Distributed Commit Protocols

We know:

There is no distributed Atomic Commit Protocol (ACP)

in an **asynchronous** system with properties:

- Uniform agreement, uniform validity, stability (A1-3)
- Non-triviality, Nonblocking (A4, A5)

Relaxation of A1 – A3 does not make sense!

$\Rightarrow$ **Relax A4** (If there is no failure and **all** local decisions where "commit" then the **overall decision is "commit"** - triviality, termination) : Paxos

or A5 (no Blocking relaxed): 2PC, 1PC, 3PC

for an ACP

Fault tolerance

Without failures: protocols next to trivial

Failures:

- Operation at node X not successful (e.g. transaction abort)
- Node X is down
- Node X does not answer: communication problem or down?
- Messages received more than once

ACP requirements

Recovery

Each node must have means to recover from

- system failure: **restart procedure**
  what is my state?
  If there is an open DTA: what is its fate?
  How can I get information about the fate?
- message / communication failure: **timeout procedure**
  What should happen with a running TA?
  Can I simply abort? Is there any node knowing about the fate of the TA?

Comparison of protocols

Typical measures:

- **number of messages** to be exchanged if nodes involved in DTA
- **number of forced write operations** in order to preserve state

  e.g. : if a node X sends an "I agree"-message to another node, X should now after a failure, that he agreed
  Does not mean, that the message was actually sent!

- "Blocking threat" would be nice to have, e.g. blocking probability.
  Not so easy: depends on failure probability

Roadmap

1. Blocking protocols

- Two phase commit (2PC) – the standard
- One phase (1PC)
- Three phase commit (3PC)
  lower blocking threat, more messages.

2. **Consensus based**

- more general problem: a community of computing nodes agrees on some value.
- PAXOS

May be used for commit processing but also for keeping replica consistent.

7.4 Two phase commit

- **2PC is a distributed handshake protocol.**

- **Goal: Atomic Commit of n** subtransactions **cooperatively** executed on **n nodes (resource managers,** participants**).**

- The standard protocol implemented in OS (Windows) as well as in DBS and transactional middleware (WebSphere, WebLogic...).

- Standardized in the **X/Open transaction model.**

2PC: Asumptions

Freie Universität Berlin

- **Subtransactions Ti, i=1..n will commit, if no error** occurred.
- Each **resource manager** (RM) called **participant** may locally abort its subtransaction, e.g. deadlock.
- **coordinator (exactly one)** taking responsibility for **unanimous outcome** (Commit processing)
- Each resource manager has a **transactional recovery system**
- A node may have the **role of coordinator and resource manager at the same time** .

Atomic Commit processing

Why is it a hard problem?

- What if resource manager **RM_i fails after a transaction commits** at RM_k?
- What if other resource managers are down when RM_i recovers?
- What if a transaction assumes that a resource manager failed and therefore aborted, when it actually is still running?

Distributed TA model

- **Application**: Start distributed transaction at participants
- Coordinator knows the set of participants
- **Work phase**: send operations to the right participant
- Errors @ a participant $\Rightarrow$ abort
- All operations successful and AP says: commit

$\Rightarrow$ **Coordinator starts final ACP**

2PC: Coordinator

1. Coordinator C requests vote (y/n) from each participant

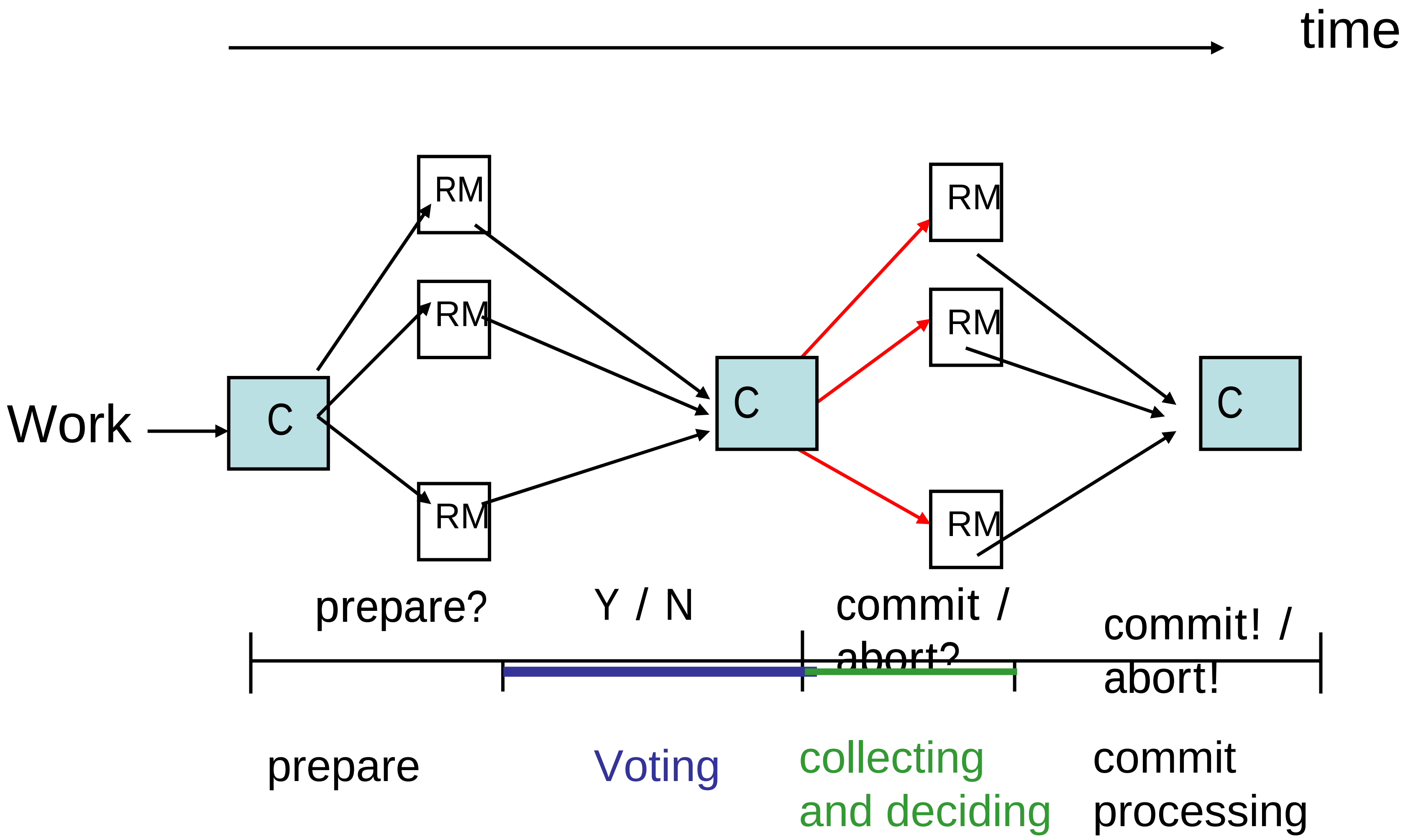
2. C collects votes and decides: **abort** if **at least one vote is abort**, else commit.

3. C sends decision to all participants

4. Participants send ack

That's it. So what?   Uncertainty phase?

Phases of 2PC



P=Participant

2PC Resource Mgr

Freie Universität Berlin

1. After **work phase**, RM waits for message from C

2. Message **prepare** from C arrives

3. RM prepares subtransaction s in a way which allows to commit or to abort it

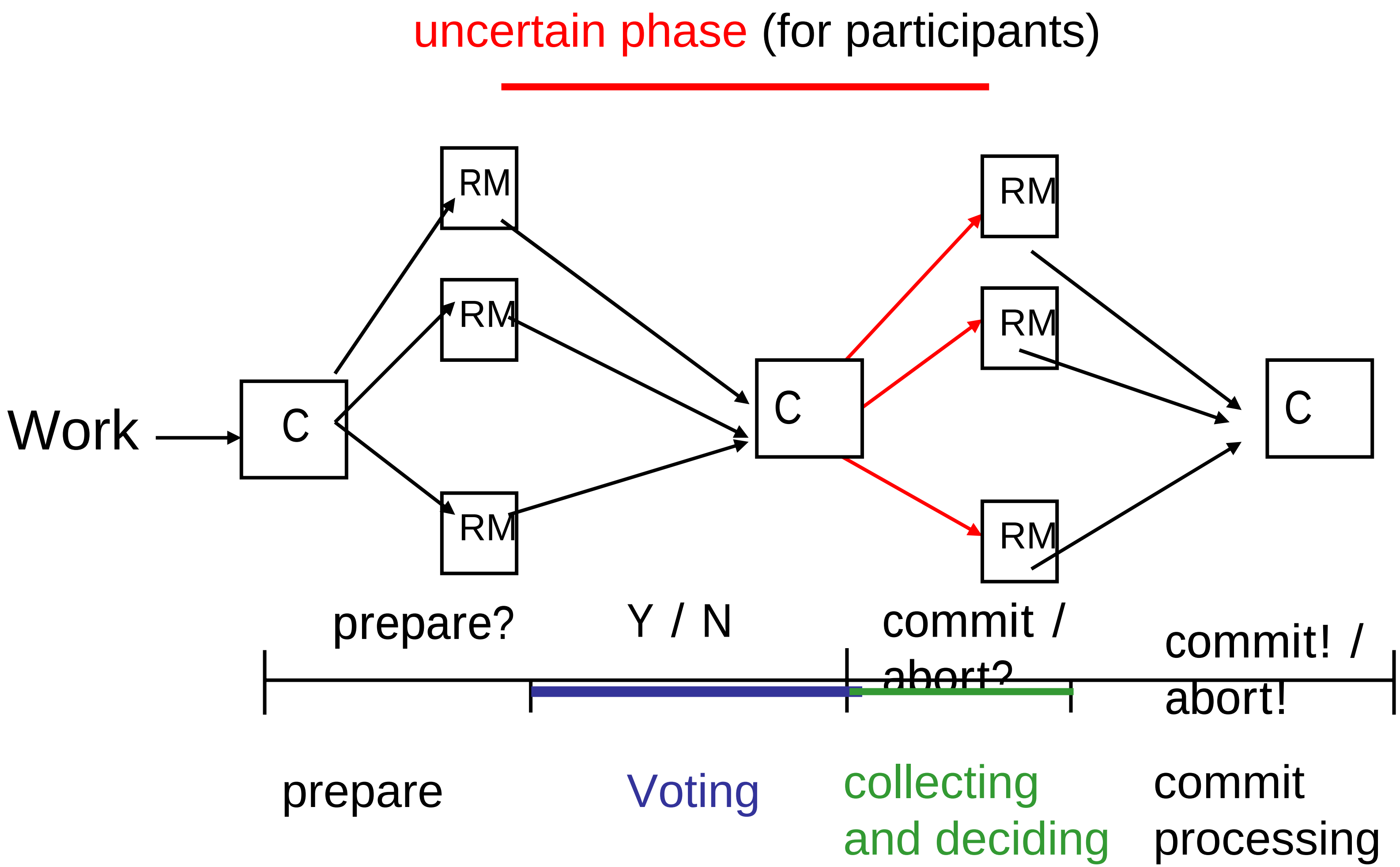
4. Send **"ready"** ("prepared") **msg** to coordinator

5. **Wait for coordinator's message**: "commit" or "abort"

6. Do what C has decided: "commit" or "abort"

When does RM give up autonomy??

Phases of 2PC



P=Participant

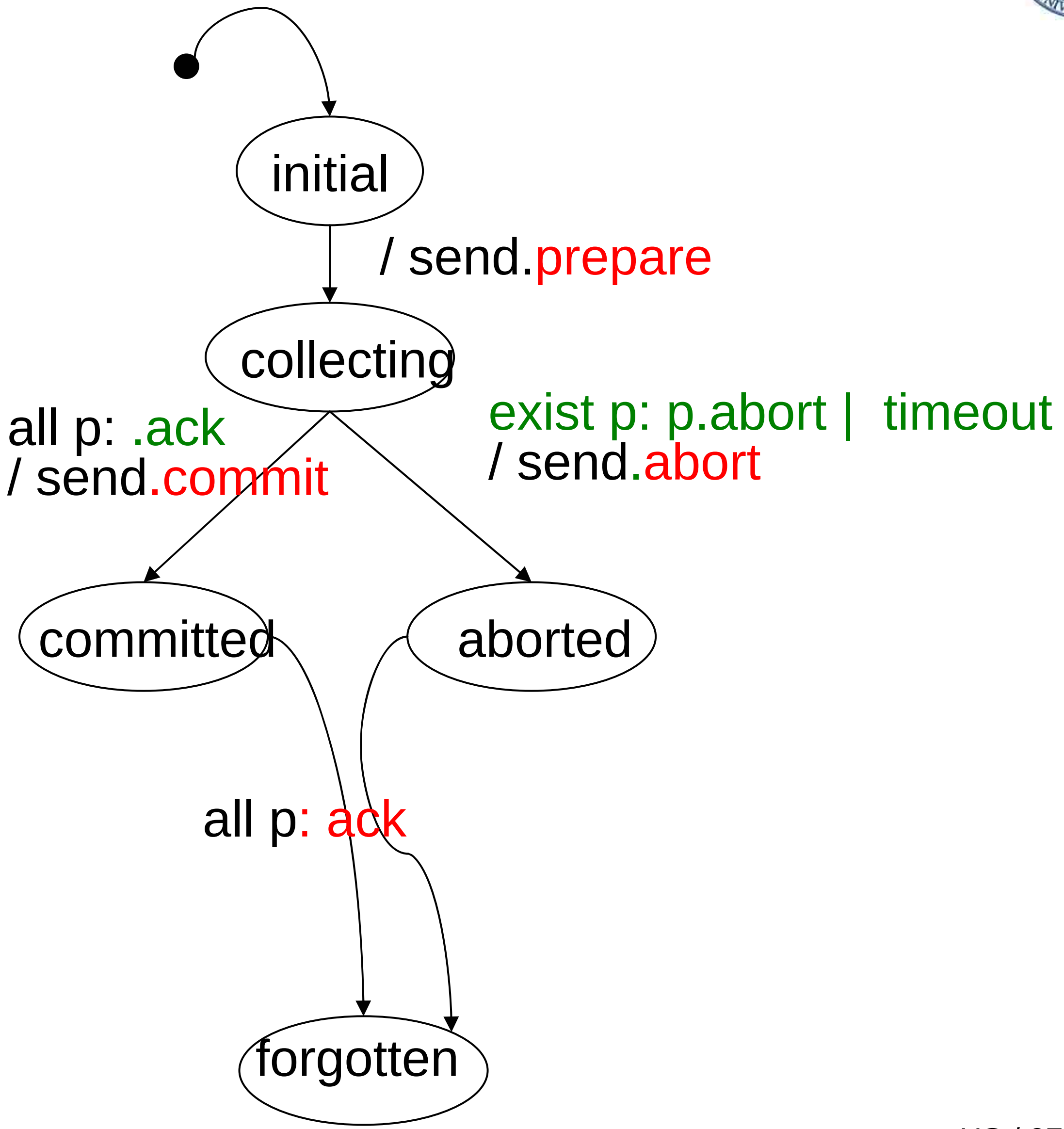
2PC as a state cart

State chart:

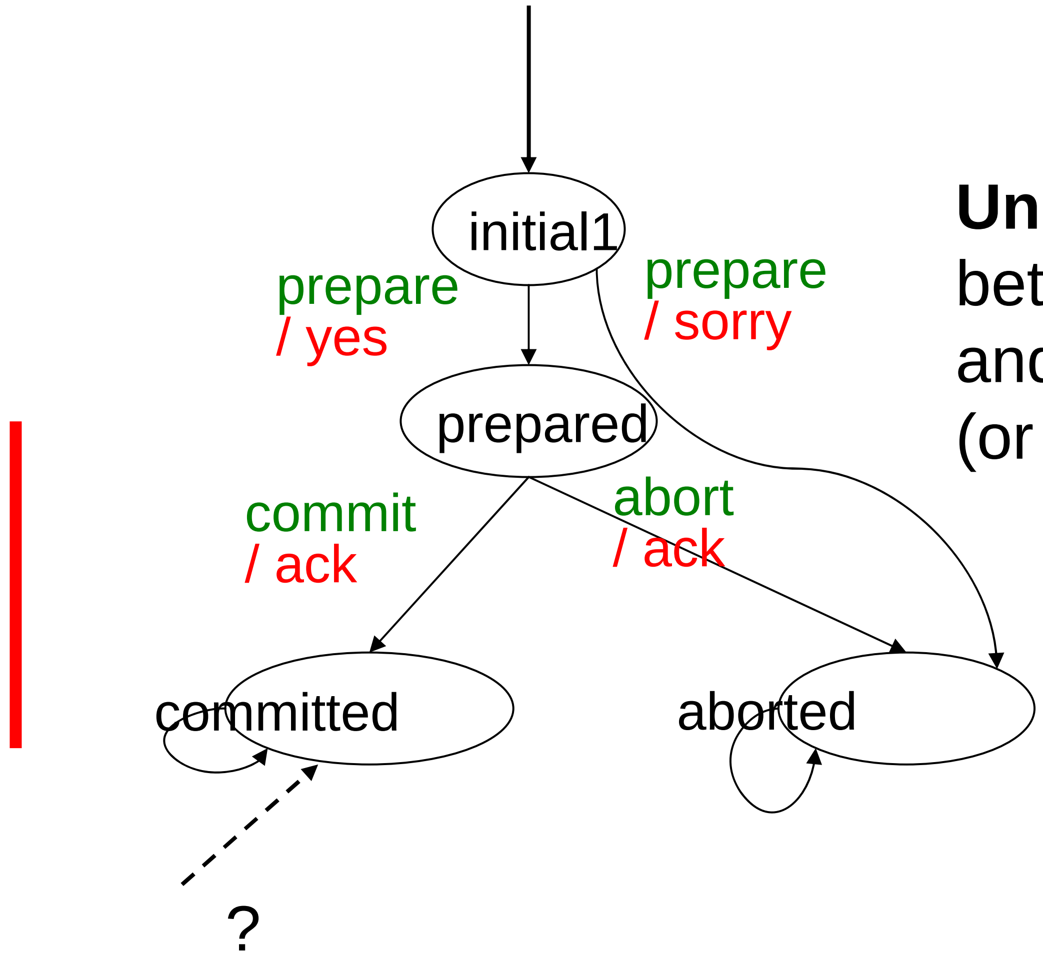
Mealy automaton: on input **in**
state transition $Z \rightarrow Z'$ and output **out**.

in may be a predicate on one or more inputs

2PC - Coordinator



2PL: Participant protocol



Uncertainty phase:
between prepared (yes) and receiving commit (or abort!)

Two phase commit steps

How to insure reliability?

Coordinator: first phase (voting):

coordinator starts protocol: sends *prepare* messages to participants and waits for *yes* or *no* votes

Coordinator second phase (decision)

- coordinator decides: sends *commit* or *abort* messages to participants and waits for *ack*s

Participant:

– promises to obey the coordinator.

What has to be logged in order to terminate successfully (i.e. with a unanimous decision in **all** cases?

2PC and fault tolerance

Is the protocol fail-safe?

- Message loss or process failure $\Rightarrow$ protocol failure

- Each process restarts after failure **at last remembered state**

$\Rightarrow$ **Forced logs** for different states in order to be able to recover

Protocol failures

Not so easy:

e.g. coordinator:

- **failed after writing prepared log entry**

$\Rightarrow$ wait for "yes / ack" of all participants

But some messages could have get lost (or where never sent!) $\Rightarrow$ wait forever?

Not decidable if message sent or not in case of failure ...

2PC and fault tolerance

Writing log-record must precede sending "commit", "ack" etc

But: No atomic disk write **and** message send

Consequence: **Reading log-record with "commit" (e.g.) does not ensure that the message has been sent**

⇨ Resend msg ⇨ duplicated messages

Datagrams used for 2 PC-TA coordination

Could reliable protocol (TCP/IP) be utilized?

Would message queues help? (delivery guarantee!)

Protocol failures

Needed:

Forced logs: what has definitely happened before crash?

Restart protocol : how to proceed a failed protocol

Termination protocol : how to react upon a time-out when waiting for some messages

Example 2PC with Log records

Freie Universität Berlin

Coordinator C Participants: P1 *P2*

force-write **Coord** *begin log entry*

send “prepare” →

send “prepare” →

Participant

force-write *prepared log entry* force-write *prepared log entry*

← send “yes”

← send “yes”

force-write **Coord** *commit log entry*

send “commit” →

send “commit” →

Participant

force-write *commit log entry* force-write *commit log entry*

← send “ack”

← send “ack”

write **Coord** *end log entry*

Logging

Freie Universität Berlin

Init and voting

Logging: Coordinator (1)

- writes *begin* log entry

Logging: Participants (1)

- write *prepared* log entries in voting phase and become *in-doubt (uncertain)*

$\rightarrow$ potential blocking danger, breach of local autonomy

Decision phase

Logging: Coordinator(2)

- coordinator writes *commit* or *rollback* log entry and can now send decision to participants freeing them from blocking

Two phase commit steps (cont)

Logging: Participants(2)

- participants write commit or rollback log entry in decision phase

Termination

Logging: Coordinator(3)

- Coordinator writes *end (done, forgotten)* log entry to facilitate garbage collection

2PC performance

Failure free case

n Participants, 1 coordinator

if **acks** are counted

→ ***4n*** **messages,**
→ ***2n+2*** **forced log writes,**

1 unforced log write

2PC and fault tolerance

Failure model:

- process failures: **transient server crashes**
- network failures: **message losses, message duplications**
- assumption that there are **no malicious commission failures** $\rightarrow$ Byzantine agreement
- no assumptions about network failure handling

  i.e. no distinction if participant server crashed or network failure

$\Rightarrow$ **Enhanced state-chart**

- **F transition:** restart after protocol failure and reading state (log)
- **T transition:** timeout received

TA 2PCommit: Correctness

Point of reference for participants and coordinator is:

log entry

forced before sending messages ⇨ state or states of servers before crash, e.g." begin" entry of coordinator c means: state is "initial" or "collecting"

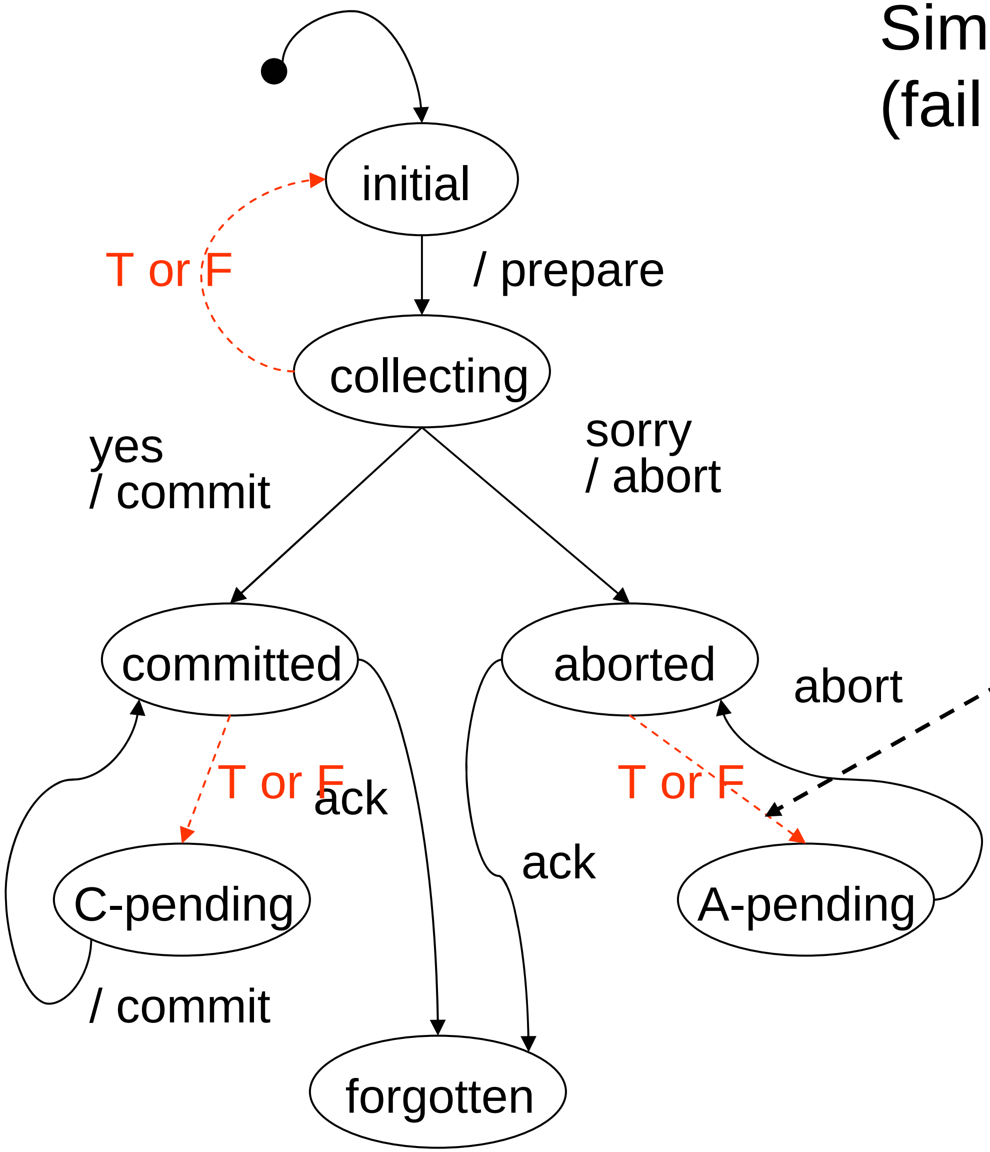
Do not know anything about actions taken after last log entry written - have all messages been sent? did any participant send a message? … -

Correctness reasoning

No failure: Commit unanimous / abort? Obvious

Check all failure situations (crash, timeout) and **show** that all participating systems will **eventually decide unanimously**

2PC and fault tolerance



Simple coordinator protocol (fail safe)

Resend messages when timeout T or restart F in "collecting" of "committed /aborted" state.

Selective resend if already received msgs from participants

Not shown: max number of timeouts

2PC failure handling

<u>**Coordinator**</u> fails... and recovers (or timed-out)

(1) **not yet in state committed | aborted**

⇨ send "prepare" to all partners,

already sent? Doesn't matter

wait for replies,

if timeout: abort

else make decision as usual

(2) **state is committed | aborted**

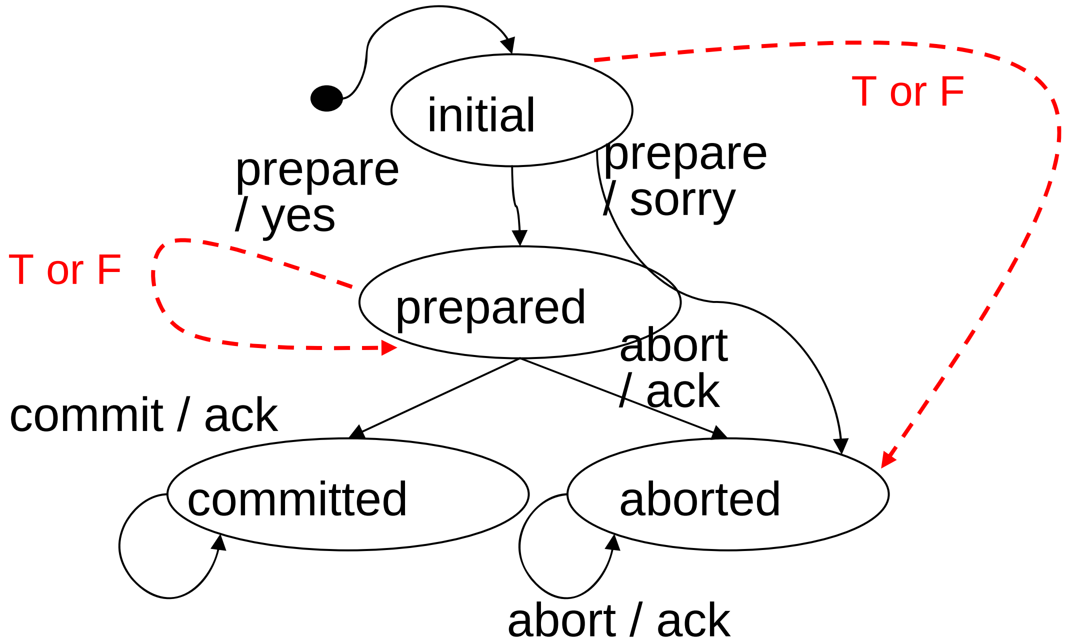
⇨ send again either "commit" or "abort"

depending on log entry

if timeout: reminder messages

2PC and fault tolerance

Participant



Autonomy: allows to cancel subtransaction in case of failure or timeout before in "prepared" state

Not shown:
wait for more than one timeout in initial state

Prepared state: **wait - and block resources :(**

Manual intervention could be necessary

2PC failure handling

<u>Participants</u> fails… and recovers (or timed-out)

(1) **Not yet prepared:**

wait for message for an open sub-TA ,

e.g. "application action" or "prepare" msg

if timeout: abort sub-TA, vote "no" if "prepare" msg arrives later

(2) **prepared (waiting for vote of coordinator)**

timeout: blocked!

// Cannot abort, since others may have
// committed already after "commit"-vote

recovery from failure: ask coordinator (may time out!) …. wait patiently….

(3) **exists log entry e "commit" | "abort"** :

action according to e; send ack to coordinator

Blocking...

.. is bad!

e.g. resources of an autonomous system which runs a subtransaction may be **blocked forever**....

Workarounds

- manual intervention
- guess the outcome
- find a participant who knows more...

2PC and heuristic commit

Participant recovers, but the termination protocol leaves T blocked.

Operator can guess whether to commit or abort

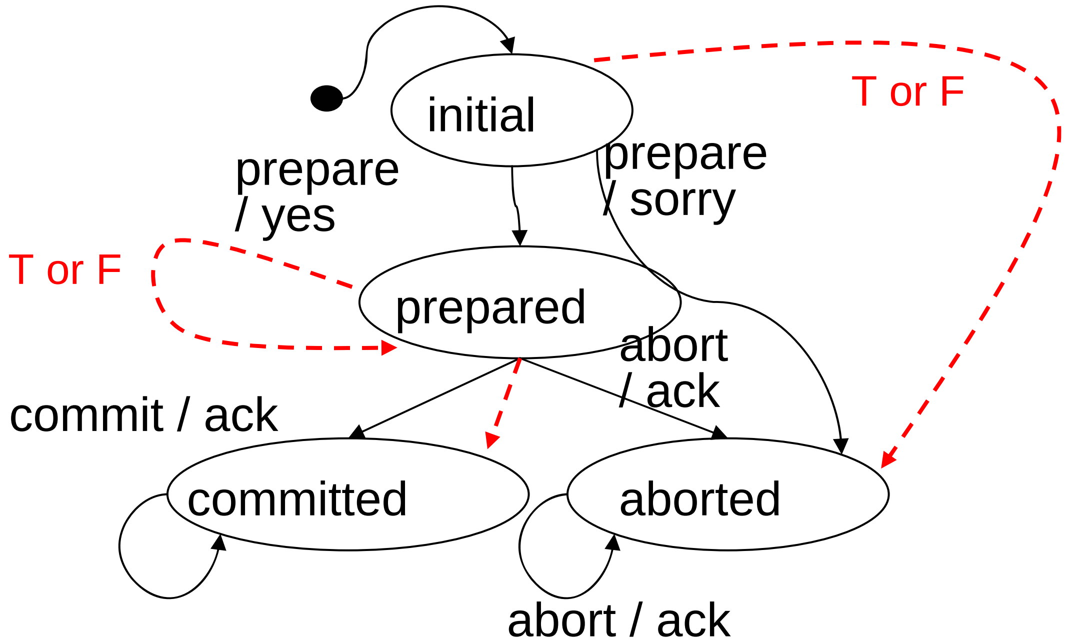
- Must detect **wrong guesses** when coordinator recovers
- Must run **compensations** for wrong guesses

Heuristic commit

- If T is blocked, the local resource manager (actually, transaction manager) guesses
- **At coordinator recovery**, the resource managers jointly detect wrong guesses
- Use **compensation transaction** of healing

2PC and fault tolerance

Participant



Heuristik commit may need **compensation** after coordinator is back

Autonomy: allows to cancel subtransaction in case of failure or timeout before in "prepared" state

Not shown:
wait for more than one timeout in initial state

Prepared state: wait - and block resources :(

Manual intervention could be necessary

Blocking

Assumption: **participants know each other**

Let P be blocked, i.e. sent "yes"- vote and does not receive answer.

if P finds another participant Q, which

has **received the final decision** from coordinator

$\Rightarrow$ P knows TAs (global) fate $\Rightarrow$ unblock

if P has **not yet voted** $\Rightarrow$ decide abort together with Q

Blocking

Can blocking be avoided?

There is no distributed commit protocol which avoids blocking in case of more than a single process failure.

- Blocking can be a serious problem which can not be solved automatically in all situations

- Cannot be avoided in the general case, e.g. network partitioning

- Bad: 2PC is fault tolerant but blocks in case of failures…

Can blocking be avoided for single process failures (no communication fault) ?

7.5 Optimizing 2PC

- Can (some) **forced logs** be relinquished?

- **Saving of messages** due to known characteristic of application?

- Read Only Transactions?

- Specialized topologies? $c \rightarrow P \rightarrow P \rightarrow .... \rightarrow P$

Example 2PC

Coordinator C Participants: S1 S2

force-write
begin log entry

send “prepare” →

send “prepare” →

force-write
prepared log entry

force-write
prepared log entry

← send “yes”

← send “no”

force-write
commit log entry

send “abort” →

send “abort” →

force-write
commit log entry

force-write
commit log entry

← send “ack”

← send “ack”

write
end log entry

all forced writes needed?

2PC with Presumption

Why **forced** "begin TA-commit" **log** (coordinator) entry?

Not a correctness issue:

if *no log entry after voting, just abort everyone*

No forced log writes of participants:

- they can inquire coordinator who has stable log
- does it work if coordinator log has been garbage collected? ("transaction forgotten"?)

No, except when a ***particular outcome is assumed*** **when no log state information is found** at the **participants / coordinators** site

2PC: Presumed abort

Recovering participants make the following **assumption**:

If no information found in coordinator's log entries about the outcome of TA, **assume** it has been **abort**ed

⇒ ACKs of participants at the end of abort not needed

saves forced log writes and acks

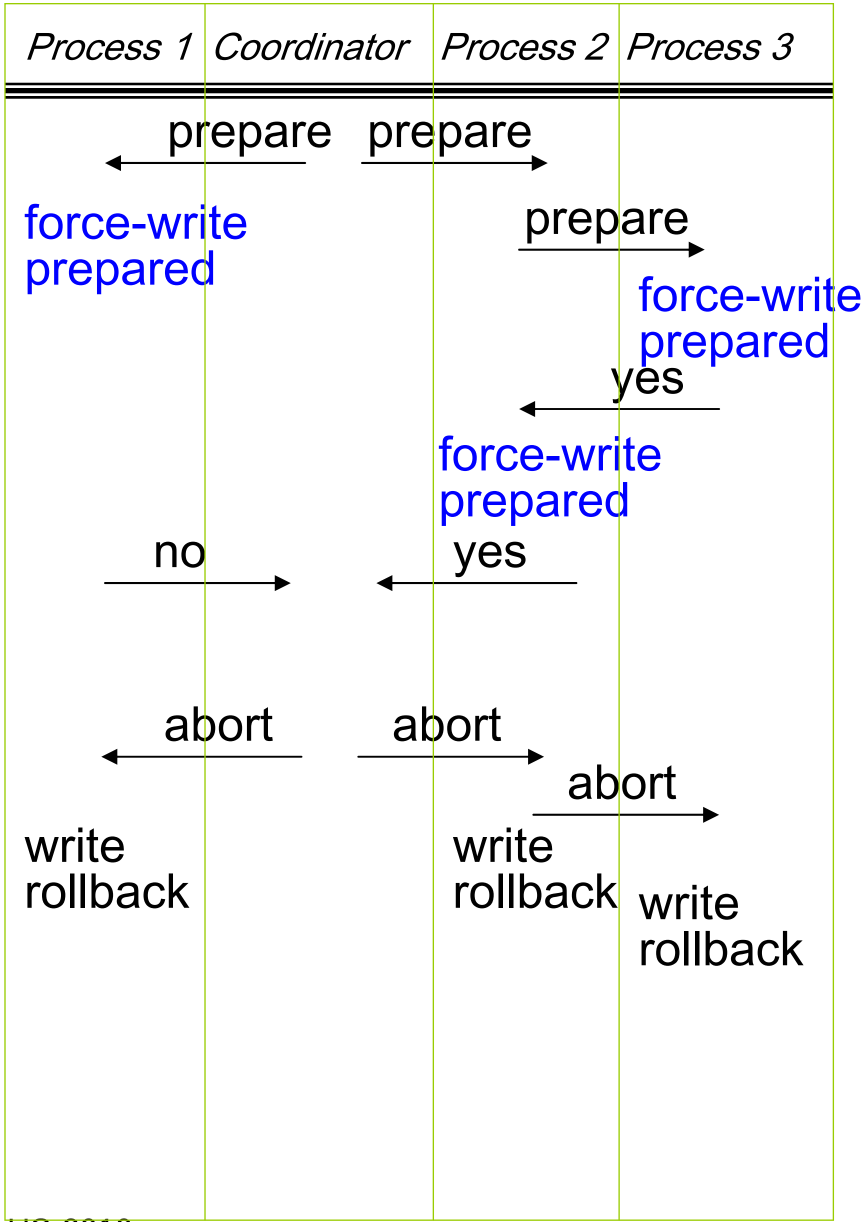
Question: could "abort" ("rollback") log entries be omitted totally??

Important: **winner log-entries ("commit") must still be forced!**

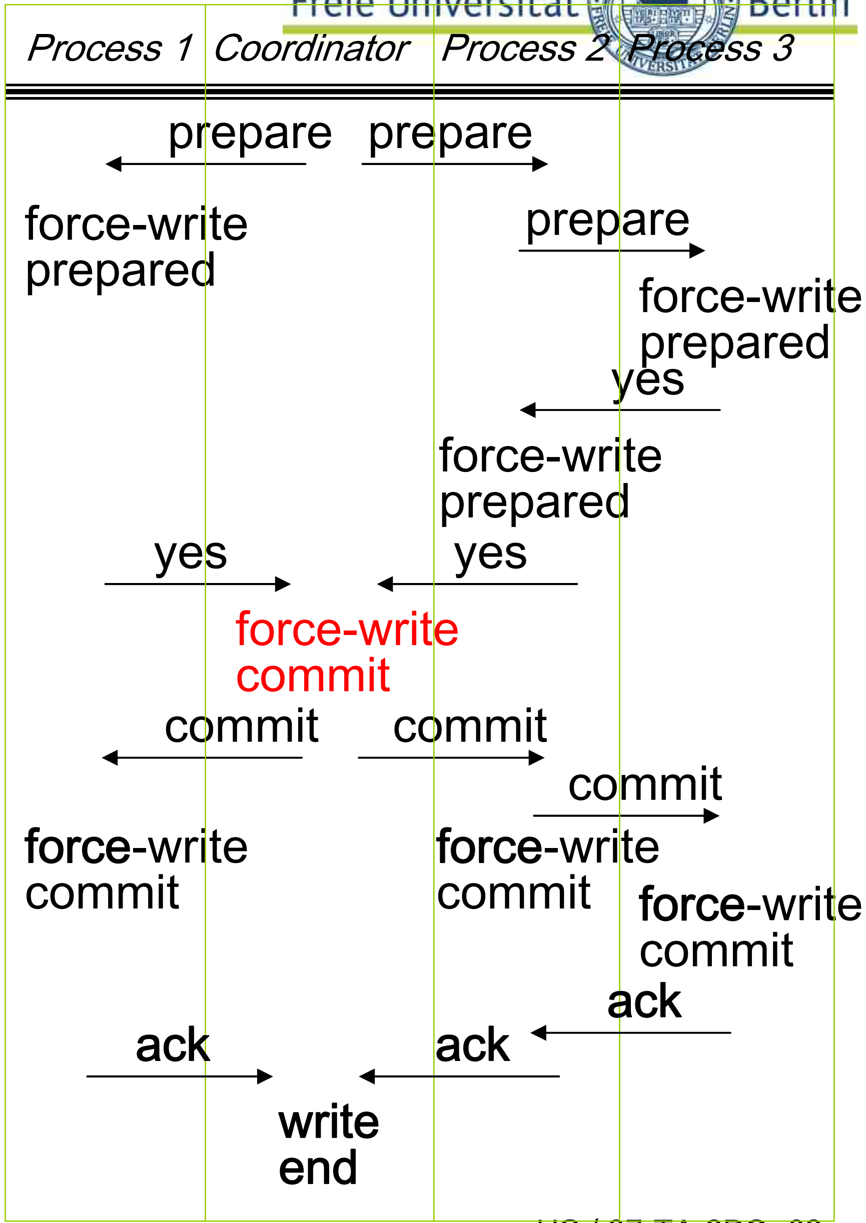
Presumed abort is employed in XA-Standard

Saved : n messages, 2n+1 forced writes

2PC assumed abort: illustration



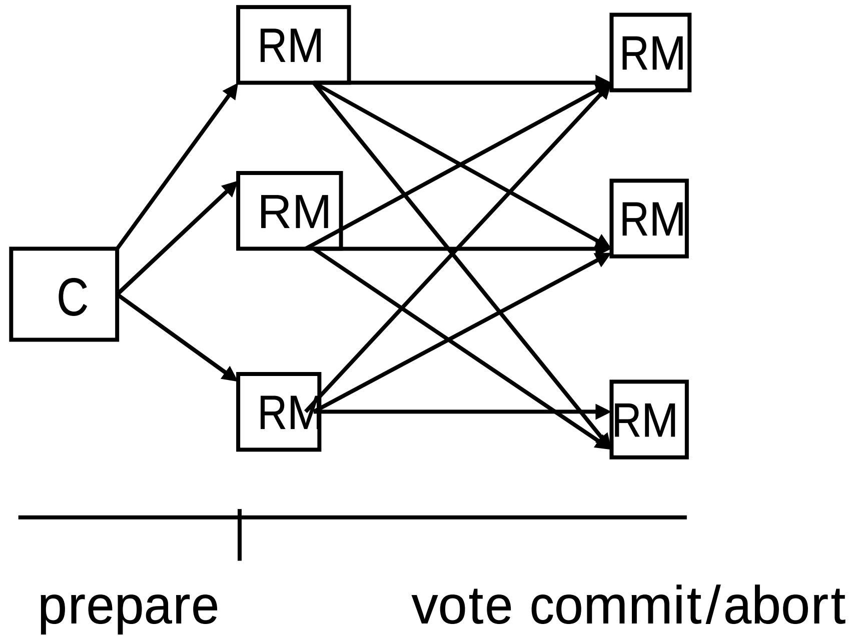
Case 1: transaction abort



Case 2: transaction commit

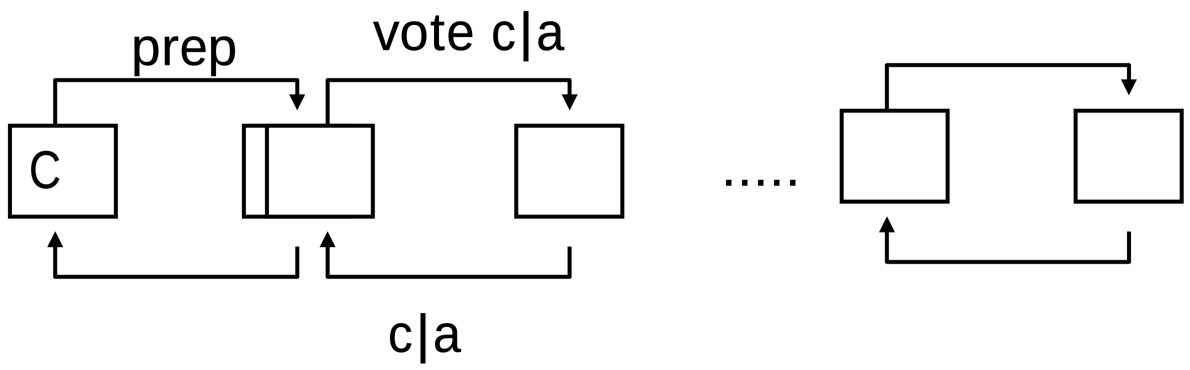
Freie Universität Berlin

Distributed commit



2 rounds
n^2+n messages

Linear commit



2 n +2 messages, 2n rounds

Hierarchical 2PC (Tree 2PC)

Hierarchical process structures

- During transaction execution the transaction forms a ***process tree*** rooted at transaction initiator with bilateral communication links according to request-reply interactions
- frequent situation in practice

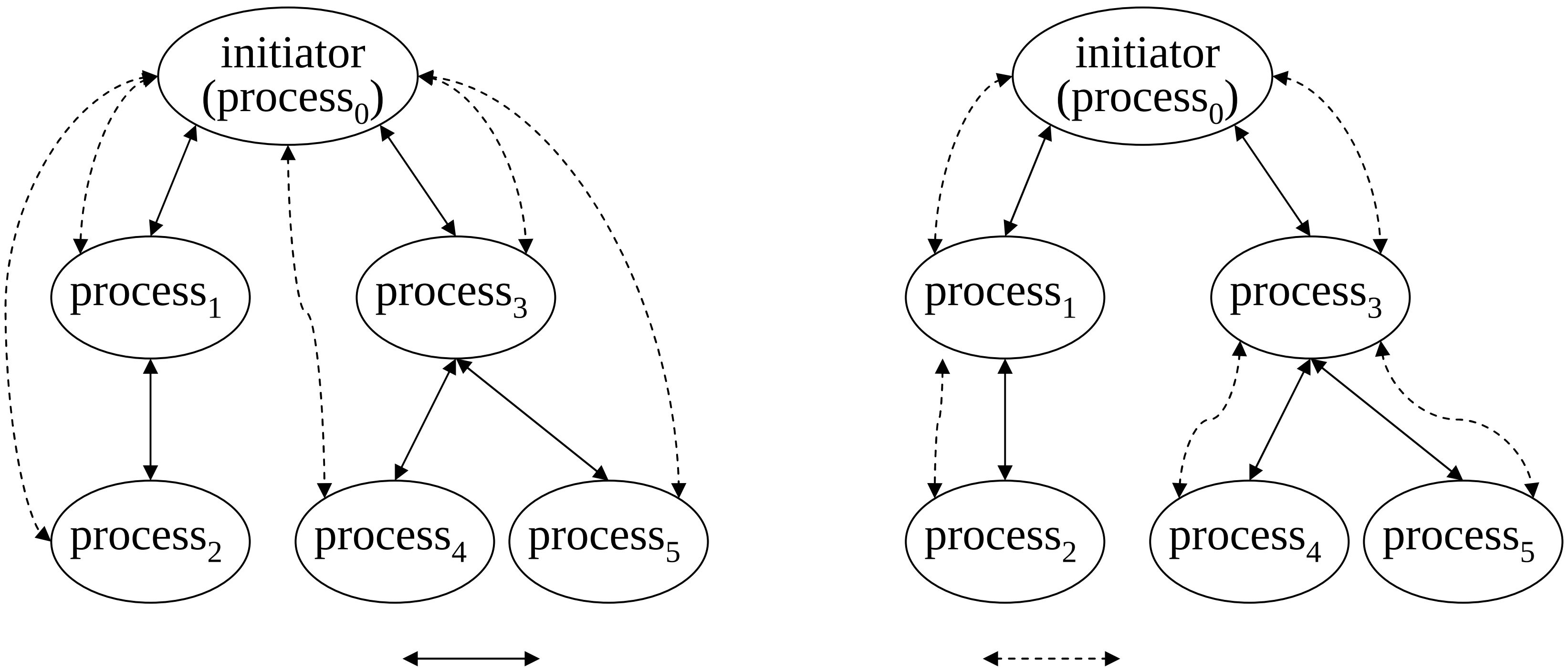
  e.g. submit an SQL request which triggers sending of a mail…

Commit processing

- as is: hierarchical form of 2PC
- flatten process-tree and use standard 2PC

Hierarchical 2PC (Tree 2PC)

Flattened 2PC: Hierarchical 2PC:



communication during transaction execution

communication during commit protocol

Need addresses of participants

Variants of 2PC: Optimization

Goals

reduce the **number of messages** and **forced log writes** for higher throughput

shorten the critical path until local locks can be released for faster response time

Possible optimizations:

fewer messages and forced log writes by **presumption** in the case of missing information

eliminating **read-only subtrees** as early as possible

(dynamic) **coordinator transfer**

Read only transaction

Read only Participants:

no action needed after "prepare"-message received
except "prepared-read-only" msg to coordinator

Semantics: release read locks, no further action
Coordinator eliminates participant

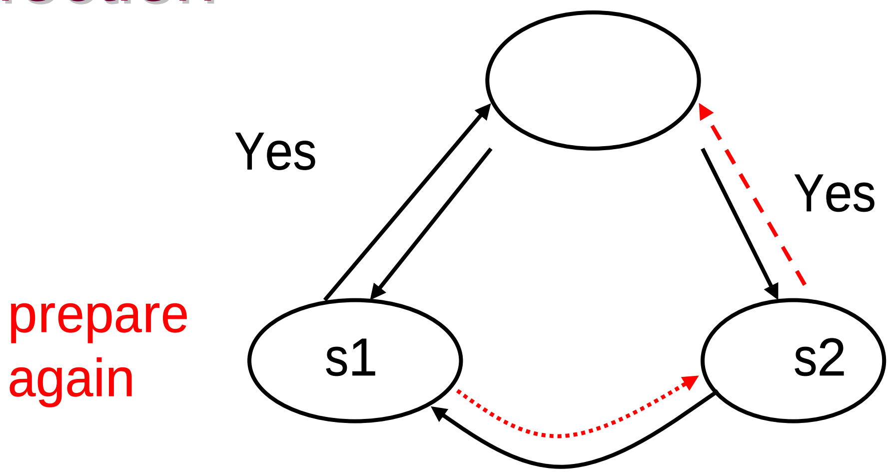
Caution: **reinfection**

Hierarchical 2PC may cause trouble:
subordinate transaction (e.g. "execute trigger") may still be active and acquire a lock: 2PL broken!

Will not happen with commit-deferred TA-protocol

("don't send commit before all actions complete")

Reinfection



Do something at site 1 during preparing and acquire lock!
Could result in a deadlock, even though site 1 has voted "Yes"

Example: Trigger processing at the end of a transaction

Solution:
- s1: do work assigned by s2
- s1: prepare again (!)
- s1: ack action to s2
- s2: vote "Yes" (or "No")

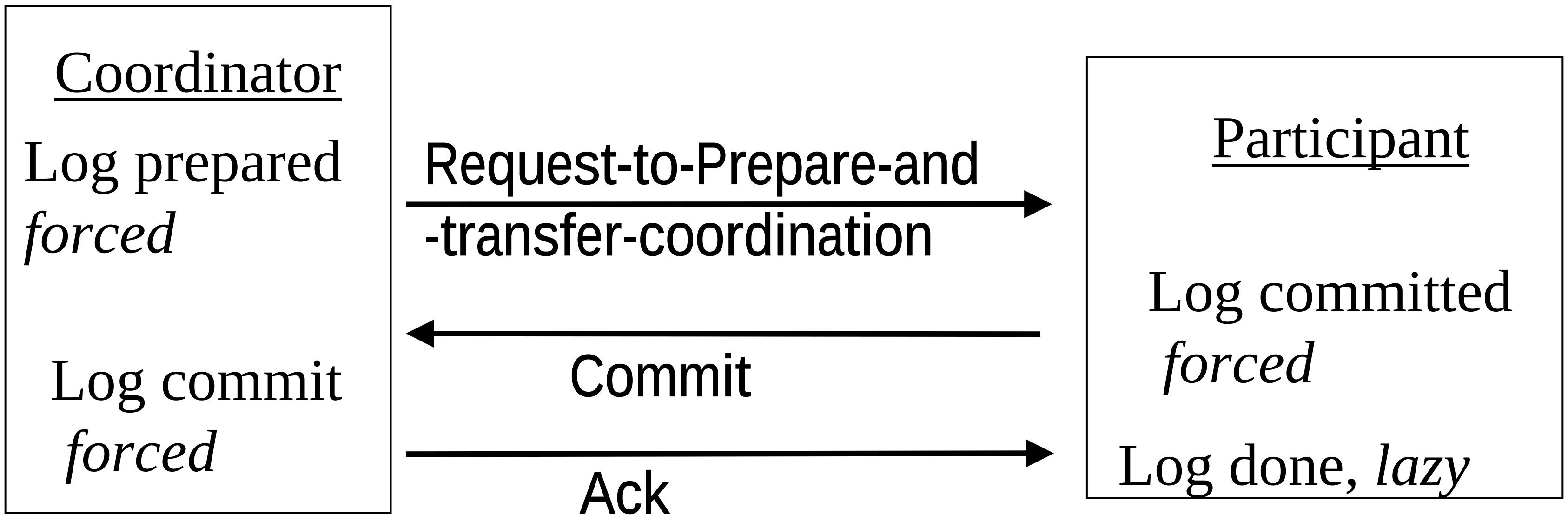
Transfer of Coordination

Assumptions: only two participants

(1) Coordinator asks participant to prepare and become the coordinator.

(2) Participant (now coordinator) prepares, commits, and tells the former coordinator to commit.

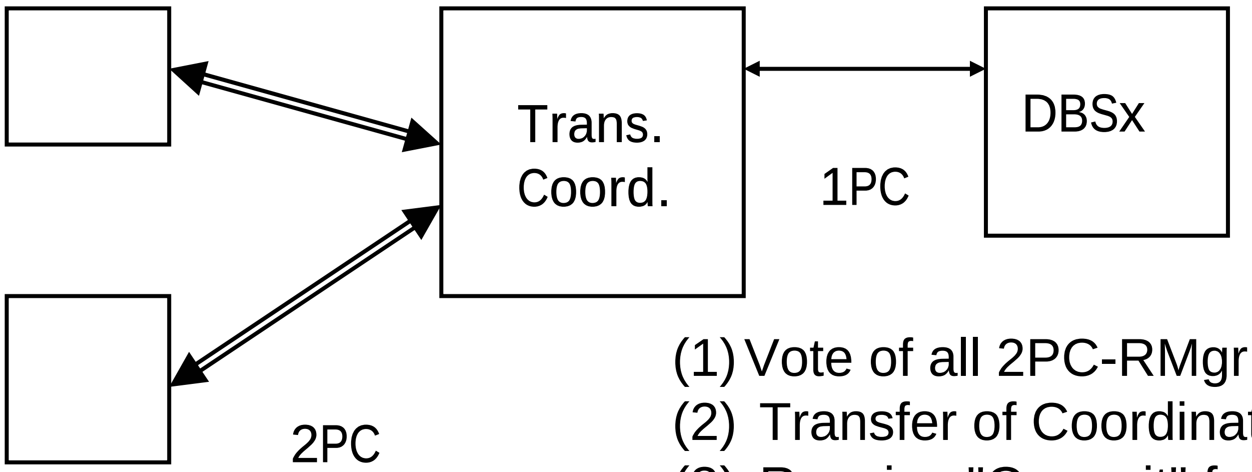
(3) Coordinator commits and replies Done.



Transfer of Coordination

Freie Universität Berlin

Transfer can be used in a situation in which one resource manager does not implement 2PC, e.g. MySQL (... not true any more ;)



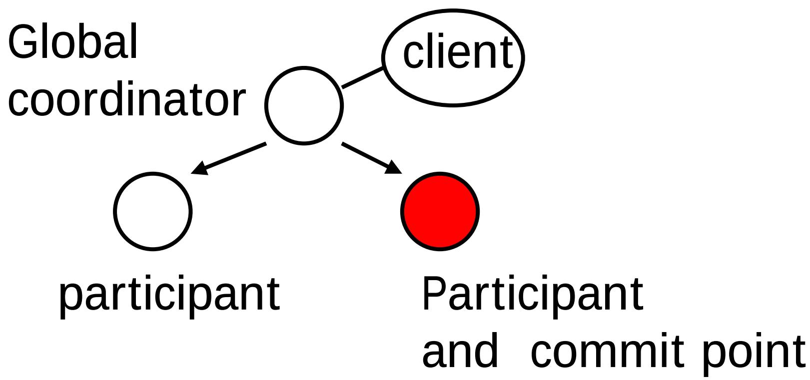
(1) Vote of all 2PC-RMgr
(2) Transfer of Coordination
(3) Receive "Commit" from DBSx
(4) Send "Commit" to 2PC systems

Choosing a commit point

Commit coordinators and commit points

Most critical aspect of 2PC: blocking of resources in case of failures

Commit point: participant which is chosen by the commit coordinator to decide on the outcome



Advantage:
no "in doubt" state at commit point site.
⇨ Chose site with most critical data as Commit point

8.2 Distributed Transactions in practice

X/Open Distributed Transaction Processing

Standardization of distributed transactional processing interfaces (since 1991)

Based on 2PC

most important: XA

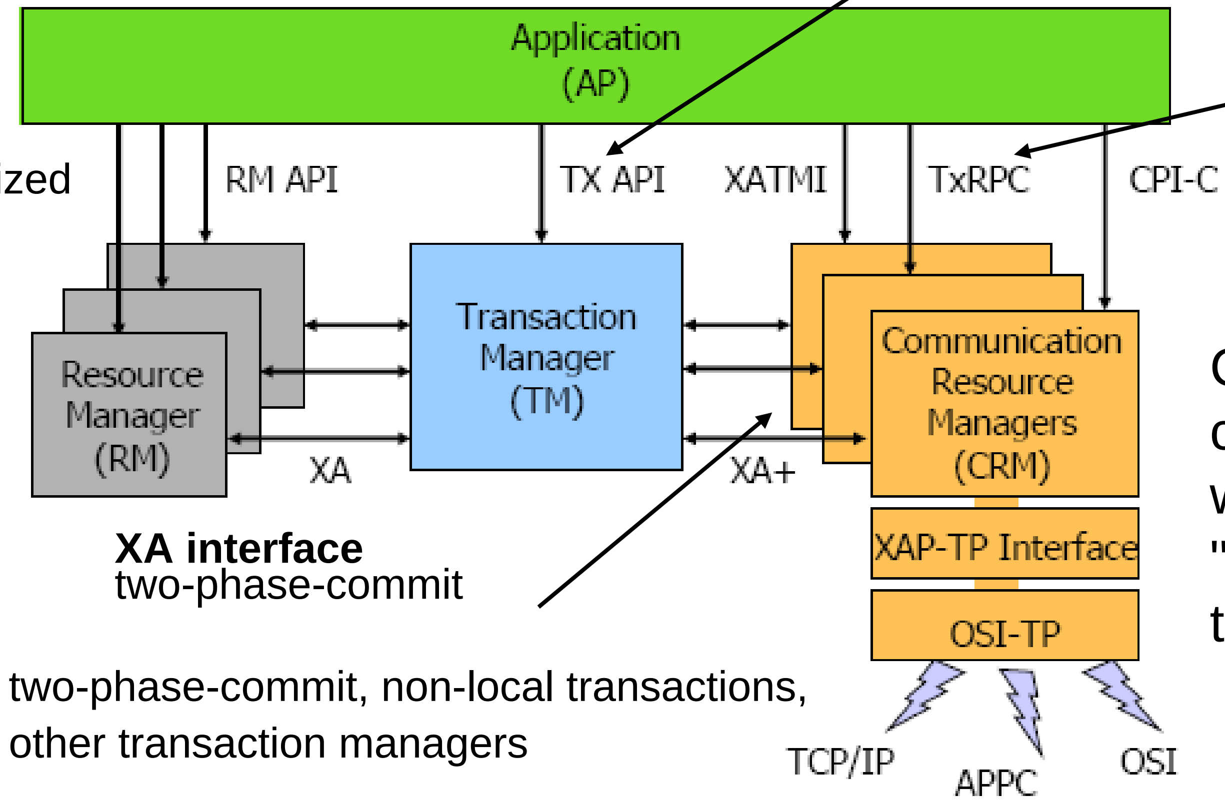
Components in a DTP environment

- Application program (AP)
- Transaction Manager (TM) responsible for atomic commit of global TA
- Resource Manager (RM), e.g. DBS
- Communications Resource manager (CRM)

X/Open DTP model

TX- interface (StartTrans, commit, Rollback..)

Microsoft: OLE transactional interface



Transactional RPC

Not standardized by X/Open, SQL etc.

XA interface
two-phase-commit

two-phase-commit, non-local transactions, other transaction managers

CRM: Allows to communicate with other "instances of the model"

Graphic from CS341/unibas

Practice: Process structuring

To support multiple RMs on multiple nodes, and *minimize communication*, use ***one transaction manager (TM) per node***

TM performs **coordinator and participant roles** for all transactions at its node.

TM communicates with local RMs and remote TMs.

TM **may be in the OS** like Distributed Transaction Coordinator (MSDTC) embedded in Windows XP,

the TP monitor (IBM CICS),

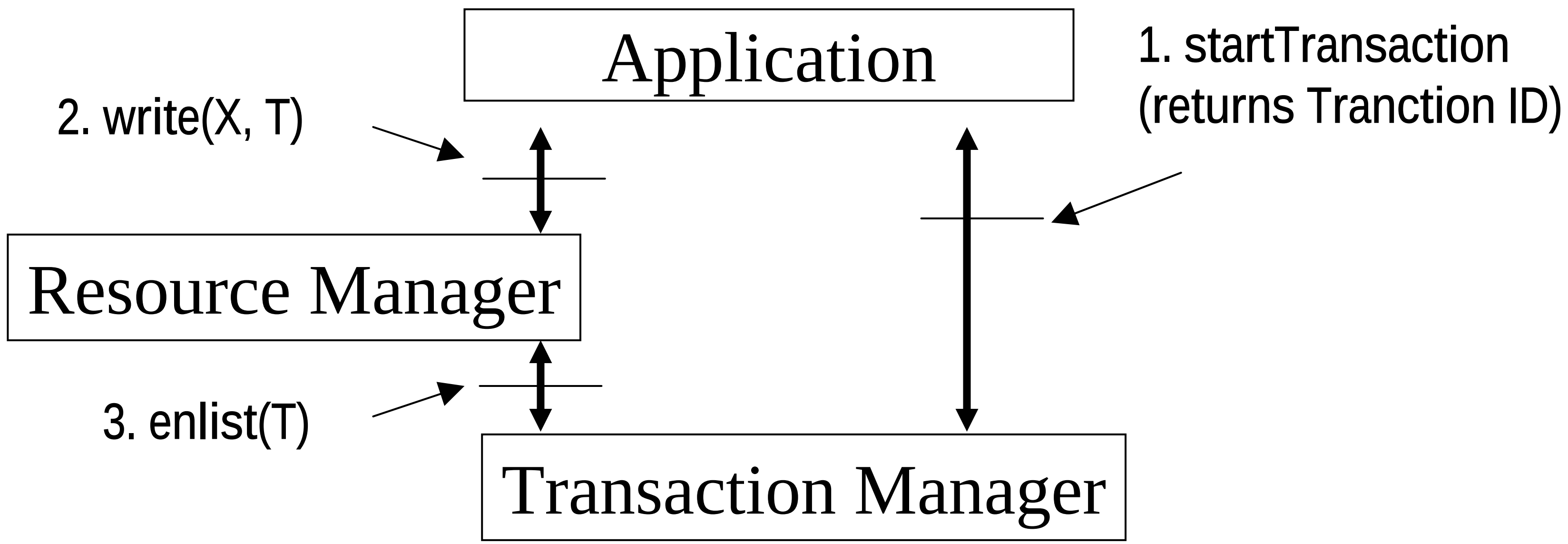
or a separate product (Encina, Tuxedo,…)

Following Newcomer, Bernstein: TP for Systems professionals, Morgan Kaufmann

Building a process tree: Enlisting a TA

When an application in a transaction T first calls an RM, the RM must tell the TM it is part of T.

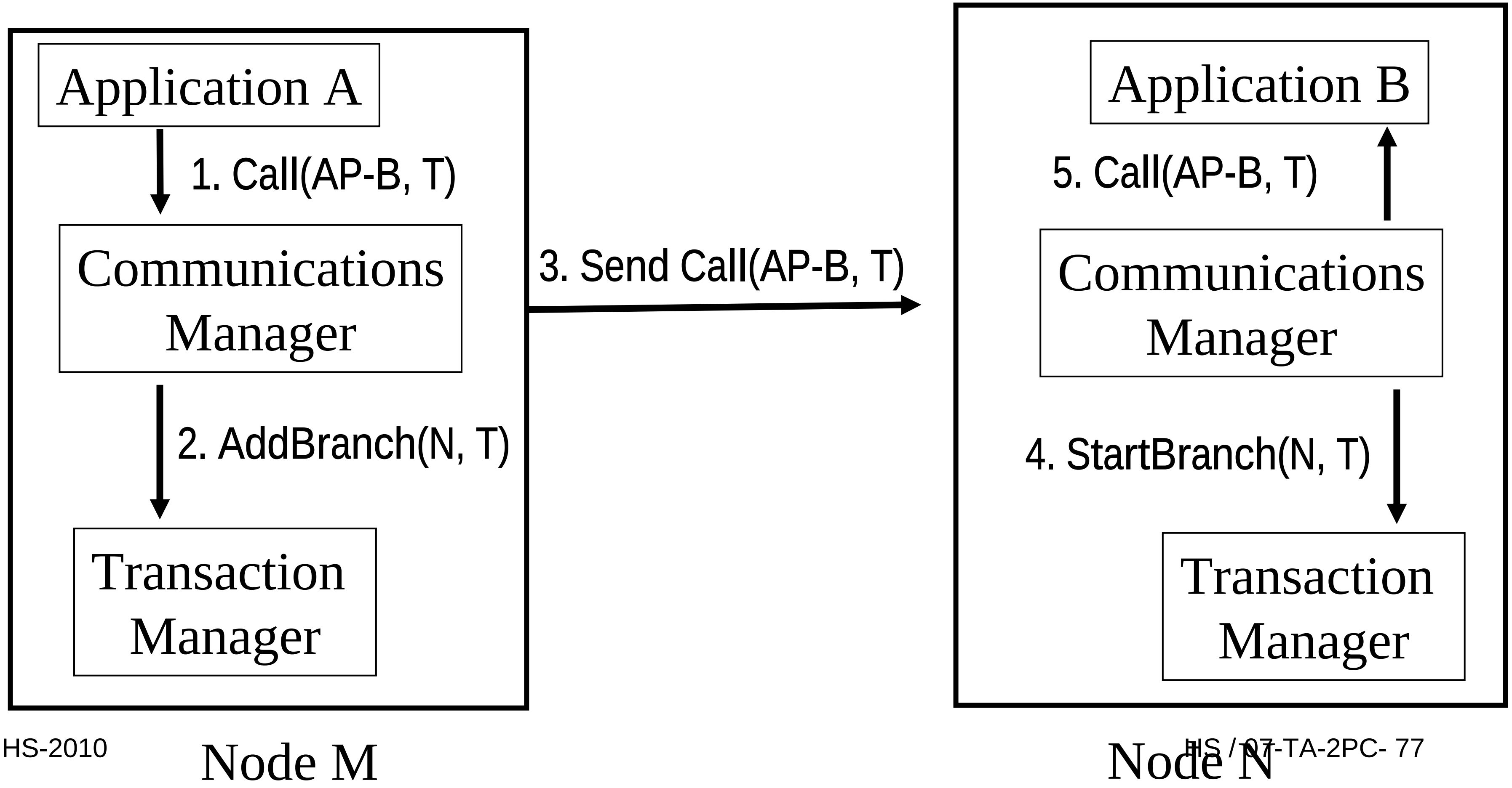
Called *enlisting* or *joining* the transaction



enlist() issued by application server, if present

Building a process tree: Enlisting a TA

When application A in transaction T first calls an application B at another node, B must tell its local TM that the transaction has arrived.

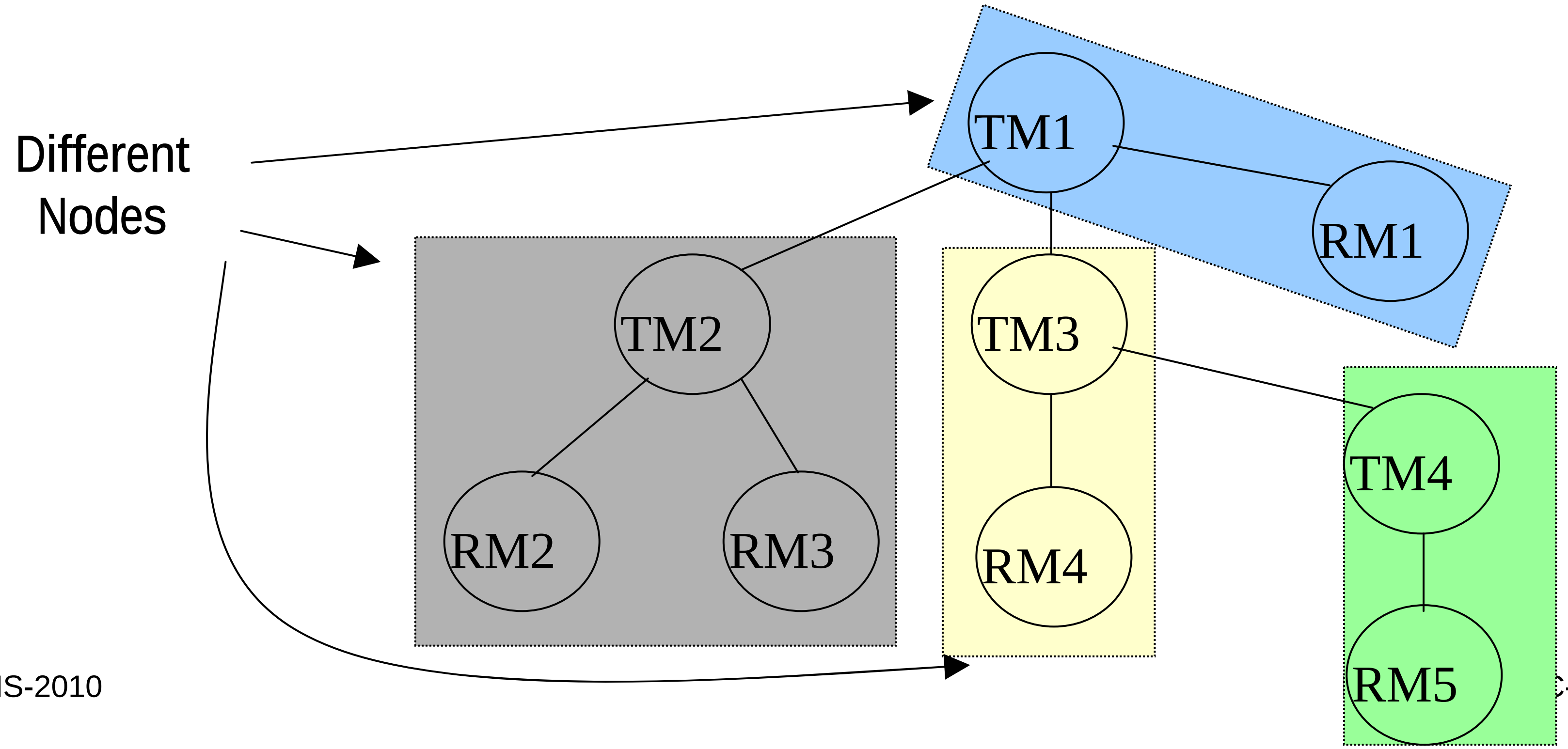


Tree of Processes

Application calls to RMs and other applications induces a *tree of processes*

Each internal node is *coordinator* for its descendants, *and participant* to its parents.

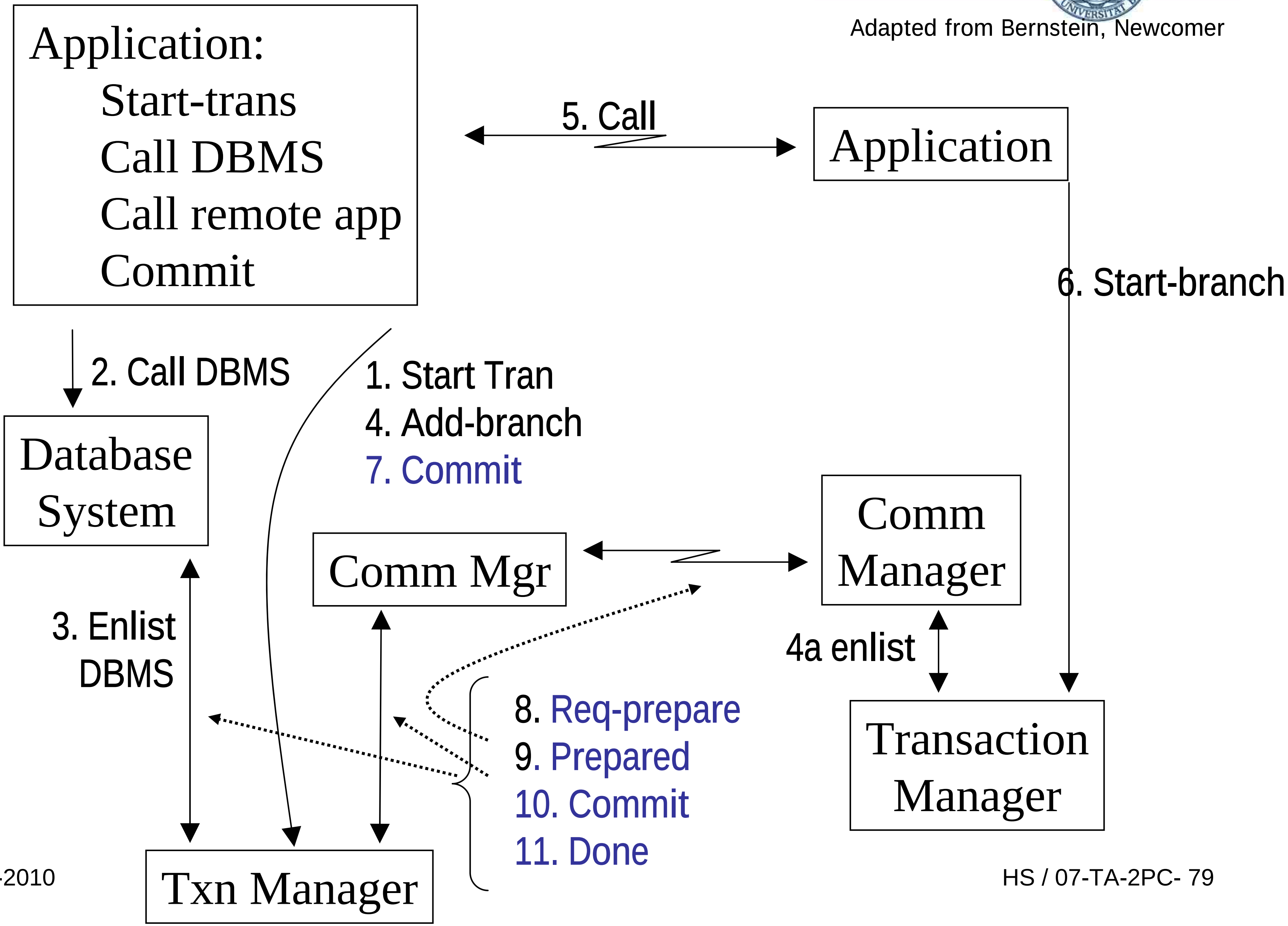
This adds delay to two-phase commit



Complete Walk through

Freie Universität Berlin

Adapted from Bernstein, Newcomer



7.3 Transactional RPC

Three different communication methods between processes:

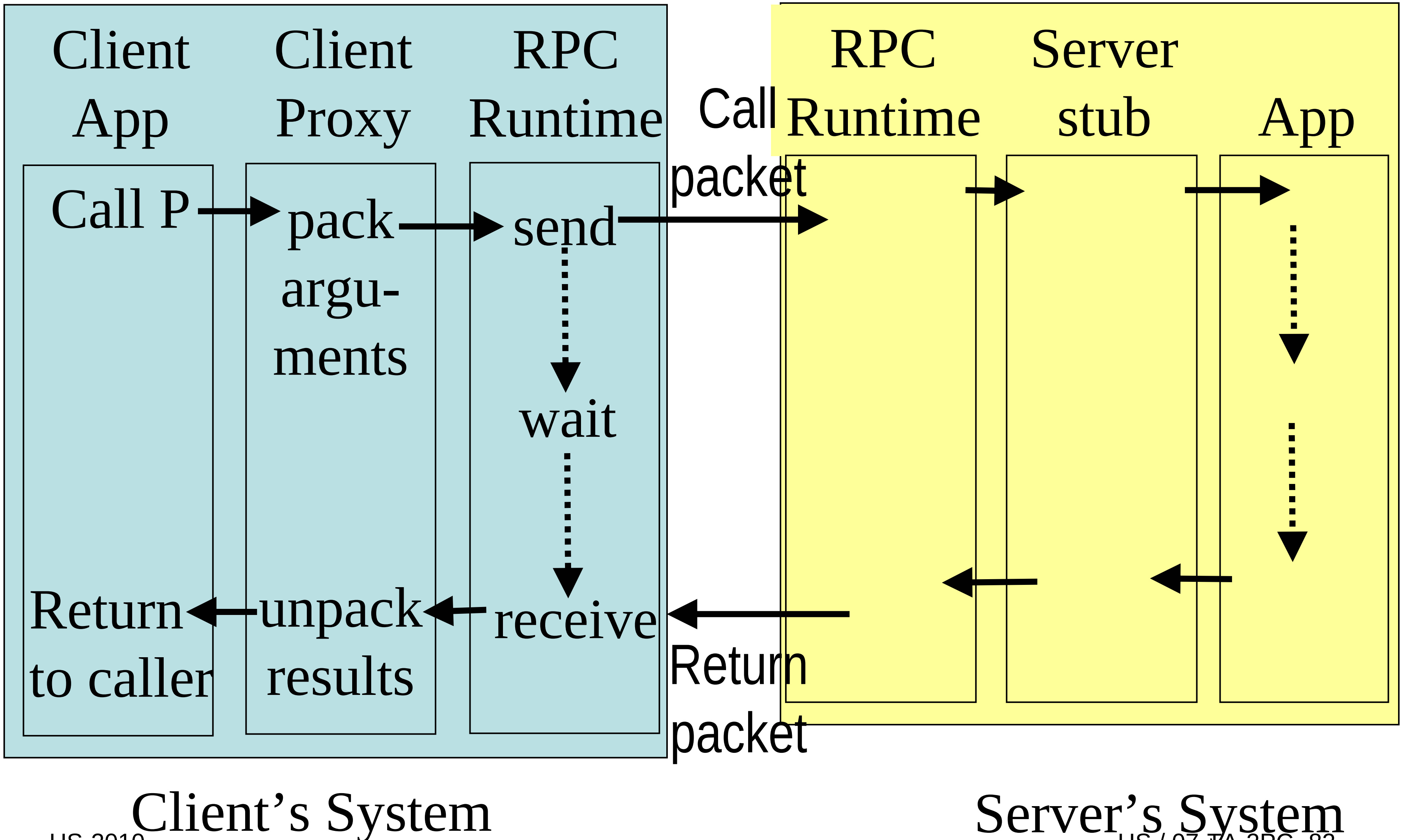
peer-to-peer msg sending (send/receive)

Message queues

Remote Procedure call (not necessarily transactional!)

RPC (non-transactional) walkthrough

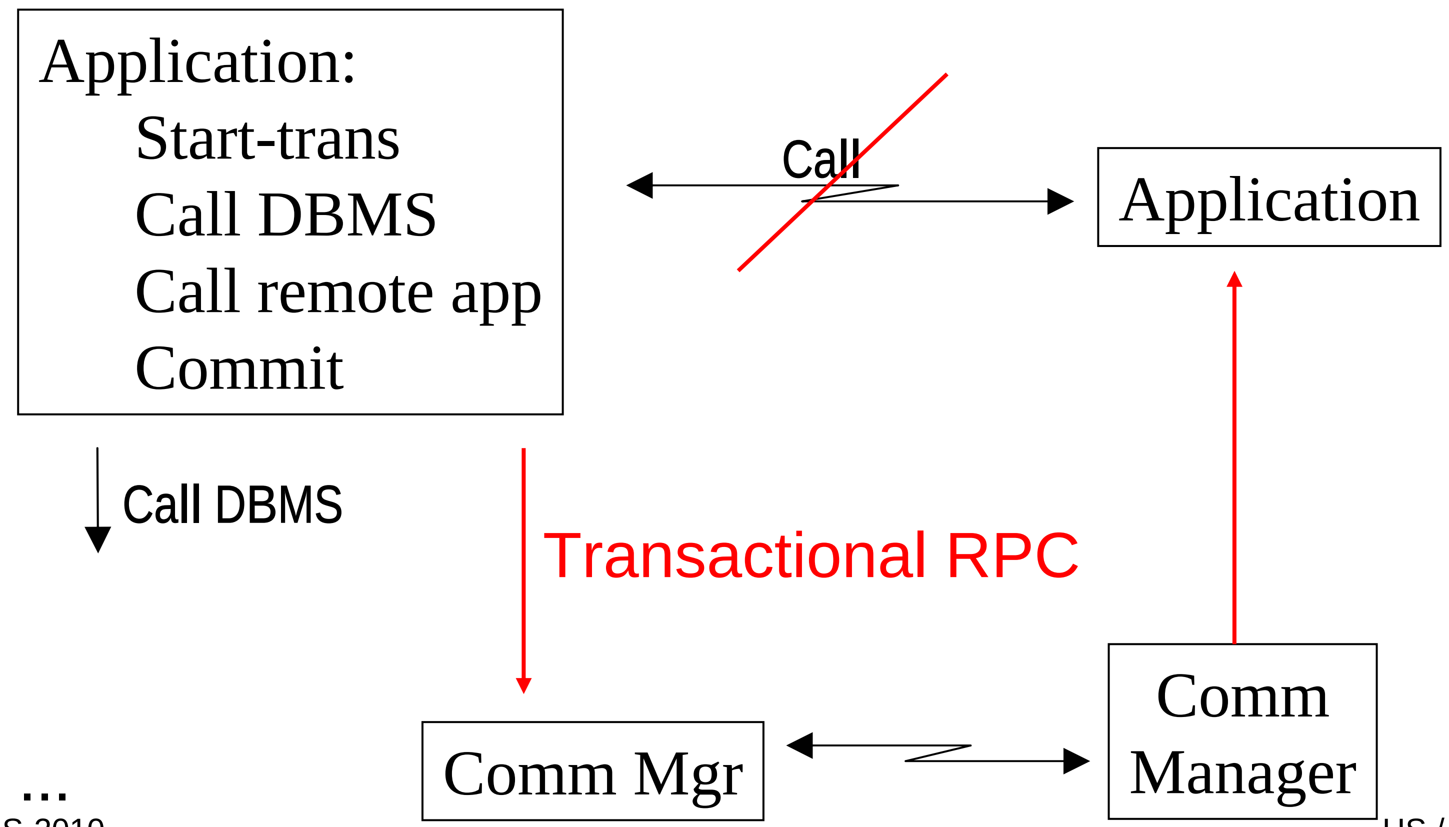
Freie Universität Berlin



Transactional RPC: TxRPC

Transactional RPC

- may be a **member of a global TA**
- **or stand alone RPC ("non-transactional")**



...

TxRPC

Benefit: guarantees **exactly once semantic**

Each call gets TXID (different from global TID!)

Call starts a client timer which may
repeat the call **with the same TXID**
$\Rightarrow$ server knows that this is

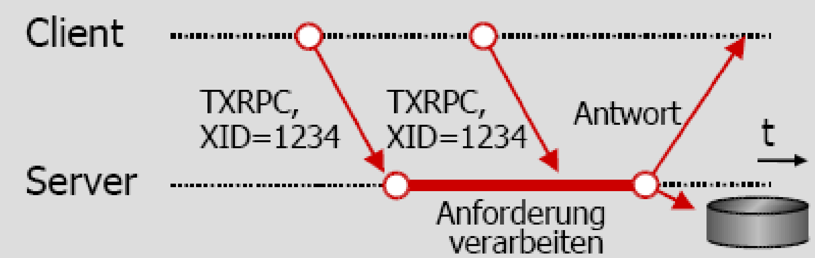
- a repeated call: ignore
- the first call (because of some failure): process

Server has to keep the result in stable store in
order to be able to resend lost result messages

Exactly once semantics of TxRPC

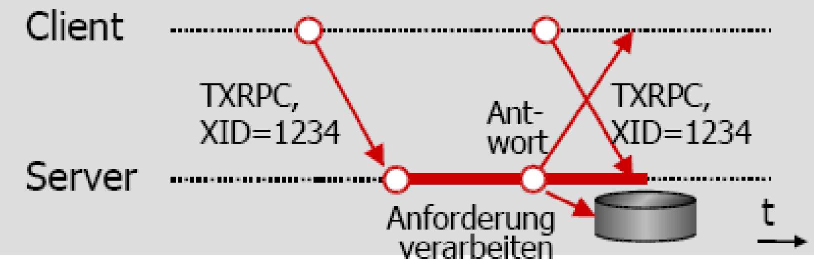
Freie Universität Berlin

Anforderung wird noch verarbeitet



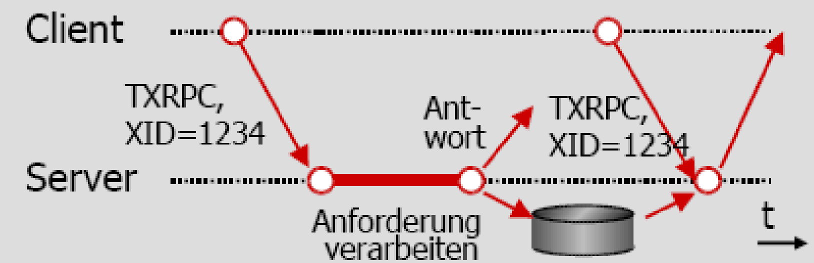
Durch die gleiche XID können erster und zweiter Aufruf als identisch erkannt werden, so dass der Server den Auftrag nicht ein zweites Mal starten muss.

Antwort gerade zurückgegeben



Durch das Eintreffen des Auftrags kurz nach Ausliefern der Antwort geht der Server davon aus, dass die Antwort noch unterwegs ist und startet auch hier den Auftrag nicht ein weiteres Mal.

Antwort ist verloren gegangen



Nach Eintreffen des Auftrags mit der XID eines bereits bearbeiteten Auftrags muss der Server davon ausgehen, dass die Antwort verloren gegangen ist und mit Hilfe der persistenten Speichers die Antwort reproduzieren.

by E. Heinz, UMIT, At

7.5 One phase commit

Example: Calendar application

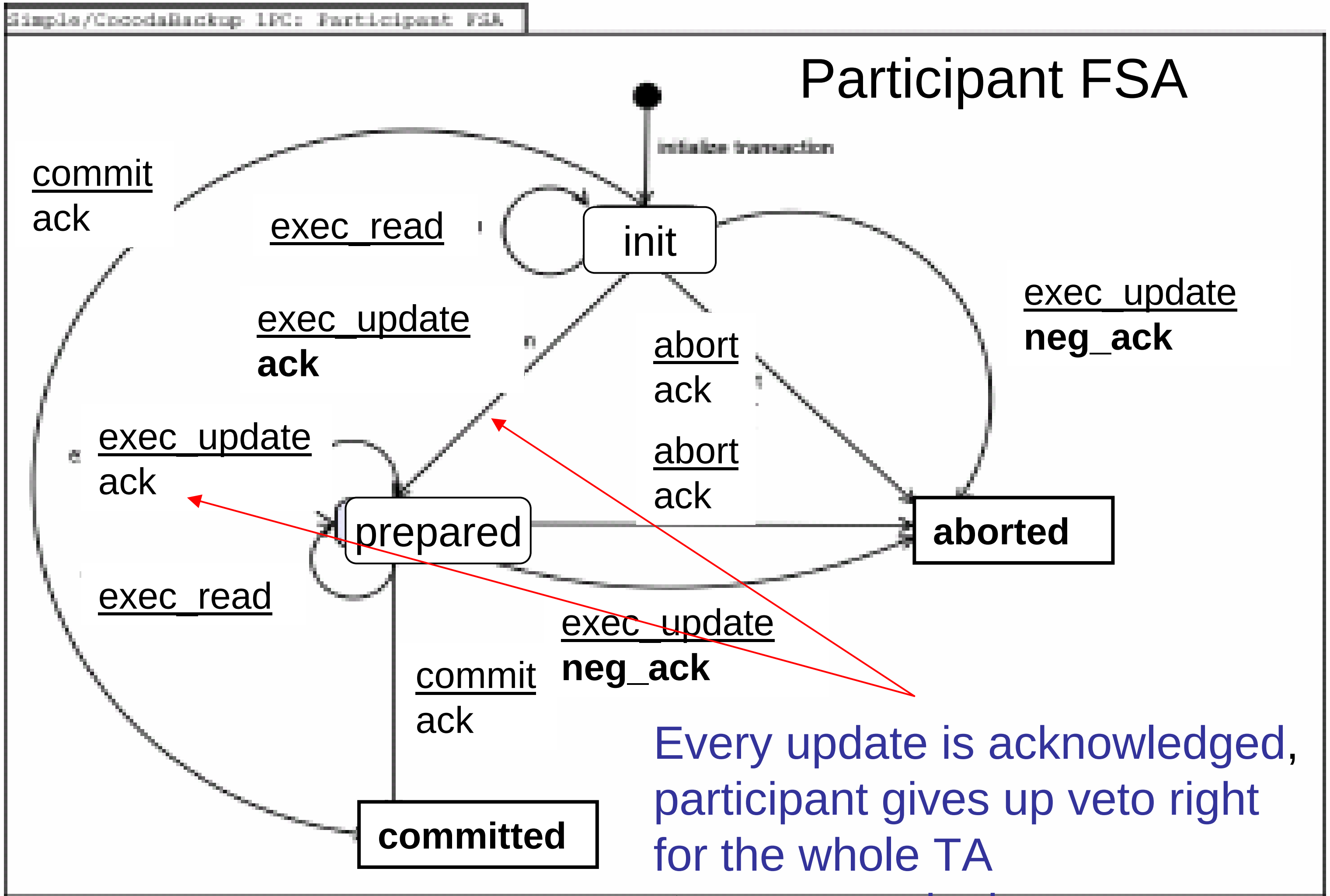
Application protocol: agreement on the date / time of some event.

e.g:

```
".. everyone happy with suggested date?
      if one participant votes no,
              coordinator makes new suggestion
      else commit  (1-phase)"
```

Agreement between nodes in processing phase, not during commit.

1PC: participant protocol



Every update is acknowledged, participant gives up veto right for the whole TA
⇨ one commit phase

slide: J. Bross

Notation

Finite state automaton

different for

- participants
- coordinator

State transition labeled by

`msg received / msg send`

`transition fct` δ**`: inputs X states -> states`**

`output fct` λ**`: inputs X states -> output`**

Any statechart type is ok

Characteristics of 1PC

Blocking?

Yes! When?

Two types of blocking:

- **participant failure**
- **coordinator failure** – more serious, why?

Window of uncertainty in failure free case?

Number of messages for commit /abort?

Suppose **n participants**.

More involved task

n participants, each having a variable x_i

clients send increments ("+j") to each of them

no individual ack of an increment operation, (but of msg received)

---- end of operation phase ------

Condition for successful operation: all increments successful (no overflow, or alike)

If not successful: participants reset x_i

Commit coordinator has to decide!

Commit phase? 1PC is not sufficient to come to a unanimous result! Why?

work phase

commit phase