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!

⇒ 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: Assumptions
- Subtransactions $T_i, i=1..n$ 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 at a participant ⇒ abort
- All operations successful and AP says: commit
  ⇒ 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?

2PC Resource Mgr
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
- Work
  - $C$ requests vote (y/n) from each participant
  - $C$ collects votes and decides: abort if at least one vote is abort, else commit.
  - Participants send ack
- Commit processing
  - $C$ waits for message from participants
  - $C$ sends “commit” or “abort” message to participants
  - Participants send ack
  - $C$ waits for all acknowledgments
  - $C$ sends final ACP to participants
  - Participants send ack

P=Participant
Phases of 2PC

- **Prepare** phase (for participants)
- **Commit** phase
- **Abort** phase

**Uncertain phase (for participants)**

**2PC as a state cart**

State chart:

```
Z -> Z'
```

Mealy automaton: on input \textit{in} in state transition \textit{Z} \rightarrow \textit{Z}' and output \textit{out}.

\textit{in} may be a predicate on one or more inputs.

2PC - Coordinator

Initial state: send prepare

Collecting:
- all p: ack / send commit
- committed
- aborted
- forgotten

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

2PL: Participant protocol

Initial state: prepare / yes

Prepare / sorry

Commit / ack

Abort / ack

2PC and fault tolerance

Is the protocol fail-safe?

- Message loss or process failure \implies protocol failure
- Each process restarts after failure at last remembered state

\textbf{Forced logs} for different states in order to be able to recover.

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 acks

**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?)
Protocol failures

Not so easy:
- coordinator:
  - failed after writing prepared log entry
  ⇒ wait for "yes / ack" of all participants

But some messages could have get lost
(or where never sent?) ⇒ 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 2PC-TA coordination
Could reliable protocol (TCP/IP) be utilized?
Would message queues help? (delivery guaranteed!)

Example 2PC with Log records

Coordinator C
Participants: P1 P2

force-write Coord
begin log entry
 send "prepare"
 send "prepare"

Participant
force-write force-write
begin log entry prepared log entry
 send "yes" send "yes"

force-write Coord
commit log entry
 send "commit"
 send "commit"

Participant
force-write force-write
commit log entry commit log entry
 send "ack" send "ack"

write Coord
send log entry

Log logging

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)
→ 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

- Participants: $n$, Coordinator: $I$
  - $4n$ 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:
- There are no malicious commission failures
- No assumptions about network failure handling

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
  - 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?

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 failure handling

Coordinator fails... and recovers (or timed-out)

1. not yet in state committed | aborted
   - send "prepare" to all partners
   - 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

Manual intervention could be necessary
2PC failure handling

Participants 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

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

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? $c \rightarrow P \rightarrow P \rightarrow \ldots \rightarrow P$

Example 2PC

Coordinators C, Participants: S1, S2

- start 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
  - send "ack", send "ack"

write 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 aborted
  - 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

Variants and optimization of 2PC

Distributed commit

2PC assumed abort: illustration

Case 1: transaction abort

Case 2: transaction commit

2 rounds

$\text{n}^2 + n$ messages
Linear commit

Hierarchical process structures

- During transaction execution, the transaction forms a process tree rooted at the 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:

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

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.

Coordinator
Log prepared
forced
Log commit
forced
Request-to-Prepare-and
-transfer-coordination
Commit
Ack
Participant
Log committed
forced
Log done, lazy

Transfer of Coordination

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

Trans. Coord.
1PC
DBSx
2PC

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

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

X/Open Distributed Transactions in Practice

8.2 Distributed Transactions in Practice

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

Global coordinator
participant
Commit point
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

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 (MDTTC) 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.

1. startTransaction (returns Transaction ID)
2. enlist(T)
3. write(X, T)

Application
Resource Manager
Transaction Manager

enlist() issued by application server, if present

Building a process tree: Enlisting a TA

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

1. Call(AP-B, T)
2. AddBranch(N, T)
3. Send Call(AP-B, T)
4. StartBranch(N, T)

Application A
Transaction Manager
Communications Manager
Application B
Transaction Manager
Node M
Node N

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.

Different Nodes

Complete Walk through

Application: Start-trans
Call DBMS
Call remote app
Commit

Client's System
Server's System

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

Call P
Pack arguments
Send
Wait
Receive
Unpack
Call runtime
RPC
Wait
Receive
Unpack
Result
Return to caller

RPC Runtime
Server's System

Client App
Client Proxy
RPC Runtime
Client's System
**Transactional RPC: TxRPC**

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

**Application:**
- Start-trans
- Call DBMS
- Call remote app
- Commit

**Comm Mgr**

**Comm Manager**

**Call**

**Application**

**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**

⇒ 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**

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**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>exec_read ack</td>
</tr>
<tr>
<td>aborted</td>
<td>exec_update msg_ack</td>
</tr>
<tr>
<td>committed</td>
<td>exec_update ack</td>
</tr>
</tbody>
</table>

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

**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