Algorithms and Distributed Systems 2023/2024 (Lecture Eight)

MIEI - Integrated Master in Computer Science and Informatics
MEI – Master in Computer Science and Informatics
Specialization block

Nuno Preguiça (nmp@fct.unl.pt)
Alex Davidson (a.davidson@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Based on slides from João Leitão

Lecture structure:

- Strong Consistency
- Eventual Consistency
- Dynamo (From Amazon)

Recently on ASD: State Machine Replication and Paxos

- Replication Strategy that, *from the outside*, of the system provides a behavior that:
  - Replication is hidden from Application/Users (**Transparency**)
  - The state of the system evolves having all operations being ordered (in an order that respects real-time).

Recently on ASD: State Machine Replication and Paxos

- Replication Strategy that, *from the outside*, of the system provides a behavior that:
  - Replication is hidden from Application/Users (**Transparency**)
  - The state of the system evolves having all operations being ordered (in an order that respects real-time).
- This is known as **Linearizability**.
- Can we be more precise?

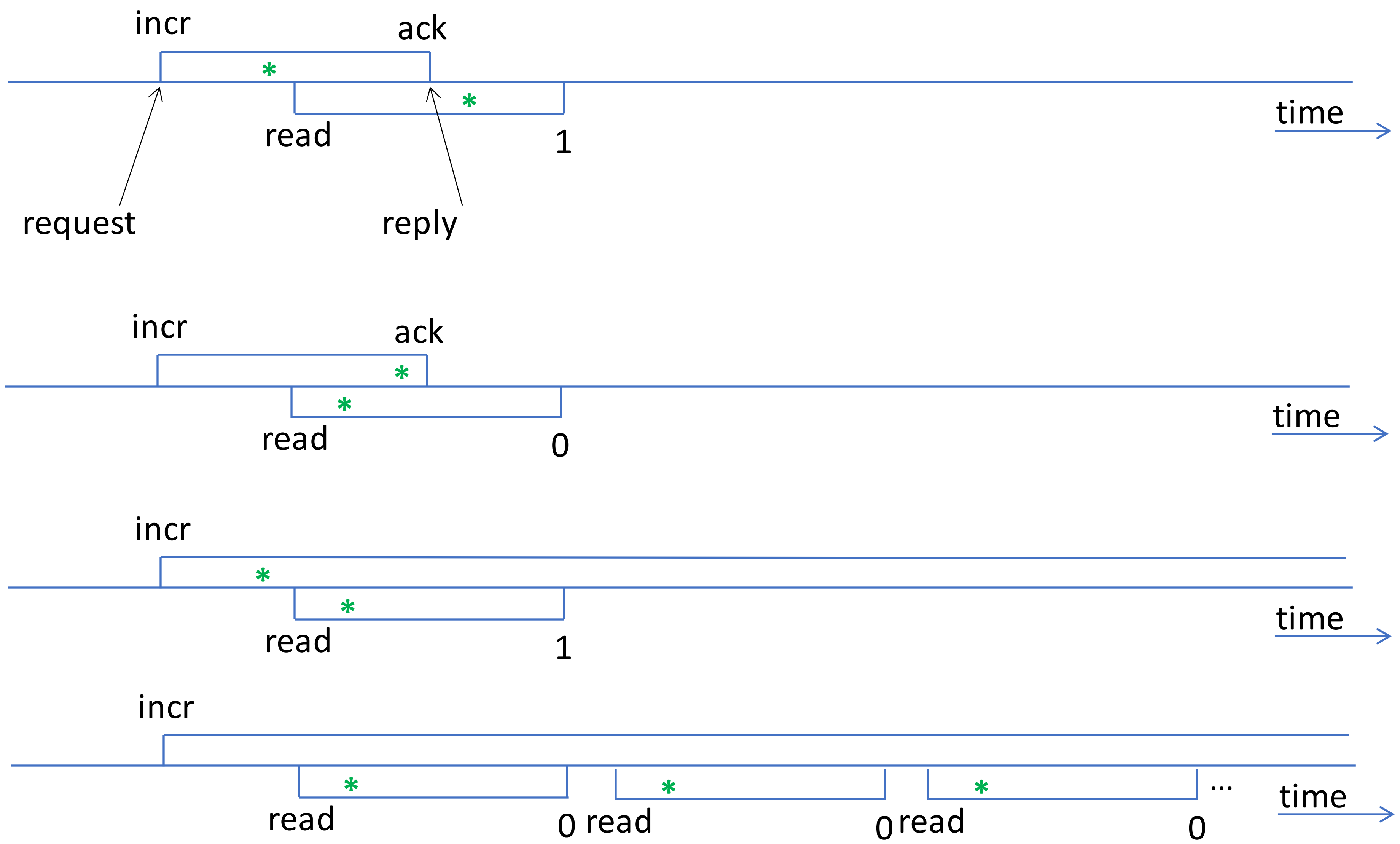
First: Atomicity

- Given a trace of a system, there exists for each operation a **serialization point** (between the start and the end of the operation).
- If we consider that the operation has happened at that point, then the state of the system follows the (ordered) effects of all operations.
- If an operation never returns a reply to the client, then such an operation **might or might not** have a serialization point.

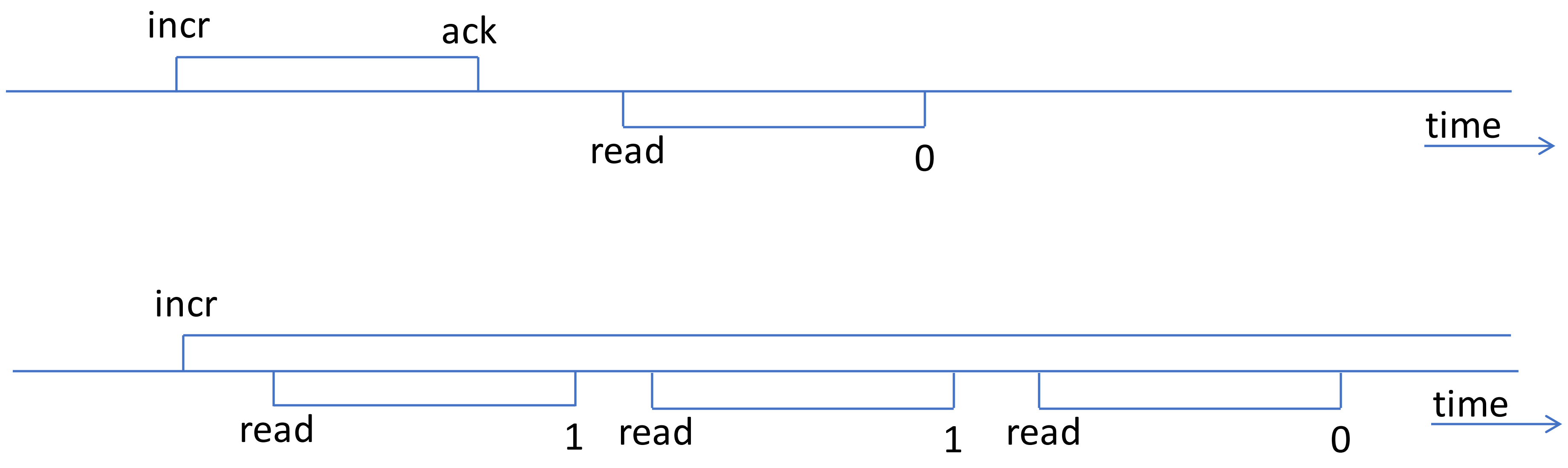
Atomicity

- Given a trace of a system, there exists for each operation a **serialization point** (between the start and the end of the operation).
- If we consider that the operation has happened at that point, then the state of the system follows the (ordered) effects of all operations.
- If an operation never returns a reply to the client, then such an operation **might or might not** have a serialization point.
- E.g., a replicated counter (initial value zero) where increment operations can be executed.

Atomic Executions over a Counter



Non-Atomic Execution



Strong Consistency

- Systems that provide this notion of strict evolution of state across all replicas (and hence Atomicity) are said to offer **Strong Consistency**.

- Multiple consistency models that fit Strong Consistency:
  - Linearizability.
  - Serializability.

Linearizability

Linearizability *is a guarantee about single operations on single objects.*

It provides a real-time (i.e., wall-clock) guarantee on the behavior of a set of single operations (often reads and writes) on a single object (e.g., distributed register or data item).

Linearizability

Under linearizability:

- Writes should appear **instantaneous**.
    - Informally, once a write completes, all later reads (where “later” is defined by a global wall-clock start time) should return the value generated by that write or the value of a later write.
- Once a read returns a concrete value, all later reads should either return that value or a value produced by a later write.

Example: ABD Quorum

Write operations can be performed immediately

Read operations require a *write-back* phase, to ensure that the most recent *observed* write is completed.

Subsequent reads return the value of this write, or a *subsequent* (i.e. *time-dependent*) write

Serializability

<u>Serializability</u> *is a guarantee about groups of* ***one or more*** *operations over* ***one or more*** *objects (usually called <u>transactions</u>).*

It guarantees that the execution of a set of transactions (usually containing a mix of read and write operations) over multiple data objects is equivalent to *some* serial execution (total ordering) of those transactions.

Serializability

Unlike linearizability, serializability does not—by itself—impose any real-time constraints on the ordering of operations.

It only requires a single order across all replicas (that might be different from the order of operations in relation to a global wall clock).

Example: Paxos

The Paxos agreement protocol ensures a consistent ordering of operations across replicas.

However, that ordering is not dependent on *when* operations were proposed.

It simply ensures that the same ordering is applied everywhere.

Why is it that we like Strong Consistency?

Positives of Strong Consistency

- Easier to reason about the evolution of the state of a replicated system (since all replicas evolve across the same set of states in the same order).
    - In ABD, via the quorum mechanic
    - In Paxos, via the explicit ordering of operations

- Some operations over the state of a replicated system require strong consistency to ensure that replicas converge to the same state (**non-commutative** operations for instance).

Negatives of Strong Consistency

Negatives of Strong Consistency

- **Strong Consistency** solutions will always require (among other things) that operations contact a quorum of replicas to execute.

- This is required to enforce that read operations always return the value of the last completed write operation **(non-empty intersection)**.

Negatives of Strong Consistency

- **Strong Consistency** solutions will always require (among other things) that operations contact a quorum of replicas to execute.

- In many cases, contacting a quorum of replicas might be an issue that creates new problems...

Negatives of Strong Consistency

In many cases, contacting a quorum of replicas might be an issue that creates new problems...

- Geo-Replicated Systems (high latency between replicas in different continents).

Negatives of Strong Consistency

In many cases, contacting a quorum of replicas might be an issue that creates new problems...

- Geo-Replicated Systems (high latency between replicas in different continents).
- Systems where there is poor connectivity between clients and server replicas (e.g., mobile applications).

Negatives of Strong Consistency

In many cases, contacting a quorum of replicas might be an issue that creates new problems...

- Geo-Replicated Systems (high latency between replicas in different continents).
- Systems where there is poor connectivity between clients and server replicas (e.g., mobile applications).
- Systems with strict requirements of performance (Service Level Agreements) such as maximum latency for an operation.

How can we deal with such scenarios?

How can we deal with such scenarios?

- Well evidently if there is Strong Consistency…
- …there is also **Weak Consistency**

Weak Consistency Solution

- Operations are only executed in a small set of replicas (smaller than a quorum, i.e., *without* intersection guarantees), eventually only one.
- Operations are then propagated between replicas in background.
- Clients obtain answer before replication concludes (i.e., relies on asynchronous replication).

Weak Consistency Solution

- What are the problems that can arise from this strategy?

Weak Consistency Solution

- What are the problems that can arise from this strategy?

- Replicas will diverge (while operations are happening and being propagated among replicas, each replica might have a different value from all the others).

Weak Consistency Solution

- What are the problems that can arise from this strategy?

- Replicas will diverge (while operations are happening and being propagated among replicas, each replica might have a different value from all the others).

- We must find ways to explicitly deal with
  replica divergence.

Weak Consistency Models

- Similar to Strong Consistency, there are multiple consistency models that fit within what is generally referred as **Weak Consistency**.

- A consistency model effectively defines a set of **safety conditions for a replication algorithm**.

- Such a model (usually) restricts the state of the system that a client can observe, given the history of the system and their own local history (i.e. the operations that the client itself has executed before)

Weak Consistency Models

- Eventual Consistency.

- Causal Consistency.

(Many others that we would need an entire course just to cover them and understand the differences).

Weak Consistency Models

- Eventual Consistency. <- Today

- Causal Consistency.

(Many others that we would need an entire course just to cover them and understand the differences).

Weak Consistency Models

- Eventual Consistency. <- Today
- Causal Consistency. <- Next Week

(Many others that we would need an entire course just to cover them and understand the differences).

Eventual consistency

- What are the safety guarantees enforced by the eventual consistency model?

Eventual consistency

- What are the safety guarantees enforced by the eventual consistency model?
- Actually: none... This is the *weakest* form of weak consistency that exists.
- So what is effectively promised (or enforced) by eventual consistency?

Eventual consistency

- Eventual consistency only promises that: eventually, when **no write operations** happen for a sufficiently long period of time, all replicas of the system will converge to the same state.

Eventual consistency

- Eventual consistency only promises that: eventually, when **no write operations** happen for a sufficiently long period of time, all replicas of the system will converge to the same state.
- Is this safety or liveness?

Eventual consistency

- Eventual consistency only promises that: eventually, when no write operations happen for a sufficiently long period of time, all replicas of the system will converge to the same state.

- Is this *safety* or *liveness*?
- Effectively this is a **liveness** property.

- This has generated some debate, where some people in the community argue that eventual consistency is no consistency model at all.

Eventual Consistency

- Since there are no safety properties, the key aspect of Eventual Consistency replication strategies is to enforce the convergence of state.

- How can we do this?

Divergence Reconciliation

- Multiple Possibilities:
  - “***last writer wins***” Policy:
    - We define an order among any concurrent write operations.
    - We lose the effects of all operations (over a given data object) except for the last one (given that order).

Divergence Reconciliation

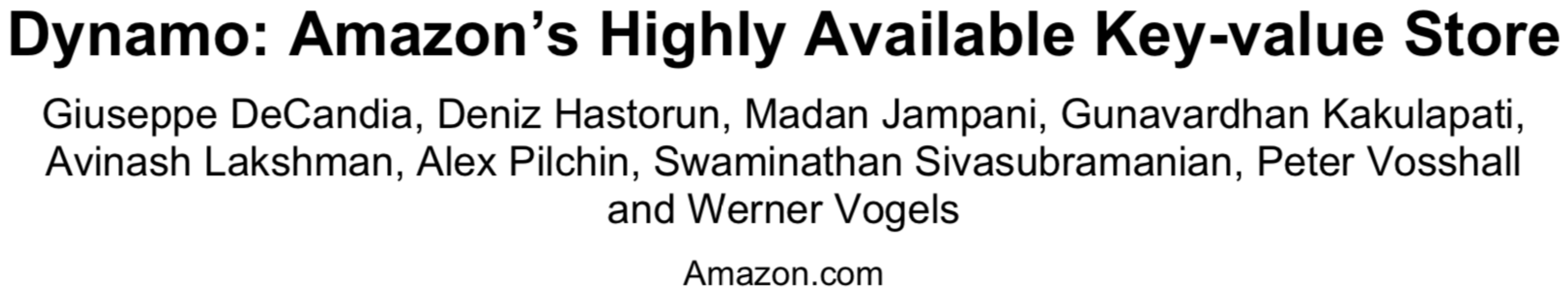
- Multiple Possibilities:
  - "***last writer wins***" Policy:
    - We define an order among any concurrent write operations.
    - We lose the effects of all operations (over a given data object) except for the last one (given that order).
  - Replicas have access to a "***merge procedure***" that given the state of two divergent replicas, knows how to compute a new state that combines both divergent states (this must be deterministic).

Divergence Reconciliation

- Multiple Possibilities:
  - "***last writer wins***" Policy:
    - We define an order among any concurrent write operations.
    - We lose the effects of all operations (over a given data object) except for the last one (given that order).
  - Replicas have access to a "***merge procedure***" that given the state of two divergent replicas, knows how to compute a new state that combines both divergent states (this must be deterministic).
  - We expose the divergence to clients (for instance by returning multiple values upon the execution of a read operation) and the client merges these values and write the new value back to the system (***application-dependent policy***, example: git merge)

Eventual consistency

- Many variants of eventual consistency exist (since there are no strict safety properties) that depend on implementation.
- We are going to focus on Dynamo (Amazon)



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

Amazon.com

Eventual Consistency: Dynamo

- Used in practice in many systems available now-a-days.
- Popularized by Amazon in the **Dynamo** System
  - Dynamo is a distributed and replicated data storage system.
  - Dynamo is used for many Amazon services, including the shopping cart.
  - Dynamo features multiple replicas, across different data centers. Operations execute by contacting only a very small number of replicas.
  - Clients are application servers (i.e., proxies) that operate within the Amazon datacenters.

Eventual Consistency: Other apps

- Also used in many mobile systems:
    - Clients execute their operations (offline) over a local copy (i.e., replica) of the (relevant) application state.
    - In background (typically when connectivity is available) they synchronize with servers.

Valid Question: Why are we focusing on an Amazon system?

Valid Question: Why are we focusing on an Amazon system?

- Amazon was the first global-scale online retailer…
- Encountered global-scale problems, such as: how to build a database for global operation?

Valid Question: Why are we focusing on an Amazon system?

- They were the first to break with the SQL paradigm and proposed Dynamo that had a much simpler No-SQL Interface (Key-Value Store).

amazon DynamoDB

Valid Question: Why are we focusing on an Amazon system?

- They were the first to break with the SQL paradigm and proposed Dynamo that had a much simpler No-SQL Interface (Key-Value Store).
- This in turn had a profound impact on how distributed system are built today…

Valid Question: Why are we focusing on an Amazon system?

ıe SQL paradigm
much simpler

mongoDB®

- This in turn had a profound im
distributed system are built to

cassandra

amazon
DynamoDB

Valid Question: Why are we focusing on an Amazon system?

ıe SQL paradigm
nuch simpler

- This in turn had a profound im
  distributed system are built to

Dynamo Requirements for Eventual Consistency (intuitive definition)

- Geographical replication with high performance and availability.

Dynamo Requirements for Eventual Consistency (intuitive definition)

- Geographical replication with high performance and availability.
- Write operations can be executed without being aware (i.e., observe the effect of) other write operations.
    - Two users are allowed to buy (”at the same time”) the last available ticket for a plane.

Dynamo Requirements for Eventual Consistency (intuitive definition)

- Geographical replication with high performance and availability.
- Write operations can be executed without being aware (i.e, observe the effect of) other write operations.
    - Two users are allowed to buy (”at the same time”) the last available ticket for a plane.
- Eventually, all operations are propagated among all replicas of the system, and the state converges.
    - In the previous example, the list of all passengers in a plane would become visible everywhere and we would observe overbooking… How to address this?

Dynamo Requirements for Eventual Consistency (intuitive definition)

- Starting Point:
  - Key-Value Store offering two operations
  - Read and Write operations (get/put)
    - get(key) → returns “value”
    - put(key,value) → returns “ack”

Dynamo Requirements for Eventual Consistency (intuitive definition)

- To deal with concurrent writes, Dynamo extends this interface.
- Add metadata that allows to associate a **put** operation to the previous **get** of **that same client.**
  - This allows to infer that a given put operation already reflects the effects of a set of previous put operations (those that were observed by the get operation).
- Get operations can return multiple values (if divergence is detected).

Dynamo: interface

- **get(key) → returns <list of values,context>**
  - Returns a list containing the values written by the most recent put operations (that have not yet observed the effects of each other). This ensures that effects of concurrent writes are not lost.
  - **Context** describes the set of put operations that are reflected in the returned list of values.
- **put(key,value,context) → returns "ack"**
  - Writes the new value for data object identified by key.
  - Provides the context of the most recent get executed by that client.

Dynamo: Implementation

- To simplify, lets just consider operations over a single key, and a fixed set of replicas N.

- In fact Dynamo deals with faults, and partitioning of data:
  - One-hop DHT.
  - Consistent hashing.
  - Specifies the protocol to deal with faults.

Dynamo: Implementation

- Read-Write Quorums over sub-set of replicas with size R, W.
- However: $R+W<N$ (*no intersection is guaranteed*)

Dynamo: Implementation

- Read-Write Quorums over sub-set of replicas with size R, W.
- However: R+W<N (*no intersection is guaranteed*)
- **Operations are executed in a single step:**
  - **Client contacts one of the replicas selected randomly.**
  - **Replica propagates the get or put request to N replicas.**
  - **Waits for the answer of R or W replicas before returning to the client (including itself)**
  - **In the particular case of the get operation, it returns the collection of most recent values obtained in its quorum if concurrent writes are detected.**

Dynamo: Implementation

- Read-Write Quorums over sub-set of replicas with size R, W.
- However: R+W<N (*no intersection is guaranteed*)
- **Operations are executed in a single step:**
  - **Client contacts one of the replica selected randomly.**
  - **Replica propagates the get or put request to N replicas.**
  - **Waits for the answer of R or W replicas before returning to the client (including itself)**
  - **In the particular case of the get operation, it returns the collection of most recent values obtained in its quorum if concurrent writes are detected.**

- **How do we know if two put operations were concurrent?**

Concurrency Detection

- Possibility: return in the context the list of puts that form the complete story of the data object identified by that key.

Concurrency Detection

- Possibility: return in the context the list of puts that form the complete story of the data object identified by that key.
- The context is sent alongside the put operation and stored by Dynamo along side the data.

Concurrency Detection

- Example: replicas A,B,C; clients: c1,c2,c3; W=1,R=2
  - **Client c1** executes ***get*** of the initial version of K **in A,B** and executed ***put*** **alpha (op c1-1)** in **A** (empty context).
  - **Client c2** executed ***get*** of the initial version of K **in B,C** and executed ***put beta*** in **B (op c2-1)** (empty context).
  - **Client c3** executes ***get*** **in A,B → detects that A and B have not seen the effects of the last write in each other, therefore these writes are concurrent.**
    - **Get returns a list with values {alpha,beta} and context {c1-1,c2-1}**
  - **Client c3 executes put of delta in any replica (op c3-1)**, **and sends context {c1-1,c2-1} (reconciliation has been achieved).**

Concurrency Detection

- Example: replicas A,B,C; clients: c1,c2,c3; W=1,R=2
  - **Client c1** executes ***get*** of the initial version of K **in A,B** and executed ***put* alpha (op c1-1)** in **A** (empty context).
  - C nd
    executed ***put beta*** in **B (op c2-1)** (empty context).

Is this perfect?

  - **Client c3** executes ***get*** **in A,B → detects that A and B have not seen the effects of the last write in each other, therefore these writes are concurrent.**
    - **Get returns a list with values {alpha,beta} and context {c1-1,c2-1}**
  - **Client c3 executes put of delta in any replica (op c3-1)**, **and sends context {c1-1,c2-1} (reconciliation has been achieved).**

Problem: Scalability

Problem: Scalability

- The solution that we saw works perfectly, but it does present a huge overhead in terms of metadata. **Is there a way to compress this information?**

Problem: Scalability

- The solution that we saw works perfectly, but it does present a huge overhead in terms of metadata. **Is there a way to compress this information?**
- **Intuition**: what if we attribute a sequence number to put operations, and memorize only the most recent one?
    - Requires that only a single replica processes write operations and propagates to the other replicas (we would have to abandon the multi-primary model).
    - This would have efficiency/centralisation complications

Problem: Scalability

- Use of **vector clocks**.
- Each write has associated a clock value, that is composed by an entry for each replica.
- The replica entry has the sequence number of the most recent put operation that was processed by that replica.

Example: Vector Clocks

- Client c1 executes get of the initial value [0,0,0] in A,B and executes put D1 in A (with context [0,0,0])
    - Replica A associates vector clock [1,0,0] to D1

Example: Vector Clocks

- Client c1 executes get of the initial value [0,0,0] in A,B and executes put D1 in A (with context [0,0,0])
  - Replica A associates vector clock [1,0,0] to D1
- Client c1 executes get of version [1,0,0] in A and [0,0,0] in B executing put operation D2 in A (with context [1,0,0])
  - Replica A associates vector clock [2,0,0] to D2

Example: Vector Clocks

- Client c1 executes get of the initial value [0,0,0] in A,B and executes put D1 in A (with context [0,0,0])
  - Replica A associates vector clock [1,0,0] to D1
- Client c1 executes get of version [1,0,0] in A and [0,0,0] in B executing put operation D2 in A (with context [1,0,0])
  - Replica A associates vector clock [2,0,0] to D2
- Client c1 executes get of version [2,0,0] in A and [0,0,0] in B executing put operation D3 in B (with context [2,0,0])
  - Replica B associates vector clock [2,1,0] to D3

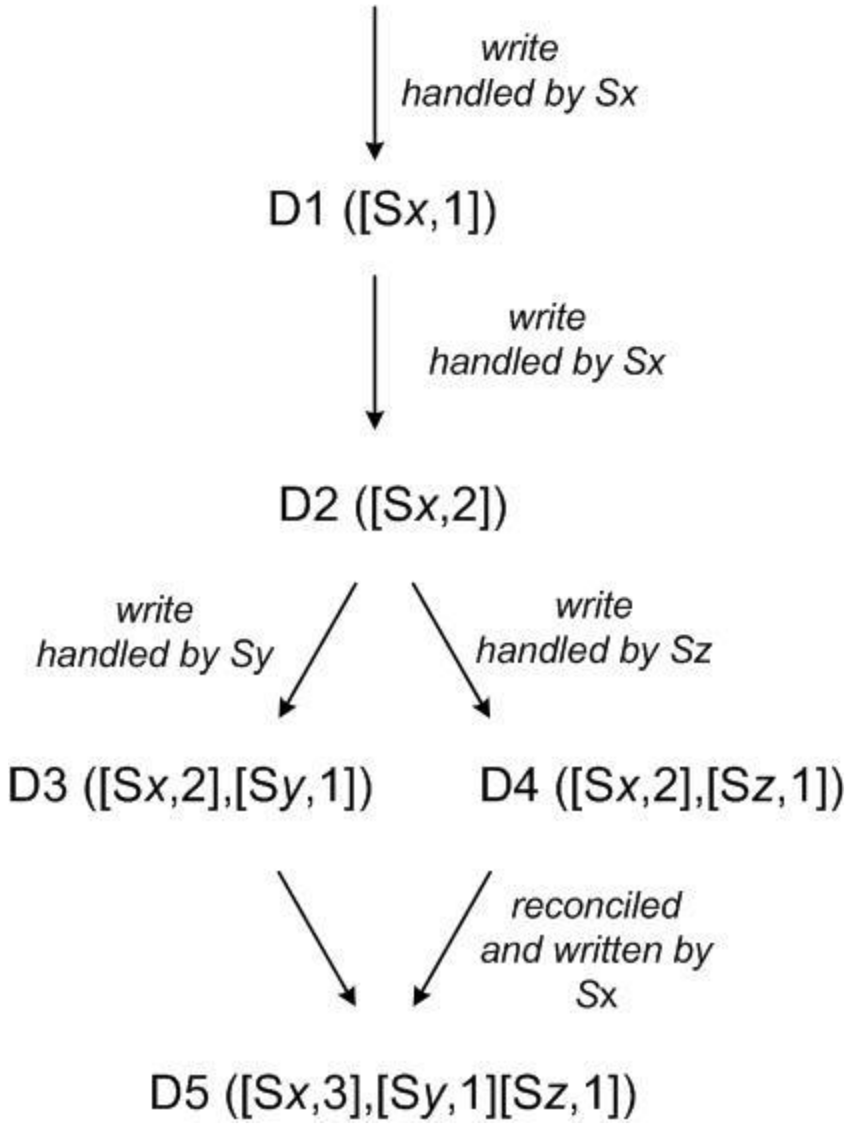
Example: Vector Clocks

- Client c1 executes get of the initial value [0,0,0] in A,B and executes put D1 in A (with context [0,0,0])
  - Replica A associates vector clock [1,0,0] to D1
- Client c1 executes get of version [1,0,0] in A and [0,0,0] in B executing put operation D2 in A (with context [1,0,0])
  - Replica A associates vector clock [2,0,0] to D2
- Client c1 executes get of version [2,0,0] in A and [0,0,0] in B executing put operation D3 in B (with context [2,0,0])
  - Replica B associates vector clock [2,1,0] to D3
- Client c2 executes get of version [2,0,0] in A and [0,0,0] in C executing put operation D4 in C (with context [2,0,0])
  - Replica C associates [2,0,1] to D4

Example: Vector Clocks

- Client c1 executes get of the initial value [0,0,0] in A,B and executes put D1 in A (with context [0,0,0])
    - Replica A associates vector clock [1,0,0] to D1
- Client c1 executes get of version [1,0,0] in A and [0,0,0] in B executing put operation D2 in A (with context [1,0,0])
    - Replica A associates vector clock [2,0,0] to D2
- Client c1 executes get of version [2,0,0] in A and [0,0,0] in B executing put operation D3 in B (with context [2,0,0])
    - Replica B associates vector clock [2,1,0] to D3
- Client c2 executes get of version [2,0,0] in A and [0,0,0] in C executing put operation D4 in C (with context [2,0,0])
    - Replica C associates [2,0,1] to D4
- Client c3 executes get of version [2,1,0] in B and [2,0,1] in C → these vector clocks are incomparable, and hence reflect concurrent writes
    - Get return both {D3,D4} values and context [2,1,1]
    - Client application is responsible for merging these values and executes a put operation of D5 with [2,1,1], on replica A.
    - Replica A associates[3,1,1] to D5

Vector Clocks in Practice…



Fonte: G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store", in the Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, WA, October 2007.

Dynamo: so how do we deal with concurrency?

- **Through the use of vector clocks**: one entry per replica
  - i^{th} entry represents the number of write operations reflected in the state (i.e., accepted) by replica i.
- If two replicas move their vector clocks independently, then both writes are concurrent.
- **Comparing vector clocks R1 & R2:**
  - **R1>=R2 if and only if R1[i]>=R2[i]** for all positions i of the vector clock
- **If neither R1>=R2 nor R2>=R1 then R1 & R2 are incomparable**
  - **Which means that the state of those replicas reflects concurrent (write) operations.**

Problem: Dealing with Divergence

- Vector clocks allow to detect concurrent operation, but how do we make sure that the state converges in these scenarios?

Problem: Dealing with Divergence

- Vector clocks allow to detect concurrent operation, but how do we make sure that the state converges in these scenarios?
- Dynamo: client resolution

Problem: Dealing with Divergence

- Vector clocks allow to detect concurrent operation, but how do we make sure that the state converges in these scenarios?
- Dynamo: client resolution
- Cassandra: "***last writer wins***":
    - **Concurrent updates are ordered in a deterministic way.**
    - **All concurrent updates are lost except one**

Problem: Dealing with Divergence

- Vector clocks allow to detect concurrent operation, but how do we make sure that the state converges in these scenarios?
- Dynamo: client resolution
- Cassandra: "***last writer wins***":
  - Concurrent updates are ordered in a deterministic way.
  - All concurrent updates are lost except one
- Alternative: Conflict-free replicated data types (CRDTs) by N. Preguiça et. al.

Summary: Weak Consistency

- In particular Eventual Consistency
  - Debatable if it conforms to the specification of a consistency model (imposes no restrictions to the observable state of the system by clients).

- Benefits of Eventual Consistency
  - Allows for faster response times for clients (by avoiding server replicas to coordinate with each client operation)
  - Improves availability (broader conditions in which the system is able to process client requests)

Negatives of Eventual Consistency

- The behaviour of the system might be hard to grasp for users/developers.

- In particular, a sequence of reads executed by a client might exhibit state that is not consistent with the evolution of time perceived by the user.

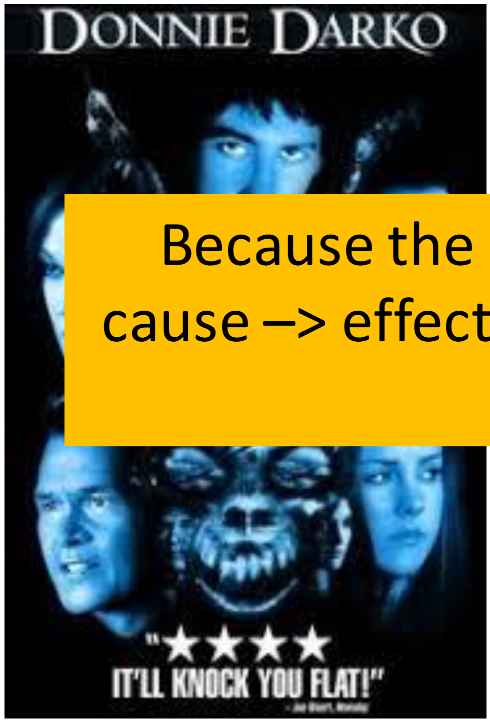
- Why is this bad?
  - Problems with non-commutative operations
  - Usability of system

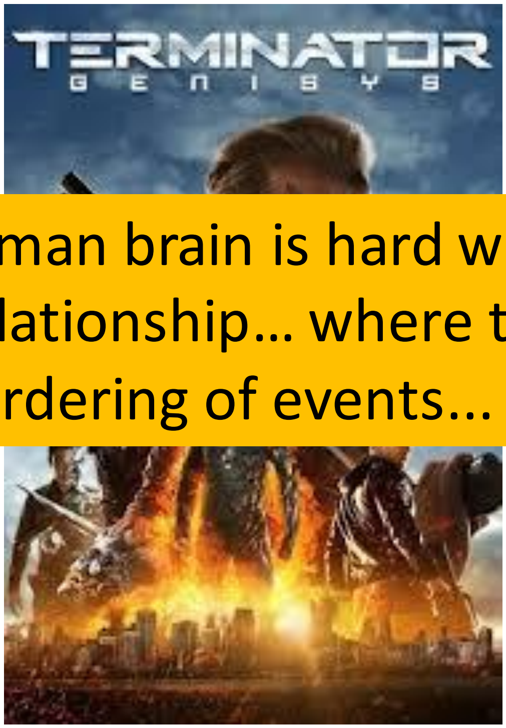
Time should follow in a single direction…

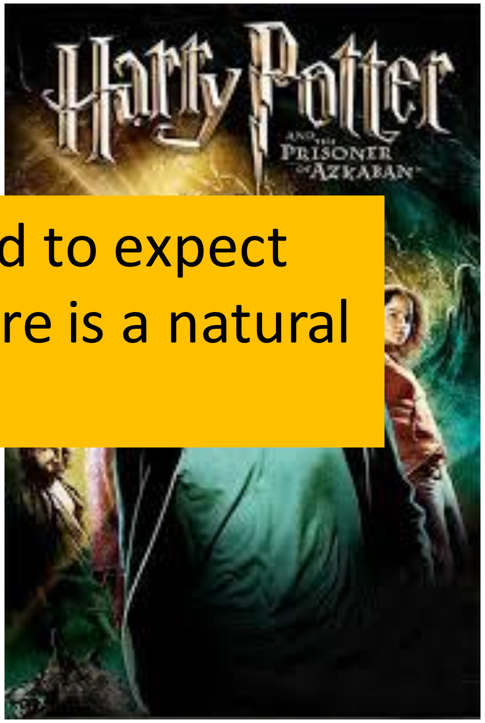
- Why is it that many time travel movies are hard to follow by the audiences?

Time should follow in a single direction…

- Why is it that many time travel movies are hard to follow by the audiences?







Because the human brain is hard wired to expect cause –> effect relationship… where there is a natural ordering of events...