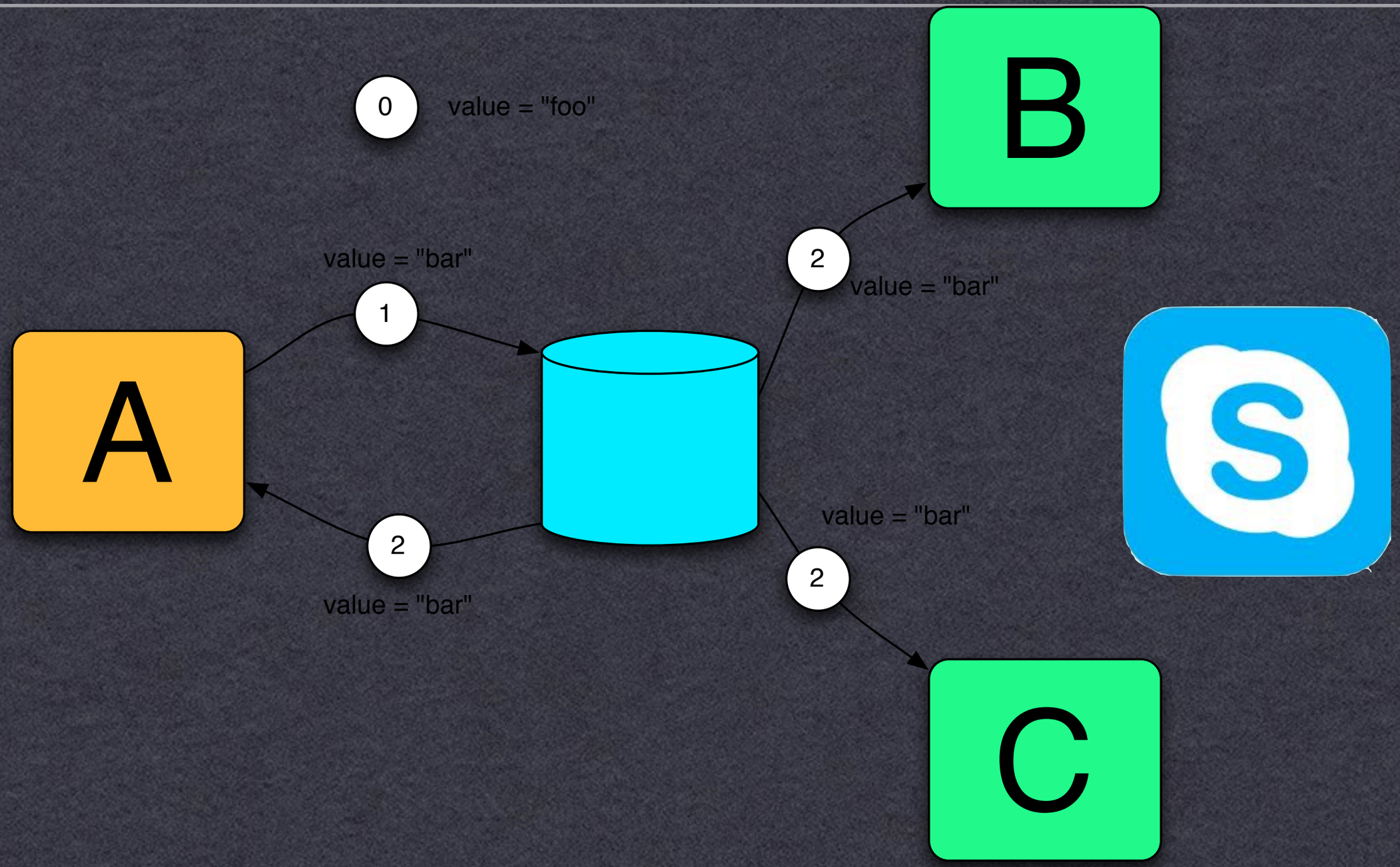


DISTRIBUTED SYSTEMS

CONSISTENCY MODELS

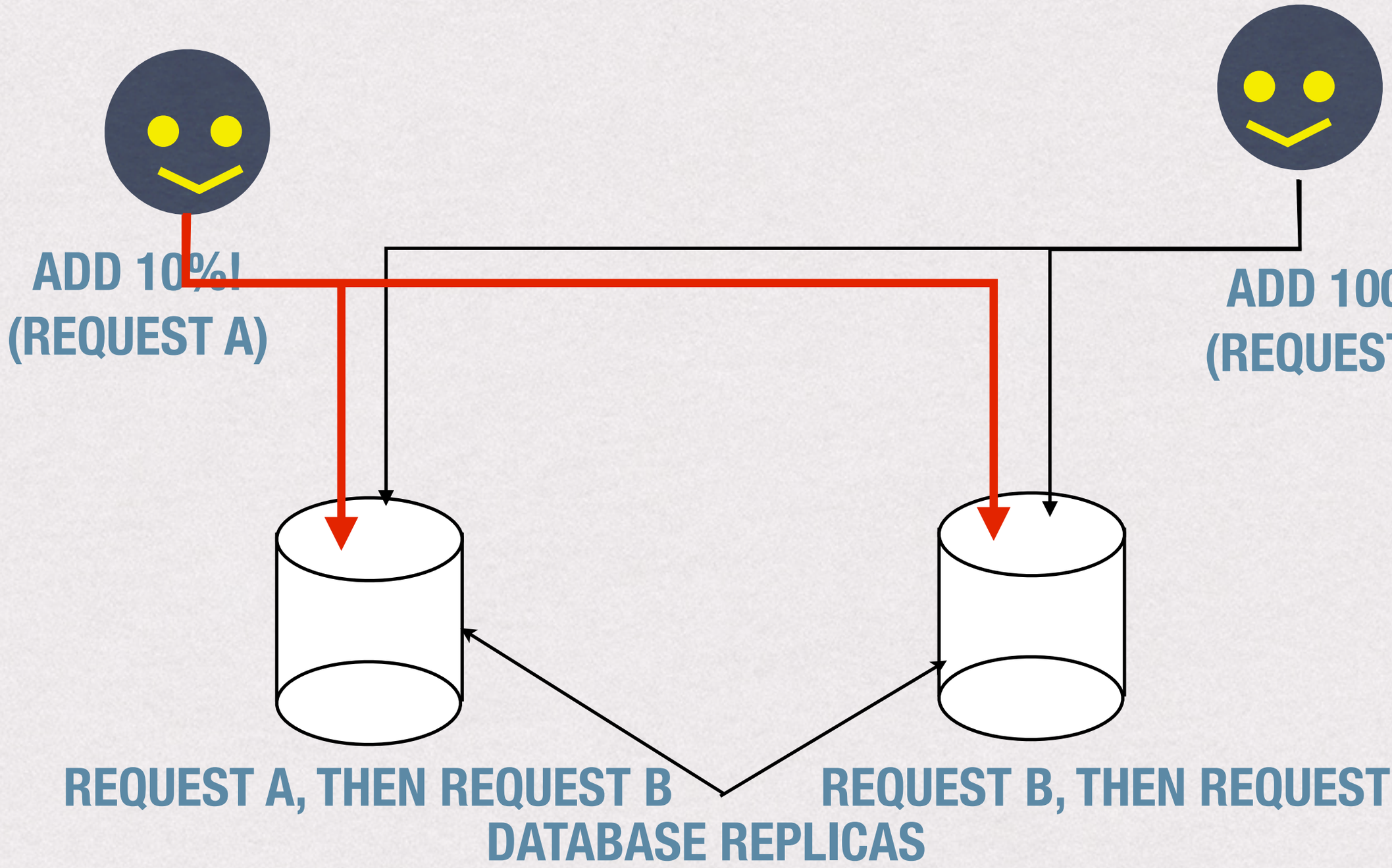
Last week ...

- Introduction to fault tolerance
  - Adaptive timeouts & probes
  - Transactions - 1PC, 2PC, 3PC
- Semantic models
  - at-most-once
  - at-least-once
  - zero-or-once
  - ...

Motivation

- Requirements for a distributed system
  - Security
  - Scalability
  - Availability
  - Performance
- Common solution: replication
  - Tradeoff: high availability and data consistency

Data Consistency



History ...

- 1970s: distribution transparency
  - Better to fail whole system than break transparency
  - An update to the data would ensure all observers saw that update
- 1990s: internet → focus on availability
  - Eric Brewer's CAP theorem: data consistency, system availability, tolerance to network partition (partial failure)
  - Problem: large-scale systems need to be failure tolerant!
  - Must choose from consistency or availability...
  - Solution: relax consistency guarantees

Impossibility Theorems

- CAP - Consistency, Availability, Partitioning
  - we cannot have all three at the same time
- Fischer, Lynch, Paterson (FLP)
  - no consensus is possible in an asynchronous network with unreliable nodes
  - can’t tell difference between message delay and failure

CAP Theorem - Tradeoffs!!

- It is impossible simultaneously to achieve always-on experience (availability) and to ensure that users read the latest written version of a distributed database (consistency) in the presence of partial failure (partitions)

- Maintaining a single-system image in a distributed system has a cost
  - If two processes (or groups of processes) cannot communicate then updates cannot be synchronously propagated to all processes without blocking.
  - Under partitions a system cannot safely complete updates and hence is unavailable to some or all of its users.
  - A system that chooses availability over consistency enjoys benefits of low latency: if a server can safely respond to a user's request when it is partitioned from all other servers, then it can also respond to a user's request without contacting other servers even when it is able to do so.

Design & Implementation Choices ...

- Both options require you to be aware of what the system is offering
- If emphasis is on consistency
  - the system may not be available to take, for example, a write
  - If the write fails because of system unavailability => you will have to deal with what to do with the data to be written.
- If the emphasis is on availability
  - it may always accept the write, but under certain conditions a read will not reflect the result of a recently completed write
  - You will have to decide whether the client requires access to the absolute latest update all the time

ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

Example

- Suppose we have an application requirement that an employee's manager is always paid more than the employee:

  emp.salary < emp.manager.salary

- This is referred to as a ***CONSISTENCY CONSTRAINT***:
  - We need to be able to write applications that can manipulate data whilst maintaining such requirements.
  - In a client/server application, we need to be able to support manipulation of such data by several remote operations executing concurrently.
  - ***Atomic transactions*** provide one way to write such applications.

Example - Incomplete Executions

- Suppose we have the code:
  - emp.salary = emp.salary * 1.1
    if (emp.salary >= emp.manager.salary) {
      emp.manager.salary = emp.manager.salary * 1.1
    }
- Now, if the server crashes after executing the first assignment, but not the if statement, it is possible that the consistency constraint will be violated.
- This situation, where only part of an execution is completed, is termed a failure to achieve ***ATOMICITY***.

Example - Incomplete Executions

- Suppose we have the code:
  - emp.salary = emp.salary * 1.1

- Clearly, since this does not check the manager's salary at all, this code can trivially lead to a violation of the consistency constraint.

- This code can fail to preserve ***CONSISTENCY***, even when it executes to completion.

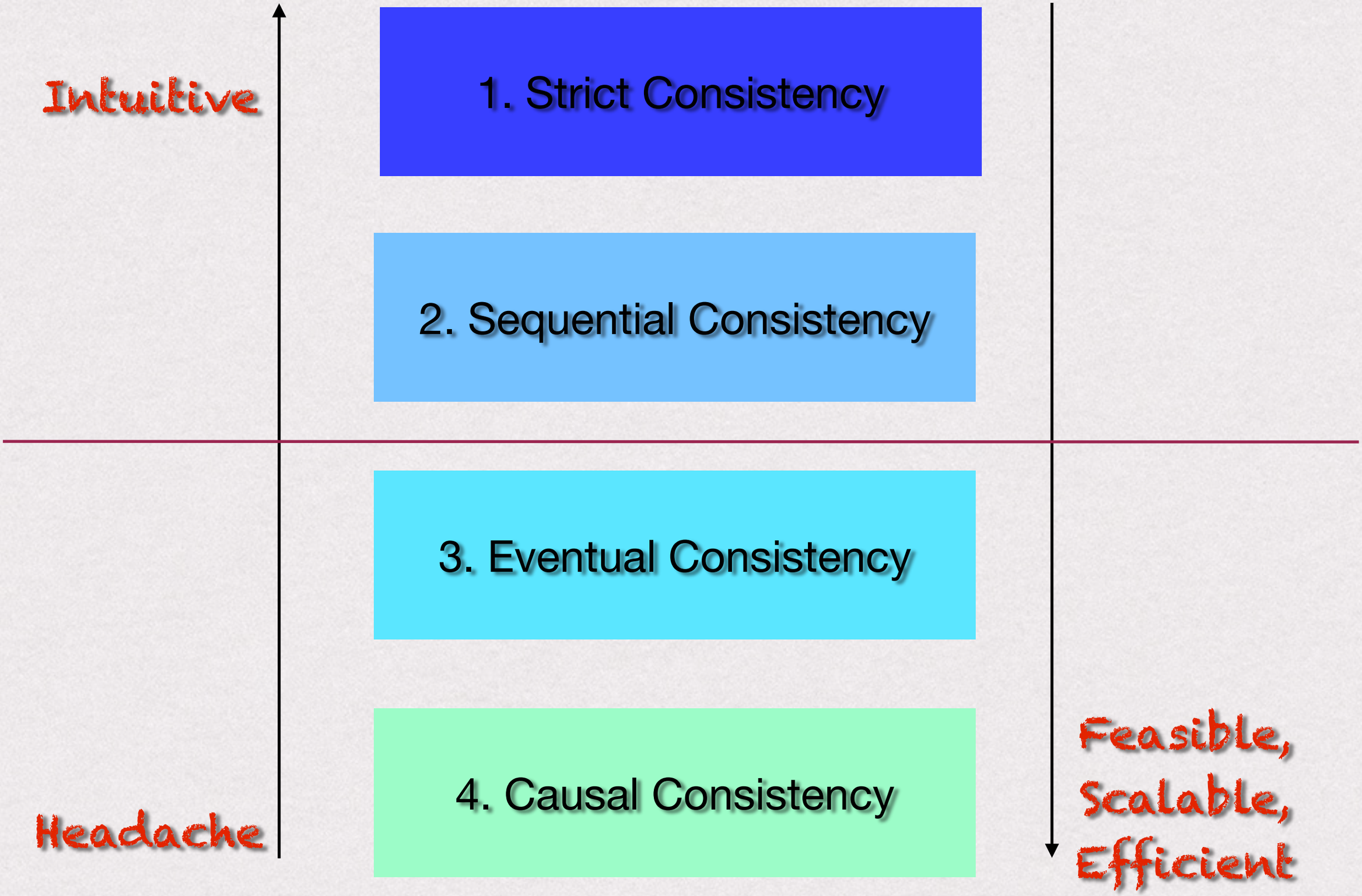
Interference

- Suppose we have two concurrent executions of the code:
  - ```
    emp.salary = emp.salary * 1.1
    if ( emp.salary >= emp.manager.salary ) {
      emp.manager.salary = emp.manager.salary * 1.1
    }
    ```
- Now, consider the following:
  - Execution A executes the first line.
  - Execution B executes the first line.
  - At this point, the salary is 1.21 times its original value.
  - A executes the if statement, and calculates, but does not assign, the new manager salary (1.1 times the original).
  - B executes the if statement, multiplying the manager's salary by 1.1.
  - A completes, assigning 1.1 times the original manager's salary.
- The employees salary has increased by 21%, the manager's by 10%, so the consistency constraint may be violated.
- This can be attributed to a lack of ***ISOLATION*** of concurrent executions:
  - An alternative (non-transactional) analysis identifies a failure of cooperation.
    ```

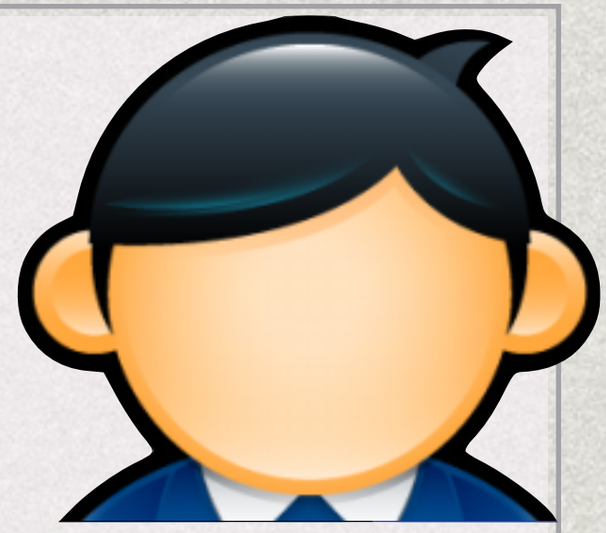
Durability

- Finally, we can't reliably do anything unless we can be sure that at some point an execution has had a permanent effect.
- If we can't guarantee this, then we can't guarantee that our system will ever get anything done in any lasting sense.
- A failure to provide this guarantee is said to be a failure to provide ***DURABILITY***.
  - guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure

Models

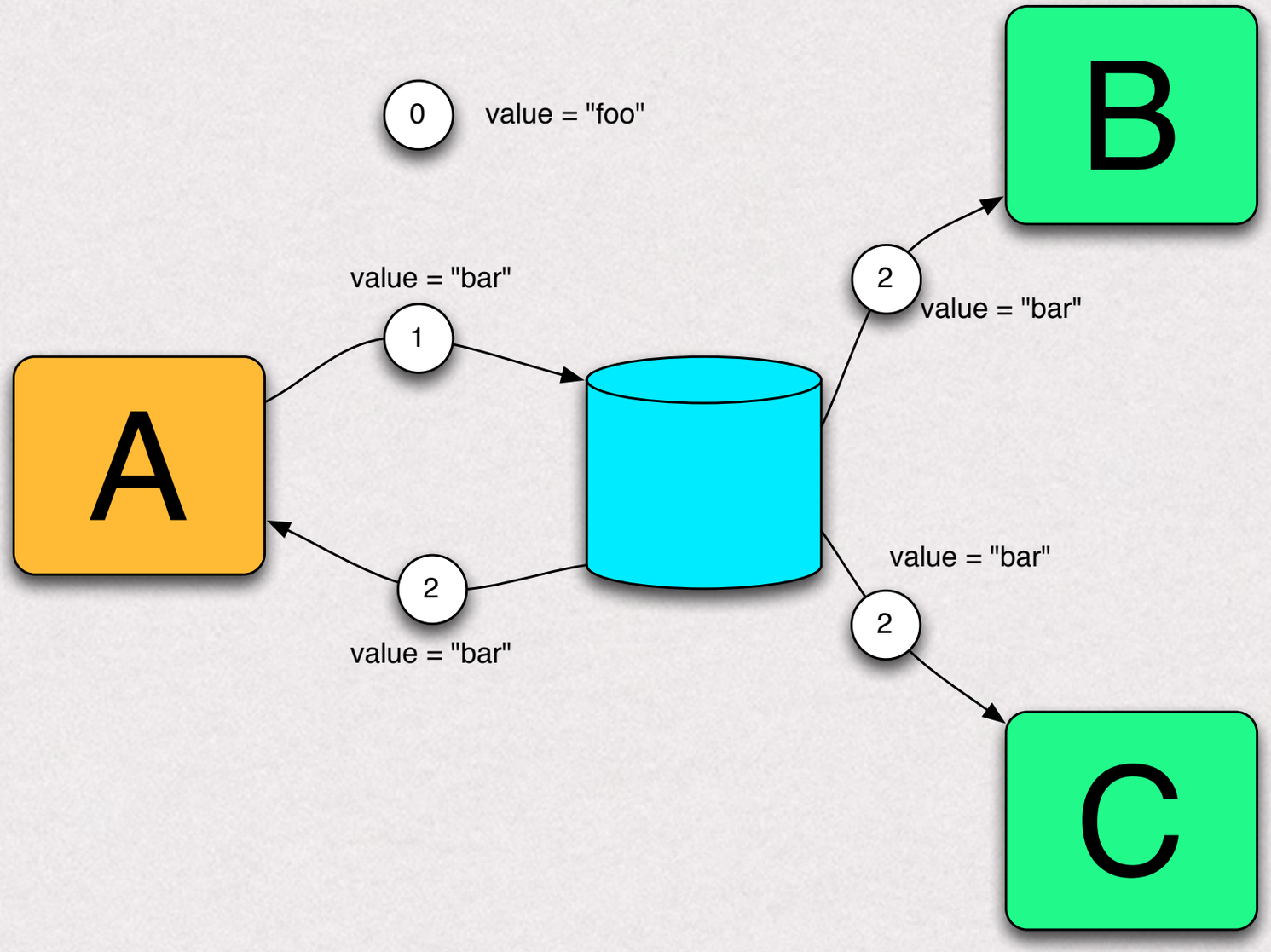

Intuitive
1. Strict Consistency
2. Sequential Consistency
3. Eventual Consistency
4. Causal Consistency
Headache
Feasible,
Scalable,
Efficient

Client perspective ...



- **A storage system:**
  - A black box, but under the covers it is something of large scale and highly distributed, and that it is built to guarantee durability and availability.
- **Process A:**
  - Writes to and reads from the storage system.
- **Processes B and C:**
  - Independent of process A and write to and read from the storage system
  - Need to communicate to share information

1. Strict Consistency



1. Strict Consistency

- Process A has made an update to a data object
- After the update, any subsequent access will return the updated value
- Any execution is the same as if write/read operations were performed in the order of wall-clock time at which they were issued
  - each operation needs to be stamped with the time moment when it was produced

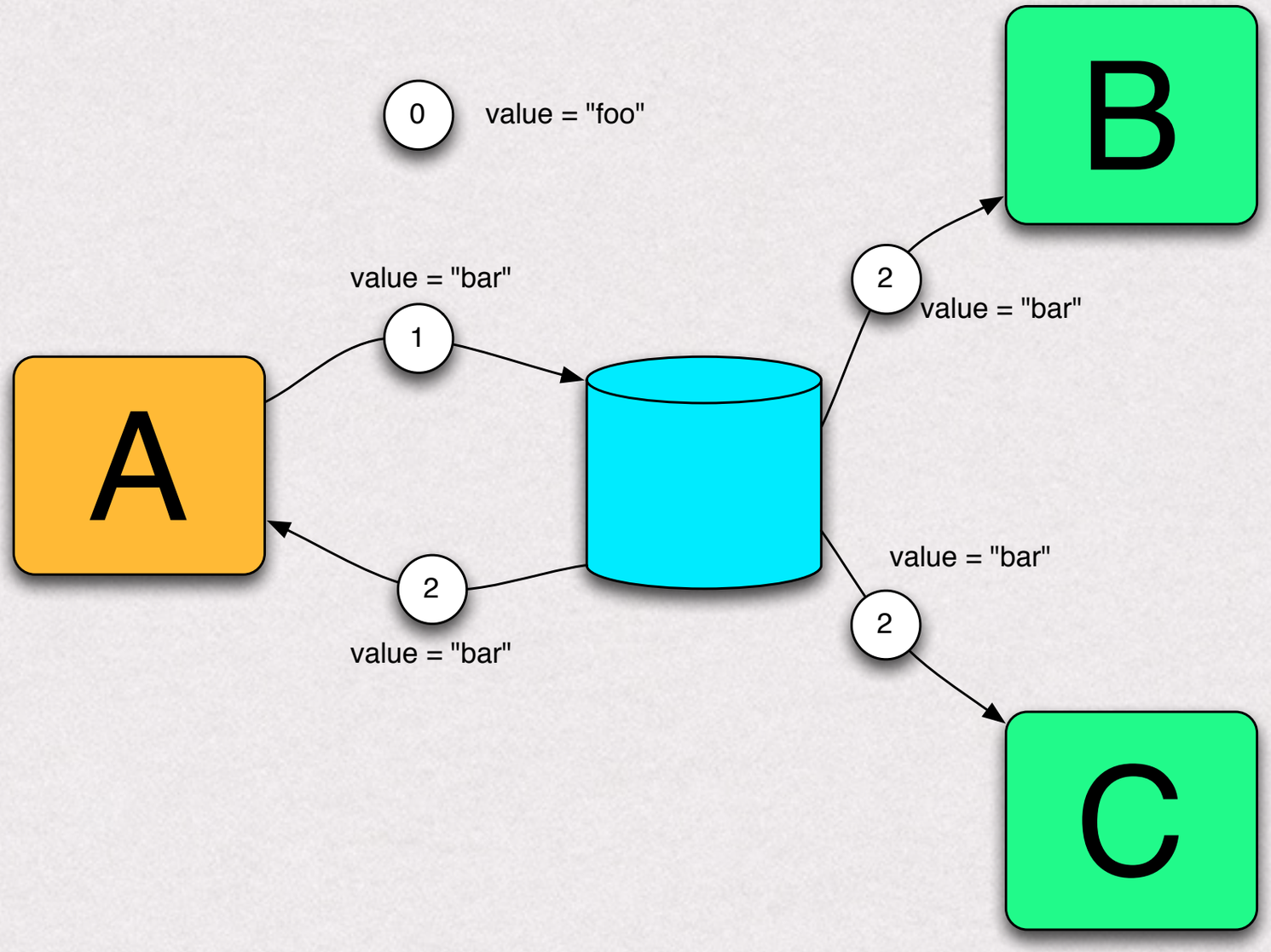
1. Strict Consistency

- Advantages?
- Disadvantages?

1. Strict Consistency

- Advantages?
  - very intuitive
- Disadvantages?
  - difficult to implement

2. Sequential Consistency

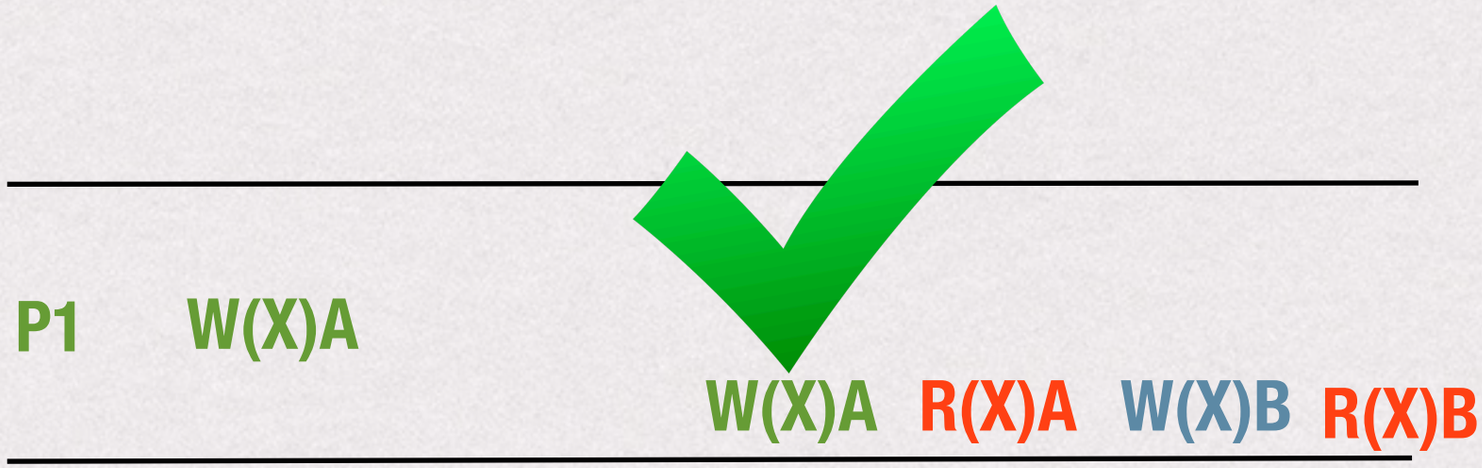


2. Sequential Consistency

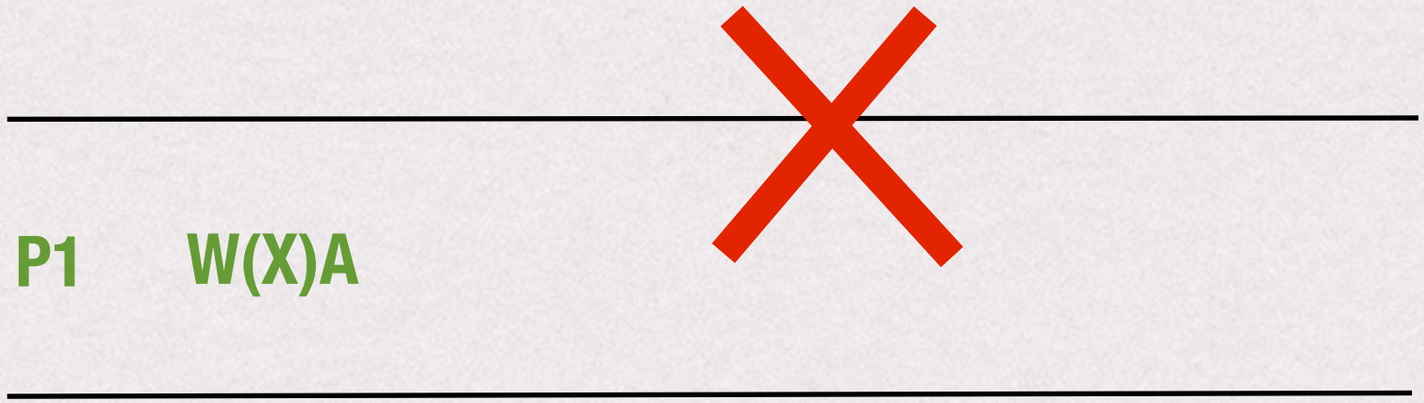
- Process A has made an update to a data object
- After the update, any subsequent access will return the updated value
- Any execution is the same as if write/read operations were performed in a logical order
  - Rule 1: Each machine's own ops appear in order dictated by their program
  - Rule 2: All machines see results according to total order (i.e. reads see most recent writes)
    - reads may be stale according to wall-clock time, but not logical time

2. Sequential Consistency

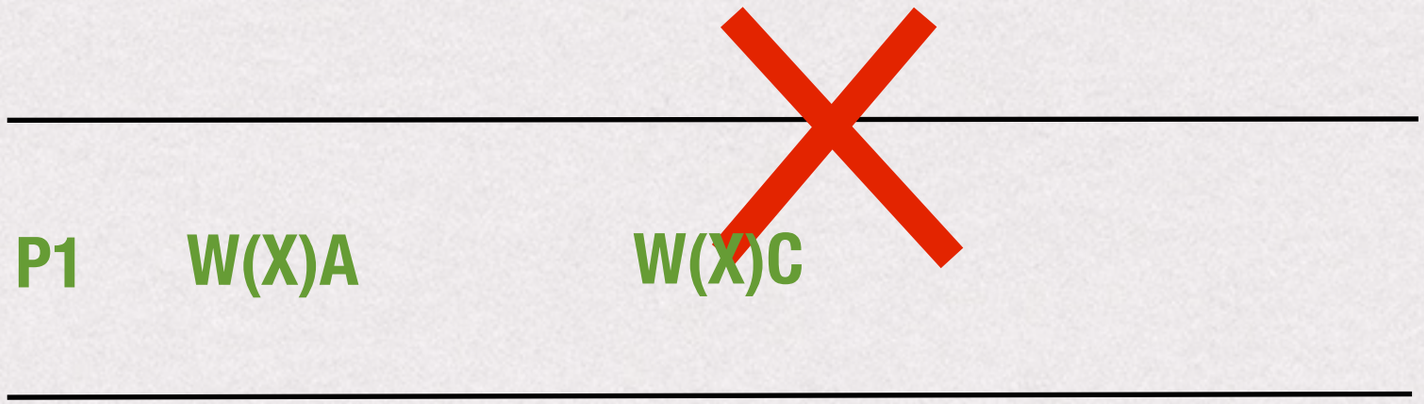
2. Sequential Consistency



2. Sequential Consistency



2. Sequential Consistency



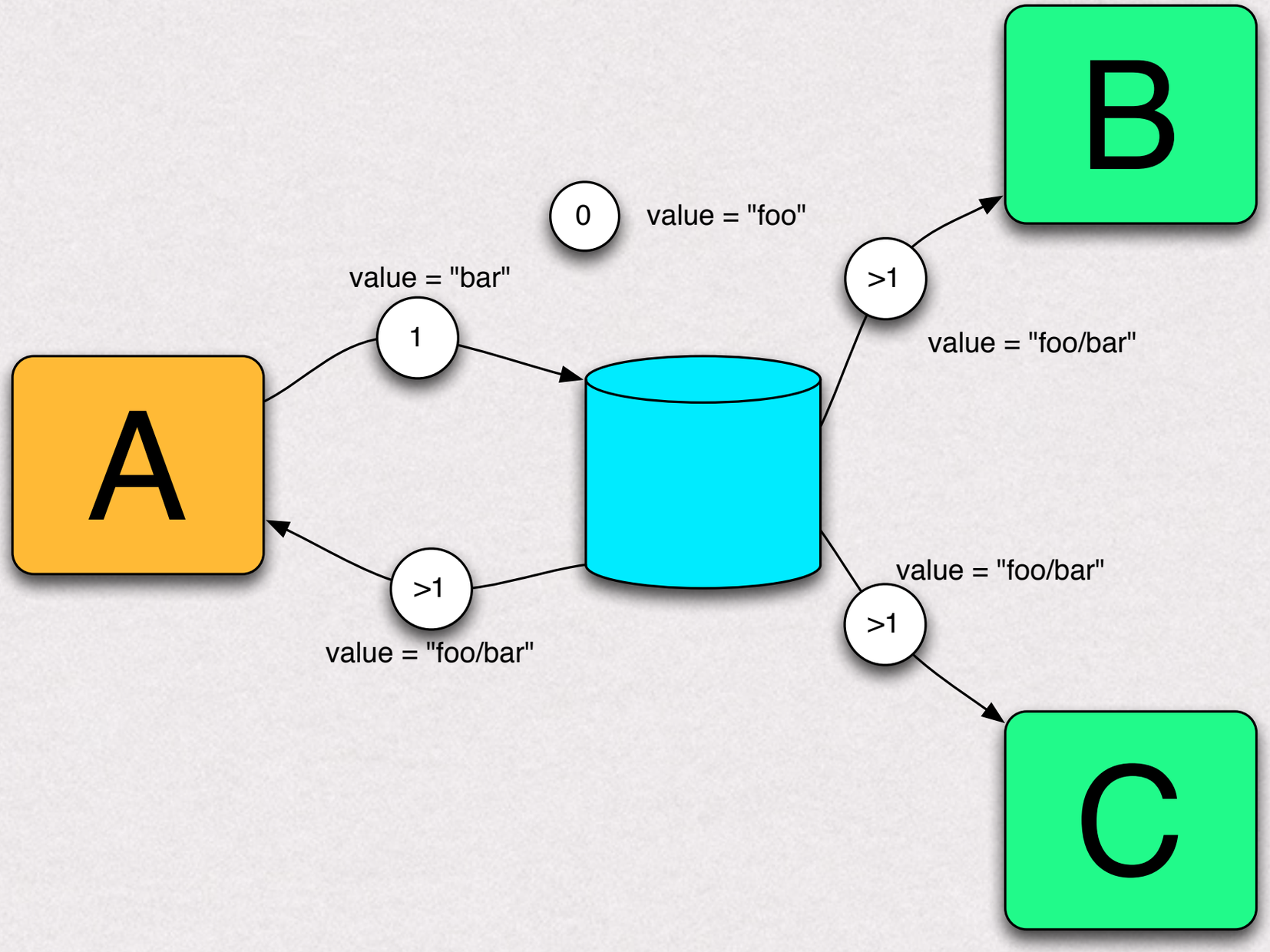
2. Sequential Consistency

- Advantages?
- Disadvantages?

2. Sequential Consistency

- Advantages?
  - still intuitive
  - no notion of real time
  - system has some leeway in how it orders operations
- Disadvantages?
  - difficult to implement
  - once a write completes, other machines reads must see new data [why?] => writes and reads will still be expensive
  - what if you disconnect from the network but still want to edit your shared document?

Weak Consistency



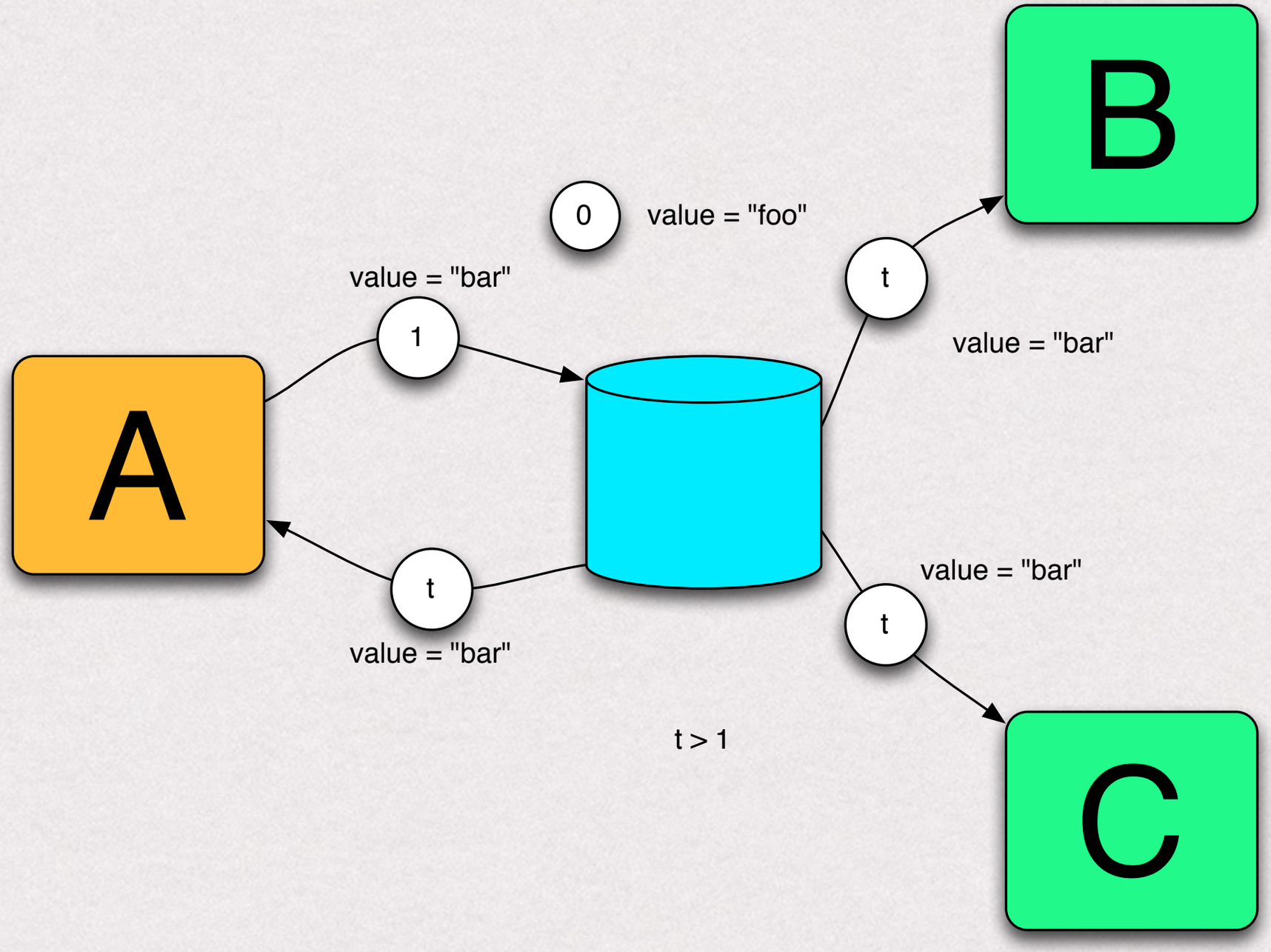
Weak Consistency

- The system does not guarantee that at any given point in the future subsequent access will return the updated value
- A number of conditions need to be met before the value will be returned.
- Inconsistency window:
  - The period between the update and the moment when it is guaranteed that any observer will always see the updated value
  - Measured either in time or in versions (how many versions old will a read be?)

Weak Consistency

- Weak consistency uses synchronization variables to propagate writes to and from a machine at appropriate points:
  - accesses to synchronization variables are sequentially consistent
  - no access to a synchronization variable is allowed until all previous writes have completed in all processors
  - no data access is allowed until all previous accesses to synchronization variables (by the same processor) have been performed

3. Eventual Consistency



3. Eventual Consistency

- A specific form of weak consistency; permits the existence of “stale” reads
- The storage system guarantees that if no new updates are made to the object, *eventually* all accesses will return the last updated value.
- Implemented in
  - DNS (Domain Name System). Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update.
  - Amazon Dynamo - key/value store that is behind Amazon services

3. Eventual Consistency

- If no failures occur, the maximum size of the inconsistency window:
  - communication delays, the load on the system, and the number of replicas involved in the replication scheme.

- If reading from asynchronous replica, inconsistency window = length of log shipment

Why?

- A data infrastructure at a social network where users post new status updates that are sent to their followers' timelines, represented by separate lists—one per user
- The database of timelines is stored across multiple physical servers [why?]
- In the event of a partition between two servers, however, you cannot deliver each update to all timelines
  - Should you tell the user that s/he cannot post an update, or should you wait until the partition heals before providing a response?
  - Both of these strategies choose consistency over availability, at the cost of user experience.

Why? (2)

- You propagate the update to the reachable set of followers' timelines, return to the user, and delay delivering the update to the other followers until the partition heals
  - No guarantee that all users see the same set of updates at every point in time (and admit the possibility of timeline reordering as partitions heal),
  - But you gain high availability and (arguably) a better user experience
  - Because updates are eventually delivered, all users eventually see the same timeline with all of the updates that users posted

3. Eventual Consistency

- Advantages?
- Disadvantages?

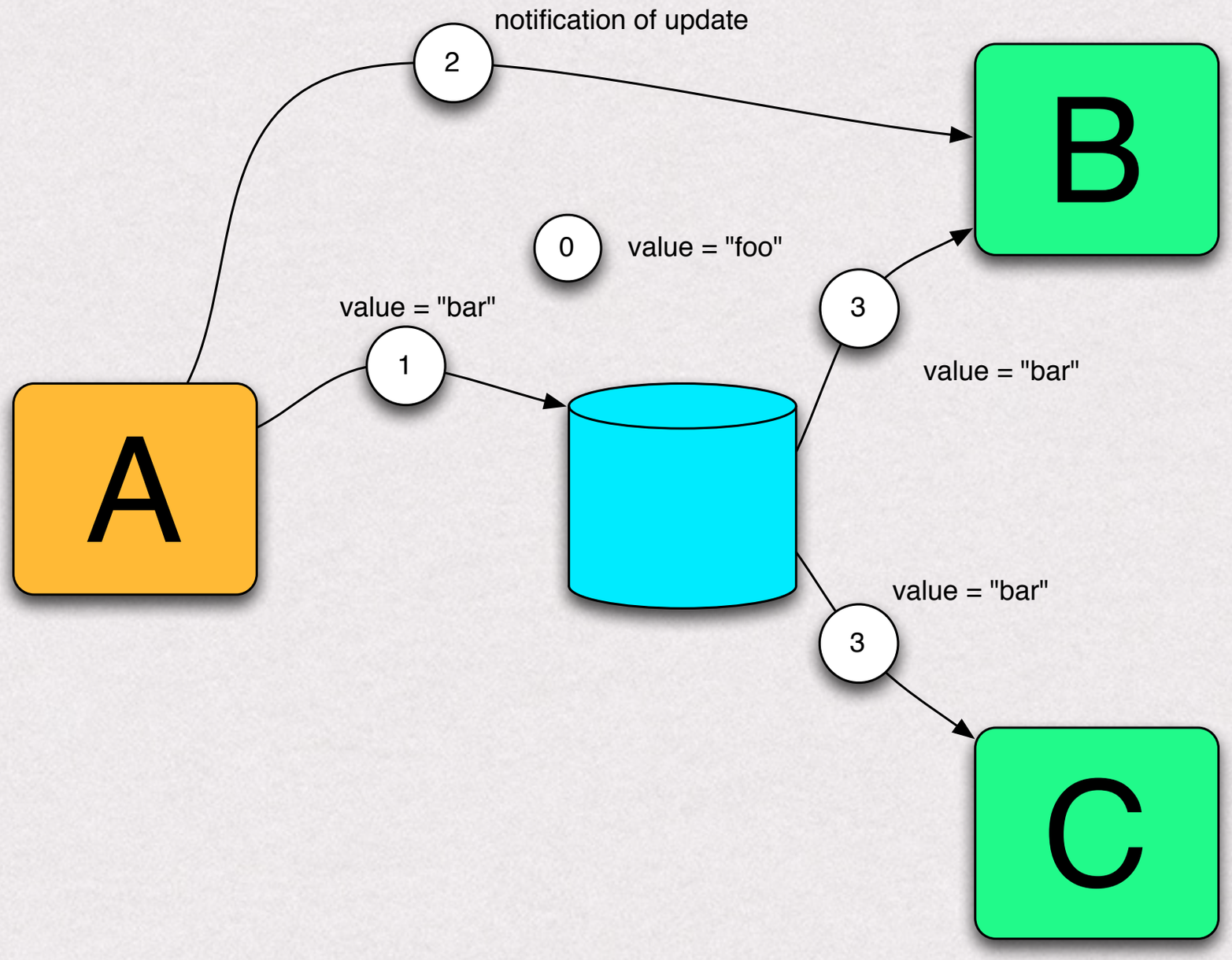
3. Eventual Consistency

- Advantages?
  - more implementation efficient - less chatty
  - stale values are available (better than nothing ...)
- Disadvantages?
  - Stale reads - need to compensate when writing application
  - Conflicting writes

3. Eventual Consistency

- Replicas must exchange information about which writes they have seen
- Can use an asynchronous all-to-all broadcast: when a replica receives a write to a data item:
  - it immediately responds to the user
  - in the background, sends the write to all other replicas, which in turn update their locally stored data items
- In the event of concurrent writes to a given data item, replicas deterministically choose a "winning" value

4. Causal Consistency



4. Causal Consistency

- Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality
- All concurrent operations might be seen in different orders
- Reads are “fresh” only with respect to the writes they are dependent on
- Only causally-related writes are ordered by all replicas in the same way
- Concurrent writes may be committed in different orders by different replicas, and hence read in different orders by different applications

4. Causal Consistency

4. Causal Consistency

- Advantages?
- Disadvantages?

4. Causal Consistency

- Advantages?
  - better performance because of concurrency
- Disadvantages?
  - not intuitive at all

Summary

- Whether inconsistencies are acceptable depends on the client application
- You need to be aware what consistency guarantees are provided by the storage system vs what your application needs to implement
- In a web application:
  - we can have the notion of user-perceived consistency
  - the inconsistency window needs to be smaller than the time expected for the customer to return for the next page load
  - this allows for updates to propagate through the system before the next read is expected.

Client consistency

- Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.
  - At location A you access the database doing reads and updates.
  - At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
- Your updates at A may not have yet been propagated to B
- You may be reading newer entries than the ones available at A
- Your updates at B may eventually conflict with those at A
- The only thing you really want is that the entries you updated and/or read at A, are in B the way you left them in A. In that case, the database will appear to be consistent to you.

- Questions?