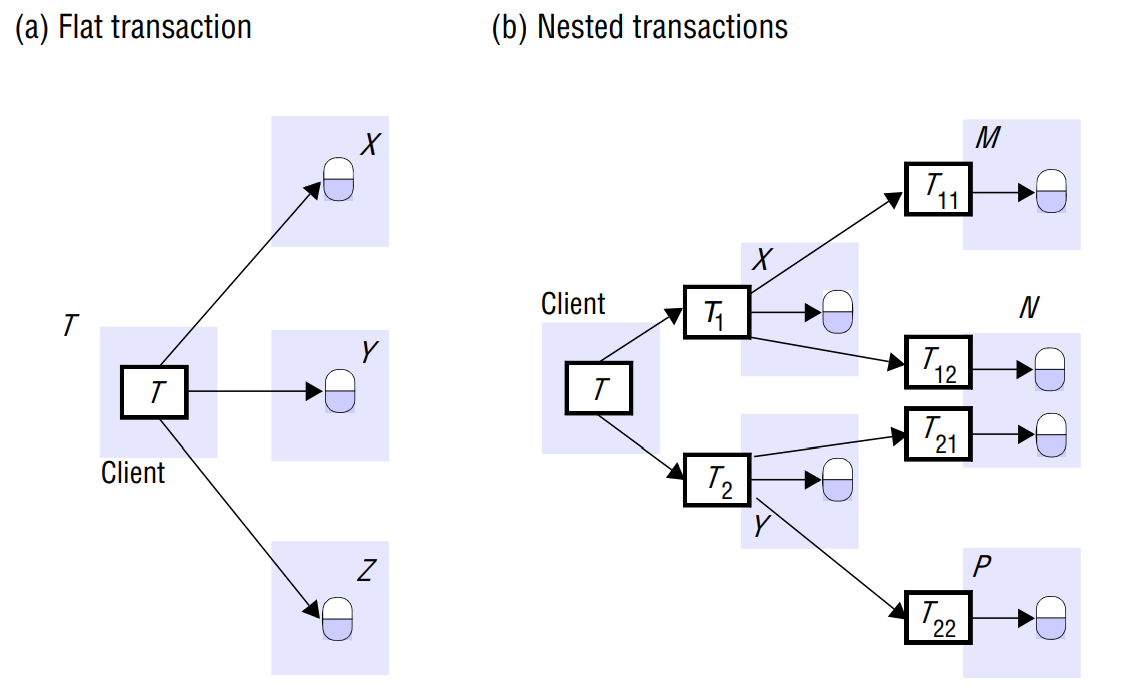
ACID

ATOMIC: must completely happened or not at all

Consistent: must take the database from one valid state to another valid state based on the roles of the database.

Isolation: means that the data should be locket for that moment, in which the transaction is occurring.

Durable: the transaction has been considered robust (if the database says that the transaction happened then you guaranteed that it has happened).

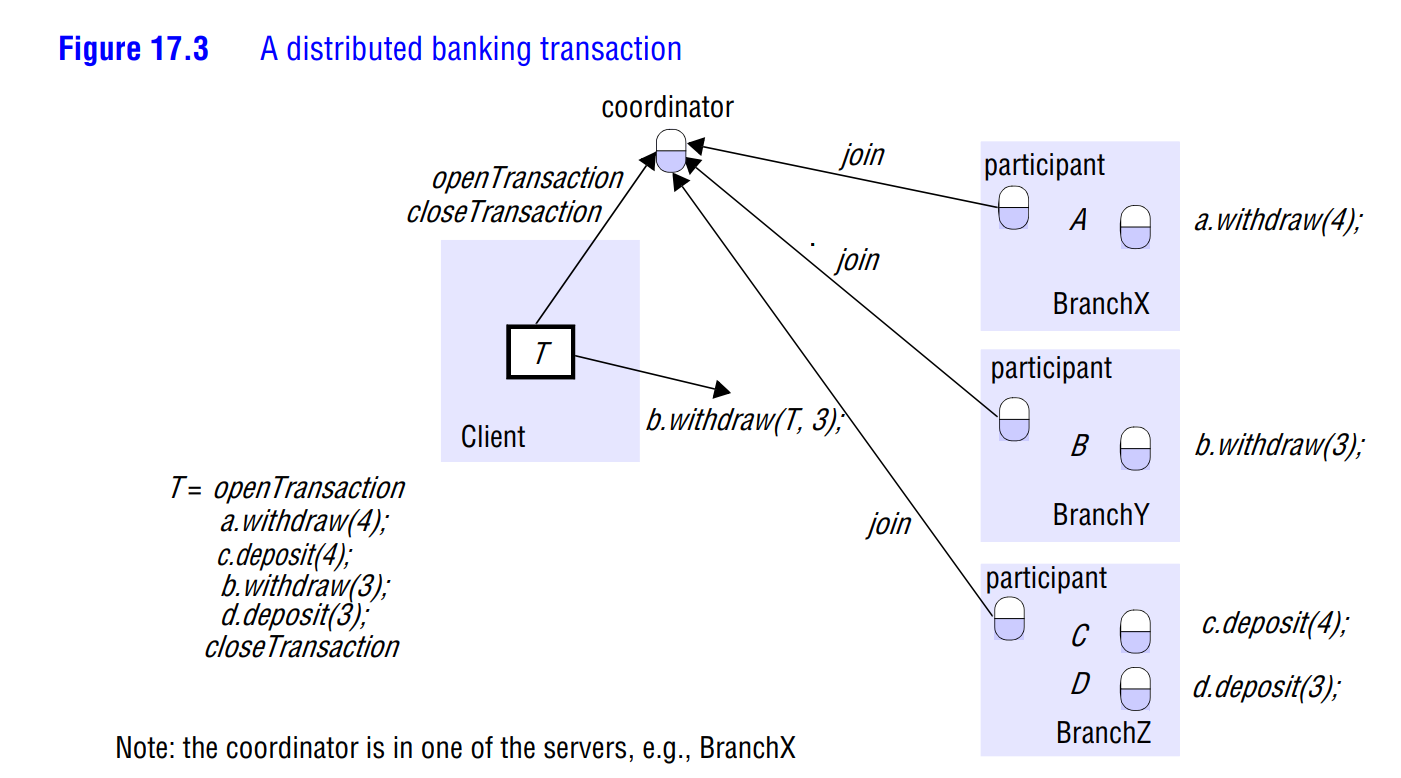
. There are two different ways that distributed transactions can be structured: as flat transactions and as nested transactions. 

In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions.

In a flat transaction, a client makes requests to more than one server. **subtransactions at the same level can run concurrently**

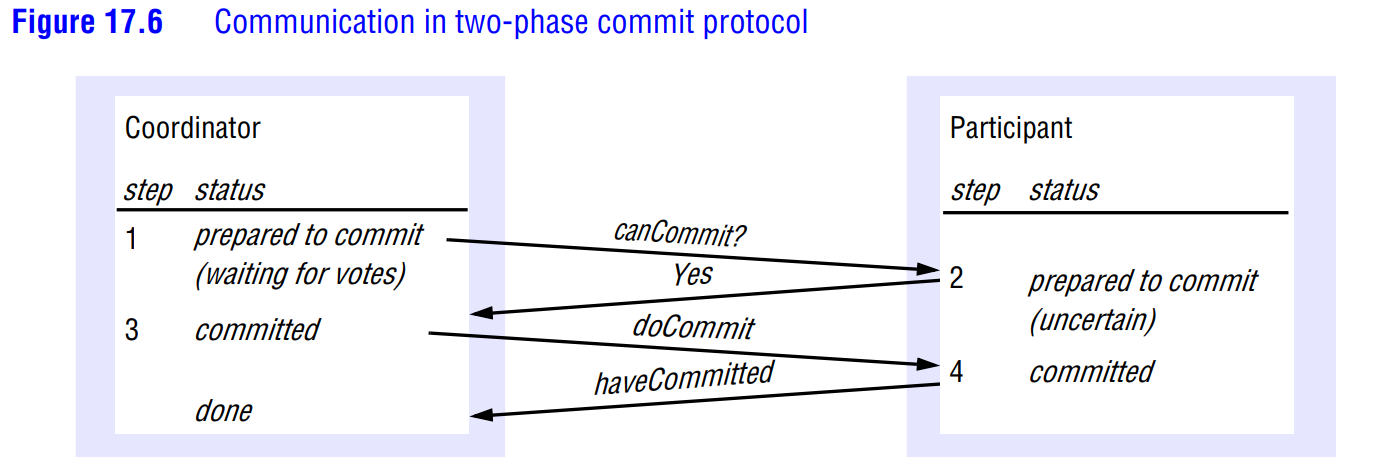
The coordinator ensure that the Transaction identifiers for distributed transactions must be unique within a distributed system. A simple way to achieve this is for a TID to contain two parts: the identifier (for example, an IP address) of the server that created it and a number unique to the server.

The coordinator records a list of references to the participants, and each participant records a reference to the coordinator



The two-phase commit protocol:

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; in the second, it tells them to commit (or abort) the transaction. If a participant can commit its part of a transaction, it will agree as soon as it has recorded the changes it has made.



The methods can Commit, do Commit and do Abort are methods in the interface of the participant. The methods have Committed and get Decision are in the coordinator interface.

Problems

* Lost update: an update is lost because the transaction is working on old data
* Inconsistent read: reading from different versions of data
* Dirty read: reading from a transaction that was later rolled back (cancelled)

Fixing the problems

* Pessimistic locking
  + Lock everything you touch (read or write)
  + Only release it when you're completely done

The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished

* + Pessimistic locking can lead to bottlenecks where the system bogs down
  + Examples
    - Windows file system
* Optimistic locking
* Let’s assume we are building an online wikipedia – like application using Couchbase Server: users can update an article and add newer articles. Let’s assume Alice is using this application to edit an article on **‘bicycles’**to correct some information. Alice opens up the article and makes those changes but before she hits save, she gets distracted and walks away from her desk. In the meantime, let’s assume Joe notices the same error in the bicycle article and wants to correct the mistake.
* If optimistic locking is used in the application, Joe can edit the article and save his changes. When Alice returns and wants to save her changes, either Alice or the application will want to handle the latest updates before allowing Alice’s action to change the document.
  + You're not allowed to write if you have an old version
    - Ex: github
  + You don't need to download the entire database if the database on the server is ahead, you just download the data you need to affect
* Avoid deadlocks:
  + lock all editable resources at transaction start (simple but not effective)
  + Use timeouts

Java Transaction API (JTA)

• Has all the code that happens in the coordinator

• Participant databases need to be jta ready (drivers)