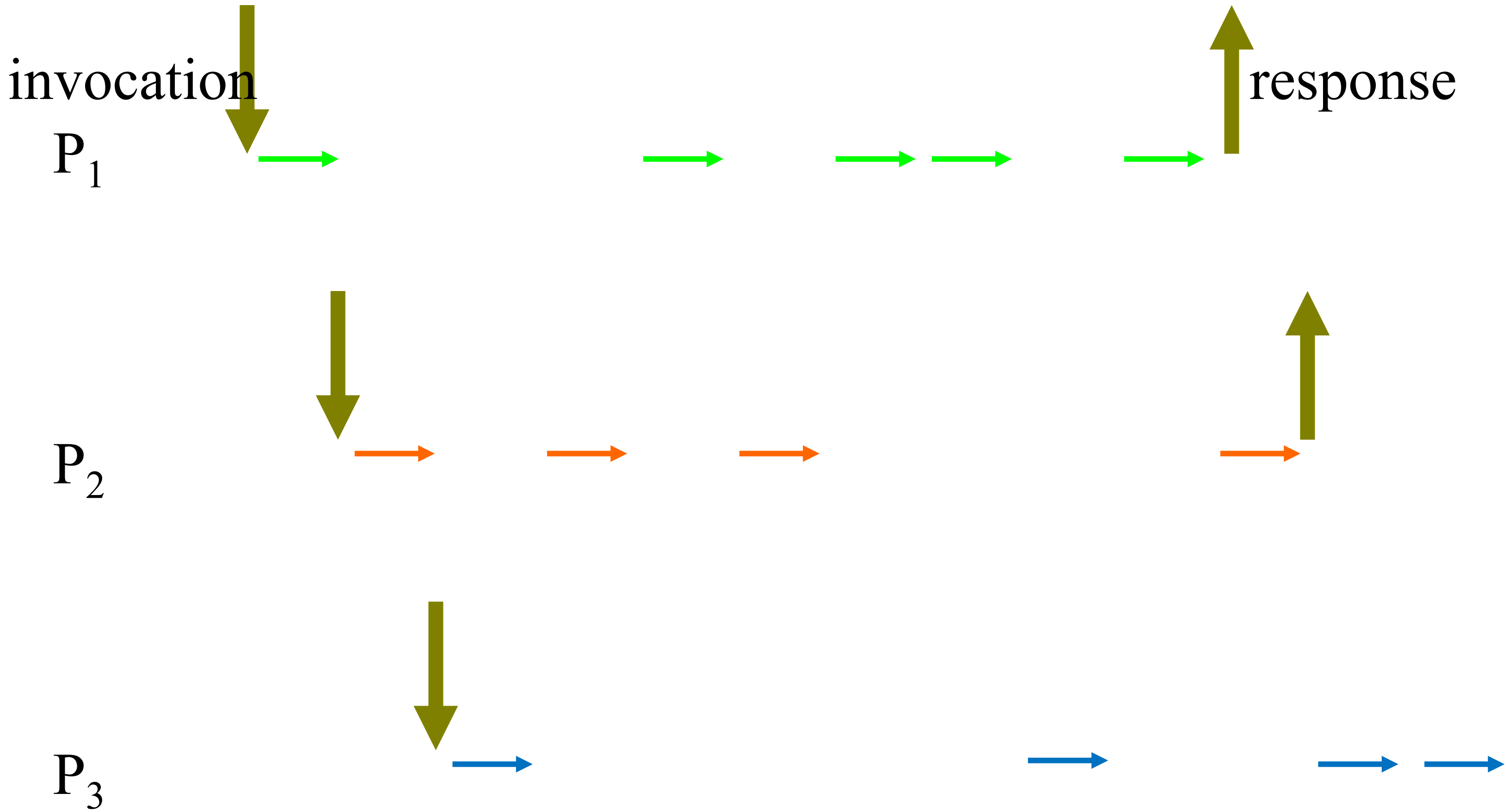
Distributed Computing and Systems
Chalmers university of technology

Prof Philippas Tsigas
Distributed Computing and Systems Research Group

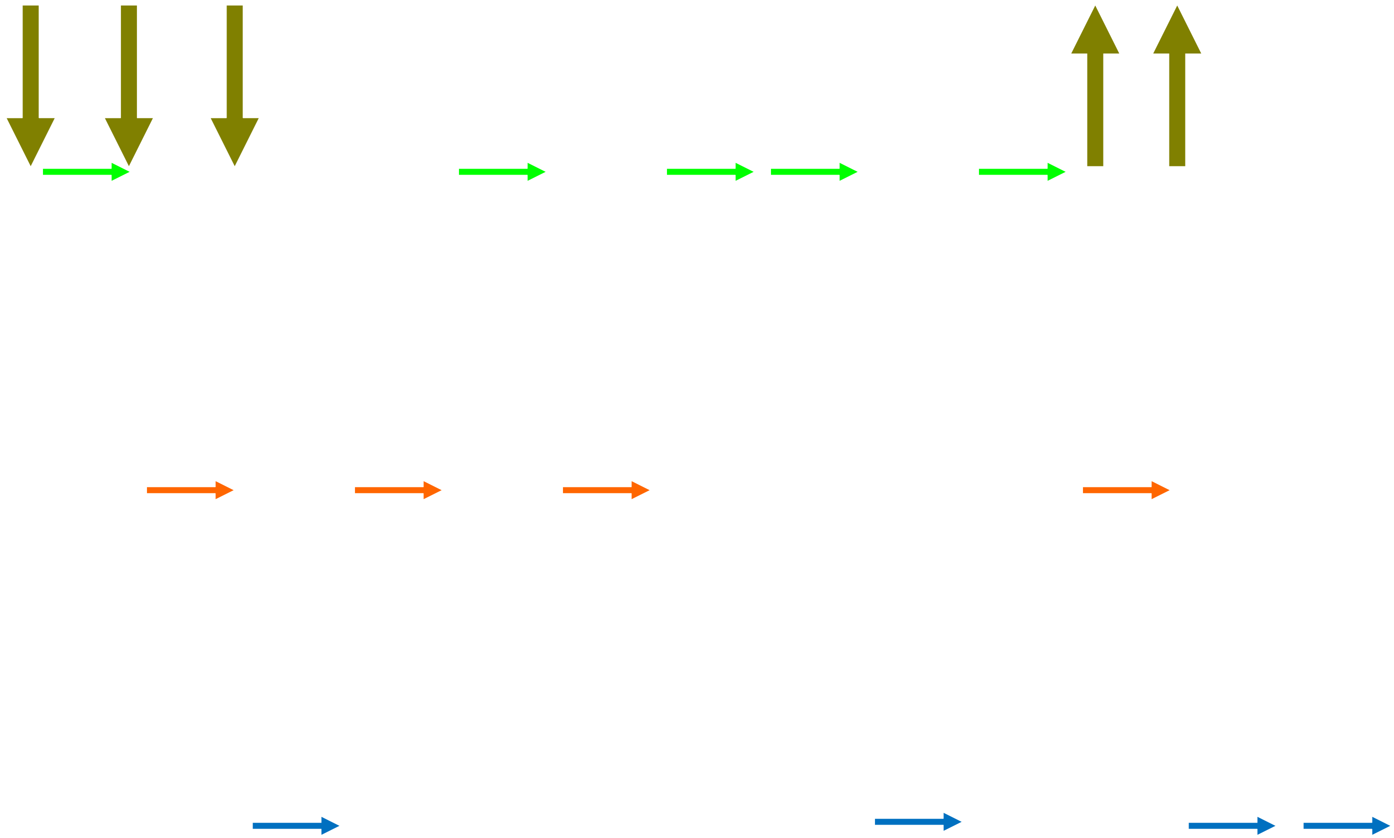
DISTRIBUTED SYSTEMS II

REPLICATION CNT.

Executing Operations



Interleaving Operations

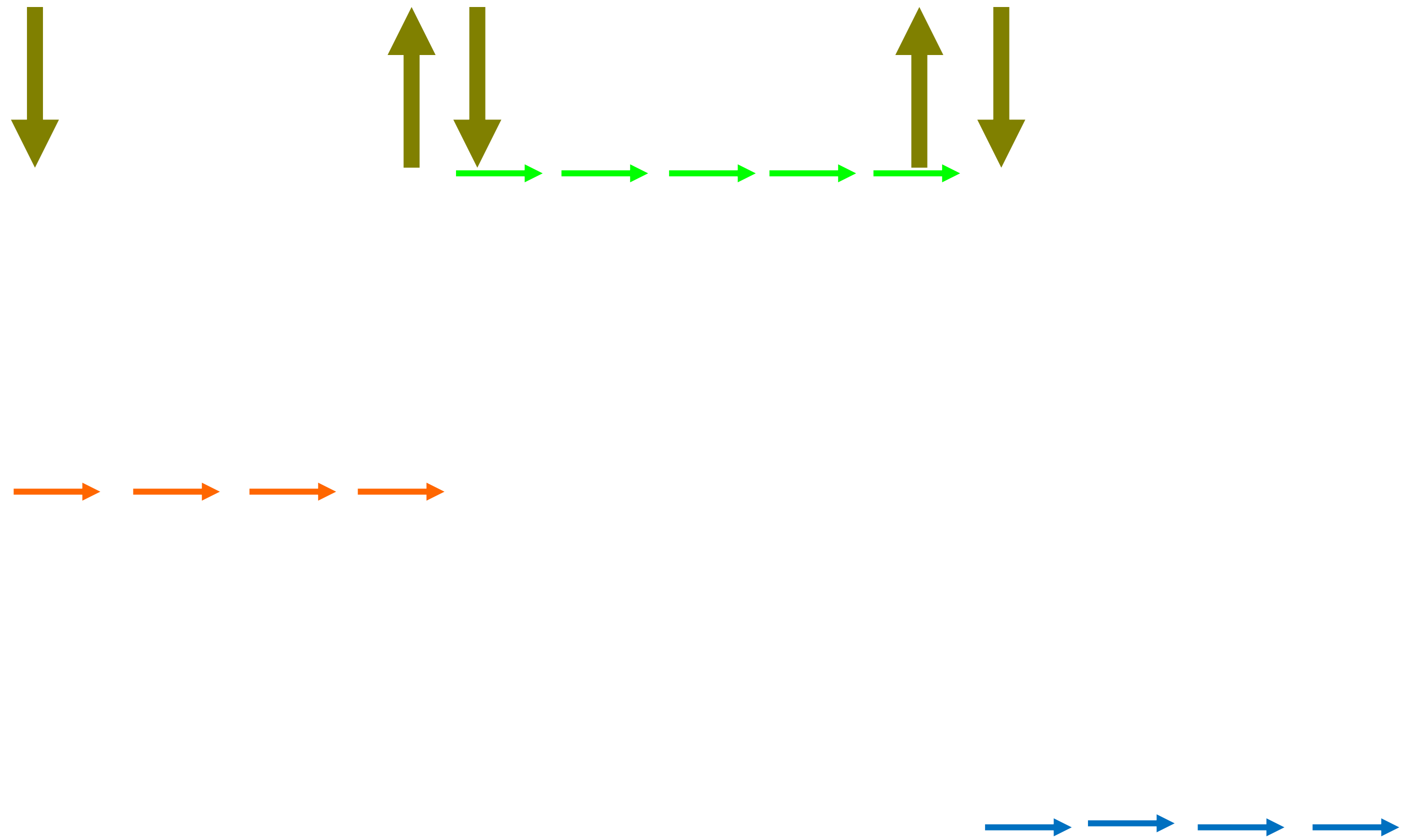


Concurrent execution

Interleaving Operations

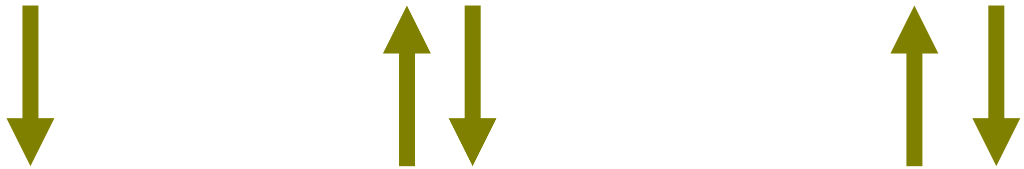
(External) behavior

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

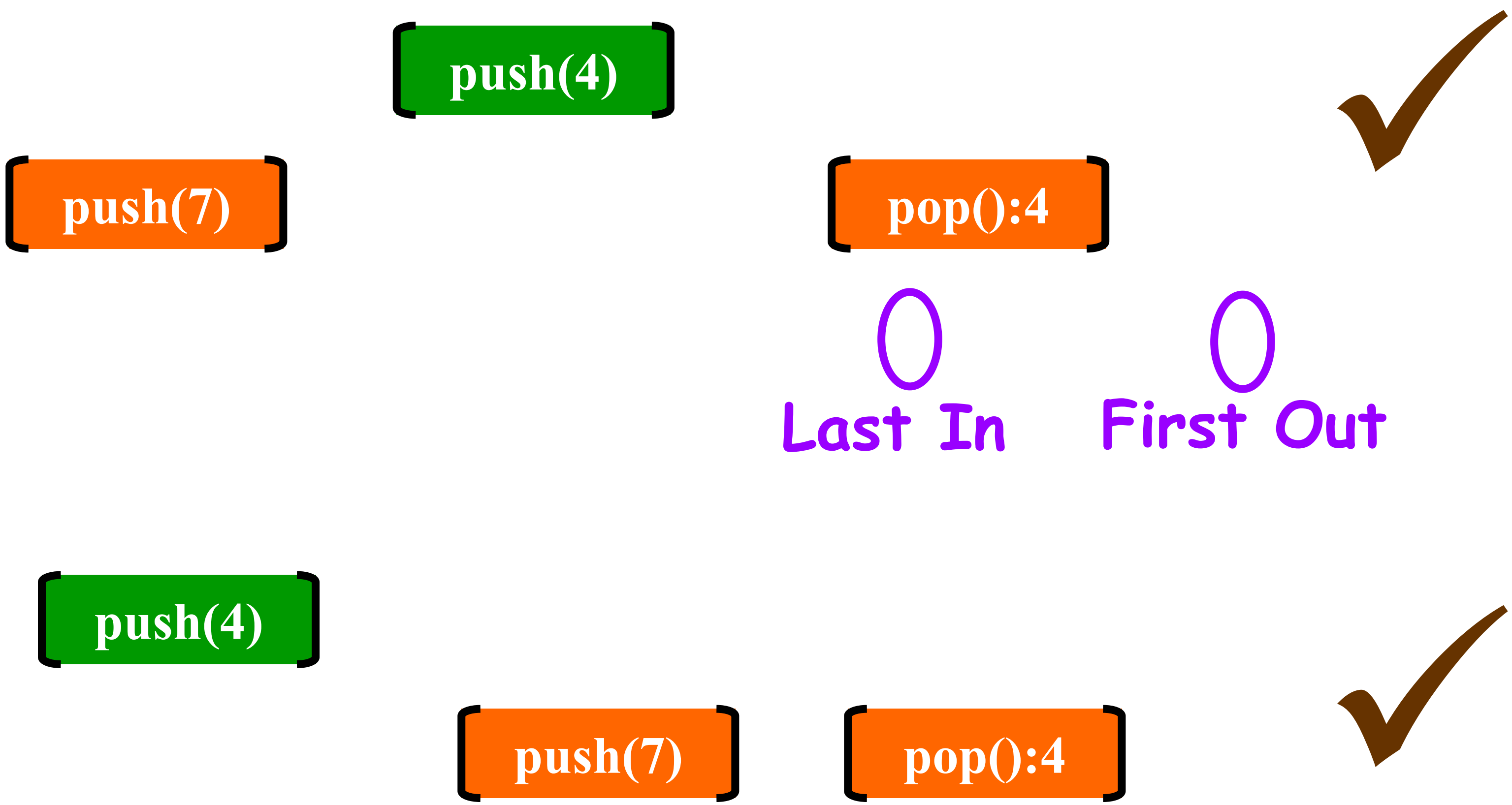
Correctness: Sequential consistency

[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

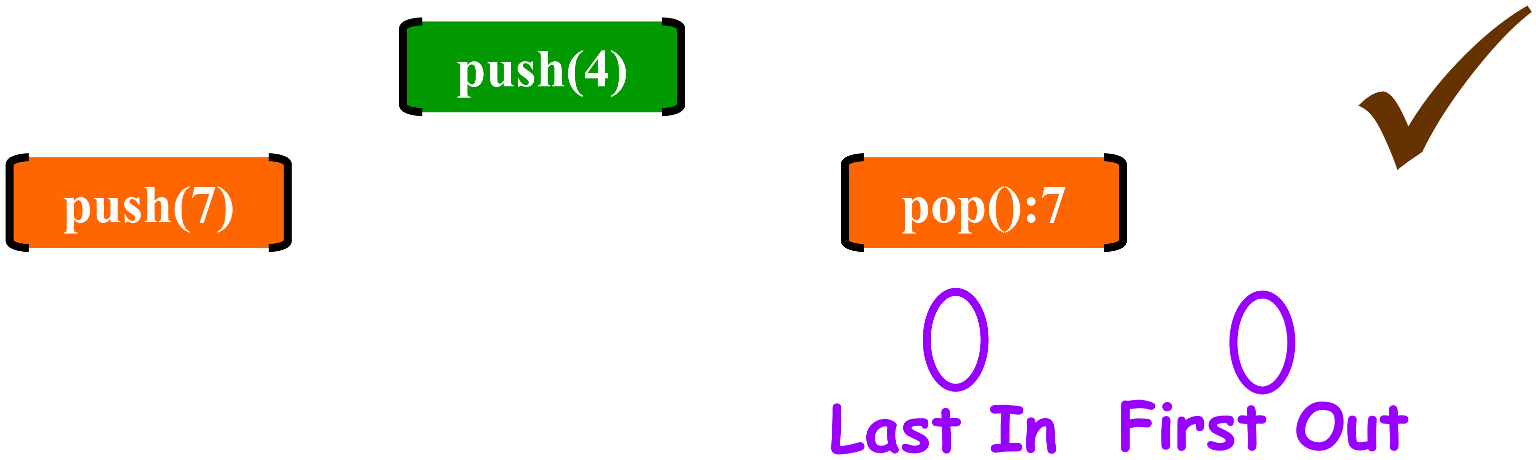
Sequential Consistency: Examples

Concurrent (LIFO) stack



Sequential Consistency: Examples

Concurrent (LIFO) stack



Sequential Consistency is not Composable

[enq(Q_1,X)] [enq(Q_2,Y)] [enq(Q_2,X)] [enq(Q_1,Y)] [Deq (Q_1,Y)] [deq(Q_2,X)]

The execution is not sequentially consistent

enq(Q_1,Y) ->enq(Q_1,X) =>

enq(Q_2,Y)->enq(Q_2,X)

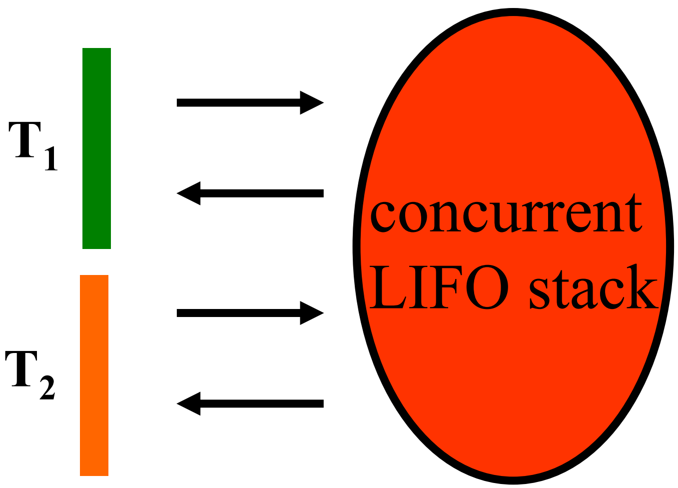
Sequential Consistency is not Composable

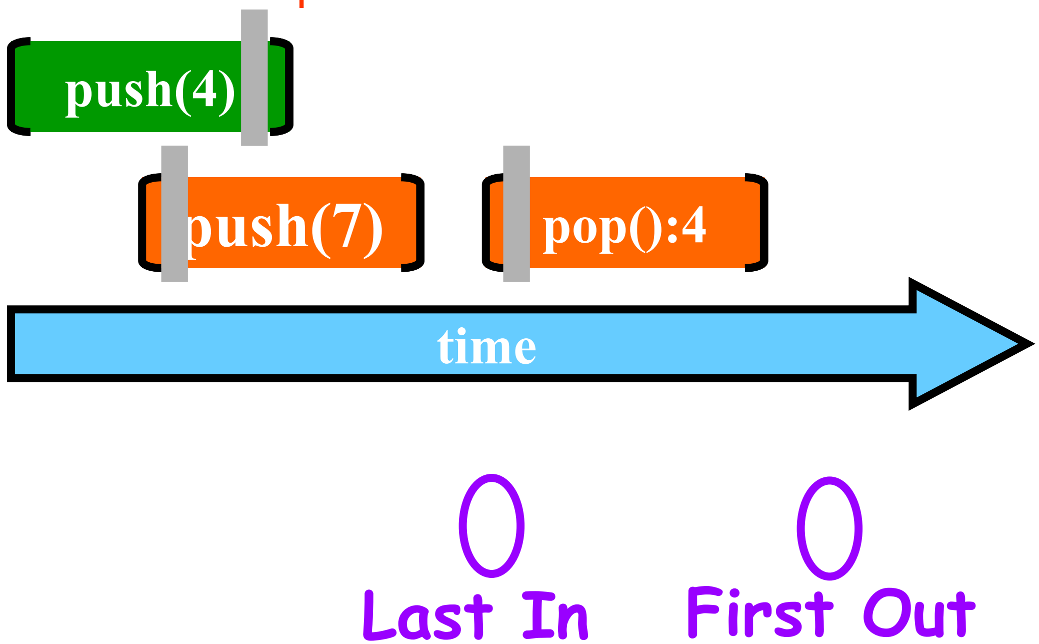
enq(Q_1,X) **enq(Q_2,Y)** **enq(Q_2,X)** **enq(Q_1,Y)** **deq(Q_1,Y)** **deq(Q_2,X)**

The execution projected on each object is sequentially consistent

Safety: Linearizability

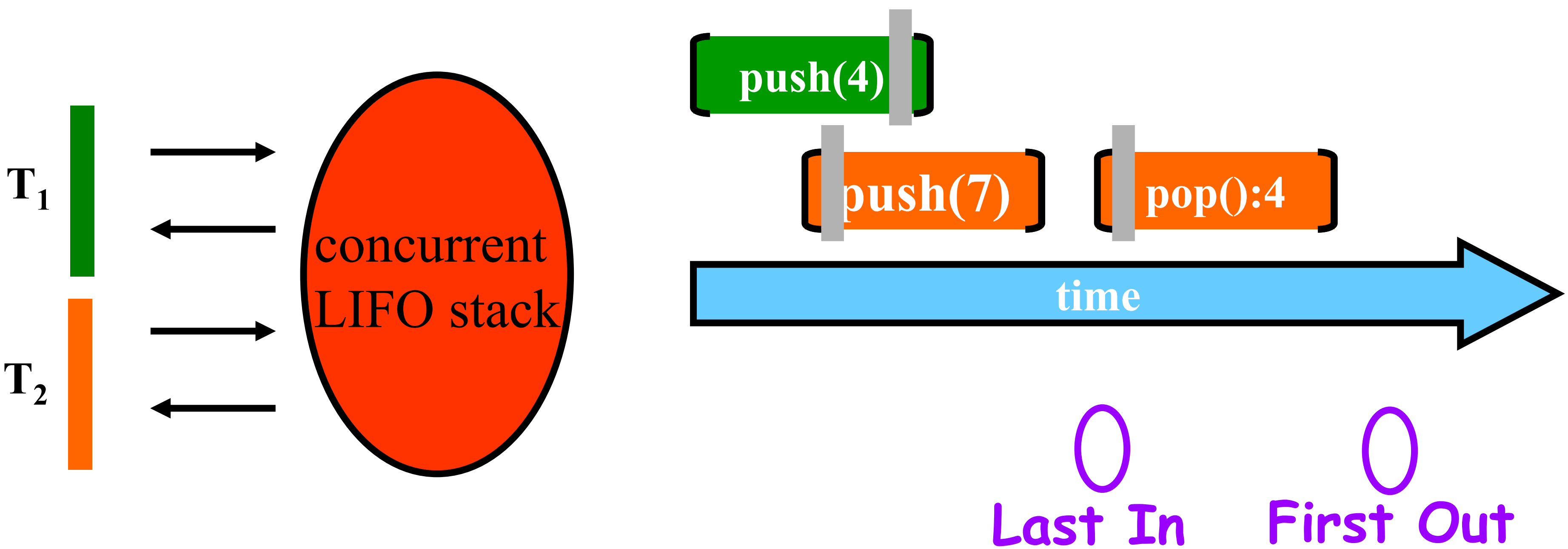
- **Sequential specification** defines legal sequential executions
- Concurrent operations allowed to be **interleaved**
- For every concurrent execution there is a sequential execution that
  - Contains the same operations
  - Is legal (obeys the sequential specification)
  - Preserves the real-time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



it is not linearizable because client2's *getBalance* is after client 1's *setBalance* in real time.

the following is sequentially consistent but not linearizable

Client 1:	Client 2:
$setBalance_B(x,1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y,2)$	

this is possible under a naive replication strategy, even if neither *A* or *B* fails -

the update at *B* has not yet been propagated to *A* when client 2 reads it

but the following interleaving satisfies both criteria for sequential consistency :

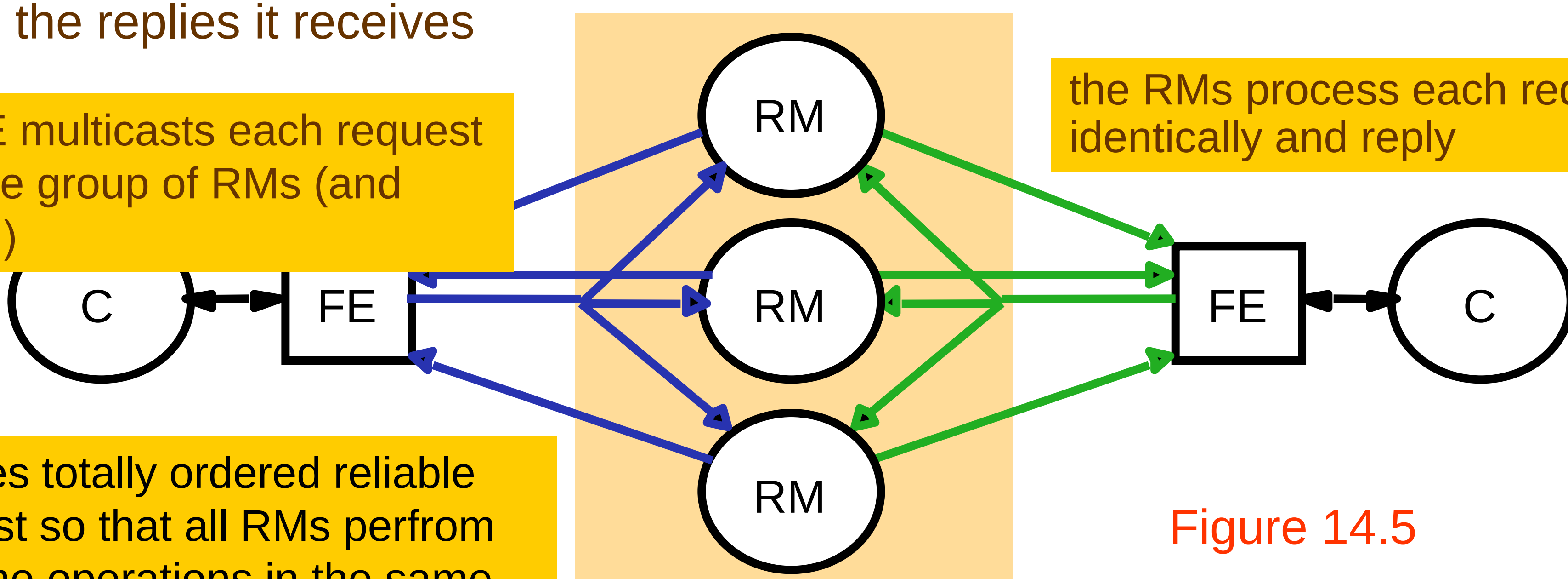
$getBalance_A(y) \rightarrow 0$; $getBalance_A(x) \rightarrow 0$; $setBalance_B(x,1)$; $setBalance_A(y,2)$

Active replication for fault tolerance: State Machine Approach

- 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



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

Figure 14.5

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.

•

Replication for Highly available services: The gossip approach

- we discuss the application of replication techniques to make services highly available.
  - we aim to give clients access to the service with:
    - reasonable response times for as much of the time as possible
    - even if some results do not conform to sequential consistency
    - e.g. a disconnected user may accept temporarily inconsistent results if they can continue to work and fix inconsistencies later
- eager versus lazy updates
  - fault-tolerant systems send updates to RMs in an ‘eager’ fashion (as soon as possible) and reach agreement before replying to the client
  - for high availability, clients should:
    - only need to contact a minimum number of RMs and
    - be tied up for a minimum time while RMs coordinate their actions
  - weaker consistency generally requires less agreement and makes data more available. Updates are propagated 'lazily'.

14.4.1 The gossip architecture

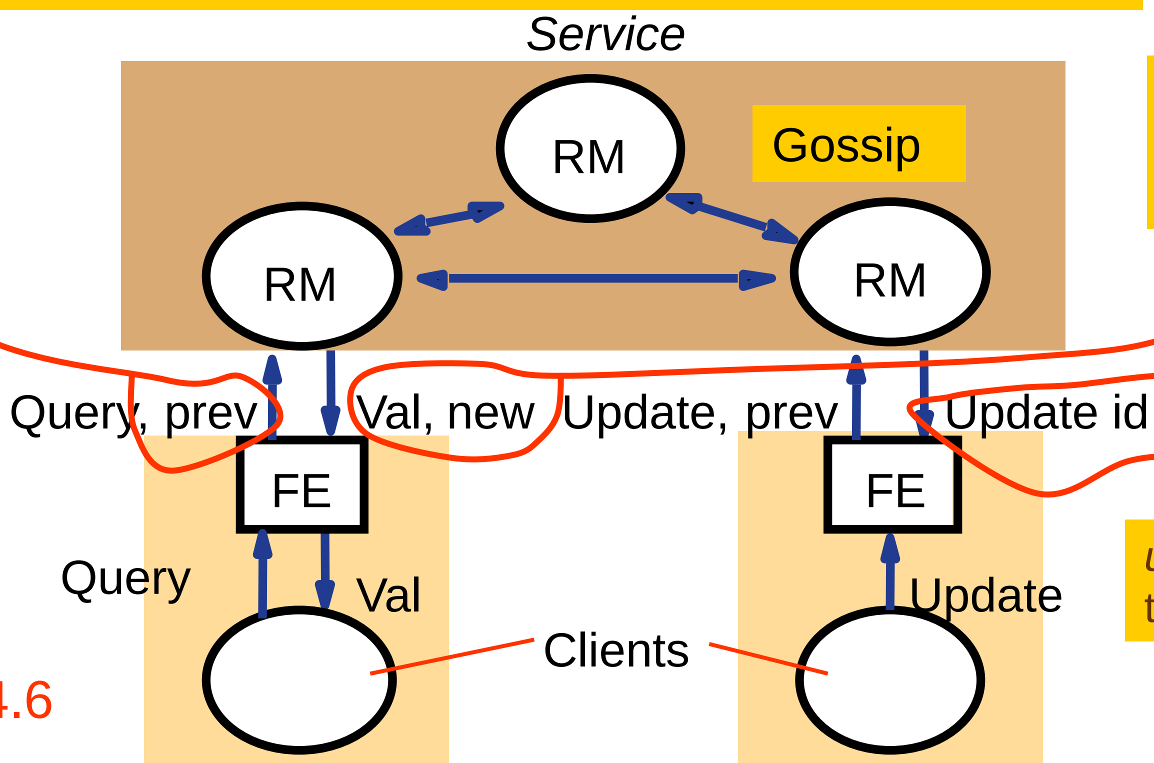
- the gossip architecture is a framework for implementing highly available services
  - data is replicated close to the location of clients
  - RMs periodically exchange 'gossip' messages containing updates
- gossip service provides two types of operations
  - queries - read only operations
  - updates - modify (but do not read) the state
- FE sends queries and updates to any chosen RM
  - one that is available and gives reasonable response times
- Two guarantees (even if RMs are temporarily unable to communicate
  - *each client gets a consistent service over time* (i.e. data reflects the updates seen by client, even if the use different RMs). Vector timestamps are used – with one entry per RM.
  - *relaxed consistency between replicas*. All RMs eventually receive all updates. RMs use ordering guarantees to suit the needs of the application (generally causal ordering). Client may observe stale data.

•

Query and update operations in a gossip service

- The service consists of a collection of RMs that exchange gossip messages
- Queries and updates are sent by a client via an FE to an RM

prev is a vector timestamp for the latest version seen by the FE (and client)



new is the vector timestamp of the resulting value, *val*

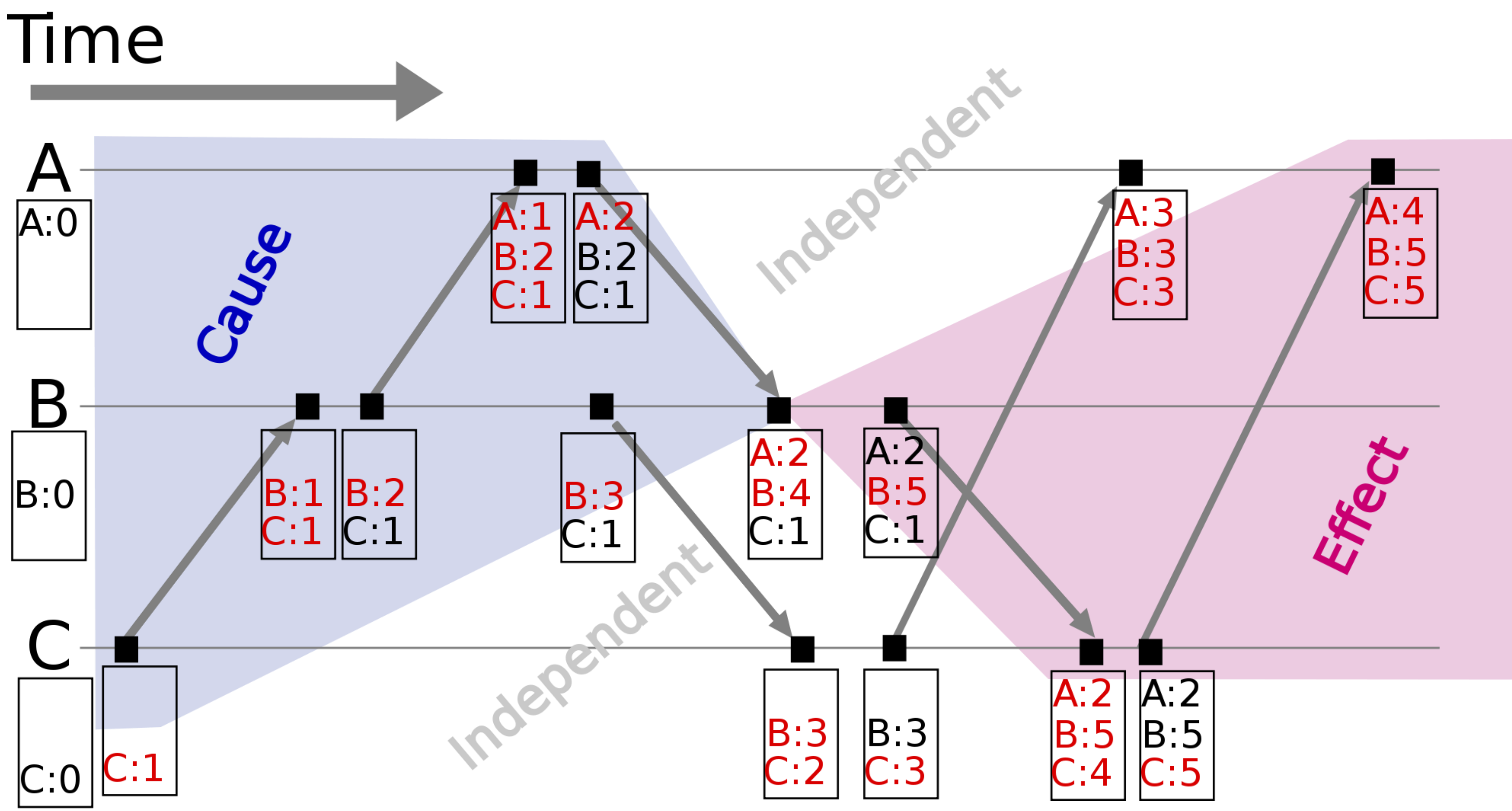
update id is the vector timestamp of the update

Figure 14.6

Causal ordering

Gossip processing of queries and updates

- The five phases in performing a client request are:
  - request
    - FEs normally use the same RM and may be blocked on queries
    - update operations return to the client as soon as the operation is passed to the FE
  - update response - the RM replies as soon as it has seen the update
  - coordination
    - the RM waits to apply the request until the ordering constraints apply.
    - this may involve receiving updates from other RMs in gossip messages
  - execution - the RM executes the request
  - query response - if the request is a query the RM now replies:
  - agreement
    - RMs update one another by *exchanging* gossip messages (lazily)
      - e.g. when several updates have been collected
      - or when an RM discovers it is missing an update



Time
Cause
Independent
Independent
Effect
A
B
C
A:0
B:0
C:0
C:1
B:1
C:1
B:2
C:1
A:1
B:2
C:1
A:2
B:2
C:1
B:3
C:1
A:2
B:4
C:1
B:3
C:2
A:2
B:5
C:1
B:3
C:3
A:3
B:3
C:3
A:2
B:5
C:4
A:2
B:5
C:5
A:4
B:5
C:5

Front ends propagate their timestamps whenever clients communicate directly

- each FE keeps a vector timestamp of the latest value seen (*prev*)
  - which it sends in every request
  - clients communicate with one another via FEs which pass vector timestamps

client-to-client communication can lead to causal relationships between operations.

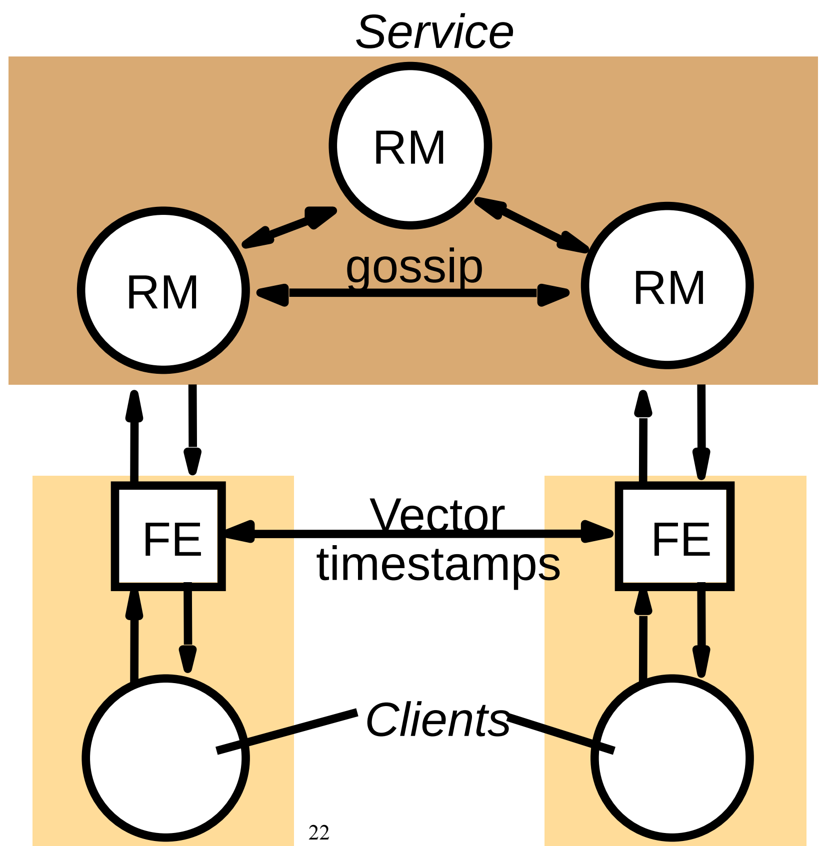


Figure 14.7

A gossip replica manager, showing its main state components

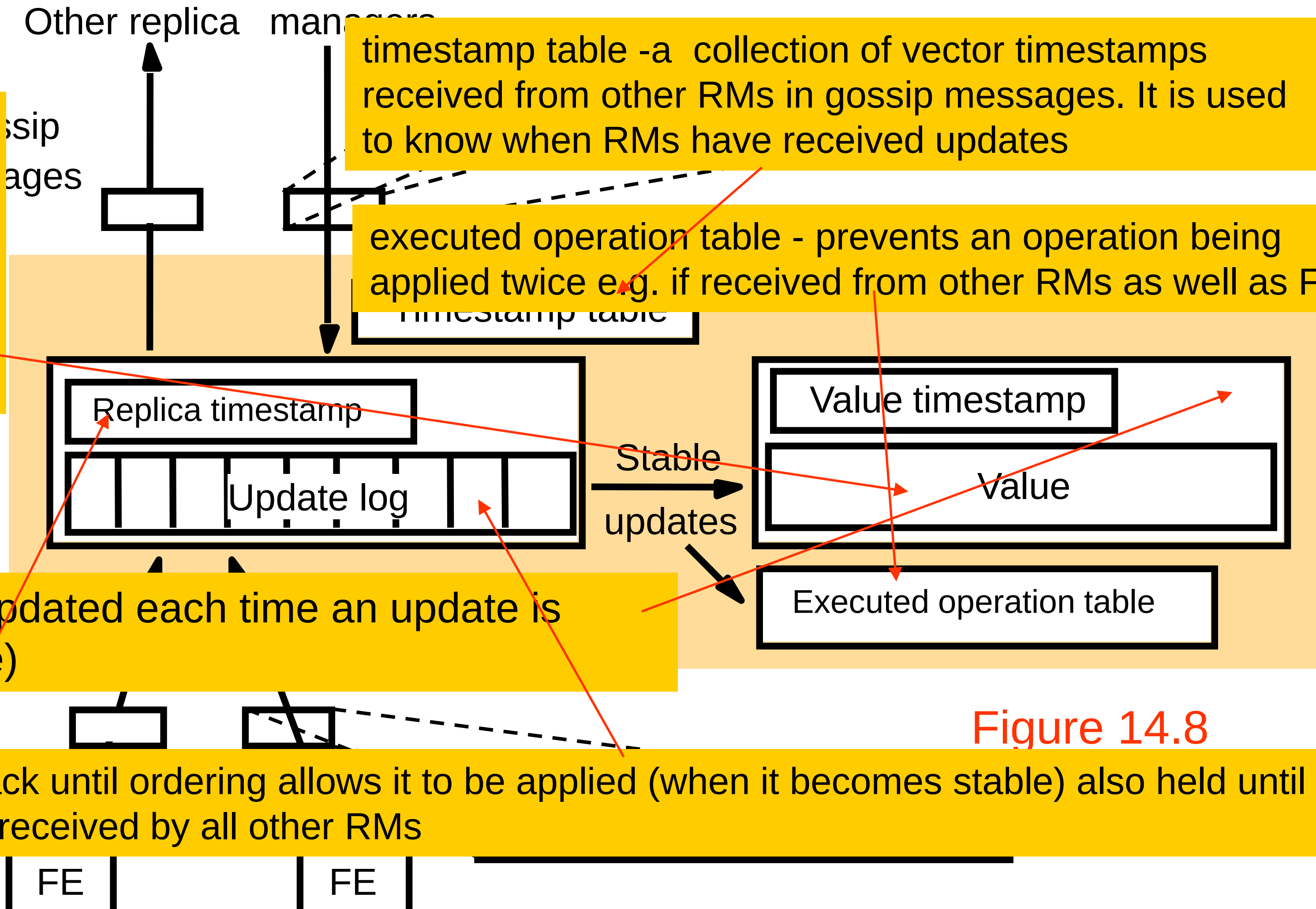


Figure 14.8

value - application state (each RM is a state machine) we are only talking about one value here

timestamp table -a collection of vector timestamps received from other RMs in gossip messages. It is used to know when RMs have received updates

executed operation table - prevents an operation being applied twice e.g. if received from other RMs as well as FE

value timestamp (updated each time an update is applied to the value)

update log - held-back until ordering allows it to be applied (when it becomes stable) also held until updates have been received by all other RMs

replica timestamp - indicates updates accepted by RM in log (different from value's timestamp if some updates are not yet stable)

•

RMs are numbered 0, 1, 2,…

Processing of query and update operations

e.g. in a gossip system with 3 RMs a value of (2,4,5) at RM 0 means that the value there reflects the first 2 updates accepted from FEs at RM 0, the first 4 at RM 1 and the first 5 at RM 2.

- Vector timestamp held by RM *i* consists of:
  - *i*th element holds updates received from FEs by that RM
  - *j*th element holds updates received by RM *j* and propagated to RM *i*
- Query operations contain *q.prev*
  - they can be applied if *q.prev* ≤ *valueTS* (value timestamp)
  - failing this, the RM can wait for gossip message or initiate them
    - e.g. if *valueTS* = (2,5,5) and *q.prev* = (2,4,6) - RM 0 has missed an update from RM 2
  - Once the query can be applied, the RM returns *valueTS* (*new*) to the FE. The FE merges *new* with its vector timestamp

•

Gossip update operations

- Update operations are processed in causal order
  - A FE sends update operation *u.op*, *u.prev*, *u.id* to RM *i*
    - A FE can send a request to several RMs, using same id
  - When RM *i* receives an update request, it checks whether it is new, by looking for the *id* in its executed ops table and its log
  - if it is new, the RM
    - increments by 1 the *i*th element of its replica timestamp,
    - assigns a unique vector timestamp *ts* to the update
    - and stores the update in its log

    logRecord = <*i*, *ts*, *u.op*, *u.prev*, *u.id*>
  - The timestamp *ts* is calculated from *u.prev* by replacing its *i*th element by the *i*th element of the replica timestamp.
  - The RM returns *ts* to the FE,which merges it with its vector timestamp
  - For stability *u.prev* ≤ *valueTS*
  - That is, the valueTS reflects all updates seen by the FE.
  - When stable, the RM applies the operation *u.op* to the *value*,updates *valueTS* and adds *u.id* to the executed operation table.

Gossip messages

- an RM uses entries in its timestamp table to estimate which updates another RM has not yet received
  - The timestamp table contains a vector timestamp for each other replica, collected from gossip messages
- an RM receiving gossip message *m* has the following main tasks
  - merge the arriving log with its own (omit those with *ts* ≤ replicaTS)
  - apply in causal order updates that are new and have become stable
  - remove redundant entries from the log and executed operation table when it is known that they have been applied by all RMs
  - merge its replica timestamp with *m.ts*, so that it corresponds to the additions in the log

Discussion of Gossip architecture

- the gossip architecture is designed to provide a highly available service
- clients with access to a single RM can work when other RMs are inaccessible
  - but it is not suitable for data such as bank accounts
  - it is inappropriate for updating replicas in real time (e.g. a conference)
- scalability
  - as the number of RMs grow, so does the number of gossip messages
  - for R RMs, the number of messages per request (2 for the request and the rest for gossip) = $2 + (R-1)/G$
    - G is the number of updates per gossip message
    - increase G and improve number of gossip messages, but make latency worse
    - for applications where queries are more frequent than updates, use some read-only replicas, which are updated only by gossip messages

The Quorum consensus method for Replication

- To prevent transactions in different partitions from producing inconsistent results
  - make a rule that operations can be performed in only one of the partitions.
- RMs in different partitions cannot communicate:
  - each subgroup decides independently whether they can perform operations.
- A *quorum* is a subgroup of RMs whose size gives it the right to perform operations.
  - e.g. if having the majority of the RMs could be the criterion
- in quorum consensus schemes
  - update operations may be performed by a subset of the RMs
    - and the other RMs have out-of-date copies
    - version numbers or timestamps are used to determine which copies are up-to-date
    - operations are applied only to copies with the current version number

Gifford's quorum consensus file replication scheme

- a number of 'votes' is assigned to each physical copy of a logical file at an RM
  - a vote is a weighting giving the desirability of using a particular copy.
  - each *read* operation must obtain a read quorum of *R* votes before it can read from any up-to-date copy
  - each *write* operation must obtain a write quorum of *W* votes before it can do an update operation.
  - *R* and *W* are set for a group of replica managers such that
    - *W* > half the total votes
    - *R* + *W* > total number of votes for the group
  - ensuring that any pair contain common copies (i.e. a read quorum and a write quorum or two write quora)
  - therefore in a partition it is not possible to perform conflicting operations on the same file, but in different partitions.