SR (Systèmes Répartis)

Unit 9: Fault Tolerance II

François Taïani

Advertisement!

- Guest lecture: Fri 8 Feb (Unit 11)

→ **Davide Frey (INRIA)**

- World-wide expert decentralised social networks. Don't miss him!

Overview of the Session

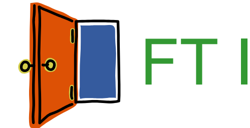
Investigate advanced issues of fault-tolerance

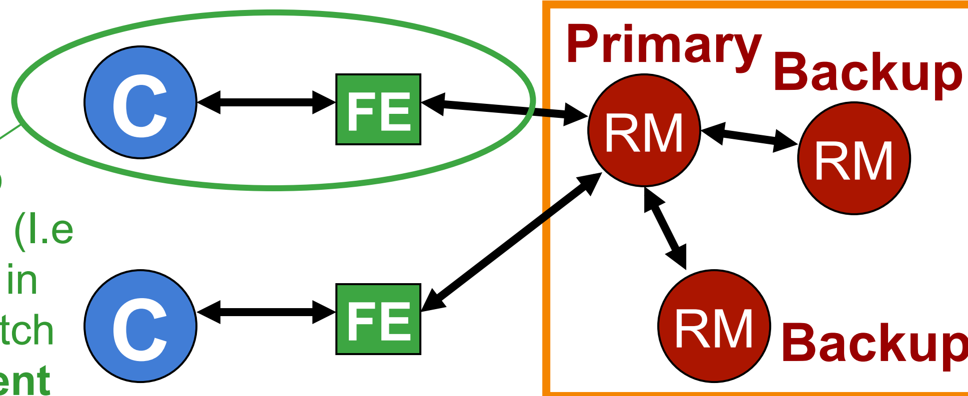
- Passive replication
  - ➔ output commit problem
  - ➔ how to provide exactly once semantics
- Active replication
  - ➔ Importance of total order multicast
  - ➔ Link total-order and consensus

Associated Reading: Section 7.5 (Distributed Commit) and 7.6 (Recovery) of Tanenbaum & van Steen; 15.4 (Hierarchical and group masking of faults) of Coulouris & al

Replication & Consistency

- Reminder: **passive replication** (aka primary backup)
  - → FEs communicate with a single **primary** Replication Managers (RM), which must then communicate with secondary RMs





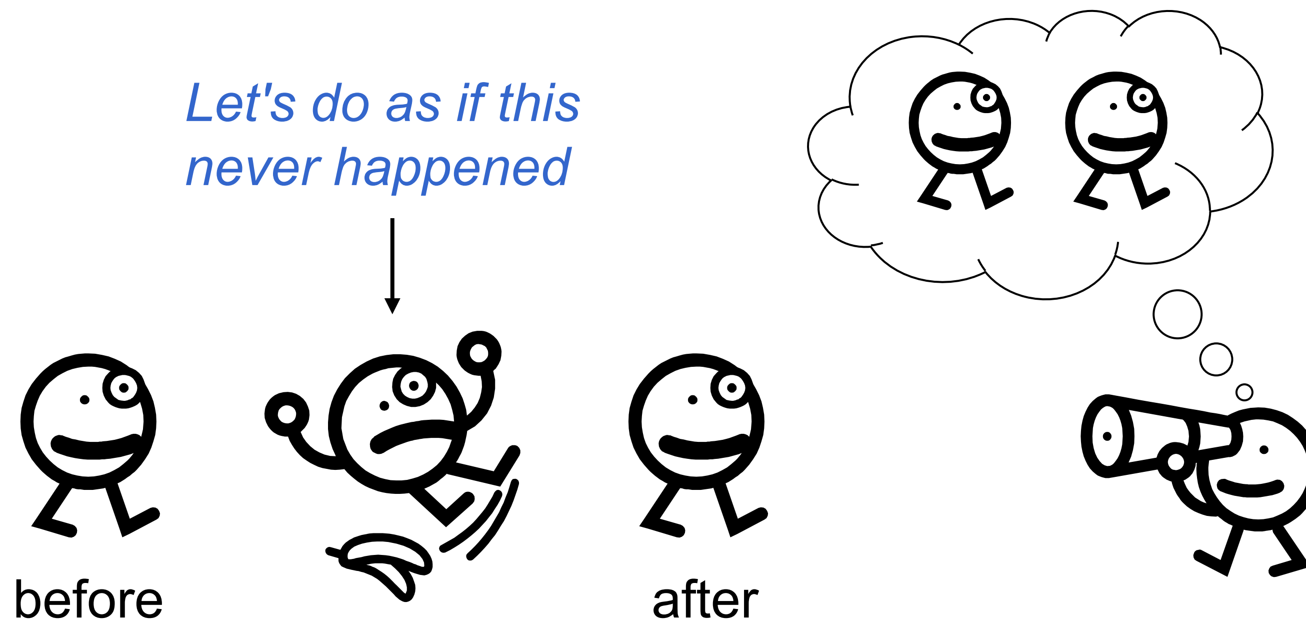
Note: for CW, OK to merge client and FE (I.e replicas hard-coded in client). However switch should be **transparent** to user.

  - → New primary must be **elected** in case of primary failure
  - → Only tolerates **crash faults** (silent failure of replicas)

Consistency & Recovery

- "A system recovers correctly if its internal state is consistent with the observable behaviour of the system before the failure"

[Strom and Yemini 1985]

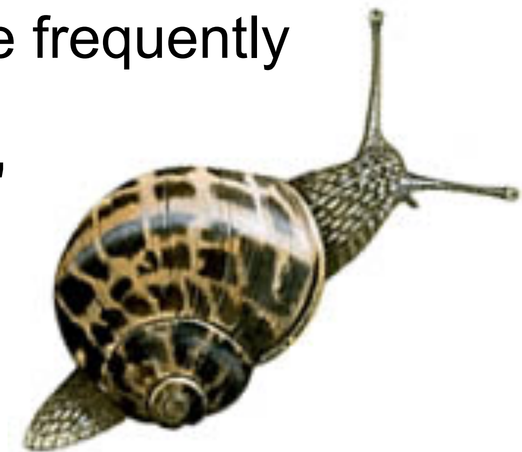


Replication & Consistency

The problem with Passive Replication

- Primary / backup hand-over ➔ **consistency** issue
  - ➔ when primary crashes, backup might be **lagging** behind
  - ➔ backup may not resume exactly were primary left
  - ➔ risk of **inconsistency** from the client point of view

- How to avoid this? *(aka checkpointing)*
  - ➔ synchronise backup with primary more frequently
  - ➔ but too frequent ➔ high overhead
  - ➔ "enough synchronisation but not more"

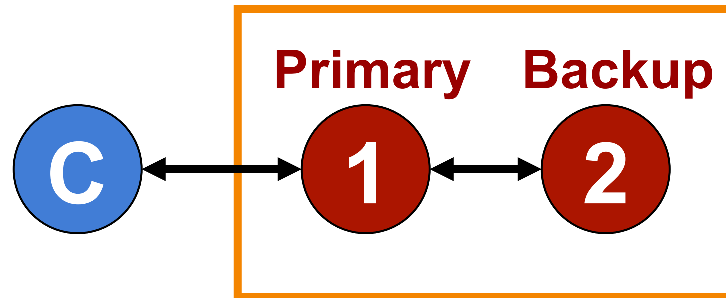


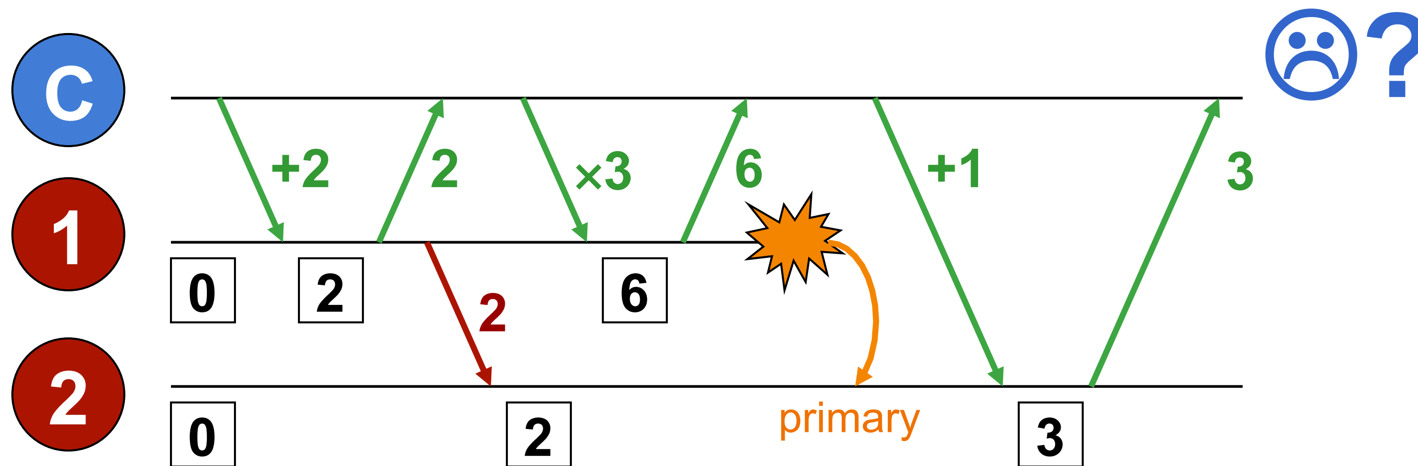
Our Assumptions

- In the following **we assume** the following
  - ➔ messages that are sent do arrive (**FIFO reliable** comm.)
  - ➔ switch from primary to backup is transparent to the client
  - ➔ client will **replay** requests for which it does not get replies
- **Our goal**
  - ➔ "smooth" hand-over to backup on primary crash
- We **don't consider** the following cases
  - ➔ backup crashes
  - ➔ client crashes
  - ➔ any arbitrary failure ("wrong" messages)

Nicolaus Copernicus (1473-1543)

Replication & Consistency

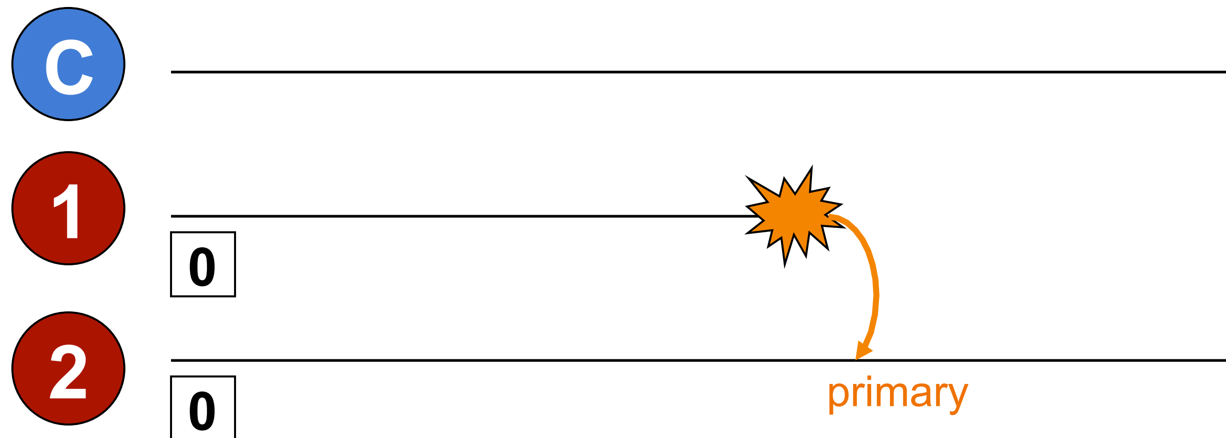




- How to avoid this?

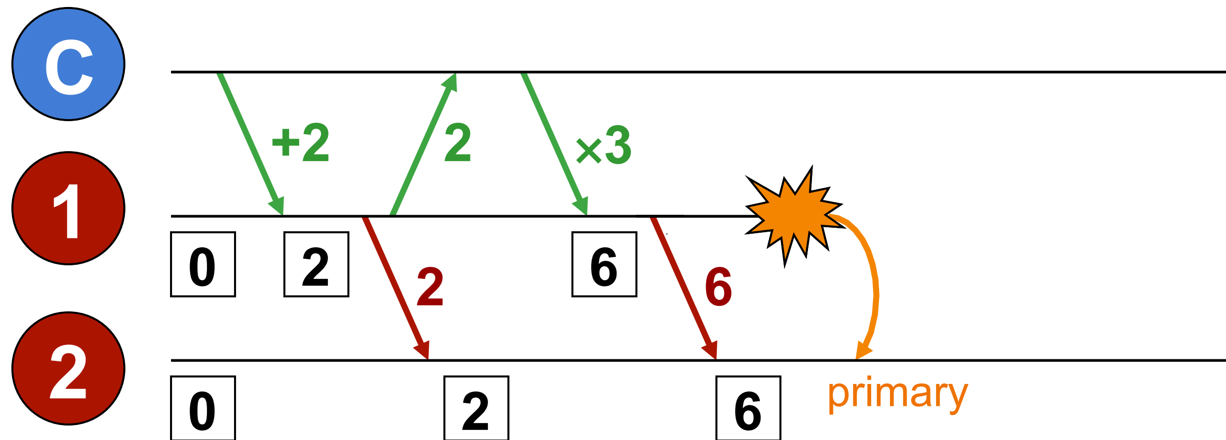
Output Commit Problem

- Algorithm

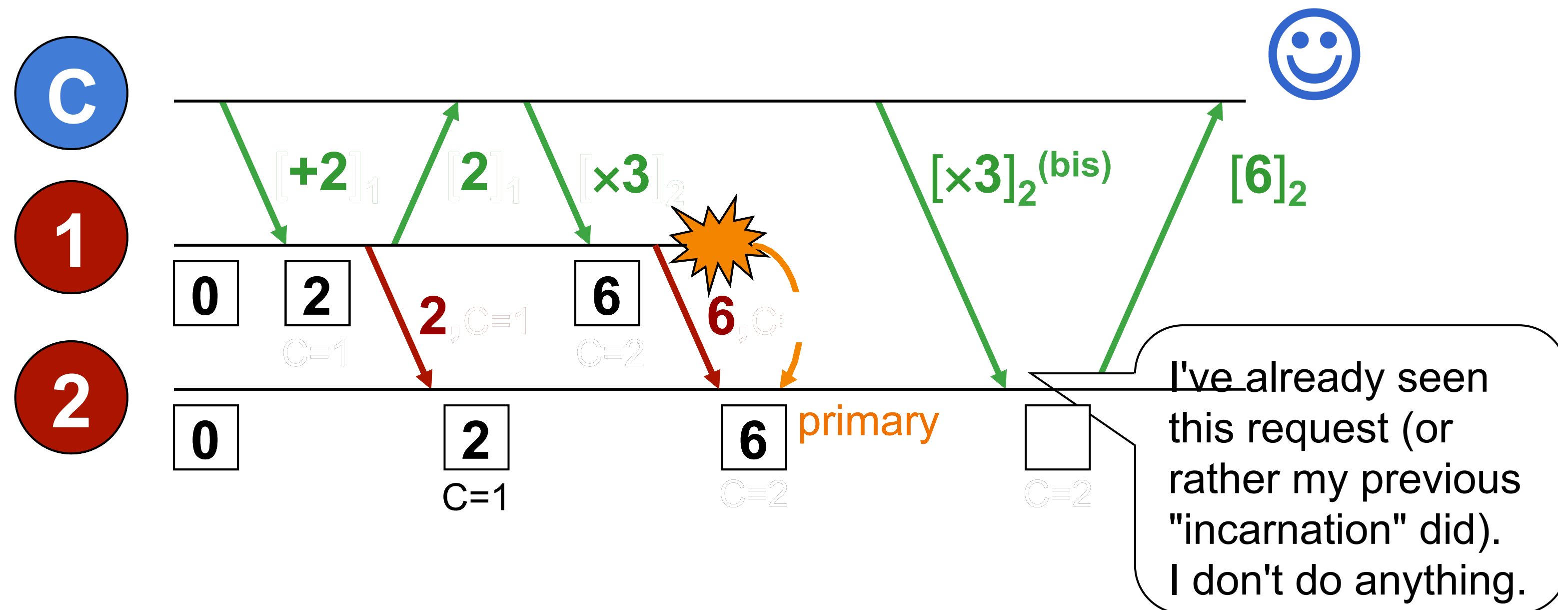


- There is still a problem with this new algorithm. Which one?

More-than-Once Problem



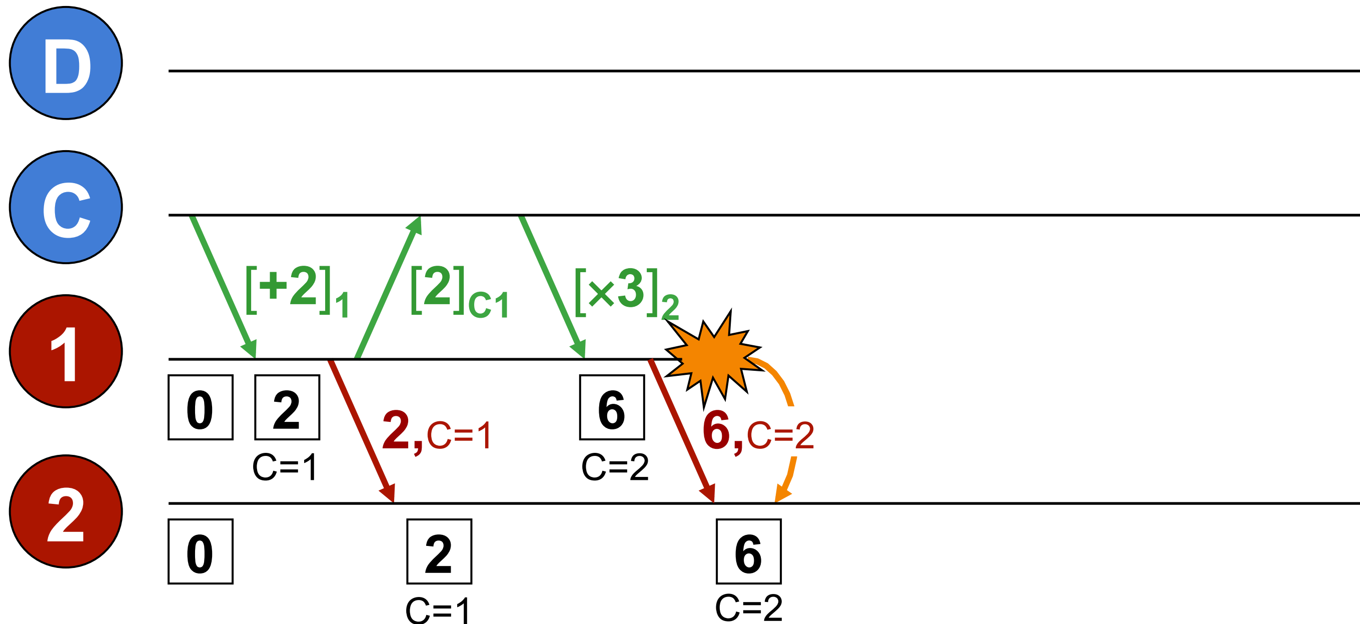
Exactly-Once: Solution 1



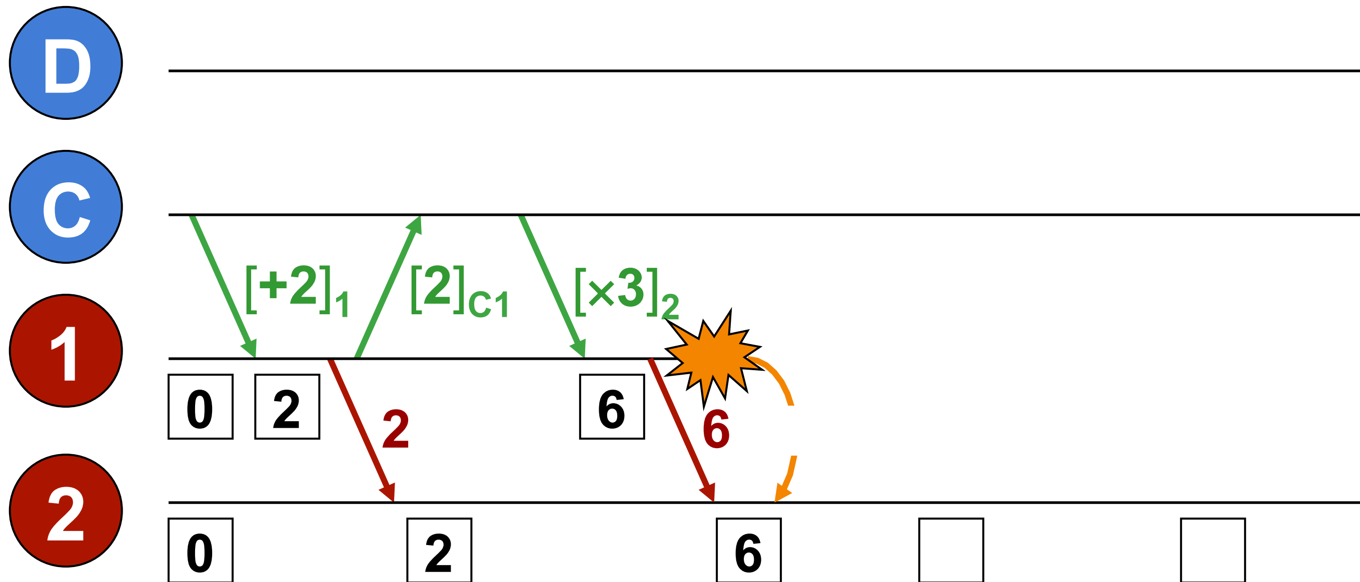
- This solution might break w/ multiple clients. Do you see how?

Multi-Client Problem

- Problem is **not** that C sees the effect of D's operation
  - ➔ This can happen in a failure-free execution. Valid result.
- Problem is that **no** failure-free execution could ever return 8



Exactly-Once: Solution 2
(multi-client safe)

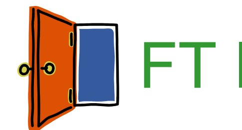


Notes on Solution 2

- *Smooth* hand-over from primary to backup on crash
  - → all operations are **executed exactly once**
- Primary failure is **masked** but not entirely transparent
  - → reply received much later than in failure-free case

→ Major disturbance (server crash) replaced by minor annoyance (network delay)

→ **Graceful degradation**: lesser quality of service but still running



FT I

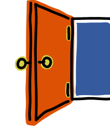
Further Comments

- Assumes bounded network delays (for switch): bad for WAN
- Previous algo **does not scale** to many clients / large state
  - ➔ If millions of clients and big database: intractable
- Solution to large state problem: use **logging**
  - ➔ “save” log of operations performed on primary
  - ➔ regularly "flush" log by checkpointing whole server state
  - ➔ on recovery: latest checkpoint + reapply current log
- All the above assume **sequential** server (i.e. monothreaded)
  - ➔ state saved when no “request in progress” (quiet state)
- Multithreading usually requires
  - ➔ "hot" checkpointing/ backup ability
  - ➔ (Not as good) alternative: Wait/create for ‘quiet’ state

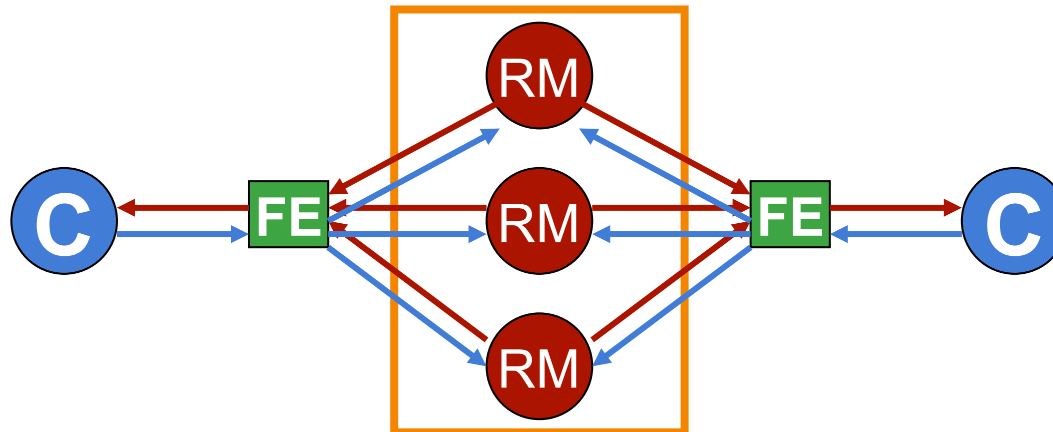
What to remember from this?

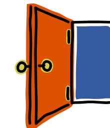
- The exact algorithms are not important
  - ➔ Although you should be able to re-analyse them
- What is important are the issues that were raised
- **Output commit problem**
  - ➔ Clients should not see changes that hasn't be made permanent
- **Duplicate requests and exactly once semantics**
  - ➔ No counter and retry ➔ at least-once-semantics
  - ➔ Counter and retry ➔ at most-once semantics
  - ➔ Exactly once requires some atomicity mechanism
  - ➔ What is atomic = "checkpointing" message to the backup
  - ➔ Either the backup receives it or it does not

Active replication



- Reminder
  - ➔ appropriate **fault-tolerant** group communications needed
  - ➔ **ordering** guarantees crucial (see Group Communication Session)

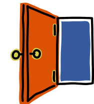




- Even more complex than passive replication
  - ➔ Some of the complexity encapsulated by group comm.
  - ➔ Pb: choosing the **right group comm.** for application & faults

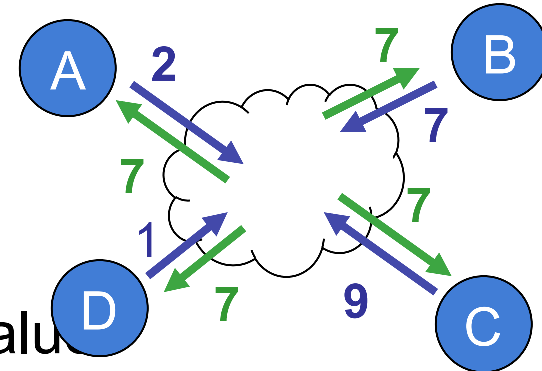
Realising Active Replication

- Group communication needed, with **total ordering** property
  - ➔ See example for what happens without total ordering
- In course for totally ordered multicast presented
  - ➔ centralised sequencer
  - ➔ based on time-stamping with logical clocks



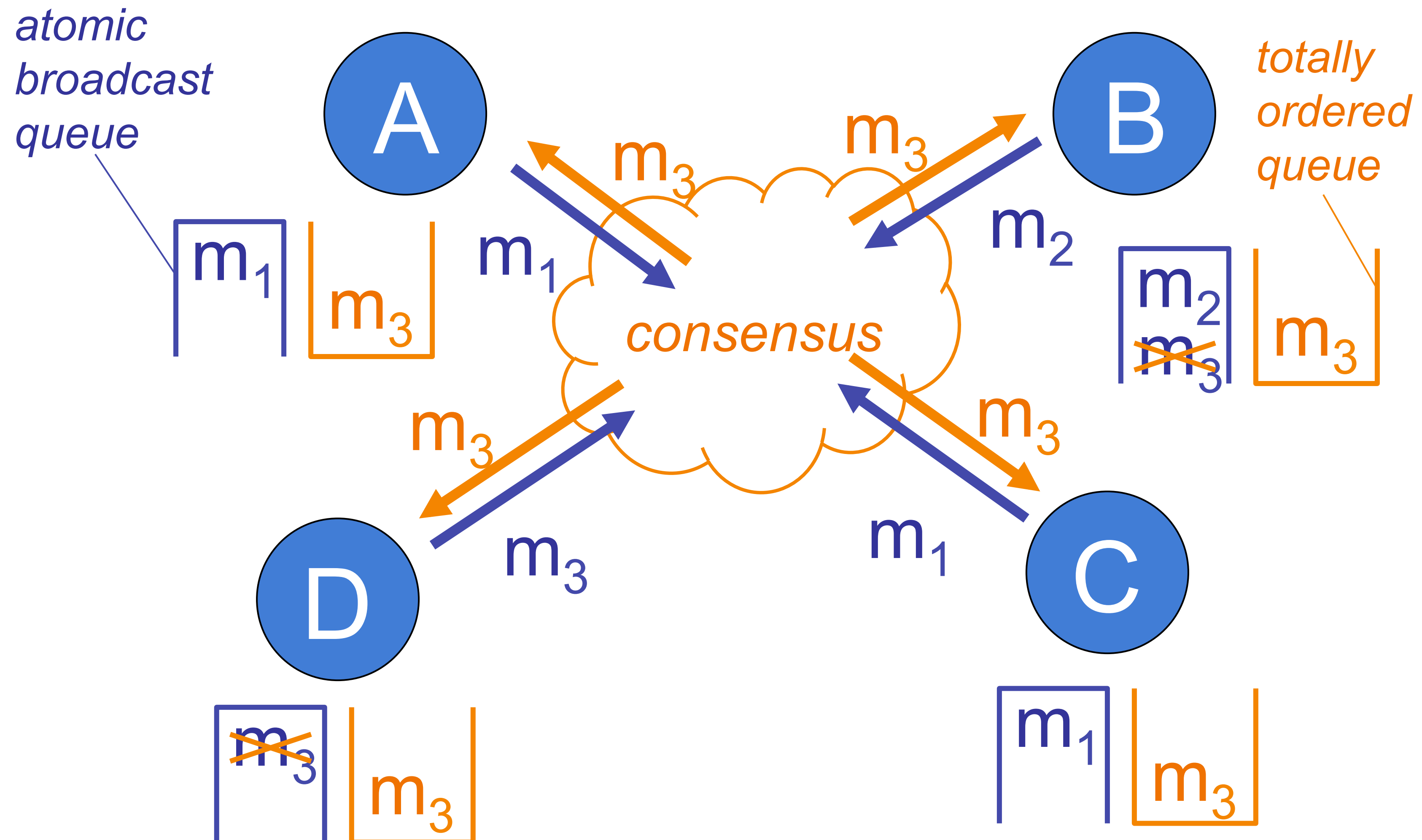
- Problem: none of them is **fault-tolerant**
  - ➔ the centralised sequencer is single point of failure
  - ➔ time-stamping: crash of any participant blocks the algorithm
- We need a fault-tolerant (atomic) totally ordered multicast
  - ➔ tolerating crash-fault if active replication used against them
  - ➔ tolerating arbitrary fault if active replication used against them

Total Ordering and Consensus



- Realising **total-ordering** is equivalent to realising **distributed consensus**
- **Distributed consensus**
  - ➔ All participants start by proposing a value
  - ➔ At the end of the algorithm one of the proposed value has been picked, and everybody agrees on this choice
- From distributed consensus to total ordering
  - ➔ Each participant proposes the next message it would like to accept
  - ➔ Using consensus everybody agrees on next message
  - ➔ This message is the next delivered
- **Fault-Tolerant consensus ➔ fault-tolerant total ordering**

Total Ordering and Consensus



Fault-Tolerant Consensus

- **Main idea 1:** not be blocked by crashed processes
  - ➔ use failure detection to stop waiting for crashed processes
- **Main idea 2**: propagate influence of crashed processes
  - ➔ before crashing a process might have comm with others
  - ➔ these messages must be share with all non-crashed processes (or with none of them). They might influence consensus outcome.
- The properties of the **failure detector** is essential
  - ➔ in reality **false positive** happen (time-out and slow network)
  - ➔ different algo for different classes of **imperfect failure detectors**
  - ➔ the more imperfect the less crashes can be tolerated
- FT Consensus algorithms even exist for **arbitrary failure**
  - ➔ Even more redundancy required

FT Consensus (strong failure detector, crash faults)

procedure $propose(v_p)$

$V_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$ $\quad\{p\text{'s estimate of the proposed values}\}$

$V_p[p] \leftarrow v_p$

$\Delta_p \leftarrow V_p$

Phase 1: $\{asynchronous\ rounds\ r_p,\ 1 \leq r_p \leq n-1\}$

for $r_p \leftarrow 1$ to $n-1$

send (r_p, Δ_p, p) to all

wait until $[\forall q:$ received (r_p, Δ_q, q) **or** $q \in \mathcal{D}_p]$ $\quad\{query\ the\ failure\ detector\}$

$msgs_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid \text{received } (r_p, \Delta_q, q)\}$

$\Delta_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$

for $k \leftarrow 1$ to n

if $V_p[k] = \perp$ **and** $\exists (r_p, \Delta_q, q) \in msgs_p[r_p]$ with $\Delta_q[k] \neq \perp$ **then**

$V_p[k] \leftarrow \Delta_q[k]$

$\Delta_p[k] \leftarrow \Delta_q[k]$

Phase 2: send V_p to all

wait until $[\forall q:$ received V_q **or** $q \in \mathcal{D}_p]$ $\quad\{query\ the\ failure\ detector\}$

$lastmsgs_p \leftarrow \{V_q \mid \text{received } V_q\}$

for $k \leftarrow 1$ to n

if $\exists V_q \in lastmsgs_p$ with $V_q[k] = \perp$ **then** $V_p[k] \leftarrow \perp$

Phase 3: $decide($ first non-$\perp$ component of $V_p)$

For information only. Not Exam material

Fig. 5. Solving Consensus using any $\mathcal{D} \in \mathcal{S}$. *[Chandra & Toueg, 1996]*

Expected Learning Outcomes

At the end of this unit:

- You should understand what the output commit problem is about.
- You should appreciate the mechanisms involved in realising exactly once semantics for passive replications.
- You should be able to explain the role played by total ordering in active replication.

References

- A survey of rollback-recovery protocols in message-passing systems
  - E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson, ACM Computing Surveys, Volume 34 , Issue 3 (September 2002), Pages: 375 - 408
  - Very good and extensive survey on algorithmic issues
- Unreliable failure detectors for reliable distributed systems
  - Tushar Deepak Chandra, Sam Toueg, Journal of the ACM (JACM), Volume 43 , Issue 2 (March 1996), Pages: 225 - 267
  - Fundamental consensus algorithms under various assumptions