Today’s Topics - Distributed Transactions

Slide 1
- Introduction to Distributed Transactions 13.1
- Atomic commit protocols 13.3
- Nested transactions and atomic commit protocols 13.2,13.3
- Concurrency control, locking with multiple servers, deadlock detection.

Distributed Transactions

Slide 2
- In the previous lecture we looked at a single server handling transactions.
- This lecture we will look a single transaction on multiple servers.
- A transaction has a number of participants.
- Because transactions need to be atomic we want to have some mechanism so that all the participants do the transaction or none of them.
One Phase commit - does not work

- A transaction should be atomic, when a distributed transaction comes to an end, either all of its operations are carried out or none of them.

Slide 3
- A transaction comes to an end when the client requests that a transaction be committed or aborted.
- A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledge that they have carried it out.

One Phase Commit - does not work

- The one phase commit is fine as long as every body wants to commit.

Slide 4
- It does not for instance allow the coordinator to decide to abort the transactions.
- The coordinator might have to abort the transaction because of concurrency control or deadlocks.
- Also participants are not allowed to abort the transactions.
Two Phase Commit

- Two phase commit gets over the problem by having two phases, a voting phase where all the participants vote and agree on an abort or commit action then in the second phase.
- Things get a little complicated because we have to allow failure of communication and servers.

Failure model for the commit Protocols

- Asynchronous systems.
- Servers may crash and messages may be lost.
- Corrupt and duplicated messages not delivered.
- No Byzantine faults: servers either crash or else they obey the message that their are sent.
The two-phase commit protocol

Phase 1 (voting phase)

1. The coordinator sends a canCommit? request to each of the participants in the transaction.

2. When a participant receives a canCommit? request it replies with its vote (yes or no) to the coordinator. Before voting Yes it prepares to commit by saving objects in permanent storage otherwise it aborts.

Two Phase commit Protocol

Phase 2

1. The coordinator collects the votes (including its own).
   - If there are no failures and all the votes are Yes the coordinator sends a doCommit request to each of the participants.
   - Otherwise the coordinator sends a doAbort request to all participants that voted Yes.

2. Participants that voted Yes are waiting for a doCommit or doAbort request from the coordinator. When participants receive a message it acts accordingly.
Failure modes of the 2-phase Commit protocol

- The protocol looks very simple, but it has some interesting failures modes.
- If a server crashes before it has send a Yes vote then the coordinator will notice.
- If it crashes after it has voted it would have saved the state so the server can be restarted.

Message loss and timeout

- The exchange of information between the coordinator and participants can fail when one of the servers crashes, or when a message is lost.
- Timeouts are used to avoid processes blocking for ever.
- At each point a process can block a timeout action is added.
- Remember in an asynchronous system a timeout might just mean that the message has been lost, not that a server has crashed.
Timeout actions

- If a participant has voted Yes and is waiting for the coordinator to report the outcome of the vote. Then such a participant is uncertain of the outcome and cannot proceed any further until it gets the outcome of the vote.

- The participant cannot decide on its own. There are a few choices:
  - Ask the server again. (Message might be lost). If the coordinator has crashed then it will have to wait until it has been replaced, this could cause some delay.
  - Another alternative strategy is for the participants to obtain a decision cooperatively.

Two Phase Commit

Slide 12
Problems with Two Phase Commit

- As it stands if the coordinator crashed the two phase commit protocol has to be modified or the other process will be waiting around for the server to wake up.
- Participants are blocked. This occurs when all participants have received and processed the `VOTE_REQUEST` from the coordinator, while the coordinator has crashed. When it restarts it is waiting for more `Vote_Request` messages that have been sent.
- The three phase commit protocol gets over these problems.

Three Phase Commit

**Phase 1** Same as two phase commit, coordinator sends a commit message and participants reply with `Yes` or `No` message.

**Phase 2** Coordinator collects the votes and makes decision if `No` then send an `abort` message. Otherwise sends a `PreCommit` message. Participants acknowledge `PreCommit` messages or abort.

**Phase 3** The coordinator collects all the acknowledgements. It is has received all the acknowledgements sends a `doCommit` to all the participants. Participants wait for a `doCommit` message when it arrives they `Commit`. 
Three Phase Commit Protocol

Slide 15

1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.

2. There is no state in which it is not possible to make a final decision and from which a transition to a COMMIT state can be made.

These two conditions are necessary and sufficient conditions for the protocol to be nonblocking.
Three Phase Commit

Blocking Situations There are only a few situations in which a process is blocked while waiting for an incoming message.

- If a participant is waiting for a vote request from the coordinator it will eventually timeout and go to abort assuming the coordinator has crashed.
- Suppose a coordinator is blocked in state `PRECOMMIT`. On a timeout it will conclude that one of the participants has crashed, but that participant is known to have voted for committing the transaction. Consequently, the coordinator can safely instruct the operational participants to commit by multi-casting a `GLOBAL_COMMIT` message.

Three Phase Commit - Blocking situations

- In addition the crashed participant can recover from its crashed state (from its saved state) and all will be well.
- A participant may block in the `READY` state or in the `PRECOMMIT` state. On a timeout the participant will conclude that the coordinator has crashed. It has to decide what to do next. If the participants contacts any other participant that is in state `COMMIT` or `ABORT` and should move to that state. In addition if all participants are in the `PRECOMMIT` state then they can safely `COMMIT`.

You can verify with time outs that the protocol will never block by going through the rest of the conditions.
Nested Transactions

- A nested transaction is nothing more than another transaction inside a transaction.
- The exact semantics of nested transactions (what to do if it aborts) depends exactly on the situation. But in general the sub-transactions must complete as well.
- In terms of distributed transactions a client calls a number of servers that call other servers in a tree (insert figure 13.1).

Two Phase Commit and Nested Transactions

Two strategies

Slide 20

1. **Hierarchical.** Each server in the tree becomes a coordinator for its subtransactions.

2. The coordinator keeps track of all the sub-transactions and keeps track of who is aborting. It is complicated if you let sub-transactions abort without the main transaction aborting.
Concurrency Control in Distributed Transactions

Locking.

- In a distributed transaction locks will be held locally.
- When locking is used, objects remain locked and unavailable during the two-phase commit protocol.
- Aborted transaction release locks after phase 1.
- As lock managers in different servers set their locks independently of one another it is possible that different servers may impose different orderings on transactions.

Locking - Distributed Transactions

Consider two transactions $T$ and $U$ on two different servers $X$ and $Y$:

<table>
<thead>
<tr>
<th>$T$</th>
<th>$U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write(A) at X locks A</td>
<td>write(B) at Y locks B</td>
</tr>
<tr>
<td>Read(B) at Y waits for U</td>
<td>Read(A) at X wait for T.</td>
</tr>
</tbody>
</table>

On one server we have $T$ before $U$ and on the other $U$ before $T$. We have created a cyclic dependency. Which we have to check for.
Distributed Deadlocks

- Each server builds a local wait graph.
- These are sent to some server which glues them together.
- Problem with phantom deadlocks. It takes time to move around the graphs, and cycles could be detected that are no longer cycles. (does not happen if two phase commit is used).
- Central server not a good idea. One point of failure.

Distributed Deadlocks : Edge Chasing

- The global wait graph is not constructed.
- A server sends probes (fragments of the wait graph). These propagate around the network with extra edges added form local information. If there is a cycle it will propagate back to the a server waiting for a lock.
- In this way deadlocks can be found in distributed fashion.

insert figure 13.15