Transactions – *locks*
*the **ACID*** properties

- **A**tomicy
- **C**onsistency
- **I**solation
- **D**urability

Conceito fundamental do processo de *locking*

- Exclusive access to data elements (critical resource) - lock
  - Each transaction promotes exclusive access to data elements
- Depending on the access mode, the locks can be:
  - Read locks (also known as Shared locks)
  - Write locks (also known as Exclusive locks)
- Access to a data element
  - Before a read, the transaction sets a ReadLock
  - Before a write, the transaction sets a WriteLock
- Conflicts
  - A ReadLocks establishes a conflict with WriteLocks
  - WriteLocks establishes conflict with ReadLocks and WriteLocks

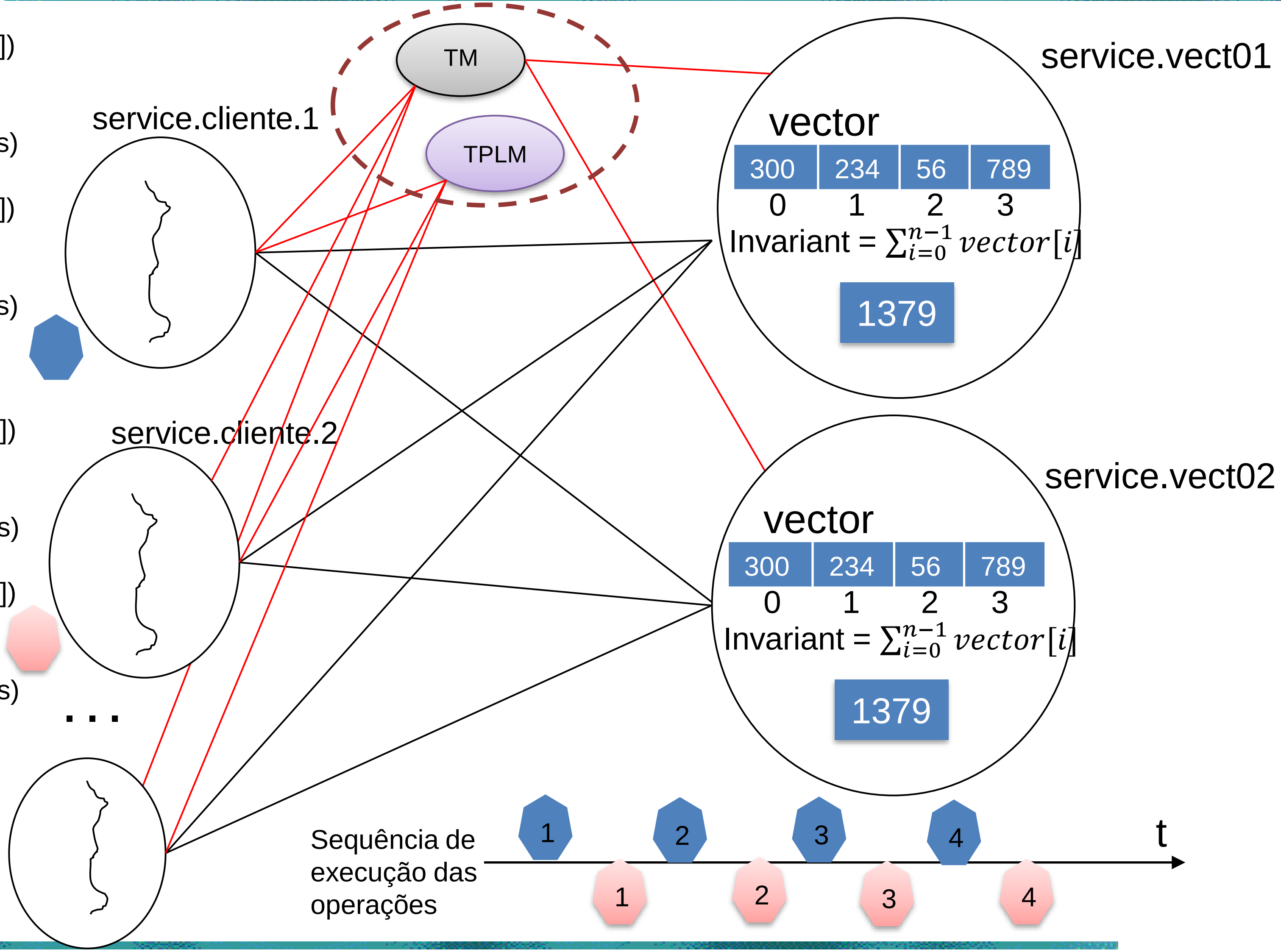
Fundamental concept of a locking process (1)

- Obtaining the locks to be associated to a transaction
  - Only if no conflict locks in relation to a same data element;
    - Can obtain a ReadLock to access X (data element) if no other transaction has a WriteLock on X (data element)
    - Can obtain a WriteLock on X (data element) if no other transaction has a ReadLock or a WriteLock on X
- Impacts on performance (parallelism)
- A lock set by a transaction
  - Delays other transactions that establishes conflict locks
- Deadlocks
  - Requires strategies to define data model structure, and coordination policies, being fundamental aspects
    - Locking “granularity”

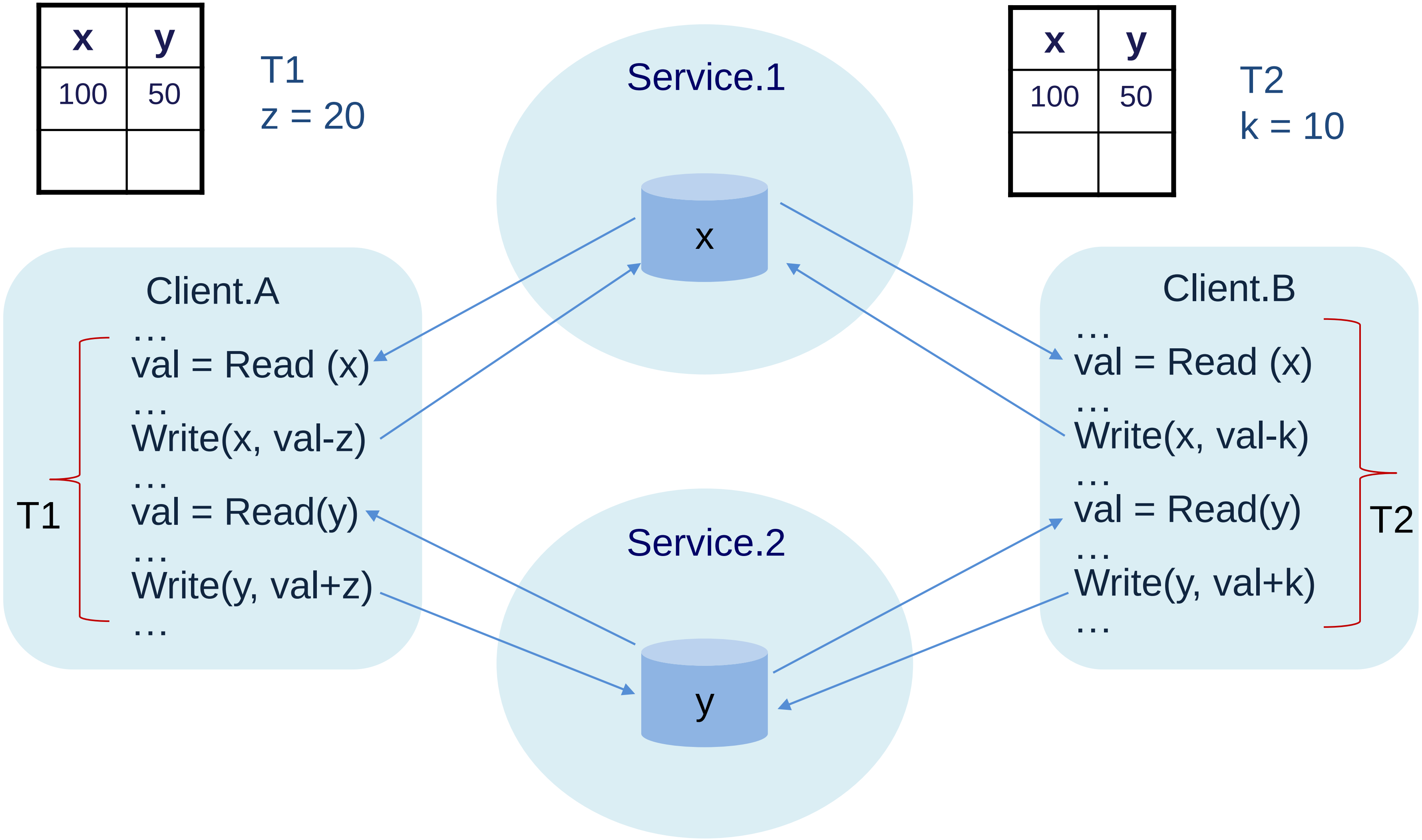
Serviços concorrentes no acesso a múltiplos ser.vectxx

1. v = read(vect01.vect[0])
res = v – x // x =100
Espera(10 segundos)
2. write(vect01.vec[0], res)
Espera(10 segundos)
3. v = read(vect02.vect[2])
res = v + x
Espera(10 segundos)
4. write(vect02.vec[2], res)

1. v = read(vect01.vect[0])
res = v – x // x =100
Espera(10 segundos)
2. write(vect01.vec[0], res)
Espera(10 segundos)
3. v = read(vect02.vect[2])
res = v + x
Espera(10 segundos)
4. write(vect02.vec[2], res)



Scenario with two data elements, x and y



Concurrent execution of T1 and T2 on x and y elements

(Concurrent execution of T_1 and T_2) E =

$\underbrace{rl_1[x]\ r_1[x]\ ru_1[x]}_{T_1 \text{ reads x …}}$

$\underbrace{rl_2[y]\ r_2[y]\ ru_2[y]\ rl_2[x]\ r_2[x]\ ru_2[x]\ wl_2[y]\ w_2[y+k]\ wu_2[y]\ wl_2[x]\ w_2[x-k]\ wu_2[x]}_{T_2 \text{ reads and writes y and x}}$

$\underbrace{wl_1[y]\ r_1[y]\ w_1[y+z]\ wu_1[y]\ wl_1[x]\ w_1[x-z]\ wu_1[x]}_{\text{…}T_1 \text{ reads y and writes y and x}}$

Serialized execution?

Demonstrating the violation of the Serializable rule

Removing the locks and unlock from the execution of E we obtain E'

$E' = r_1[x]\ r_2[y]\ r_2[x]\ w_2[x\text{-}k]\ w_2[y\text{+}k]\ r_1[y]\ w_1[y\text{+}z]\ w_1[x\text{-}z]$

x	y
100	50

Invariant = x+y (150);

T1 z = 20
T2 k = 10

Serialized execution?

The two possible serialized executions:

$T_1T_2 = r_1[x]\ r_1[y]\ w_1[y\text{+}z]\ w_1[x\text{-}z]\ r_2[y]\ r_2[x]\ w_2[x\text{-}k]\ w_2[y\text{+}k]$

x	y
100	50

$T_2T_1 = r_2[y]\ r_2[x]\ w_2[x\text{-}k]\ w_2[y\text{+}k]\ r_1[x]\ w_1[x\text{-}z]\ r_1[y]\ w_1[y\text{+}z]$

x	y
100	50

- The Transaction T_1 unlocks x before obtaining a write lock on y
- T_1 opens an opportunity window for other concurrent transaction to meanwhile access the data element

- A transaction shall obtain all the locks before release them
  - Founded on a theorem demonstrated by Ullman (1982)
    - Principles of Transaction Processing, by Philip A. Bernstein - Chap. 6
- In the previous example:
  - T1 shouldn't release the lock on x so early (and it should be a write lock)
  - As there is a pending update (the K value) $wu_1[x]$ obtained at the beginning should be released only at the end
- Important architectural decisions
  - The granularity of the data elements
  - The need for a LockManager to guarantee the consistency during transactions execution (concurrency coordinator)

Possible interface for a LockManager

- Operations (methods)
  - getLocks(transactionID, lockElements)
  - releaseLocks(transactionID, lockElements)
  - unlock(transactionID) // release all locks associated to the transaction
- LockElements[]
  - LockElement
    - DataItem, LockMode
      - lockMode = read | write
- Questions
  - How to implement it on the Jini framework?
- Evaluation scenario
  - Transferences between accounts from different bank (services), based on low level read and write operations.

Alternative Concurrency Coordination Strategies

- Optimistic concurrency control (OCC)
  - "*Optimistic concurrency control (OCC) …, operates on the principle that multiple transactions can frequently complete without interfering with each other*." [ref].
- Multiversion concurrency control (MVCC)
  - "*MVCC (Multi-Version Concurrency Control) is a way to improve database concurrency by storing multiple row versions as they have been at different points in time*." [ref]

Long lived transactions (LLTs) - SAGA

- Molina defined in 1987 [ref] the concept of Long-lived transactions (LLTs) and suggested the name of ***Saga*** when
  - *"A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions T"*
- The concept was meanwhile adopted in the context of microservices [ref] considering the case of long transactions, e.g., when the idea of Roll-back is not applicable because a failure requires the execution of a process to return to the prior state.
  - An example of organizing a trip where reservation services, e.g., flight, hotel, transfer, require hours or days to execute, and if necessary to cancel, there is commonly a reimbursement payment associated that needs to be processed.
- The term ***Long Running Transactions*** is also applied to web services [ref], that this paper associate to Saga concept.
  - *"With sagas, many small atomic transactions are wrapped by a larger longer running transaction. Each small atomic transaction is paired with a **compensation** handler that is capable of reversing the activity done in the atomic transaction."*

Bank transference operations