Distributed Computing and Systems
Chalmers university of technology

Prof Philippas Tsigas
Distributed Computing and Systems Research Group

DISTRIBUTED SYSTEMS II

REPLICATION

Teaching material based on Distributed Systems: Concepts and Design, Edition 3, Addison-Wesley 2001.

Viewing: These slides must be viewed in slide show mode.

Distributed Systems Course

Replication

Introduction to replication

Replication of data :- the maintenance of copies of data at multiple computers

- replication can provide the following
- performance enhancement
  - e.g. several web servers can have the same DNS name and the servers are selected in turn. To share the load.
  - replication of read-only data is simple, but replication of changing data has overheads
- fault-tolerant service
  - guarantees correct behaviour in spite of certain faults (can include timeliness)
  - if *f* of *f*+1 servers crash then 1 remains to supply the service
  - if *f* of 2*f*+1 servers have byzantine faults then they can supply a correct service
- availability is hindered by
  - server failures
    - replicate data at failure- independent servers and when one fails, client may use another.
  - network partitions and disconnected operation
    - Users of mobile computers deliberately disconnect, and then on re-connection, resolve conflicts

Availiability

is used for repairable systems

- It is the probability that the system is operational at any random time t.
- It can also be specified as a proportion of time that the system is available for use in a given interval (0,T).

Requirements for replicated data

- Replication transparency
  - clients see logical objects (not several physical copies)
    - they access one logical item and receive a single result
- Consistency
  - specified to suit the application,
    - e.g. when a user of a diary disconnects, their local copy may be inconsistent with the others and will need to be reconciled when they connect again. But connected clients using different copies should get consistent results. These issues are addressed in Bayou and Coda.

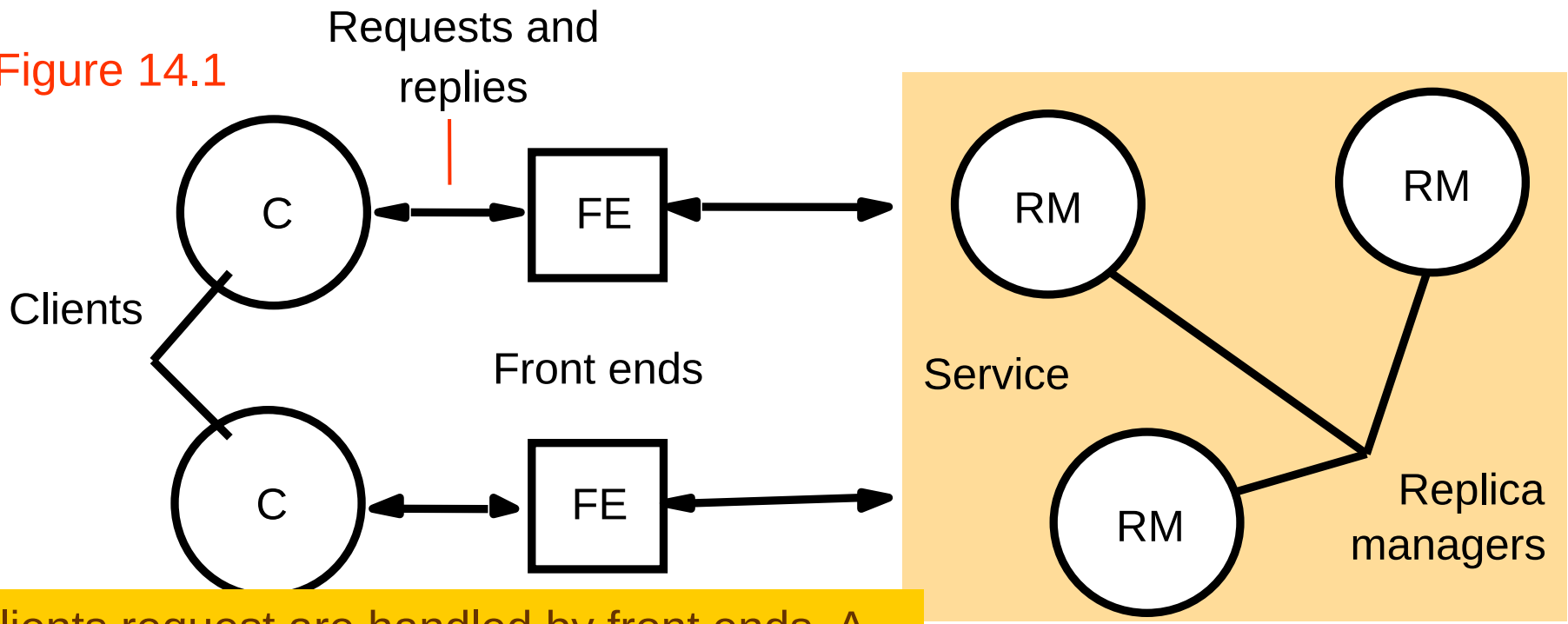
A basic architectural model for the management of replicated data

A collection of RMs provides a service to clients

Clients see a service that gives them access to logical objects, which are in fact replicated at the RMs

Clients request operations: those without updates are called *read-only* requests the others are called *update* requests (they may include reads)

Figure 14.1



Clients request are handled by front ends. A front end makes replication transparent.

14.2.1 System model

- each *logical* object is implemented by a collection of *physical* copies called *replicas*
  - the replicas are not necessarily consistent all the time (some may have received updates, not yet conveyed to the others)
- we assume an asynchronous system where processes fail only by crashing and generally assume no network partitions
- replica managers
  - a RM contains replicas on a computer and access them directly
  - RMs apply operations to replicas recoverably
    - i.e. they do not leave inconsistent results if they crash
  - objects are copied at all RMs unless we state otherwise
  - static systems are based on a fixed set of RMs
  - in a dynamic system: RMs may join or leave (e.g. when they crash)
  - a RM can be a *state machine*, which has the following properties:

State Machine Semantic Characterization

- Outputs of a state machine are complitely determined by the sequence of requests it processes indepedent of time and any other activity in the system.

- Vague about internal structure

State Machine: Examples

State machine

- Server:
- Word store[N]

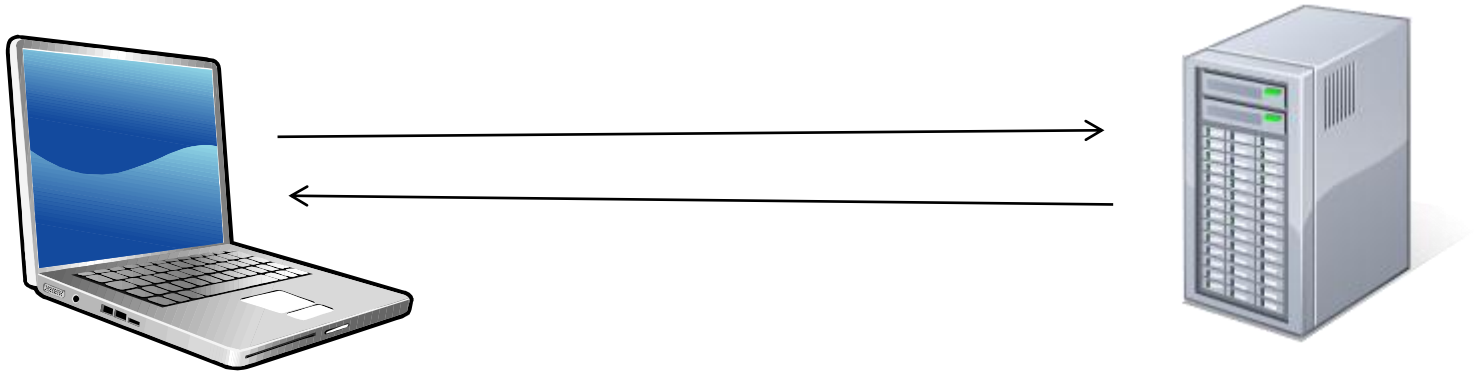
- ```
  Read(int loc) {
      send store[loc] to client;
   }
   Write[int loc, word val] {
   store[loc]=val
   }
   Client

   memory.write(100, 4)
   Memory.read(100)
   Receive v from memory
  ```

Not a state machine

- ```
  while true do
      read sensor
      q := compute adjustment
      send q to actuator
      end while
  ```

State Machine no Replication Response Guarantees

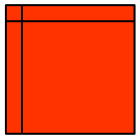


Client

Server

Response Guarantees

1) Requests issued by a single client to a state machine are processed in the order issued (FIFO request delivery)
2)
   - Request *r* to state machine *s* by client *c1*
   - could have caused request *r'* to s by client *c2, then*
   - *s* processes *r* before *r'*



Requests are buffered until they become stable to be processed

All replicas process the same ***sequence*** of requests

1. Uniquely identify the requests.
2. Order the requests. Do not forget the guarantees that we expect.
    1. Server have to know when to service a request. (When a request is stable)

When to process a reguest – Stability Detection

- 3 methods:
  - Logical clocks
  - Real-time clocks
  - Server-generated ids

Logical Clocks

- Assign integer T(e,p) to event e from processor p:
  - If e is a sending of a message
  - If e is a receiving of a message
  - Importanat event

  Properties:

  T(e,p) < T(e1,q) or vice-versa

  If e could have caused e1, then T(e,p)<T(e1,q)

p<q<r

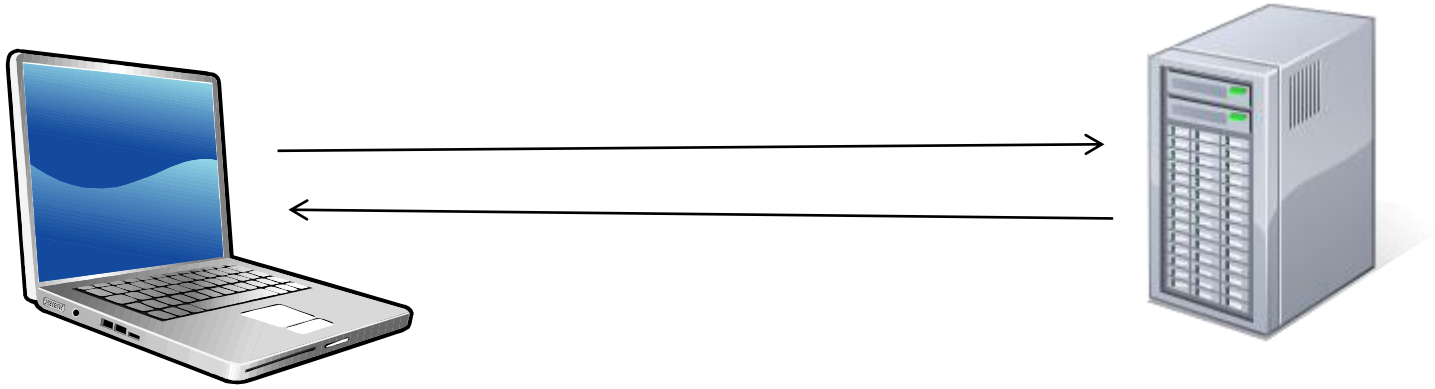
Synchronized Real-Time Clocks

- If a message sent with uid t will be received no later than t+D by local clock.
- Uids differ by D at most at any time

Server-generated ids

- Clients first get an id from the server then issue the id to issue a request (like a sequencer).

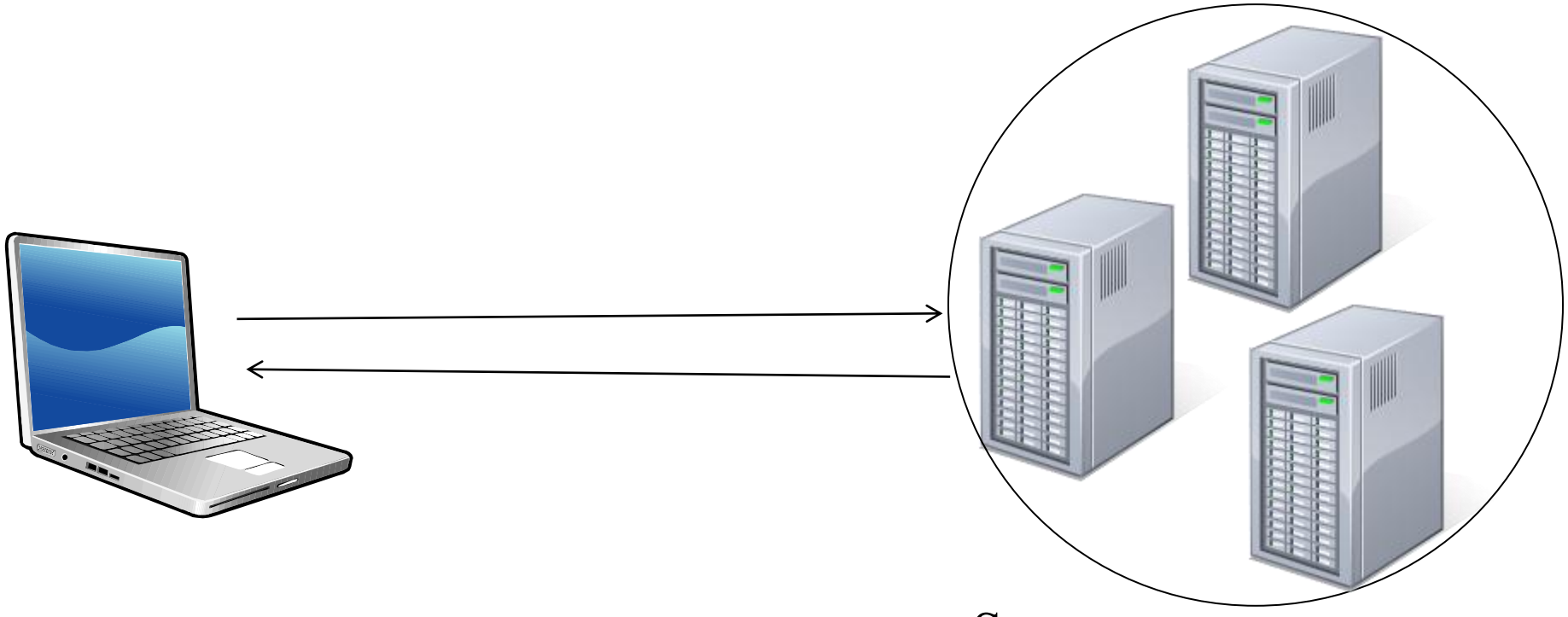
State Machine



Client

Server

State Machine



State Machine approach to Replication

Each RM

- applies operations atomically
- its state is a deterministic function of its initial state and the operations applied
- all replicas start identical and carry out the same sequence of operations
- Its operations must not be affected by clock readings etc.

Replication

- Place a copy of the server state machine on multiple network nodes.

- ? Communication of the requests?
- ? Coordination ?

- Want:
- All replicas start in the same state
- All replicas receive the same set of requests
- All replicas process the same ***sequence*** of requests

Faults

Four phases in performing a request

- issue request
  - the FE either
    - sends the request to a single RM that passes it on to the others
    - or multicasts the request to all of the RMs
- coordination + agreement
  - the RMs decide whether to apply the request; and decide on its ordering

Total ordering: if a correct RM handles r before *r'*, then any correct RM handles *r* before *r'*

  - the RMs execute the request (sometimes tentatively)
- response
  - one or more RMs reply to FE e.g.

RMs agree - I.e. reach a consensus as to effect of the request. In Gossip, all RMs eventually receive updates.

What sort of system do we need to perform totally ordered reliable multicast?

13.3.2. Active replication for fault tolerance

- the RMs are *state machines* all playing the same role and organised as a group.
  - all start in the same state and perform the same operations in the same order so that their state remains identical
- If an RM crashes it has no effect on performance of the service because the others continue as normal
- It can tolerate byzantine failures because the FE can collect and compare the replies it receives

a FE multicasts each request to the group of RMs (and FE's)

the RMs process each request identically and reply

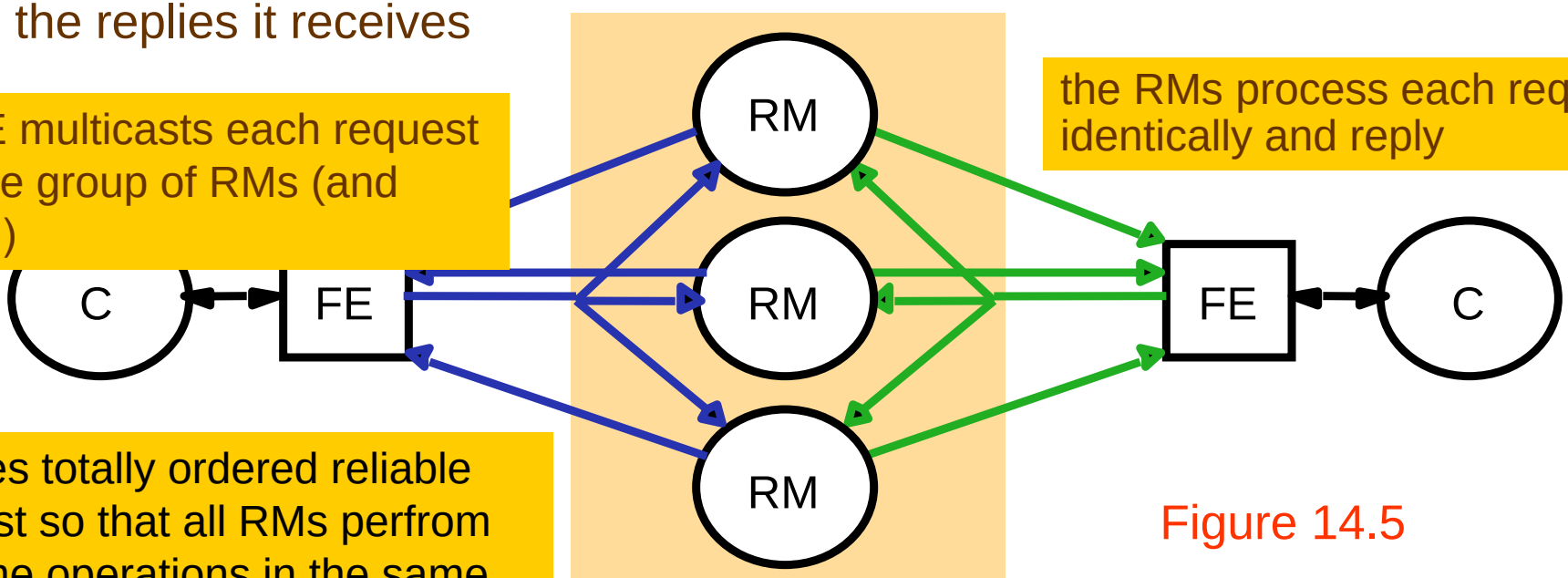


Figure 14.5

Requires totally ordered reliable multicast so that all RMs perfrom the same operations in the same order

Active replication - five phases in performing a client request

- Request
  - FE attaches a unique *id* and uses *totally ordered reliable multicast* to send request to RMs. FE can at worst, crash. It does not issue requests in parallel
- Coordination
  - the multicast delivers requests to all the RMs in the same (total) order.
- Execution
  - every RM executes the request. They are state machines and receive requests in the same order, so the effects are identical. The *id* is put in the response
- Agreement
  - no agreement is required because all RMs execute the same operations in the same order, due to the properties of the totally ordered multicast.
- Response
  - FEs collect responses from RMs. FE may just use one or more responses. If it is only trying to tolerate crash failures, it gives the client the first response.

Requirements for replicated data

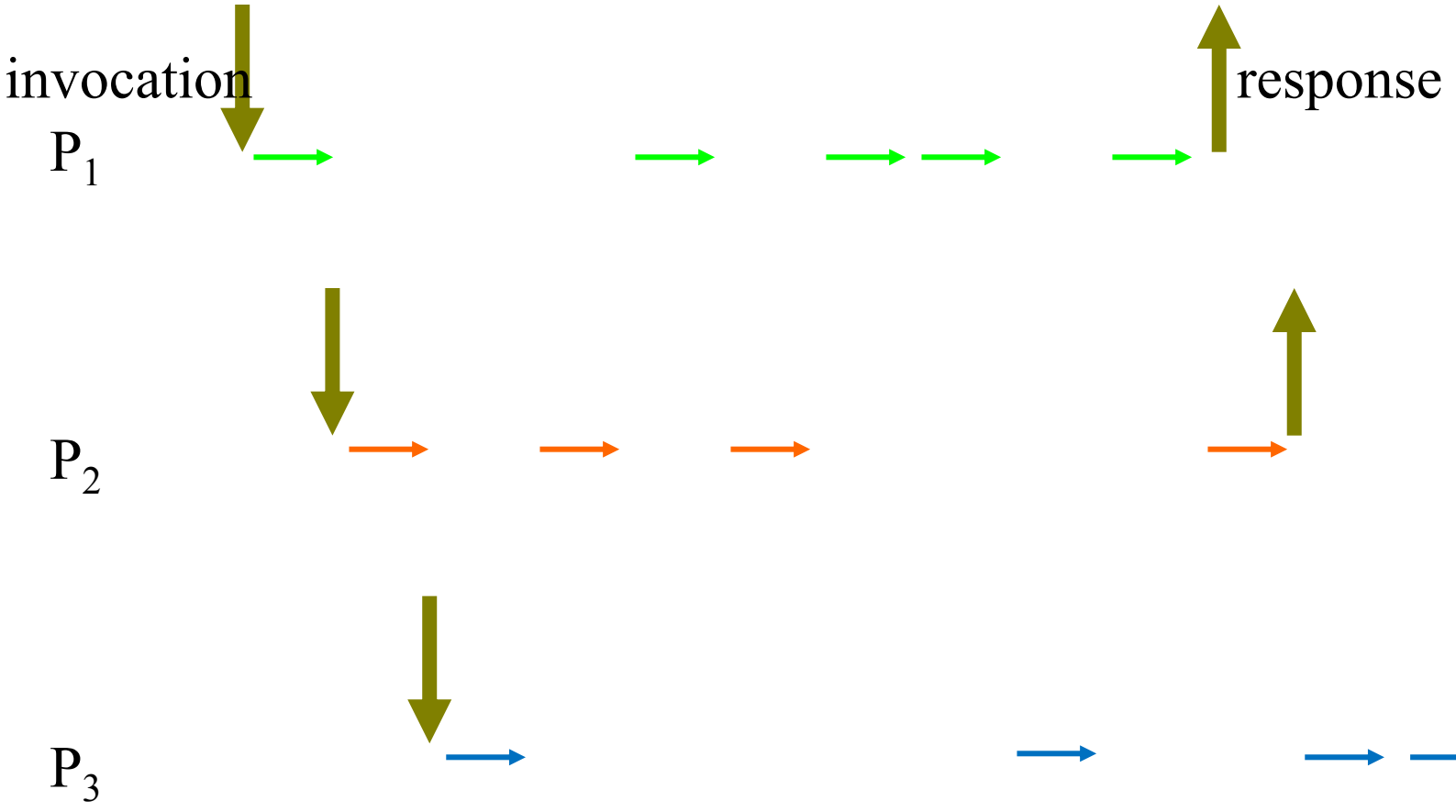
- Replication transparency
  - clients see logical objects (not several physical copies)
    - they access one logical item and receive a single result
- Consistency

  (General Consistency Models)

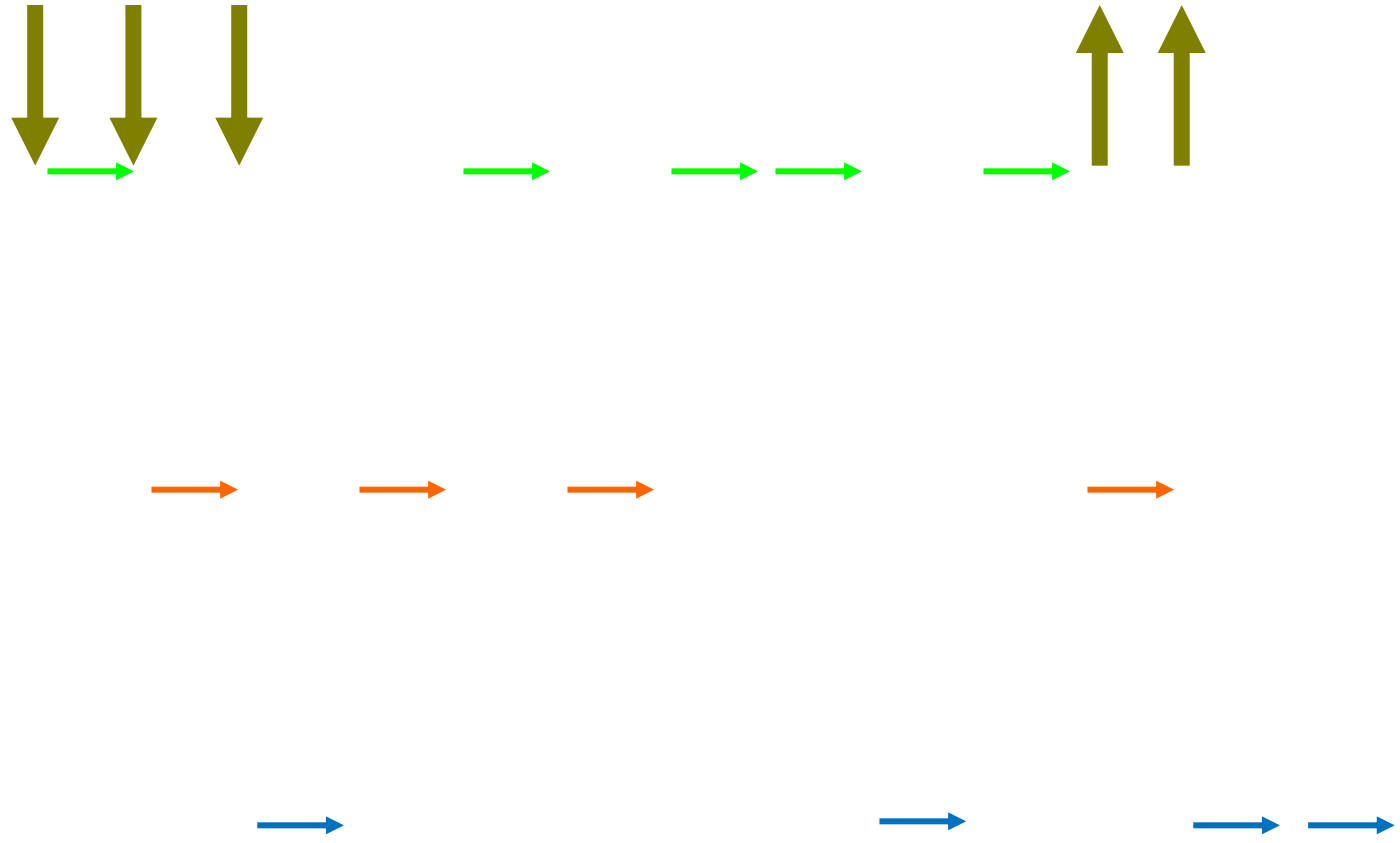
General Consistency Models

- What consistency do we expect from concurrent operations

Executing Operations

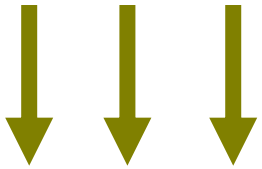


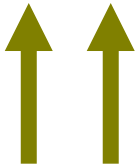
Interleaving Operations



Concurrent execution

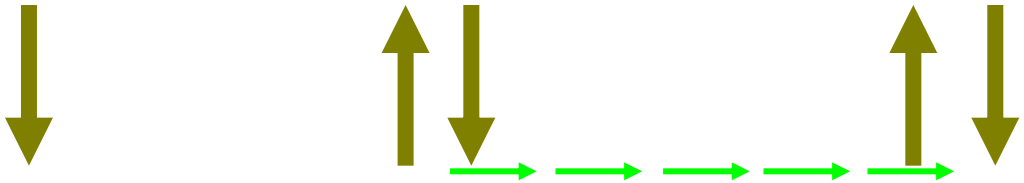
Interleaving Operations





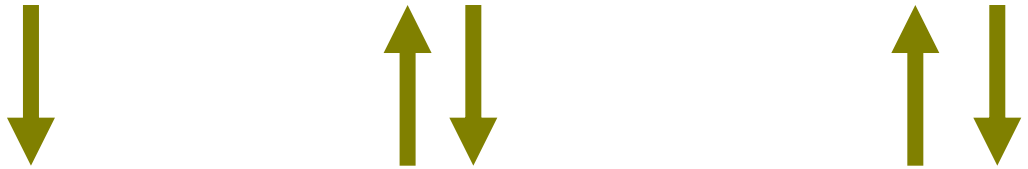
(External) behavior

Interleaving Operations, or Not



Sequential execution

Interleaving Operations, or Not



Sequential behavior: invocations & response alternate and match (on process & object)

Sequential specification: All the legal sequential behaviors, satisfying the semantics of the ADT

- E.g., for a (LIFO) stack: pop returns the last item pushed

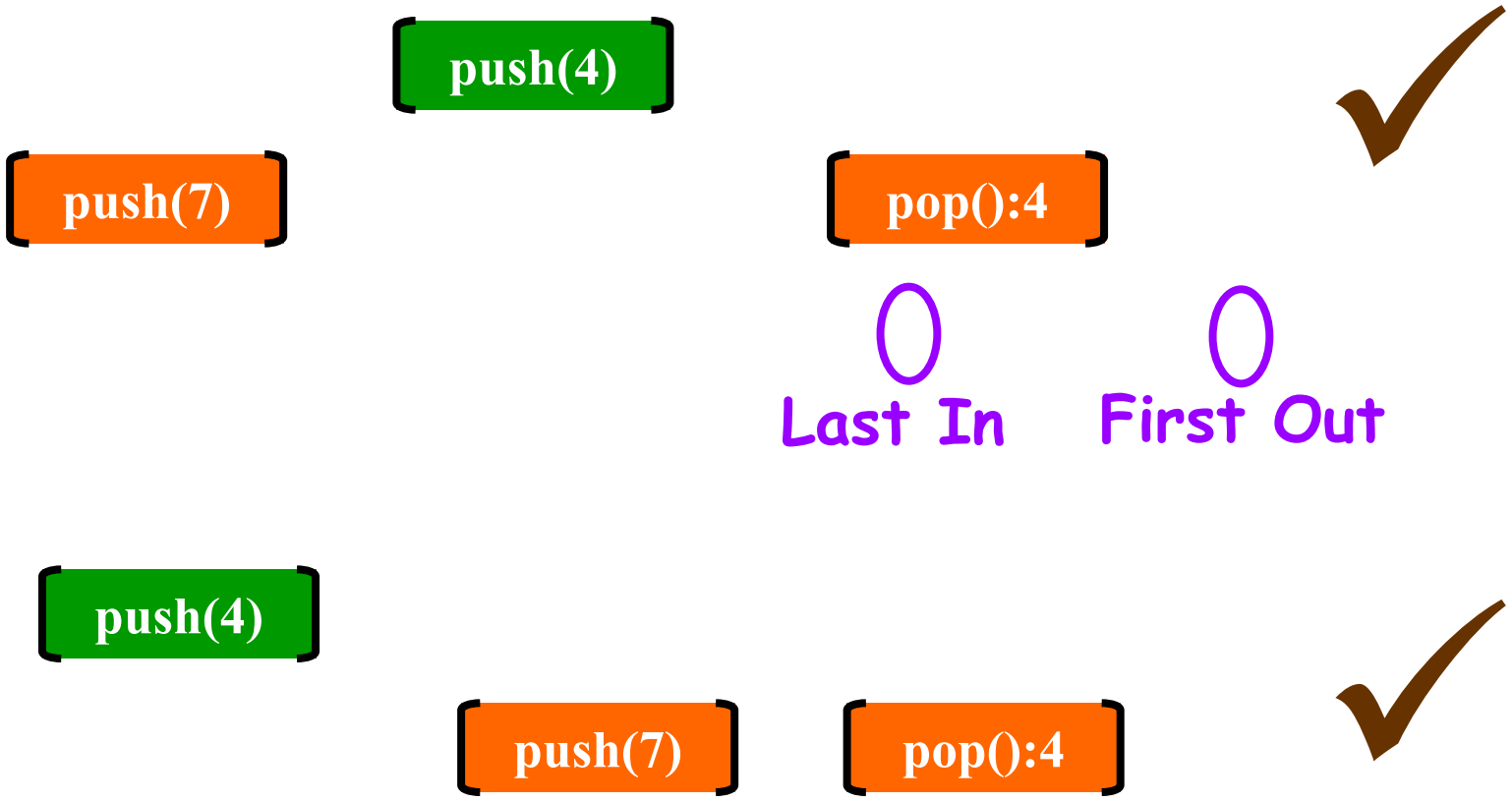
Correctness: Sequential consistency

[Lamport, 1979]

- For every concurrent execution there is a sequential execution that
  - Contains the same operations
  - Is legal (obeys the sequential specification)
  - Preserves the order of operations by the same process

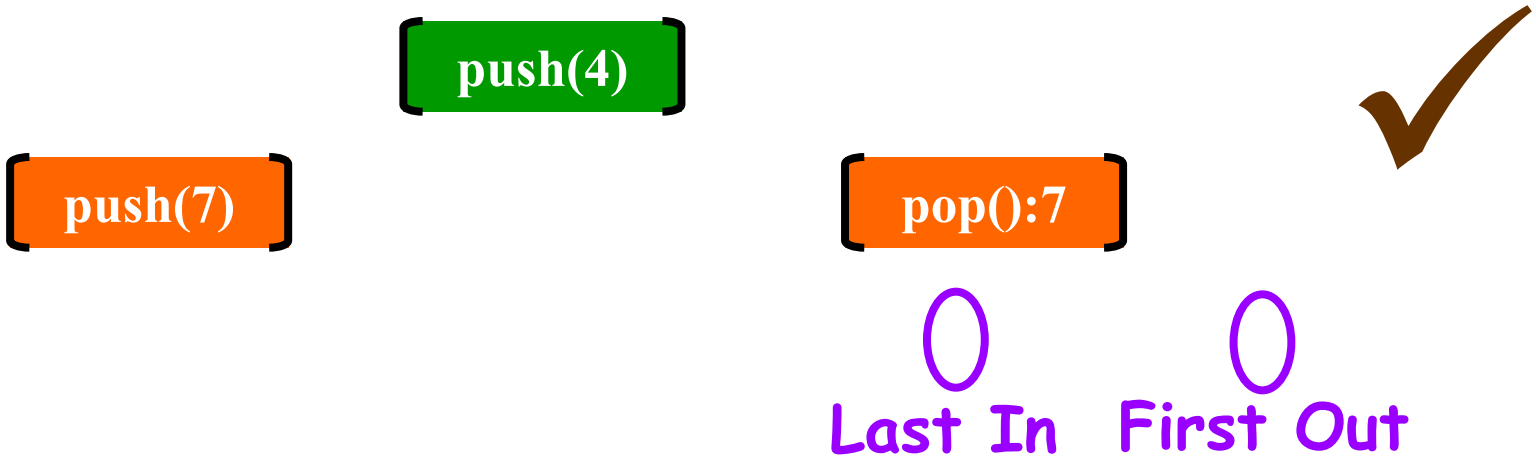
Sequential Consistency: Examples

Concurrent (LIFO) stack



Sequential Consistency: Examples

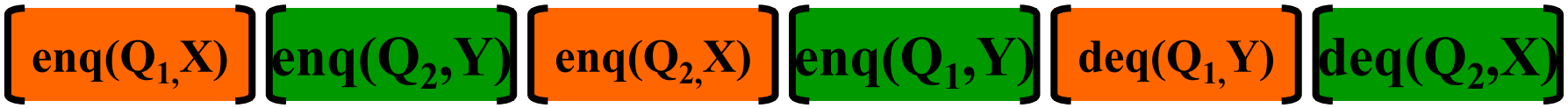
Concurrent (LIFO) stack



Sequential Consistency is not Composable

The execution is not sequentially consistent

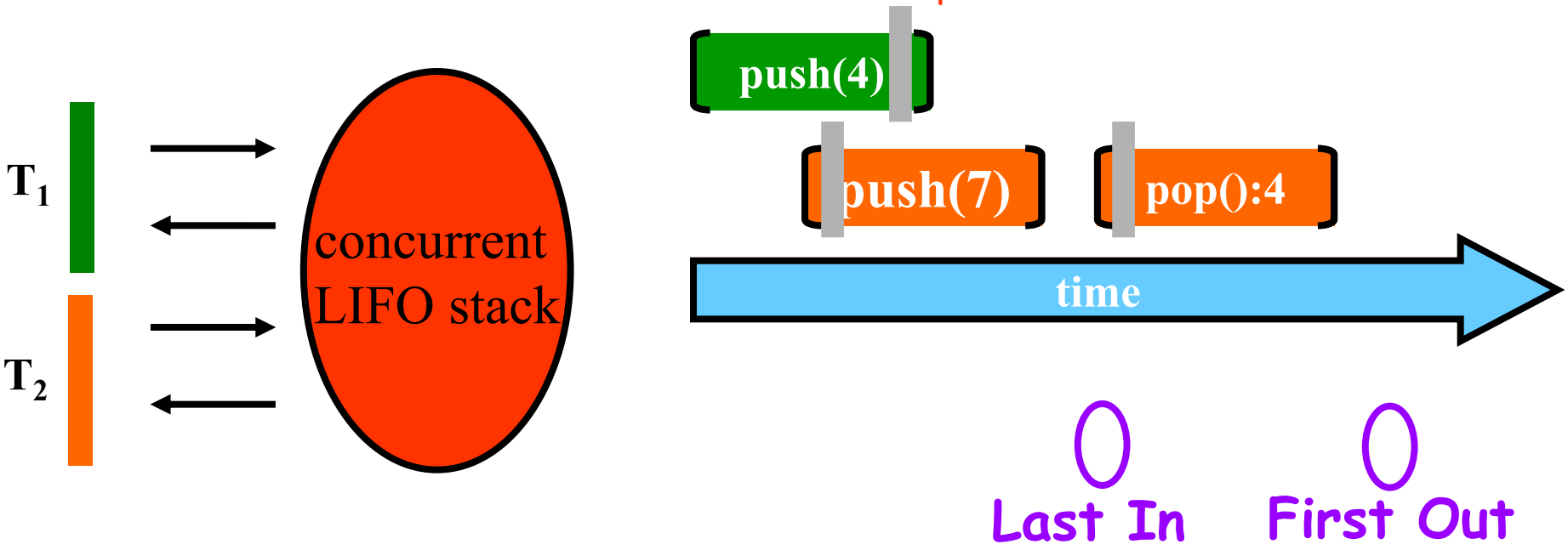
Sequential Consistency is not Composable



The execution projected on each object is sequentially consistent

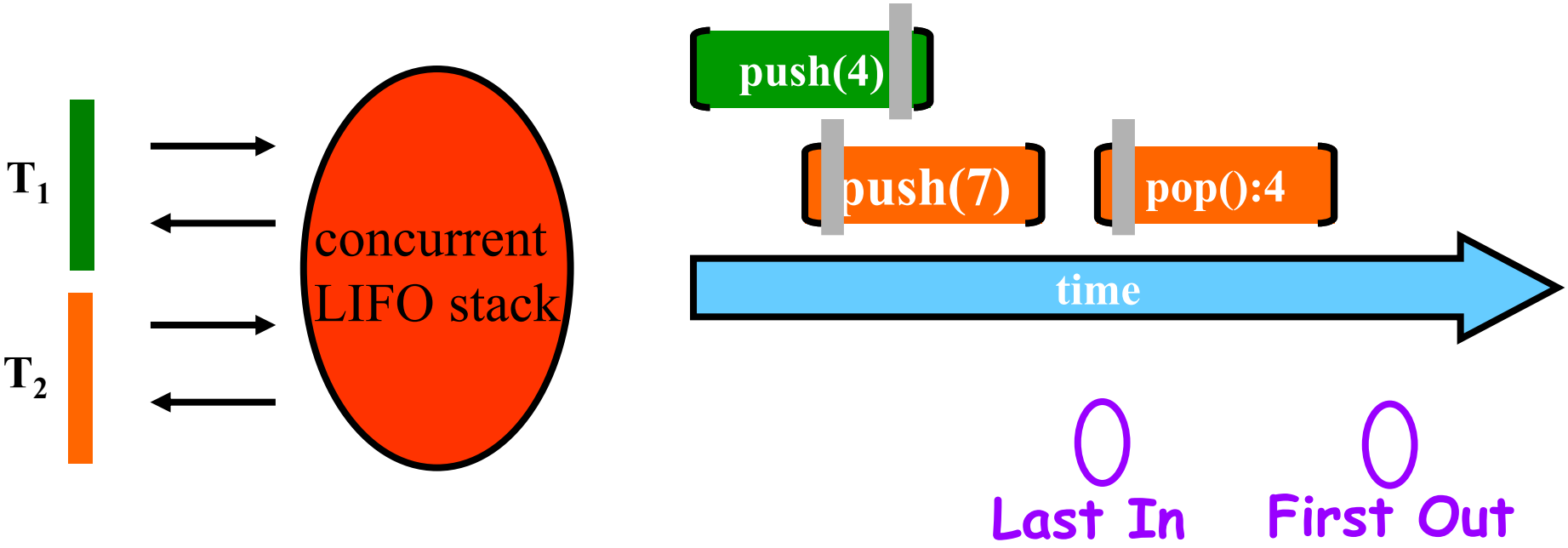
Safety: Linearizability

- **Sequential specification** defines legal sequential executions
- Concurrent operations allowed to be **interleaved**
- For every concurrent execution there is a sequential execution that
  - Contains the same operations
  - Is legal (obeys the sequential specification)
  - Preserves the real-time order of all operations



Safety: Linearizability

- **Sequential specification** defines legal sequential executions
- Concurrent operations allowed to be **interleaved**
- Operations **appear to execute atomically**
  - External observer gets the **illusion** that each operation **takes effect instantaneously** at some point **between** its **invocation** and its **response**



Linearizability (p566) the strictest criterion for a

linearizability is not intended to be used with transactional replication systems

–The real-time requirement means clients should receive up-to-date information

- but may not be practical due to difficulties of synchronizing clocks
- a weaker criterion is sequential consistency

a replicated object service is *linearizable* if *for any execution* there is some interleaving of clients' operations such that:

–the interleaved sequence of operations meets the specification of a (single) correct copy of the objects

–the order of operations in the interleaving is consistent with the real time at which they occurred

- For any set of client operations there is a virtual interleaving (which would be correct for a set of single objects).
- Each client sees a view of the objects that is consistent with this, that is, the results of clients operations make sense within the interleaving
  - the bank example did not make sense: if the second update is observed,the first update should be observed too.