Distributed Systems 1 - Project 2021:
Optimistic Concurrency Control for Distributed Transactions

Distributed transactions (TXNs)

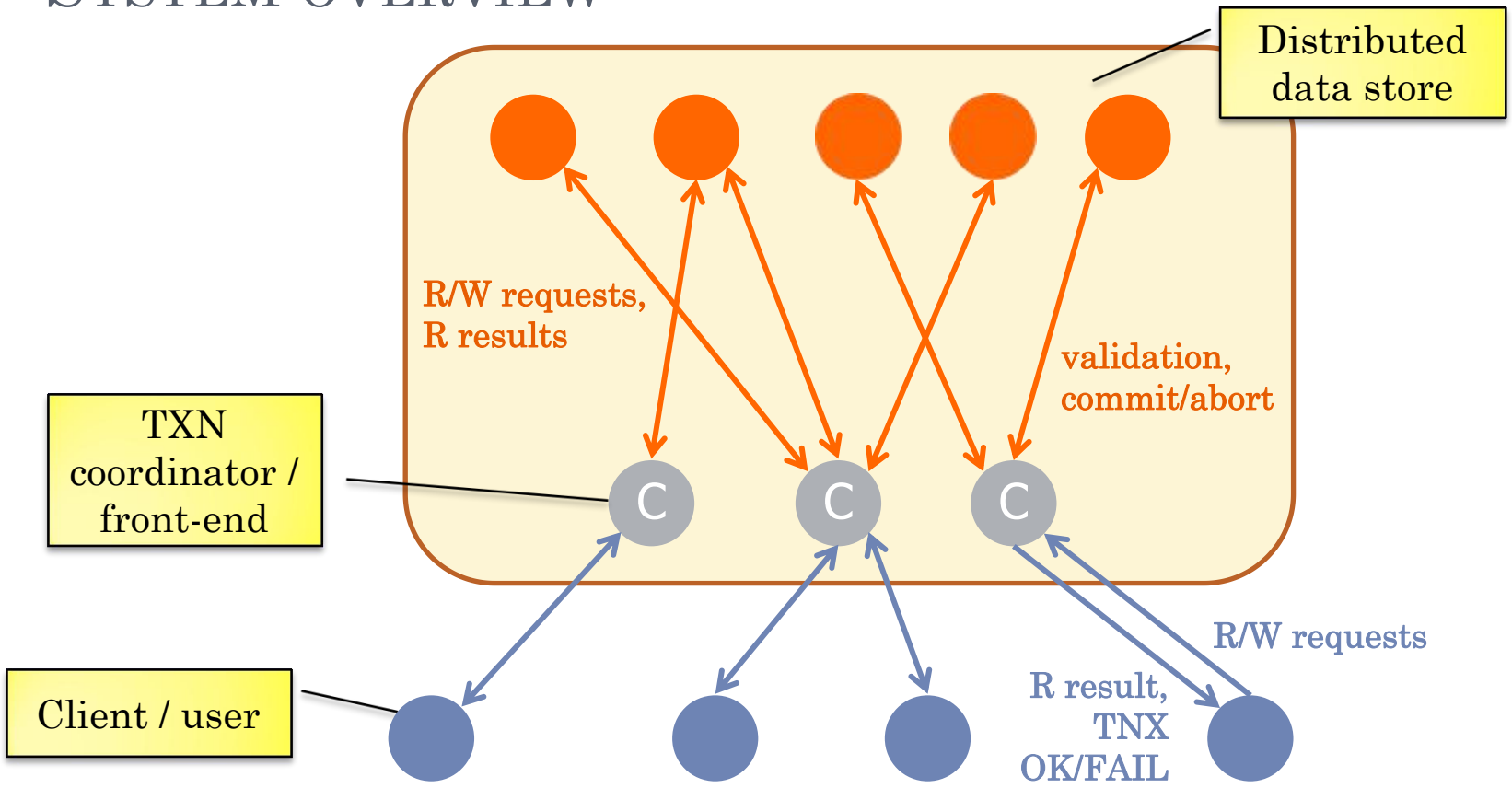
- A sequence of READ/WRITE (R/W) operations on one or more database elements.
- A TXN can be COMMITED (be applied and modify the state of the database), or ABORTED (no trace left in the system).
- When the database is partitioned, we need a strategy for atomic commitment (like 2PC).
- In Lab 4 we implemented 2PC for a single round of atomic commitment... But what if multiple TXNs run concurrently?

In this project, you will implement a protocol for concurrent distributed TXNs, ensuring that access conflicts do not turn into inconsistencies!

Concurrency control

- WRITE operations can result in conflicts with other transactions (R/W, W/R or W/W conflicts).
- One option is to lock database elements so that only one transaction at a time can access them (as in 2PL). However, 2PL is wasteful for low contention (and requires deadlock detection).
- We will implement an **optimistic** approach that allows transactions to modify a private workspace without locking in advance, and only validates the TNX when ready to commit.

System overview



- Clients sends BEGIN/END and R/W requests (provided in the project template) to any of the coordinators. A coordinator may serve multiple clients.
- The coordinator forwards R/W to the database partition that holds the data item and returns R values to the client.
- The partition applies changes to its private workspace dedicated to the transaction.

System overview - server

- For simplicity, the distributed data is simply a collection of versioned key-value pairs. Clients read and write the value identified by a unique key.

- Each data store server S_i with ID *i* holds data with keys in the range [10*i* .. 10*i*+9]. The number of servers created should be configurable. You can create all key-value pairs directly when you create the server, and initialize all versions to 0 and values to 100.

0

KEY	VERSION	VALUE
0	4	7
1	7	129
...	...	...
9	2	16

1

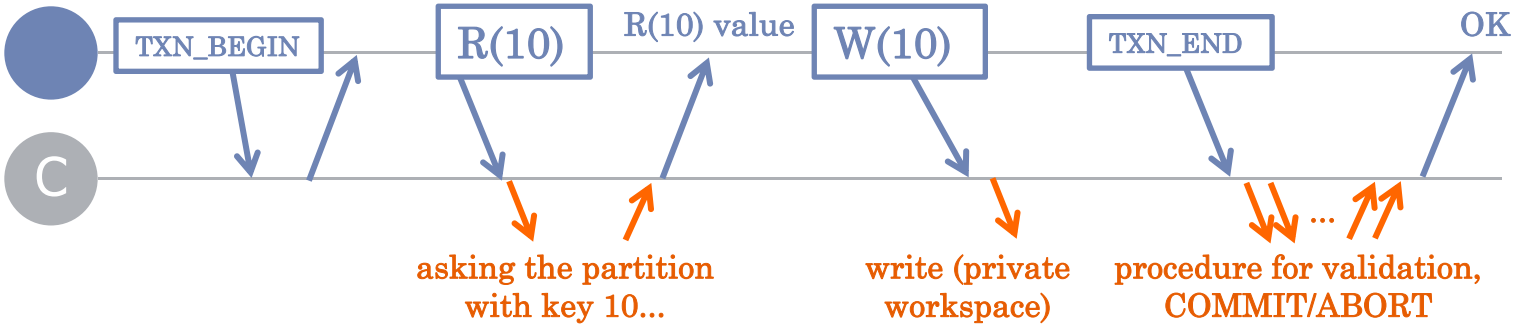
KEY	VERSION	VALUE
10	3	36
11	14	8
...	...	...
19	8	647

2

KEY	VERSION	VALUE
20	10	54
21	3	206
...	...	...
29	11	19

System overview - client

- Each client tries to get **1 transaction done at a time**, but multiple clients can be active simultaneously.
- For each TXN, it chooses a **random coordinator** that will handle it.
- The client sends its TXN_BEGIN, then waits for confirmation from the coordinator. Once confirmed, the TXN is expected to eventually commit or abort (even in case of transient failures).
- The client waits for R values ("conversational" API).
- Once all its operations are done, it sends a TXN_END message to have its changes committed.



System overview - client

The client actor is provided.

- To help you testing your implementation, the clients always do simple operations: they try to reduce some value by a given amount and increase another by the same amount.

  → the sum of visible values should be constant.

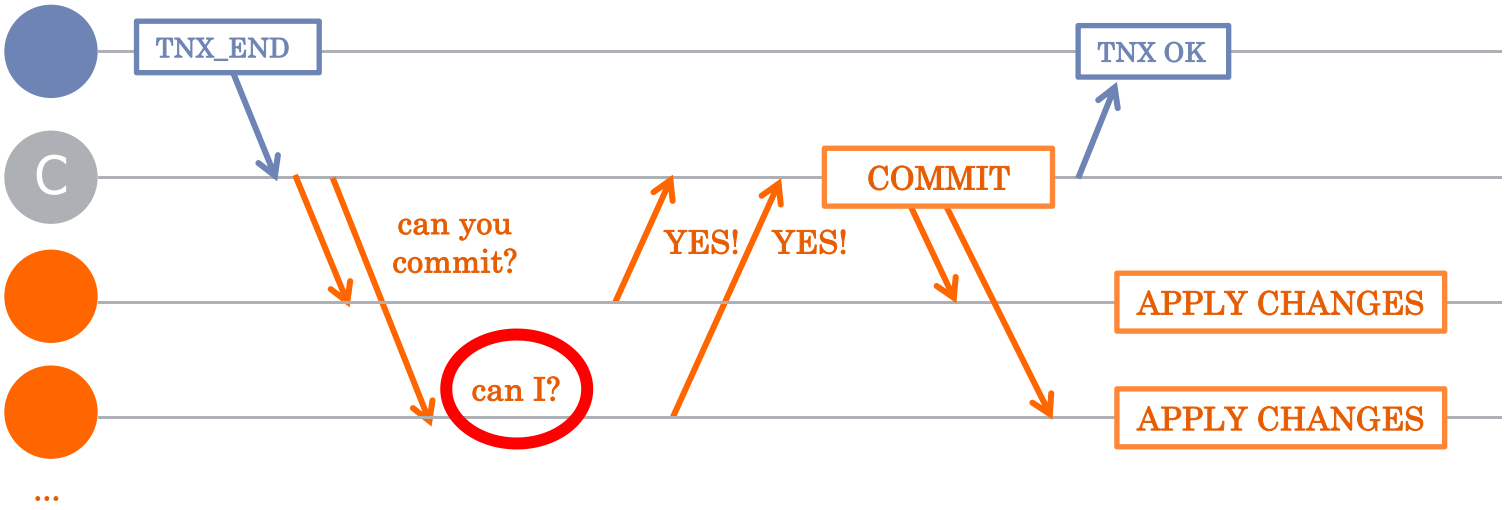
- However, do not make server-side assumptions on what the transactions will do, just execute the reads and writes as requested.
- You can modify the client, as long as you do not fundamentally change the transaction style or trivialize some aspect of the protocol.
- Find more details in the code!

System overview - coordinator

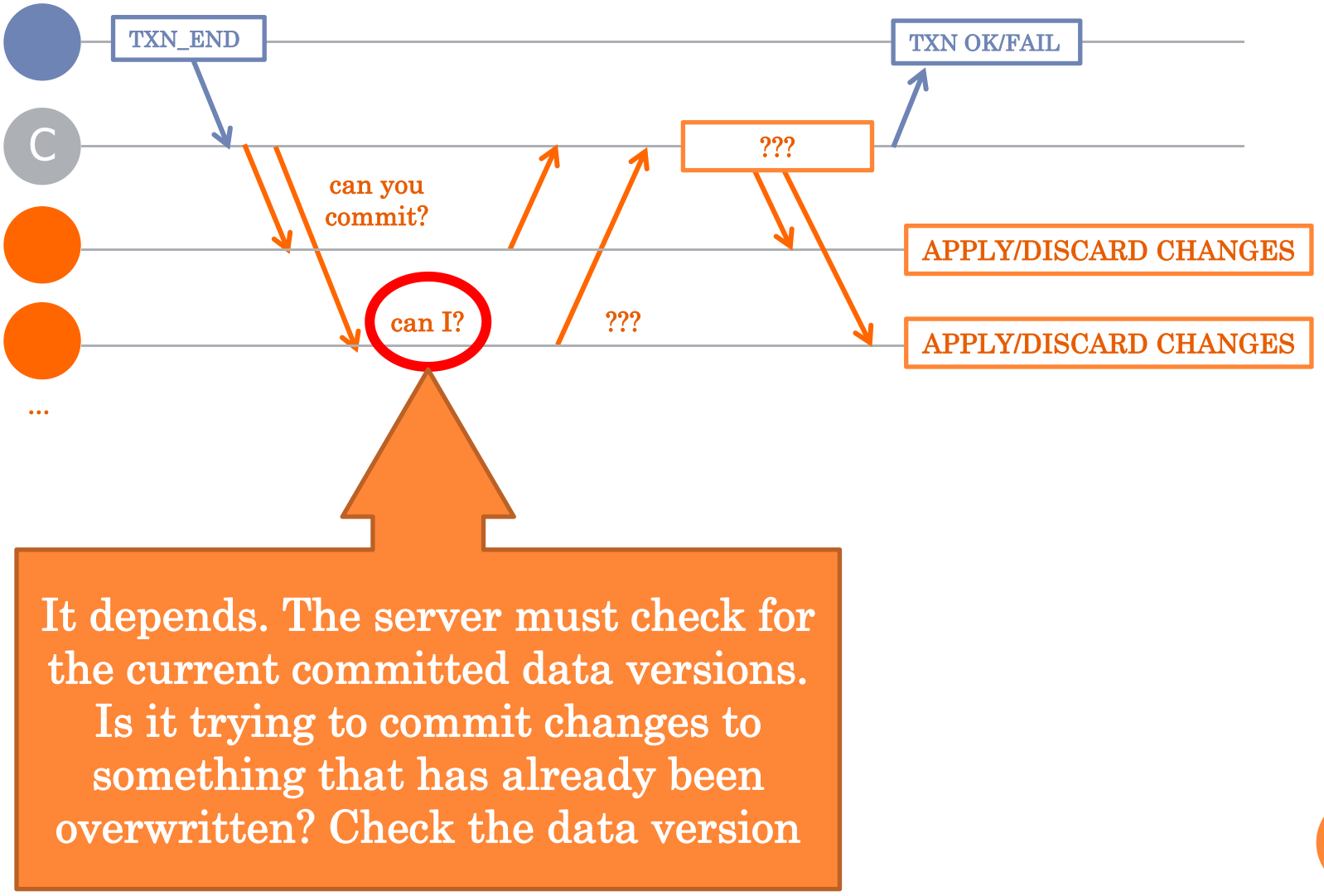
- The coordinator is responsible for:
  - globally-unique IDs for its TXNs (so that you can distinguish the operations associated to each one)
  - forwarding R/W requests from the client(s)
  - replying to the client requests
  - change values (as written by the client) and their version through commits
  - **...orchestrating distributed commit!**
- When multiple TXNs (possibly issued by different coordinators) compete for the same resource, concurrent access must be scrutinized to avoid inconsistencies.
- A plethora of techniques for distributed transactions...

Ensuring serializability

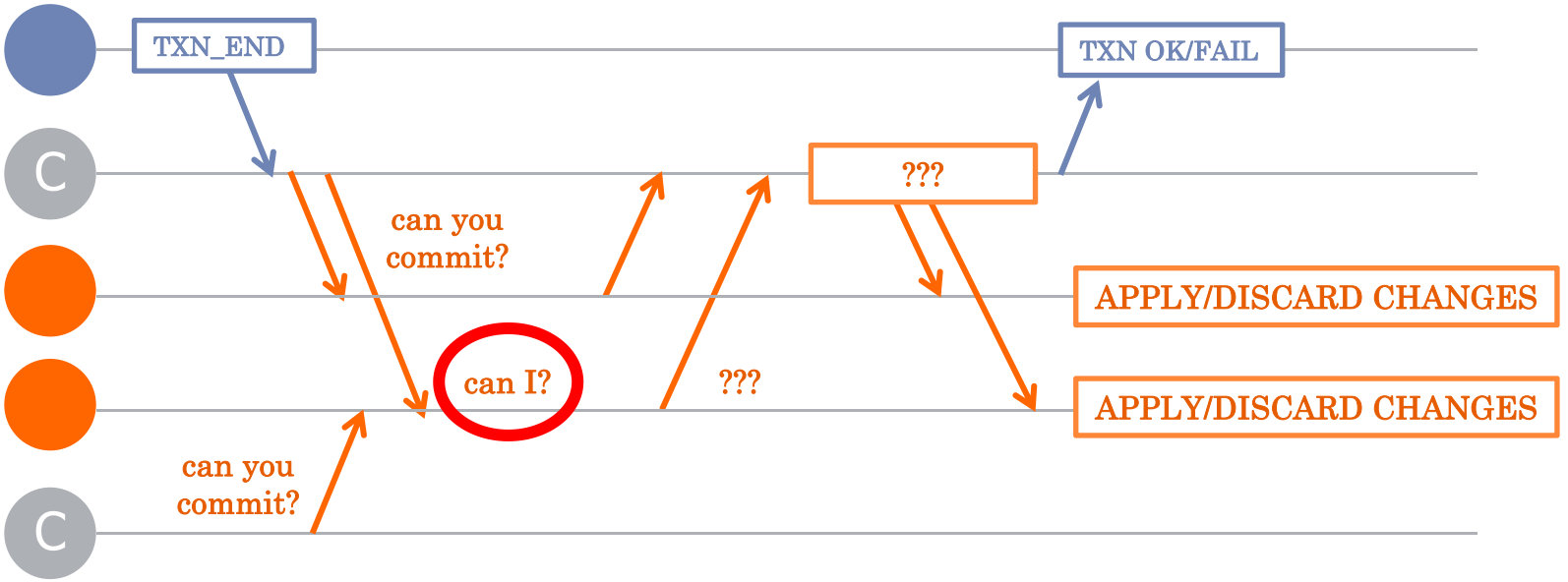
- Concurrent requests on the same data may lead to a state that is inconsistent with a serial order of conflicting TXN operations.
  - Ex.: T_1, T_2 read a value (same version) and both change it.
- Your task is to ensure **strict serializability**. TXNs that would break serialization must abort.
- We operate under optimistic assumptions: we let TXNs run and modify the private workspaces, and check if they are valid later when the coordinator wants to commit.



TXN VALIDATION (1): DATA VERSION



TXN VALIDATION (2): CONCURRENT VALIDATION



If the server is in the middle of another validation, it may soon commit changes. Are these changes in conflict with the proposed ones?

Locking?

- To protect a prospective commit, a server may need to "lock" some objects so that they cannot be modified during validation.
- However, this is very different from pessimistic locking, since the validation phase is generally very short.
- If a TXN cannot acquire the needed locks during validation, it should reply "NO" to the coordinator. The coordinator will then abort.

Hint: is locking ALL objects always necessary?

Strict serializability

- Serializability: TXNs appear to have occurred in some total order.
- Strict serializability: the total order is consistent with the real-time order. A read in a committed TXN must see the most recent values if it starts after the commit of another one that updates them.

Reads should be atomic, but only those of committed transactions.

Interfacing with clients

- When you create your ActorSystem, you can also create any number of clients.
- Once you have initialized the rest of the system, start the clients. To do so, send to the clients a WelcomeMsg including:
  - ActorRefs of the coordinators
  - Maximum key in the store
- You can stop the client actor by sending it a StopMsg (see the provided code).

Assumptions

- Reliable and FIFO channels.
- The strict serializability requirement is only enforced for *committed* TXNs.
- Coordinators and data servers can crash, but recover after an arbitrary delay. Upon recovery, they have access to non-volatile memory (in this prototype it's simply the program variables!) to retrieve their state before crashing, like in the 2PC lab.
- Simulate crashes only during validation/commit; the other cases (i.e., interaction with the client) are not that interesting - just make sure that clients are not left waiting forever.
- The client does not crash.

Implementation suggestions

- Start by implementing the handling of TXN_BEGIN, R/W forwarding, R replies from store servers to the coordinator, and from the coordinator to the client. Assign a unique ID object to each TXN.
- It may be helpful to use custom objects as keys of a Map. Java tip: you need to override `equals()` and `hashCode()` in your custom class.
- Use caution: == vs. `equals()`!
- Handle W requests at the servers. Design an appropriate object for private workspace(s) associated to transactions.
- And recall Lab 1: be careful not to reference visible items from the private workspace!
- Then implement two-phase commit. Do not consider crashes initially. Create appropriate structures to track ongoing validations and locked items.
- Your project must include a way to show that the algorithm is correct. Rely on the fact that the sum of values is constant to show that TXNs are either fully applied or leave no trace at all.

Project report

- Structure of the project
- Main design choices
  - How do you associate operations to their transaction?
  - How do you manage a server's private workspace?
  - Detailed description of the validation/commit phase.
  - How are crashes simulated and what do nodes do upon recovery?
  - …
- Discussion of the implementation
  - Is it safe? Does it achieve (strict) serializability? How?
  - Are there any assumption about the system that you need to make?
  - Comments on liveness/deadlocks; availability upon node failure.
  - Can it be improved (in general, or in specific scenarios)? Just some examples.
  - …