *scaling* needs, Amazon has developed a number of storage technologies.
  - S3, EBS, SimpleDB, RDS, Redshift, and **Dynamo**

- ***Dynamo is used to manage the state of services that have very high reliability/availability and scalability requirements. What doesn't it have???***

Key requirements

- Key requirement: *reliability/availability*
  - Slightest outage has significant financial consequences and impacts customer trust.
- To support continuous growth, platform needs to be highly *scalable*.
- Key organization lesson learned:
  - *The **Reliability** and **scalability** of a system is dependent on how its **application state is managed**.*

Observation

- Many services require only a simple *primary-key access* to a data store:
  - Best seller lists
  - Shopping carts
  - Customer preferences
  - Session management
  - Sales rank
  - Product catalog

System Requirements

- The Dynamo distributed storage system:
  - Scalable
  - Simple: *key-value*
  - ***Highly available***
  - ***Partition tolerance*** via redundancy
  - Able to guarantee ***Service Level Agreements (SLA)*** to internal and external customers.

Dynamo

- Dynamo uses a synthesis of well known techniques to achieve *scalability* and *availability*:
  - Data is partitioned and replicated using hashing.
  - Consistency is facilitated by object versioning.
- Consistency among replicas during updates is maintained by a:
  - Quorum-like technique
  - Decentralized replica synchronization protocol

System Interface

Stores objects associated with a key through two operations:
get() and *put().*

- get(key)
  - locates the object replicas associated with the key in the storage system and returns a single object, or a *list of objects with conflicting versions* along with a context.
- put(key, context, object)
  - determines where the replicas of the object should be placed based for the associated key, and writes the replicas to disk.

- Notes:
  - **The context encodes system metadata about the object** that is opaque to the caller and includes information such as the *version of the object*.
  - Context info is stored with the object to verify the validity of the put request.
  - Key (128 bits) and object is treated as BLOB.

Summary of techniques used in *Dynamo* and their advantages

Problem	Technique	Advantage
Partitioning	Consistent Hashing*	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Logical clocks for read consistency, Version is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees (hash tree)	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node *liveness* information. Bounded time.

*In most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped. Only K/n (keys/slots) needs remapping.

Consistent Hashing

- Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table.
- It powers many high-traffic dynamic websites and web applications.
- Distributed caches that power many high-traffic dynamic websites and web applications, typically consist of a particular case of distributed hashing.
- These take advantage of an algorithm known as consistent hashing.

Consistent Hashing

- Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table.
- It powers many high-traffic dynamic websites and web applications.
- Distributed caches that power many high-traffic dynamic websites and web applications, typically consist of a particular case of distributed hashing.
- These take advantage of an algorithm known as consistent hashing.

What's Hashing All About

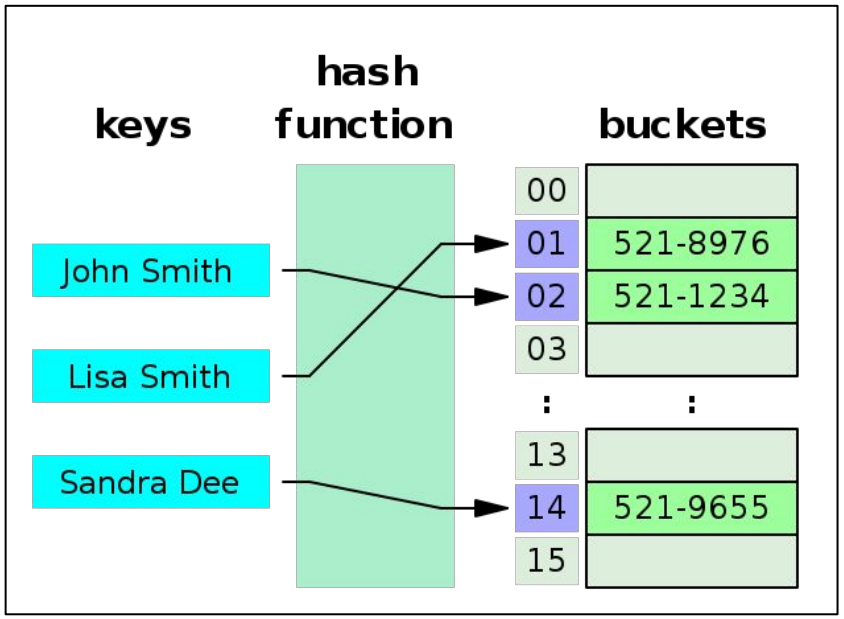
- A hash function is a function that maps one piece of data—typically describing some kind of object, often of arbitrary size—to another piece of data, typically an integer, known as hash code, or simply hash.
- ~O(1) (or O(n) with linked list) time complexity.

Example:

```
index = hash(object) mod N
```

Properties of good hash functions?

What's with the mod op?



Sample Hash Functions

```
def my_hash(string):
    """
    Hash function that returns the sum of the ASCII values
    of each character in the string.
    """
    hash_value = 0
    for char in string:
        hash_value += ord(char)
    return hash_value

def my_hash_to_100(string):
    hash_value = my_hash(string) % 100  # Calc hash value between 0 and
100
    return hash_value

def hash_to_100(string):
    hash_value = hash(string) % 100  # Calc hash value between 0 and 100
    return hash_value
```

Problems with these hashing functions?

Distributed Hashing

- In some situations, it may be necessary or desirable to split a hash table into several parts, hosted by different servers.
- One of the main motivations for this is to bypass the memory limitations of using a single computer, allowing for the construction of arbitrarily large hash tables (given enough servers).
- In such a scenario, the objects (and their keys) are distributed among several servers, hence the name.
- A typical use case for this is the implementation of in-memory caches, such as **Memcached**.

Partitioning Using Consistent Hashing

- *Consistent hashing is used* to *partition* the load across multiple storage hosts.
- The output range of the hash function is treated as a *fixed circular ring*.
- *Each node is assigned a random value* within this space (ring).
  - For example, a hash function that outputs 360 degrees.
- Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring.
- Walk the ring clockwise to find the first node with a position larger than the item's position.

Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web by David Karger et al).

Ketama algorithm.

Consistent Hashing 1

- The need for consistent hashing arose from limitations experienced while running collections of caching machines - web caches, for example.
- If you have a collection of *n* cache machines then a common way of load balancing across them is to put object o in cache machine number ***hash(o) mod n***.
- This works well until you add or remove cache machines (for whatever reason), for then ***n* changes and every object is hashed to a new location**.

Consistent Hashing 2

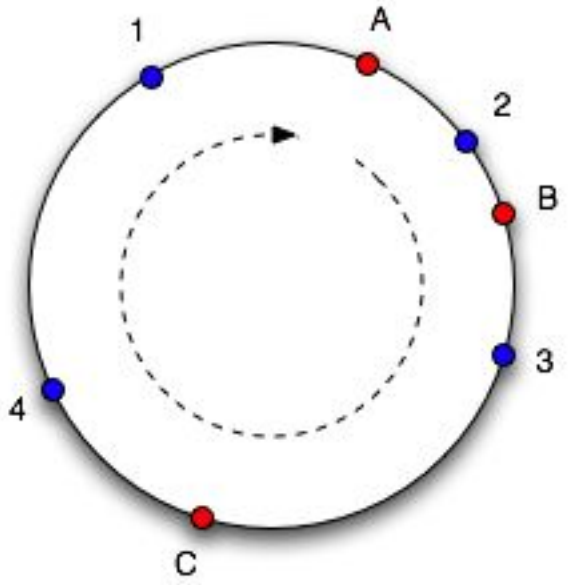
- It would be nice if, when a cache machine was added, it took its fair share of objects from all the other cache machines.
- Equally, when a cache machine was removed, it would be nice if its objects were shared between the remaining machines.
- This is what consistent hashing does - consistently maps objects to the same cache machine, as far as is possible.

Consistent Hashing 3

- The basic idea behind the consistent hashing algorithm is to ***hash both objects and caches using the same hash function***.
- The reason to do this is to ***map the cache to an interval***, which will contain a number of object hashes.
- ***If the cache is removed then its interval is taken over by a cache with an adjacent interval***.
- All the other caches remain unchanged.

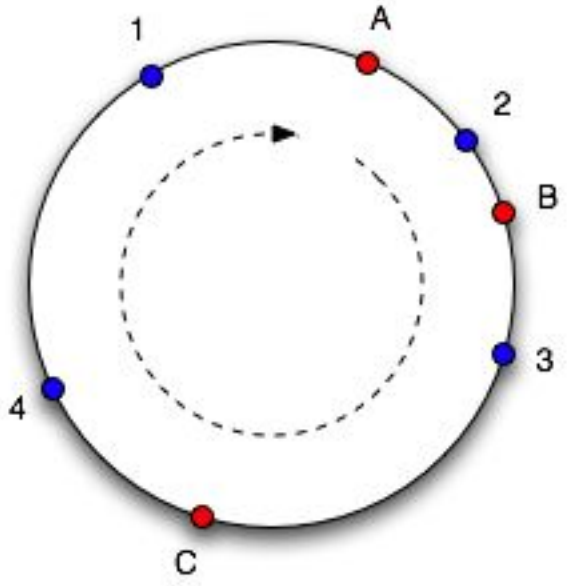
Consistent Hashing Example 1

- The hash function actually maps objects and caches to a number range.
- The circle has a number of objects (1, 2, 3, 4) and caches (A, B, C) marked at the points that they hash to.
- To find which cache an object goes in, move clockwise around the circle until we find a cache point. So object 1 and 4 belong in cache A, object 2 belongs in cache B and object 3 belongs in cache C.

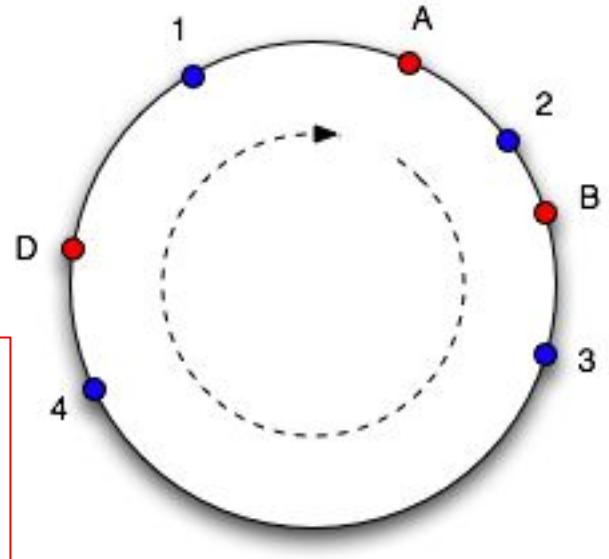


Consistent Hashing Example 2

- Consider what happens if cache C is removed: object 3 now belongs in cache A, and all the other object mappings are unchanged (left).
- If then another cache D is added in the position marked it will take objects 3 and 4, leaving only object 1 belonging to A (right).

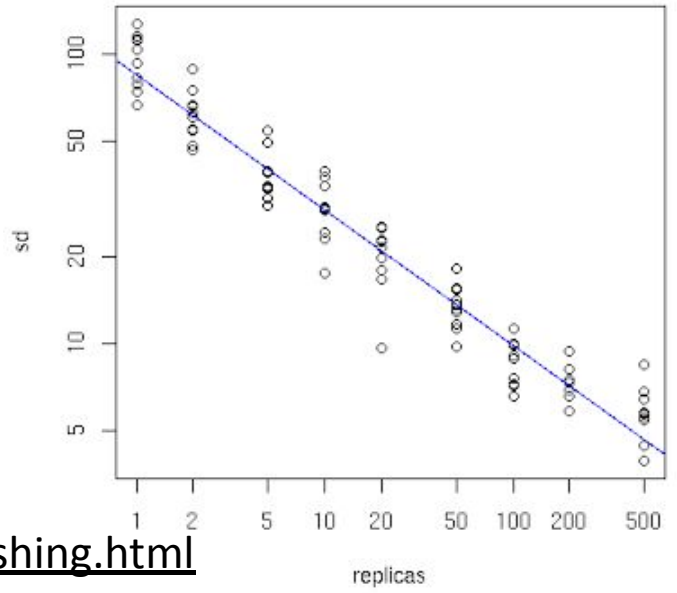


Problem: size of the intervals assigned to each cache is hit and miss



Consistent Hashing Example 2

- This works well, except the **size of the intervals assigned to each cache is hit and miss**.
- The solution to this problem is to introduce the idea of "**virtual nodes**", which are replicas of cache points in the circle.
- So whenever we add a cache we create a number of points in the circle for it.
- Distribute each node around circle.



Java implementation:
http://www.tom-e-white.com/2007/11/consistent-hashing.html

Data Versioning

- Dynamo provides ***eventual*** consistency.
- Allows updates to be propagated to all replicas asynchronously.
- A ***put()*** call may return to its caller before the update has been applied to all of the replicas.
- A ***get()*** call may return many versions of the same object.
- Challenge:
  - an object having distinct version sub-histories, which the system will need to reconcile in the future.
- Solution:
  - use ***vector clocks*** in order to capture causality between different versions of the same object.

Vector Clocks

- Algorithm for generating a partial ordering of events in a distributed system and detecting causality violations.
- Based on ***Lamport timestamps***: interprocess messages contain the state of the sending process (node's) logical clock.
- A vector clock of a system of ***N*** processes is an array/vector of ***N*** logical clocks.
- One clock per process; a local "smallest possible values" copy of the global clock-array is kept in each process.

Note: a partial order on a set is an arrangement such that, for certain pairs of elements, one precedes the other.

Vector Clocks – Update Rules

- Initially all clocks are zero.
- Each time a process experiences an internal event, it increments its own *logical clock* in the vector by one.
- Each time a process prepares to send a message, it increments its own *logical clock* in the vector by one and then sends its entire vector along with the message being sent.
- Each time a process receives a message, it increments its own *logical clock* in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

Happened Before

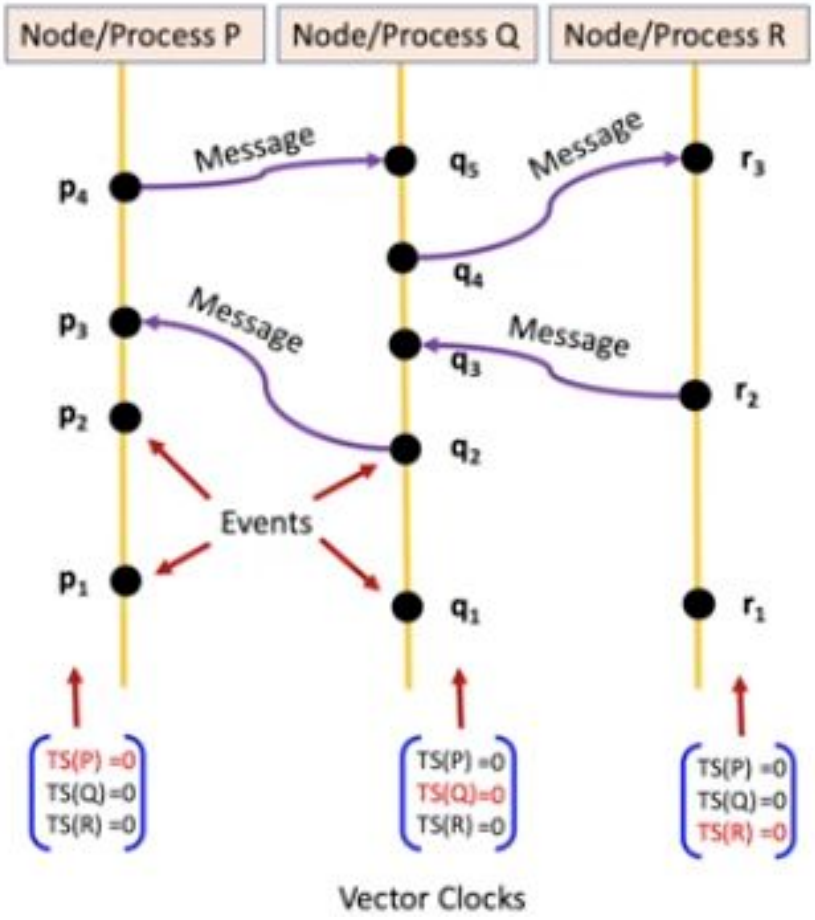
The happened-before relation is formally defined as the least strict partial order on events such that:

- If events a and b occur on the same process, a -> b if event a preceded event b.
- If event a is the sending of a message and event b is the reception of the message sent in event a, a -> b.

If two events happen in different isolated processes (that do not exchange messages directly or indirectly via third-party processes), then the two processes are said to be concurrent, that is neither a -> b nor b -> a is true.

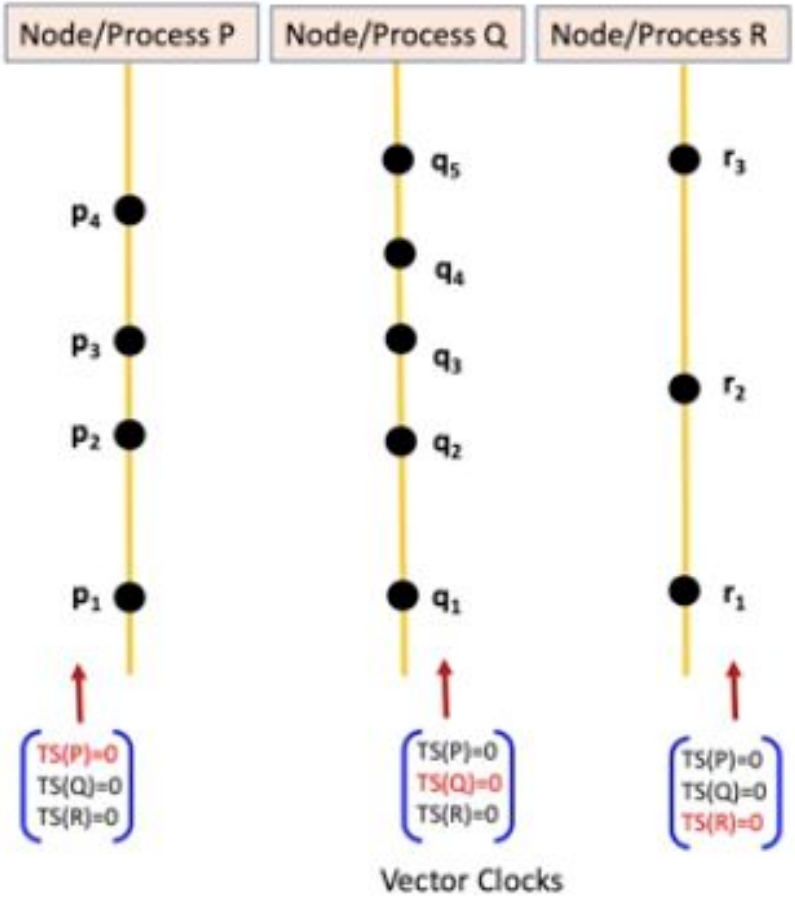
Vector Clocks: Basics

- Each event is stamped with a vector of timestamps/counters (TSs), one for each node/process (e.g., Vector = [TS(P), TS(Q),TS(R)]).
- Initially, all the timestamps in the vector are set to zero (e.g., Vector = [TS(P)=0, TS(Q)=0,TS(R)=0]).
- Each node/process increments its own timestamp and updates other timestamps in the vector when a new event occurs.
- When a node/process sends a message, the entire vector is sent with that message, in order to keep vector clocks in synchronization.



Vector Clock Conditions and Rules: Local Events

- Each node increments its own timestamp/counter in the vector by 1, when a new local/internal event occurs (i.e., local_timestamp = local_timestamp + 1).
- A sent message event is also a local event, and a timestamp/counter is incremented in the same way.



Vector Clock Conditions and Rules: Local Events

- Example1: Vector Clock (Timestamps) at local event p_1 =

$$\begin{pmatrix} TS(P)=1 \\ TS(Q)=0 \\ TS(R)=0 \end{pmatrix}$$

Update

$$\begin{pmatrix} TS(P)=0 \\ TS(Q)=0 \\ TS(R)=0 \end{pmatrix}$$

Initial value of a Vector Clock (Timestamps) on the node/process P

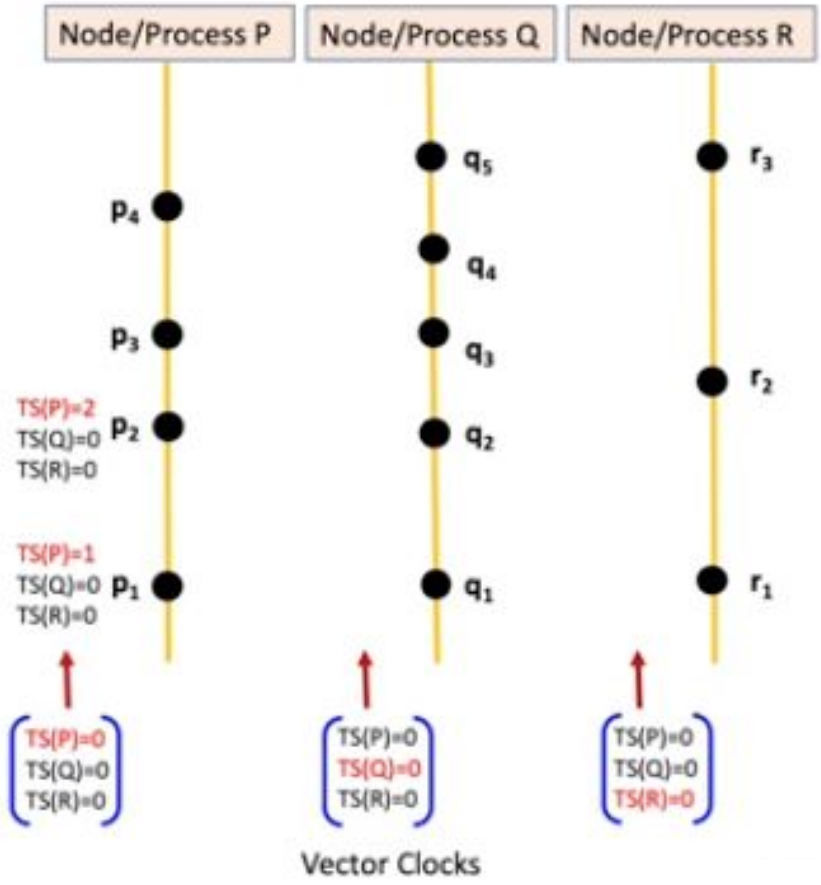
- Example2: Vector Clock (Timestamps) at local event p_2 =

$$\begin{pmatrix} TS(P)=2 \\ TS(Q)=0 \\ TS(R)=0 \end{pmatrix}$$

Update

$$\begin{pmatrix} TS(P)=1 \\ TS(Q)=0 \\ TS(R)=0 \end{pmatrix}$$

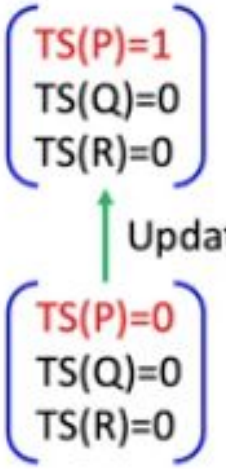
Vector Clock (Timestamps) at local event p_1



Vector Clocks

Vector Clock Conditions and Rules: Local Events

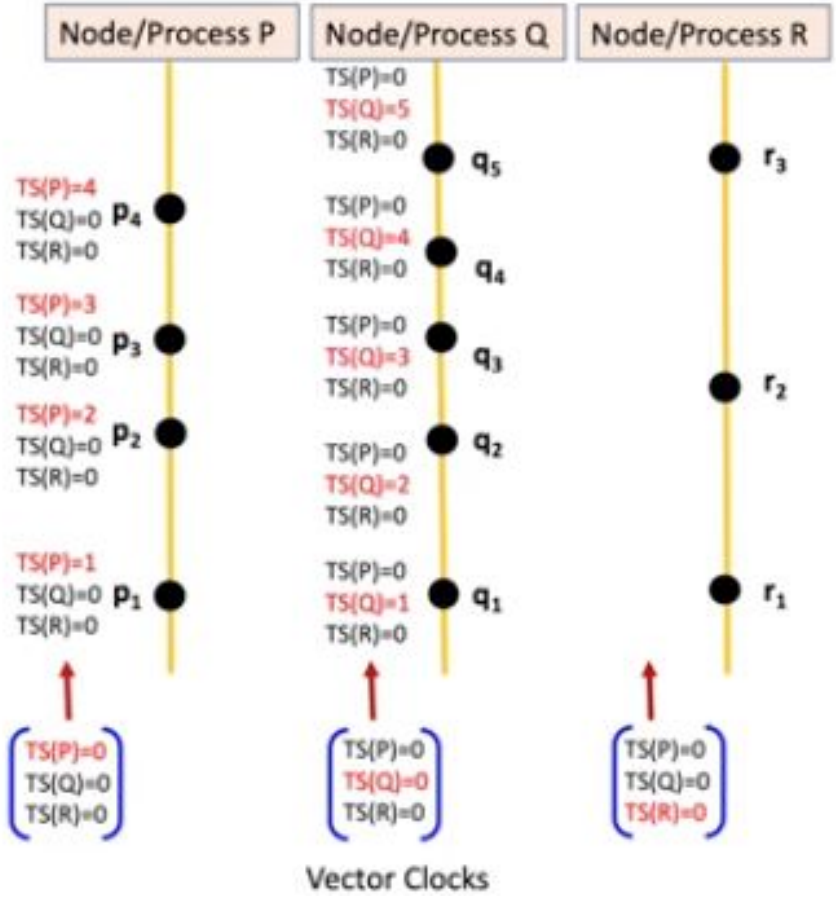
- Example1: Vector Clock (Timestamps) at local event p_1 =



Initial value of a Vector Clock (Timestamps) on the node/process P

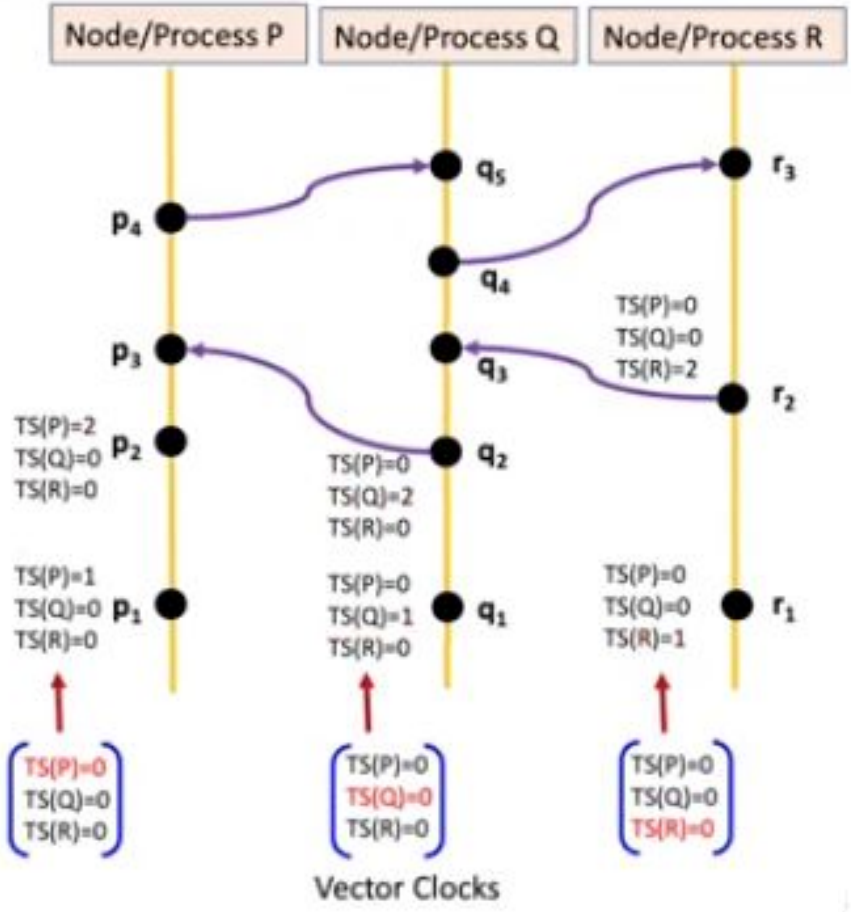
- Example2: Vector Clock (Timestamps) at local event p_2 =

Vector Clock (Timestamps) at local event p_1



Vector Clocks

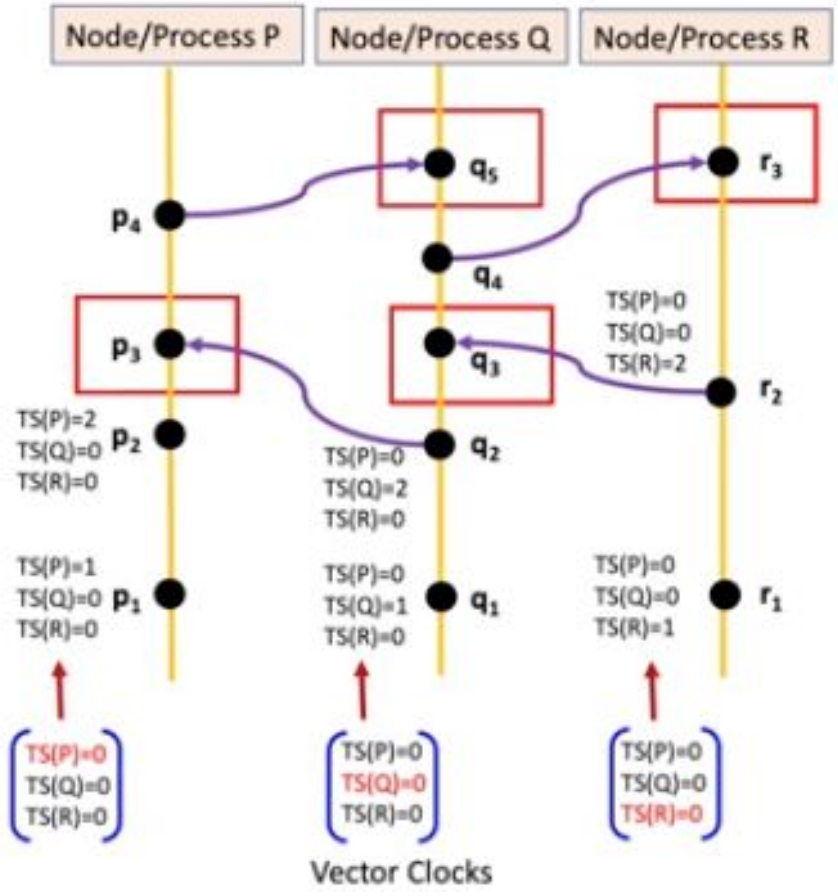
Vector Clock Conditions and Rules: External Events/Received Messages



Vector Clock Conditions and Rules: External Events/Received Messages

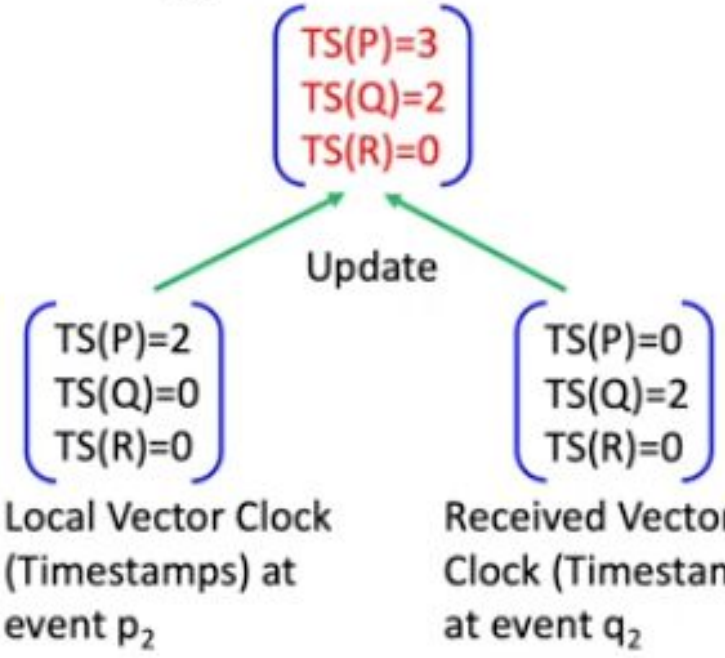
- When a message is received by any node which causes an event, then:

1. The node receiving the message increments its own timestamp/counter in the vector by 1 (i.e., local_timestamp = local_timestamp + 1).
2. It also updates each node's timestamp/counter in the vector to the maximum value based on the value of that timestamp in the local vector and received vector (i.e., max(local_timestamp, received_timestamp)).



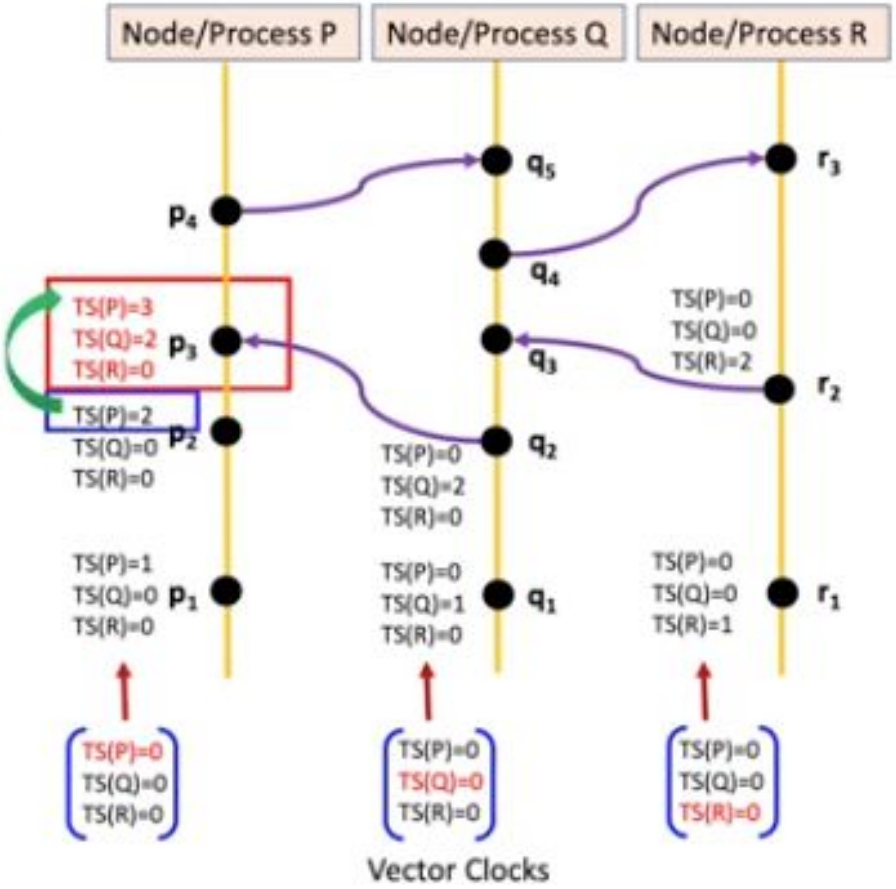
Vector Clock Conditions and Rules: External Events/Received Messages

- Example1: Vector Clock (Timestamps) at received message event p_3 =



Local Vector Clock (Timestamps) at event p_2

Received Vector Clock (Timestamps) at event q_2



Vector Clock Conditions and Rules: External Events/Received Messages

- Example1: Vector Clock (Timestamps) at received message event p_3 =

TS(P)=3
TS(Q)=2
TS(R)=0

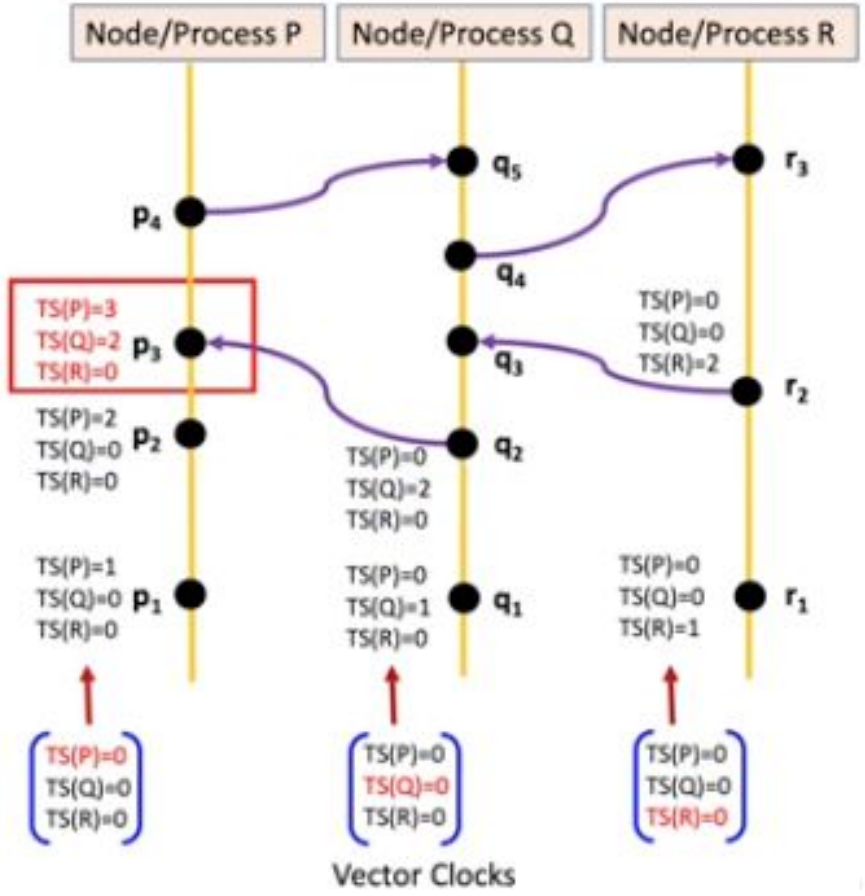
Update

TS(P)=2
TS(Q)=0
TS(R)=0

Local Vector Clock (Timestamps) at event p_2

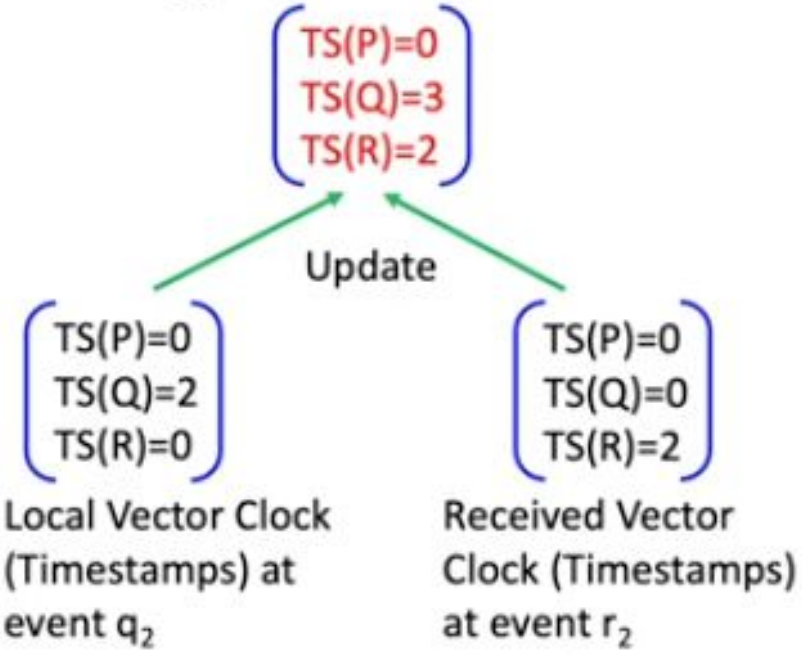
TS(P)=0
TS(Q)=2
TS(R)=0

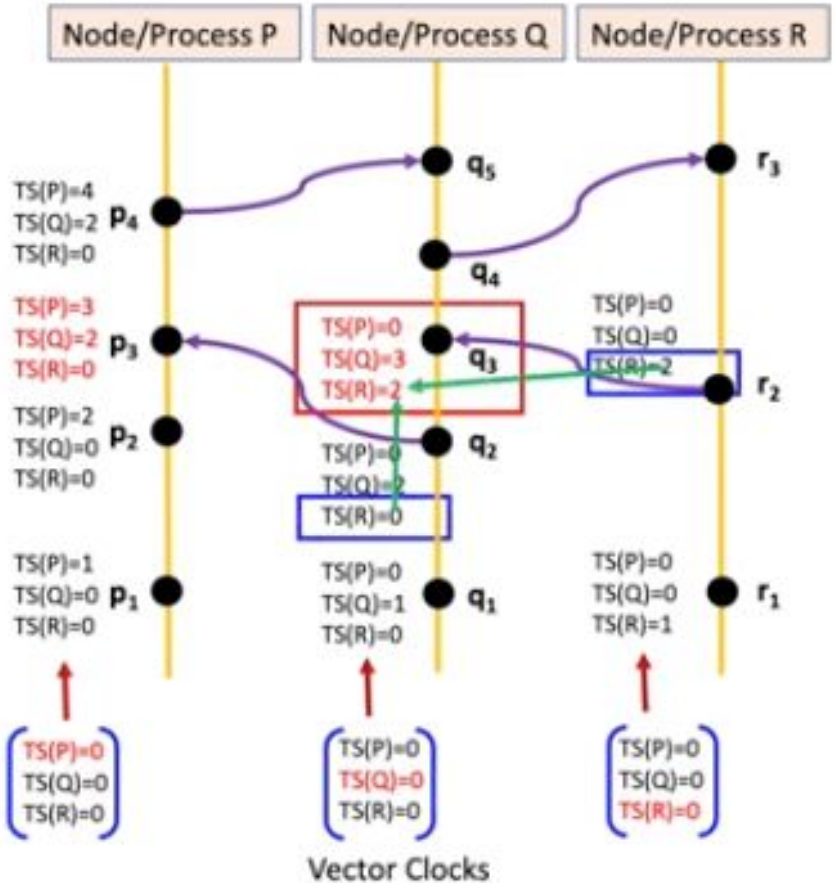
Received Vector Clock (Timestamps) at event q_2



Vector Clock Conditions and Rules: External Events/Received Messages

- Example2: Vector Clock (Timestamps) at received message event q_3 =





Vector Clock Conditions and Rules: External Events/Received Messages

- Example3: Vector Clock (Timestamps) at received message event q_5 =

TS(P)=4
TS(Q)=5
TS(R)=2

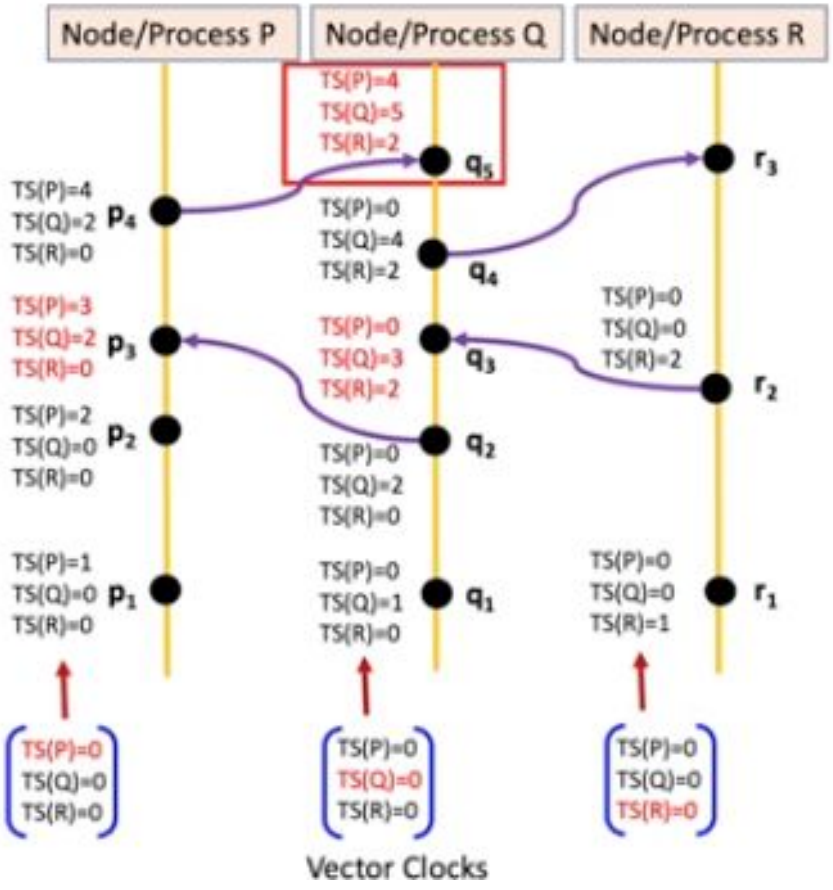
Update

TS(P)=0
TS(Q)=4
TS(R)=2

Local Vector Clock (Timestamps) at event q_4

TS(P)=4
TS(Q)=2
TS(R)=0

Received Vector Clock (Timestamps) at event p_4



Vector Clock Conditions and Rules: External Events/Received Messages

- Example4: Vector Clock (Timestamps) at received message event r_3 =

TS(P)=0
TS(Q)=4
TS(R)=3

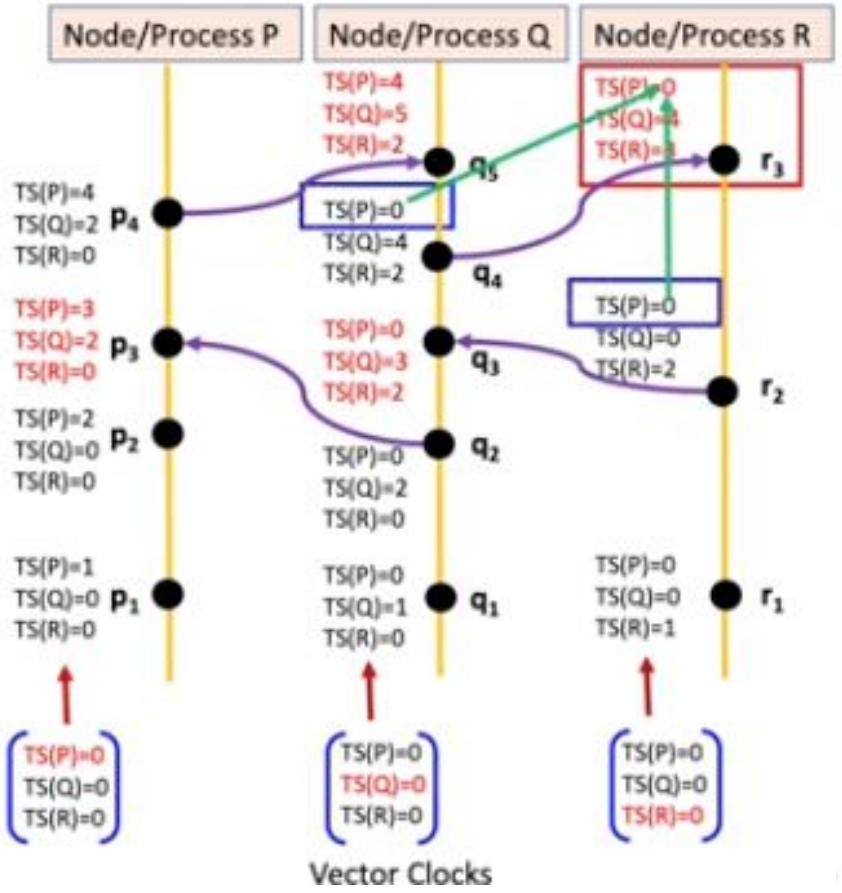
Update

TS(P)=0
TS(Q)=0
TS(R)=2

Local Vector Clock (Timestamps) at event r_2

TS(P)=0
TS(Q)=4
TS(R)=2

Received Vector Clock (Timestamps) at event q_4



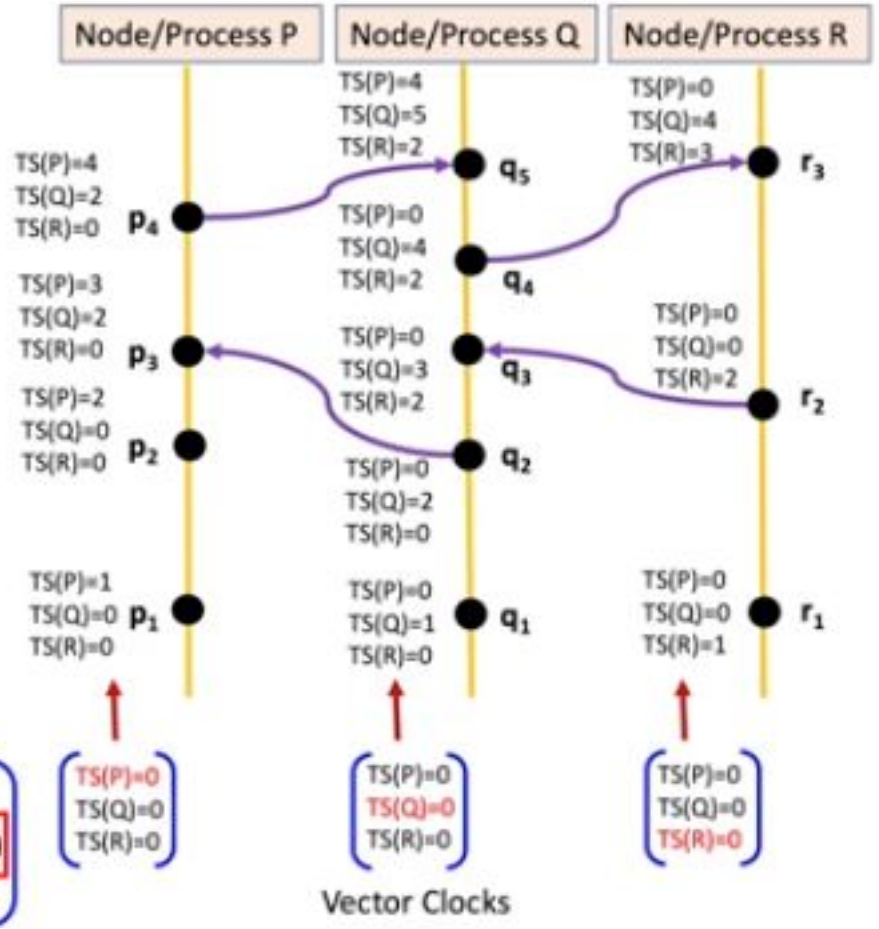
Vector Clock Conditions and Rules : Ordering of Events

- **Happened-Before Relationship (→)**: An event a happens before b (a→b), if and only if V(a) ≤ V(b) and a ≠ b.
- Example: $V(q_2) \leq V(p_3)$, therefore $q_2 \rightarrow p_3$

$$\begin{pmatrix} TS(P)=0 \\ TS(Q)=2 \\ TS(R)=0 \end{pmatrix} \leq \begin{pmatrix} TS(P)=3 \\ TS(Q)=2 \\ TS(R)=0 \end{pmatrix}$$

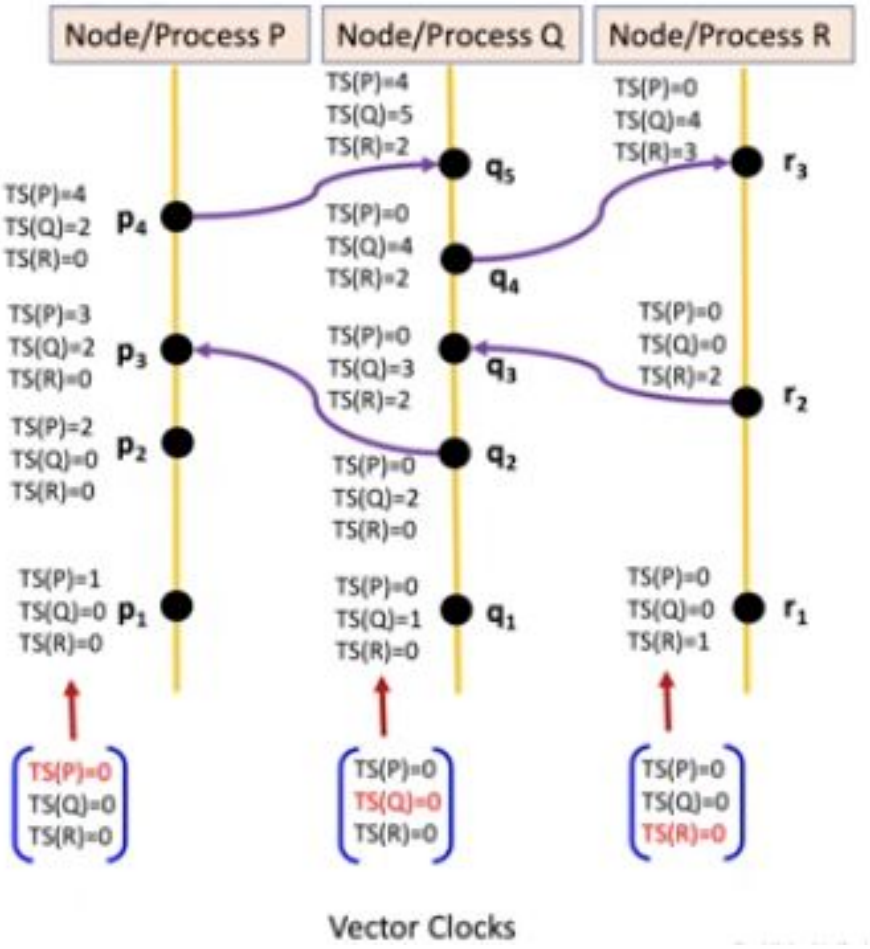
- **Concurrent Events**: Two events a and b are concurrent (a||b), if and only if V(a) ⋦ V(b) and V(b) ⋦ V(a).
- Example: $V(p_1) \not\leq V(q_1)$ and $V(q_1) \not\leq V(p_1)$, therefore $p_1 \,||\, q_1$

$$\begin{pmatrix} TS(P)=1 \\ TS(Q)=0 \\ TS(R)=0 \end{pmatrix} \not\leq \begin{pmatrix} TS(P)=0 \\ TS(Q)=1 \\ TS(R)=0 \end{pmatrix} \;\&\; \begin{pmatrix} TS(P)=0 \\ TS(Q)=1 \\ TS(R)=0 \end{pmatrix} \not\leq \begin{pmatrix} TS(P)=1 \\ TS(Q)=0 \\ TS(R)=0 \end{pmatrix}$$



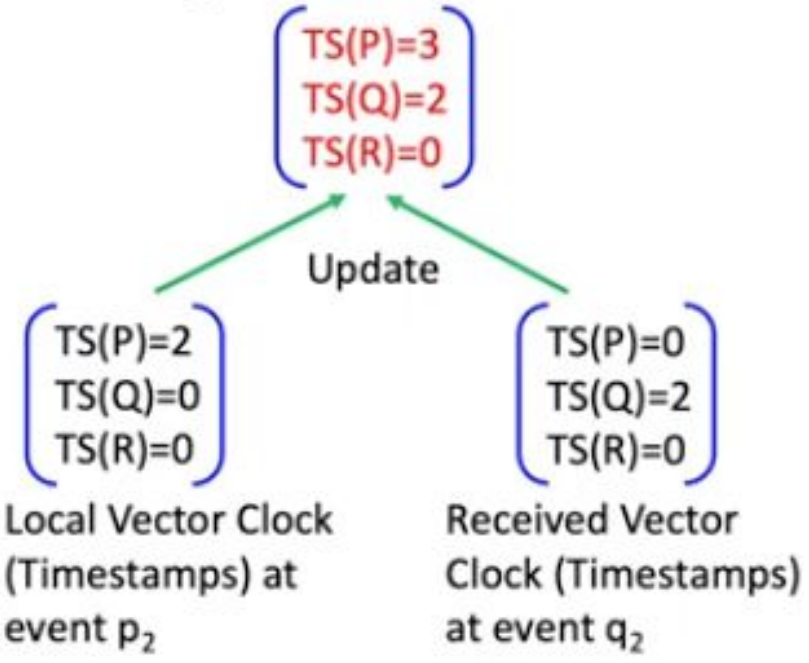
Limitations of Vector Clocks

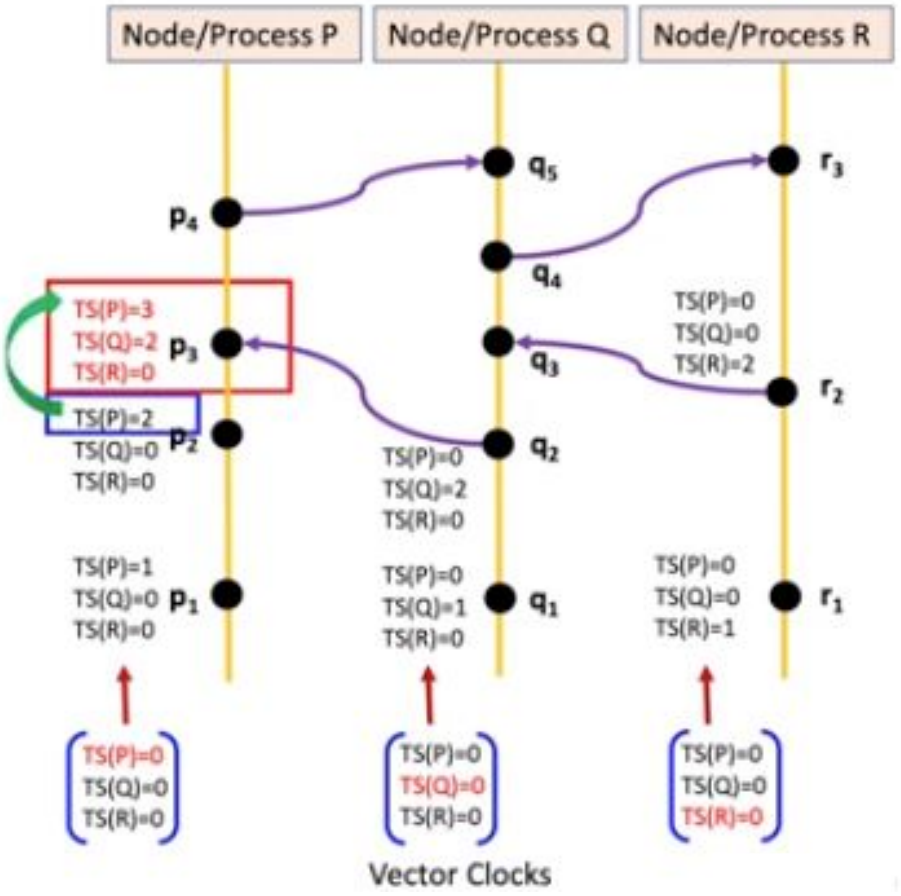
- The vector clock consists of one entry (timestamp/counter) per node, that means it can potentially become very large for large systems.
- The entire vector is sent with each message each time which increases communication overheads.
- All vector elements have to be checked on every received message which increases processing overheads.
- A variety of techniques have been applied to reduce the size of vector clocks (e.g., either by performing periodic garbage collection, or by reducing accuracy by limiting the size).



Vector Clock Conditions and Rules: External Events/Received Messages

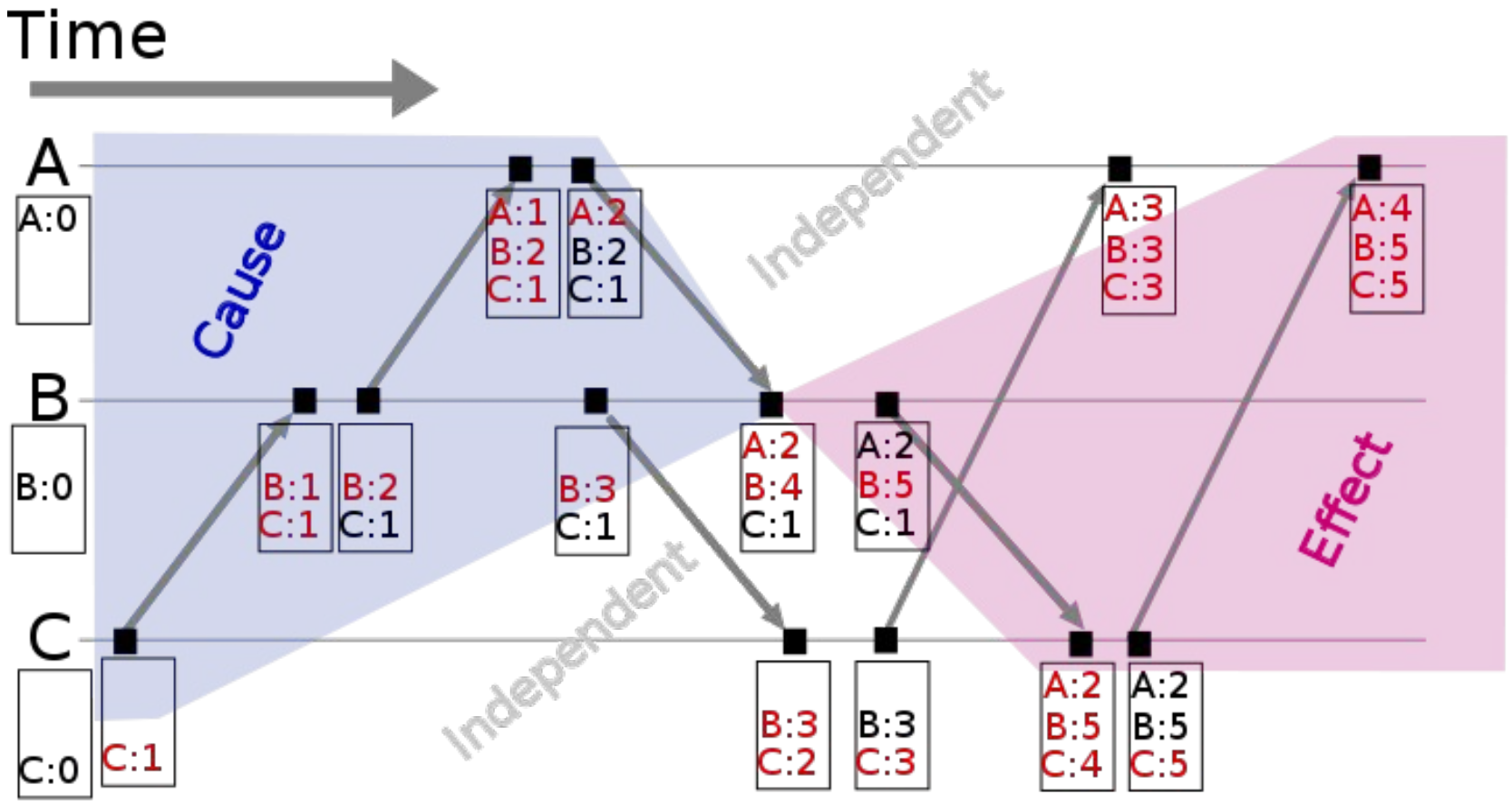
- Example1: Vector Clock (Timestamps) at received message event p_3 =





Vector Clocks – Update Rules

Events in the blue region are the causes leading to event B4, whereas those in the red region are the effects of event B4

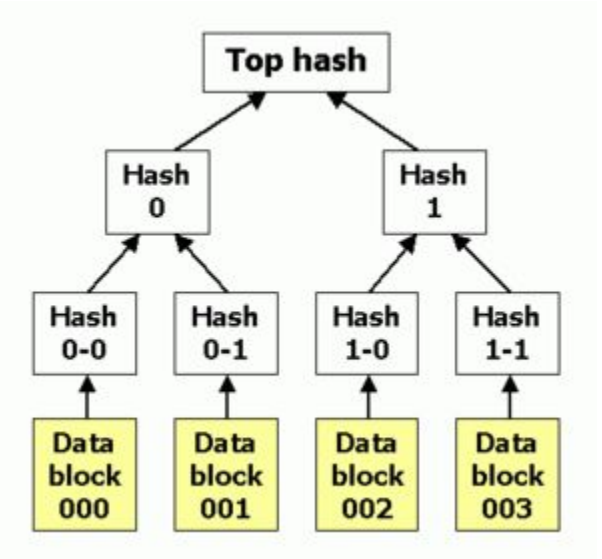


Sloppy Quorum

- To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems.
- Two configurable values: R and W.
- R/W signify the minimum number of nodes that must participate in a successful read/write operation (from top N in preference list).
- Setting R + W > N yields a quorum-like system.
- In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas.
- For this reason, R and W are usually configured to be less than N, to provide better latency. I.e., sloppy.

Mekle Hash Trees

- Replica synchronization:
  - Every non-leaf node is labeled with the hash of the labels of its children nodes, i.e., each node maintains a separate *Merkle (hash) tree* for each key range.
  - Allows two nodes to compare trees to see if they are consistent when synchronizing replicas.
  - Allow efficient and secure verification of the contents of larger data structures.



Gossip-based Message Protocol

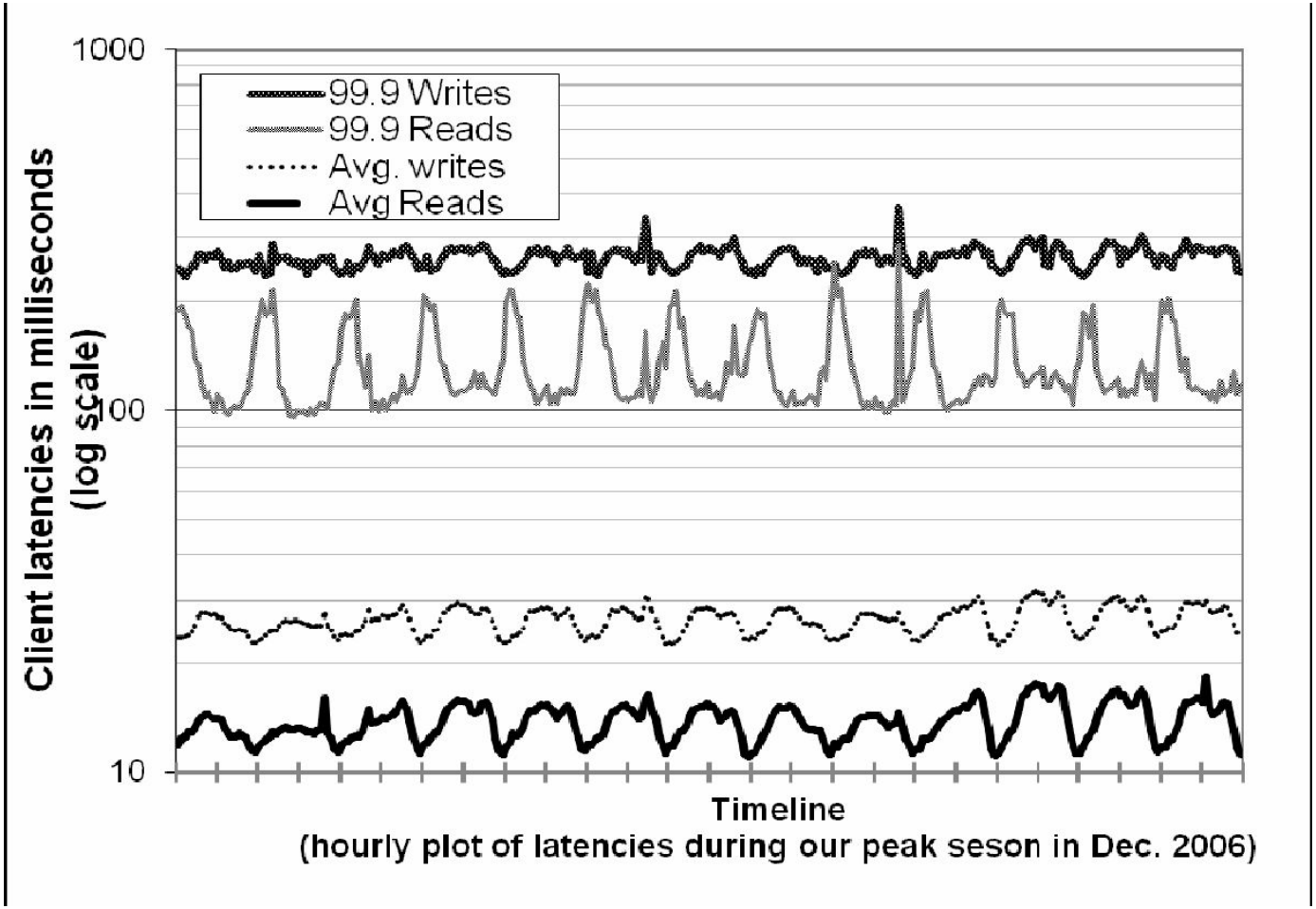
- Membership and Failure Detection:
  - *Gossip-based message* protocol to propagate membership changes in a ring.
  - Inspired by the form of gossip seen in social networks.
  - Involves periodic, pairwise, inter-process interactions.
  - Information exchanged during interactions of bounded size.
  - Reliability not assured.

Amazon Implementation

- Each storage node has three main software components:
  1. Request coordination
  2. Membership and failure detection
  3. Local persistence engine
- All components implemented in Java
- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object < tens of kilobytes.
  - MySQL: object of > tens of kilobytes.
  - BDB Java Edition, in memory, etc.
- Pluggable persistence component best suited for application's access patterns.

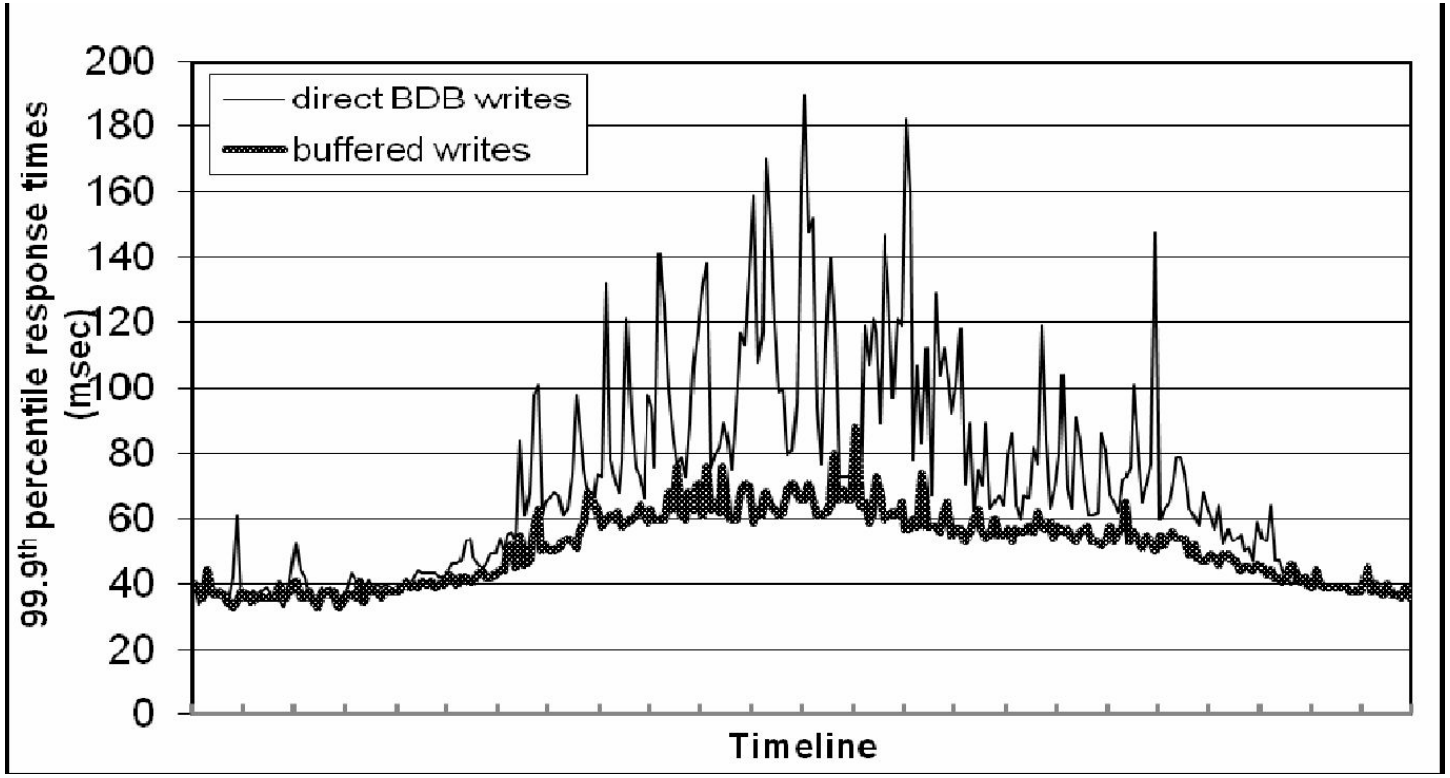
Evaluation – Shows consistent latency

Write latencies @99.9th percentile ~200ms



Evaluation

Can trade off Durability guarantees for performance and use buffered writes



Lessons Learned

- Business logic reconciliation
  - Client object performs its own reconciliation logic.
- Timestamp based reconciliation
  - Last write wins
- High-performance read engine
  - R=1, W=N

STOP

Visual Guide to NoSQL Systems