Programação em Sistemas Distribuídos

MEI-MI-MSI

2018/19

2. Distributed Systems Paradigms

Prof. António Casimiro

Concurrency and Atomicity

Register Consistency Models *[Lamport:86]*

- **Register** is a kind of abstract data type (ADT):
  - A single value that can be read and written
- Safe register:
  - **Read not concurrent with any write returns the most recent write**
  - Read concurrent with some writes returns any value $\Rightarrow$ not very meaningful…
- Regular register :
  - **Read not concurrent with any write returns the most recent write**
  - Read concurrent with some writes returns a value of the concurrent writes (including the most recent write)
- Atomic register:
  - **Every read returns the value of the most recent write**

Sequential Consistency

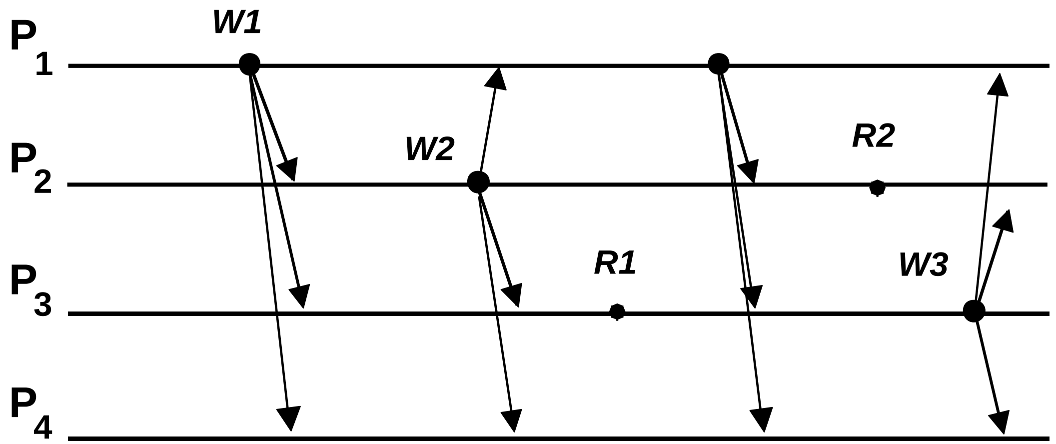
Of shared and/or replicated data registers *[Lamport:79]*

- When registers are concurrently accessed and possibly replicated (caches, replicas, DSM), what are the useful consistency criteria?
- **Sequential consistency**
  - The result of any execution is the same as if all operations were executed in some sequential order, and
  - The operations of each individual process occur, in this sequence, in the order specified by its program
  - I.e.: history H is sequentially consistent if it is equivalent to some legal sequential history S that preserves process order
- Requirements:
  - **Write atomicity**: all writes (to any location) should appear to all processors to have occurred in the same order
  - **Program order requirement**: memory operations of a process must appear (to itself and others) in program order (FIFO)

Atomic Consistency

Of replicated data registers *[Lamport 86, Herlihy 90]*

- An obvious approach is to mimic a single register
- **Atomic consistency**, a.k.a. **Strict Consistency**, a.k.a. **Linearizability**
  - Every read to a data x returns the value most recently written to x
  - Writes and reads are ordered according to the physical (real time) order



Consistency in executions

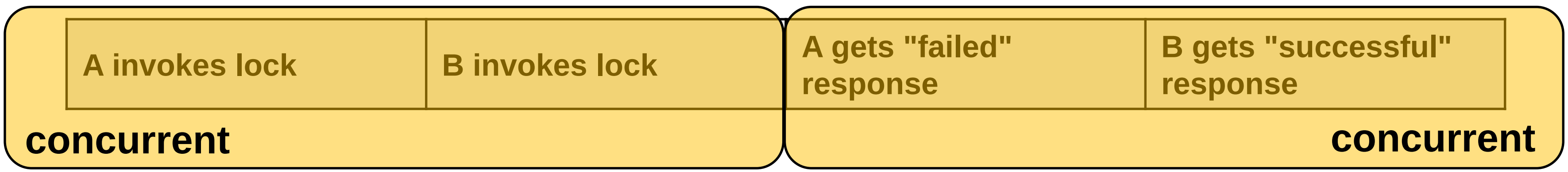
Sequential vs. concurrent

- Consider **history** is a sequence of invocations and responses made of an object by a set of threads
- Each invocation of a function will have a subsequent response
- A **sequential history** is one in which all invocations have immediate responses
- Individual concurrent accesses to objects may be managed by **total ordering (atomic consistency)**
- To keep it simple, if threads do not communicate, one may opt by sender-based total order, which is nothing else than..... **FIFO! (sequential consistency)**

Consistency in executions

Concurrency Control

- Concurrency disturbs our reasoning about “sequential”
- Multiple concurrent accesses to objects must be encapsulated to manage concurrency, e.g. by **mutual exclusion mechanisms** (critical sections)



- Long-lasting multiple accesses (sequential transactions) take parallelism away, so they should interpenetrate when they have little conflict (concurrent transactions)
- But if conflict is not well-managed, interpenetration may get you into a mess!

Consistency in executions

Making concurrent executions sequential

- Concurrency disturbs our reasoning about “sequential”

A invokes lock	B invokes lock	A gets "failed" response	B gets "successful" response
concurrent			concurrent

- This reordering of execution is not seq. consistent, because it is sequential but violates program order (initially lock is free):

A invokes lock	**A gets "failed" response**	**B invokes lock**	**B gets "successful" response**

- This reordering of concurrent execution is seq. consistent, and more, linearizable (because it also preserves temporal order between requests and responses):

B invokes lock	**B gets "successful" response**	**A invokes lock**	**A gets "failed" response**

Serializability

- Serializability
  - Property of a set of concurrent transactions executing simultaneously (concurrently) whereby the end result is the same that would be achieved by at least one sequential execution of the same transactions
  - **Serializability is like sequential consistency, except that operations may access an arbitrary number of objects**
- How to enforce?
  - Concurrency control algorithms
- 2-phase lock
  - Two-phase lock, 2PL
  - Acquire, release lock
  - One write, multiple reads, i.e. only write locks are exclusive of a transaction, read locks can be shared

Serializability
Linearizability vs. Serializability

- **Serializability** is a global property
  – A property of an entire history of operations/transactions
- **Linearizability** is a local property
  – A property of a single operation/transaction flow
- Every linearizable execution is sequentially consistent, but not the converse
  - **This history is not linearizable (in temporal order, A and B have the lock simultaneously at the beginning):**

A invokes lock	A successfully locks	B invokes unlock	B successfully unlocks	A invokes unlock	A successfully unlocks
	B holds lock				

  - **But is serializable (see the reordering):**

B invokes unlock	B successfully unlocks	A invokes lock	A successfully locks	A invokes unlock	A successfully unlocks

Serializability

Linearizability vs. Serializability

- This history is not linearizable (in temporal order, A reads 0 from x, after setting it to 1 !):

A **setBal(x,1)**	**A** **getBal(y)⟶0**	**A** **getBal(x)⟶0**	**A** **setBal(y,2)**

- But it is serializable (see that the reordering is sequentially consistent):

A **getBal(y)⟶0**	**A** **getBal(x)⟶0**	**A** **setBal(x,1)**	**A** **setBal(y,2)**

- Now, is sequential consistency universally useful?
- What if “Bal” above is a bank balance?
- And the bank reorders your salary deposit to after you tried to make a withdrawal?

Serializability

Replication: 1-copy Serializability

- Back to concurrent access to objects, if these are replicated, one will like to keep consistency among replicas (**one-copy equivalence**)
- In the case of replicated transactions, the above implies preserving serializability, and the same serializability in all copies (**one-copy serializability**)
- Problem is ... what are the good algorithms to prevent conflicting copy updates?
- It can get worse.... think about what happens when partitions occur
- How can one make non-conflicting concurrent updates?
- What about **majority partition**?...

Atomicity

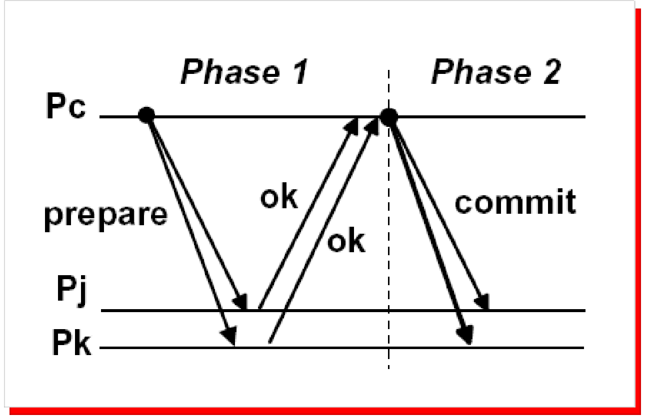
- **Atomicity** is the property of an indivisible operation
- **Transactional atomicity** is that property extended to a set of operations that are made **look like** indivisible
- **Atomic transaction** is an operation exhibiting that property
- Several techniques concur to achieve it:
  - Either all operations are performed, or the whole transaction is aborted
  - Intermediate results cannot be seen before the end
  - Results must be stored in non-volatile memory to be persistent

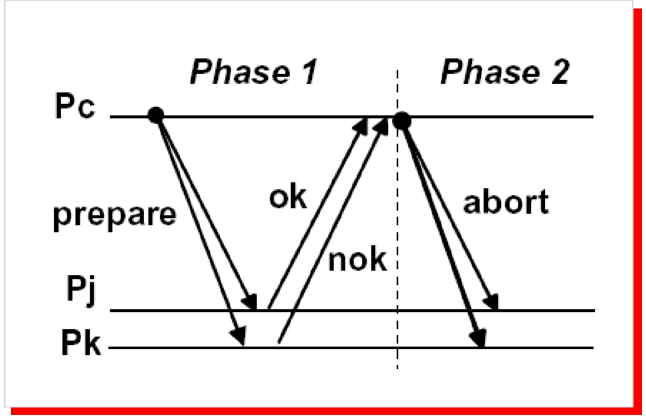
Distributed Atomicity

- How to ensure atomicity of transactions that run across several nodes?
- **Problem**:
  - Partial failure of nodes, and partitions, leading to inconsistent termination
- **Solution**:
  - Distributed atomic commitment
  - Most used protocol: **two-phase commit (2PC)**

2-phase Distributed Atomic Commitment

- Two-phase commit
  - commit
  - abort
- Problem:
  - Subject to blocking
  - Think about blocking scenarios





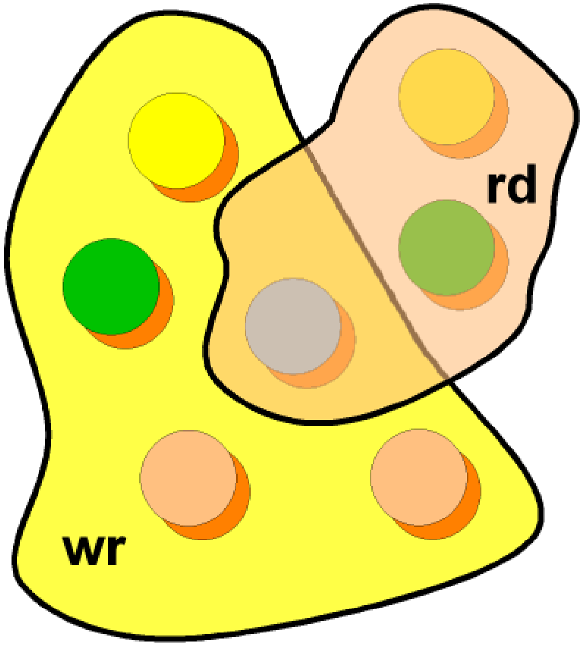
Consistency in Large-scale and/or Partitionable Systems

Problem

- Partitions cannot be detected accurately in asynchronous systems or systems with poor synchrony
  - E.g. they may appear to come and go too frequently
- Partitions can happen in bad ways by accidents or by manipulation under malicious attacks
  - E.g. primary partition may not work if they lead to all minority partitions
- Need weaker predicates

Voting and Quorums
Partitionable applications

- An operation is considered executed if a **minimum quorum** of replicas performed it, so that consistency can be maintained
- Quorum formation rules:
  - **Two conflicting operations must always intersect in at least one replica**
  - This common replica ensures the outcome of the first operation is available to all replicas executing the next operation
  - Most recent state is identified by version numbers incremented by each replica upon an update



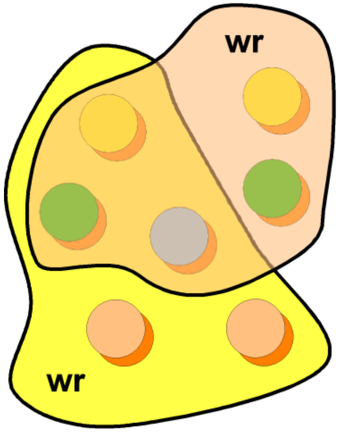
Quorum algorithms
Weighted and non-weighted voting

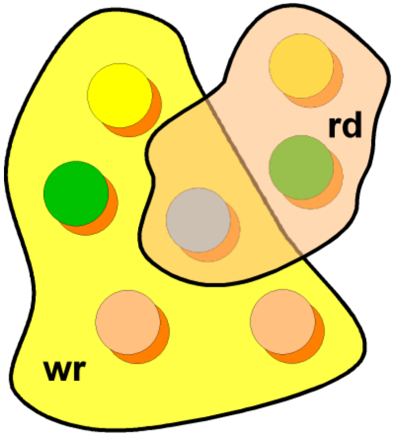
- The system has a total of ***n*** copies of a register
- Weighted quorums:
  - Each copy is assigned a **number of votes**, quorums are defined based on the number of votes instead of the number of replicas
  - **Intuition for votes**: you can give more importance to registers residing on replicas which you deem more available
- Simplifying: non-Weighted quorums:
  - Let total number of votes be n, 1 per replica
  - Writes are executed on a quorum of w replicas
  - Reads are executed on a quorum of r replicas
- Overlapping guarantee rule: **2w > n** and **w + r > n**
- **Why?**
  - Sum of quorums for conflicting operations on an item should exceed the total number of votes for that item (**to yield a common replica**)

Quorum algorithms
Non-weighted voting example

- **2w > n**
  - Suppose n = 7, w = 5 and r = 3
  - Write to quorum of at least 5 replicas
  - 2 replicas left, not enough for a divergent write
- **w + r > n**
  - Reads and writes to the same register, different partitions, are serialized
  - E.g.: write occurs first (5 replicas), so read must wait (2 replicas left, read needs 3 replicas)
  - Read is sure to include at least one of the replicas that have seen previous write
  - This replica can update the others, ensuring sequential consistency of the history of operations

n=7, w=5, r=3





CAP Theorem
[Fox&Brewer]

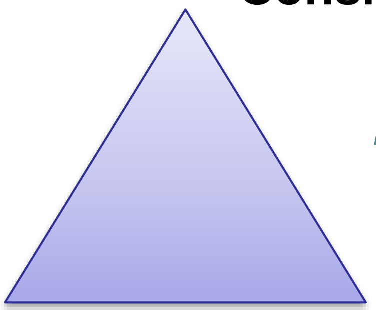
Ciências ULisboa

- **Claim**: every distributed system is on one side of the triangle, can only have simultaneously two of three properties: Consistency; Availability; Partitionability
- C-A-P: choose two.

C **Consistency**

***CA**: available, and consistent, unless there is a partition*

***CP**: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum)*



A **Availability**

P **Partitionability**

***AP**: a reachable replica provides service even in a partition, but may be inconsistent if there is a failure*

CAP, BASE and large-scale systems

- The types of large systems based on CAP are not ACID (strong consistency) they are BASE (weak consistency):
  - **Basically Available** - system seems to work all the time
  - **Soft State** - it doesn't have to be consistent all the time
  - **Eventually Consistent** - becomes consistent at some later time
- Current large-scale applications are built on CAP and BASE: Google, Yahoo, Facebook, Amazon, eBay, etc

Weak and Eventual Consistency

- Strong consistency drawbacks
  - Atomic and sequential consistency are faithful to the programmer's view, but performance suffers in distributed/replicated data
- Weak consistency
  - The system does not guarantee that immediately subsequent accesses will return the updated value
  - A number of conditions need to be met before the value will be returned.
  - Inconsistency window - period between the update and the moment when it is guaranteed that any observer will always see the updated value
- **Eventual consistency** (form of weak consistency)
  - If no new updates are made to the object, eventually all accesses will return the last updated value
  - If no failures occur, the maximum size of the inconsistency window can be determined (function. of communication delays, load, number of replicas)