SR (Systèmes Répartis)

Unit 6: Synchronisation in distributed systems

François Taïani

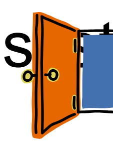
Introduction

- Distributed execution -> coordination needed
  - ➔ but not more than needed
  - ➔ as potentially very costly
  - ➔ must handle potential problems of DS (notably failures)

- Different types of coordination for different properties
  - ➔ ordering (consistency of message order, Unit 5)
  - ➔ mutual exclusion (this unit)
  - ➔ distributed transactions (this unit)

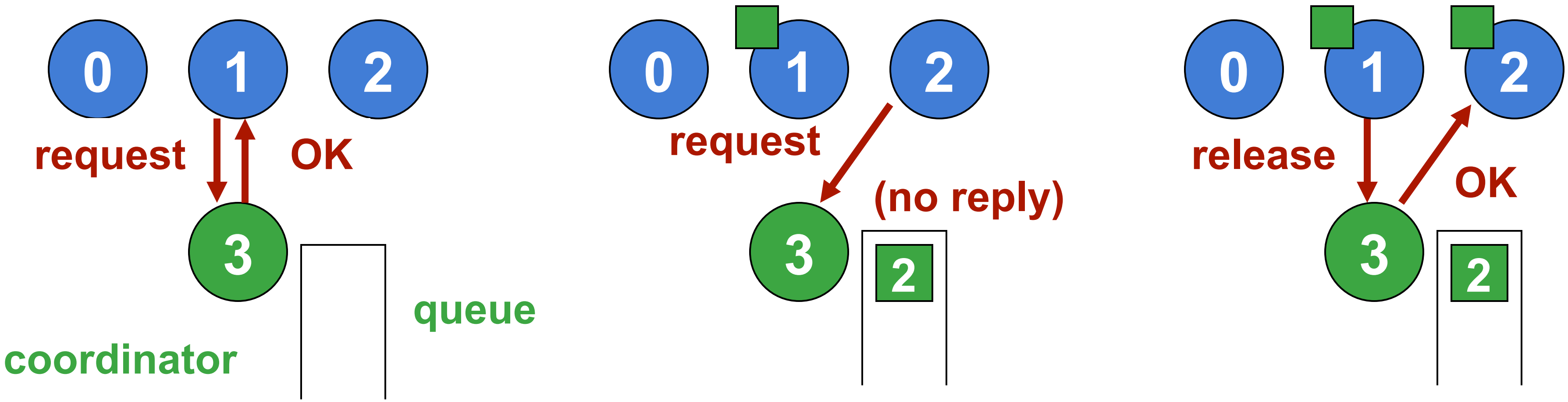
Mutual Exclusion

- **Mutual exclusion**
  - ➔ only one process can use it or access it at a time
  - ➔ example: a printer, a flight seat, a communication link
- Mutual exclusion not limited to distributed systems
  - ➔ cf. concurrency programming (multithreading)
  - ➔ cf. databases and transaction processing
- Mutual exclusion very important in distributed systems



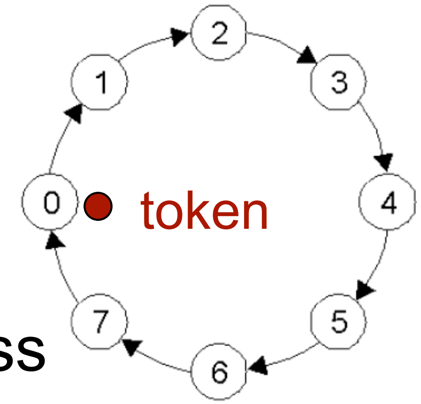
  - ➔ multiple hosts/processes executing in parallel
  - ➔ some orchestration needed to avoid chaos

Mutual Exclusion: A Centralized Algorithm



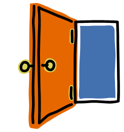
- A central server processes all lock allocation requests
- ☺: Easy to realise
- ☹: Not scalable, not fault-tolerant
  - ➔ client 1 could die or keep the lock (can be addressed)
  - ➔ server is a point of failure & a bottleneck (replication can help, e.g. Chubby at Google)

A Token Ring Algorithm



- Distributed processes organised in a ring
- A logical token circles the ring
  - ➔ only one process has the token at any time
- If a process want to mutual-exclusive access
  - ➔ **waits** to get the **token**
  - ➔ **keep** the **token** as long as mutual access needed
  - ➔ to release mutual access, **release** the **token**
- ☺: Process crashes easier to handle
  - ➔ "just" detect and rebuild the ring
- ☹: No bottleneck but not really scalable (ring size)
- ☹: Token can get lost (solutions exist)

(Distributed) Transactions



- We must first understand what a transaction is
  - ➔ see database module
- Transaction: **Synchronisation** mechanisms to access (and modify) shared data concurrently
  - ➔ heavily used in **databases**
- Similar to a **commercial business "transaction"**
  - ➔ first negotiations on what is to be done
  - ➔ at any point during negotiations any party may back out
  - ➔ if agreement is found, all parties need to commit (contract)
- Computer Science: the same
  - ➔ between a client and a data repository (database usually)

(Distributed) Transactions

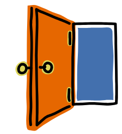
- A transaction engine provides the following operations
  - ➔ Begin_Transaction: start a transaction
  - ➔ End_Transaction: end transaction + try to commit
  - ➔ Abort: kill transaction & forget everything about it
- Examples of client behaviour:

Begin_Transaction
  book flight MAN-NYC → OK
  book hotel NYC 1 week → OK
End_Transaction

Begin_Transaction
  book flight MAN-NYC → OK
  book hotel NYC 1 week → full
Abort

ACID Properties

- ACID central to the DB course

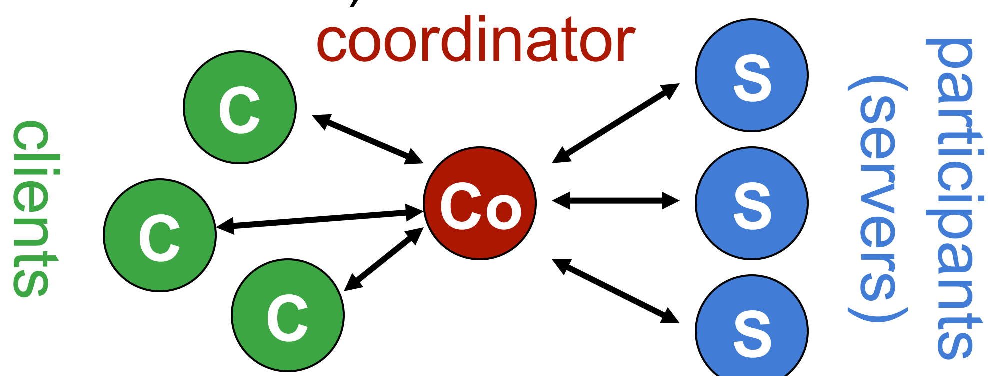


- Backstage the transaction engines ensures that
  - → transactions are atomic: all or nothing
  - → consistent: leave the system in a valid state
  - → isolated: don't interfere with each other
  - → durable: once successful, changes permanent

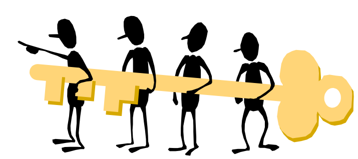
ACID

Distributed Transactions

- What is a distributed transaction?
  - ➔ A transaction where operations involve multiple servers (e.g.: **airline** + **hotel**)



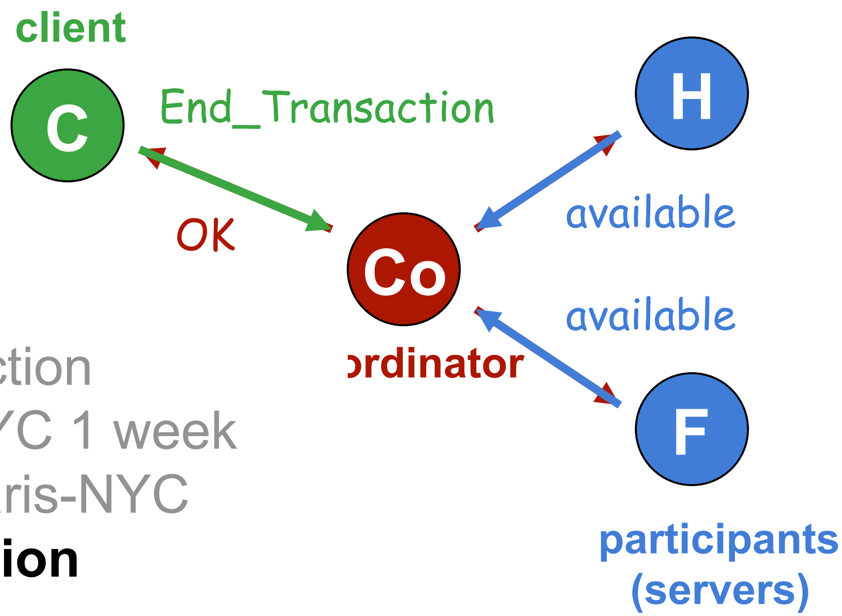
- Issues
  - ➔ Need distributed algorithms for concurrency control and recovery schemes mentioned above
  - ➔ Must deal with added difficulty of distributed deadlock
  - ➔ Crucial issue of **atomic commit protocols**



The 2-Phase Commit Protocol

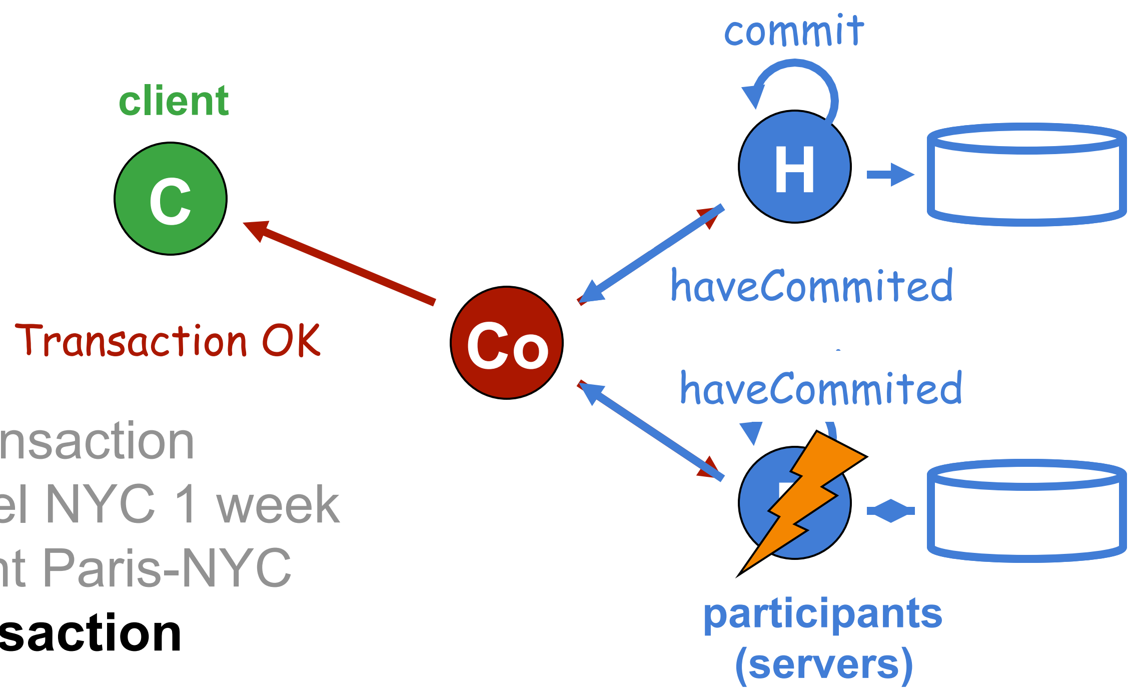
- **Phase 1** (voting phase):
  1. coordinator sends a canCommit? request to all participants
  2. on receiving canCommit? each participant replies yes or no
     if yes it saves the transaction state into **permanent storage** *before* sending the yes reply
     if no it aborts immediately
- **Phase 2** (completion according to vote)
  3. **if** all participants answer yes (including the coordinator) the coordinator sends doCommit to all participants
     else coordinator sends doAbort to participants that voted yes
  4. yes-voting participants wait for a doCommit or doAbort from coordinator, and act accordingly. In case of doCommit they send a haveCommited acknowledgement

Distributed Transaction: An Example



Begin_Transaction
  book hotel NYC 1 week
  book flight Paris-NYC
End_Transaction

(a) Commit Succeeds

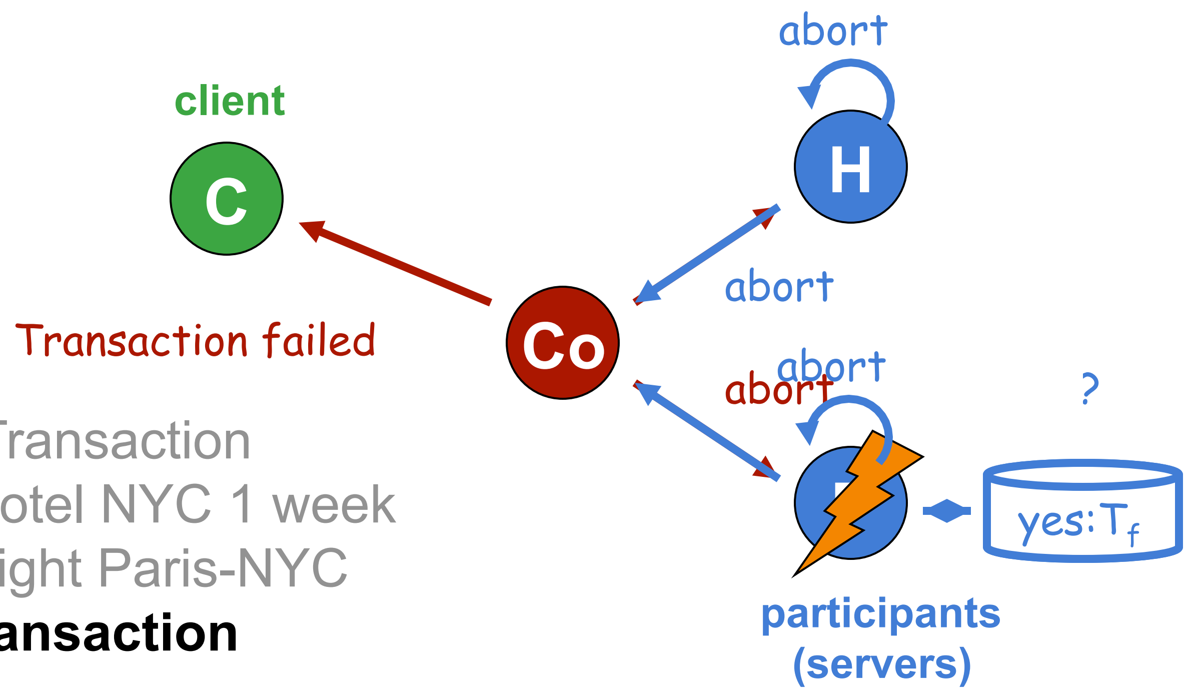


Begin_Transaction
  book hotel NYC 1 week
  book flight Paris-NYC
End_Transaction

Important Points

- Once the coordinator has sent **doCommit** only a commit is possible
- If one of the participants fails:
  - → Transaction is **blocked** until the participant resumes
  - → **Consistency** is ensured because of transaction state being saved on **stable storage**
  - → On resuming the failed participants check its disk to know which transactions to ask the coordinator about
- Many details *not* represented
  - → The coordinator uses stable storage as well against failure
  - → Distributed locking protocol omitted
  - → Typically *multiple* clients performing transaction at the same time

(b) Commit Fails



Begin_Transaction
  book hotel NYC 1 week
  book flight Paris-NYC
End_Transaction

Commit fails: important points

- Same basic mechanisms as when all agree
- Except here **no need to wait** for failed servers:
  - ➔ It is their responsibility to catch up
- Failed servers still need to **check** with coordinator
  - ➔ They do not know the outcome of the transaction
  - ➔ The transaction could have succeeded

Transactional Middleware: Transaction Processing Monitors

- Standards
  - ➔X/Open DTP, OMG Corba OTS, J2EE
- Key products
  - ➔IBM's CICS and Encina (Transarc)
  - ➔Oracle's Tuxedo
  - ➔Microsoft's MTS (included in COM+)
  - ➔SUN's Enterprise JavaBeans (EJB)
  - ➔JBoss, JOnAS (open source J2EE products)

Summary

- Two types of coordination for distributed systems
  - ➔ mutual exclusion
  - ➔ distributed transactions
- Solution for mutual exclusion
  - ➔ centralised
  - ➔ token-based
  - ➔ versions we have seen = no fault tolerance!
- Solution for distributed transactions
  - ➔ 2PC (OK if coordinator not permanently failed)
  - ➔ better protocol (not seen): 3PC