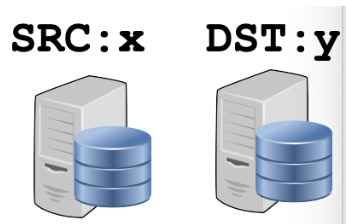
CAS CS 350
Lec 15

Distributed Commit

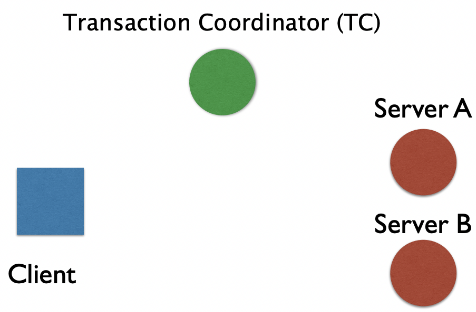
1. Transaction (Database)
    a. Unit of work (read-write operations that is):
        i. Atomic: Either the whole translation succeeds or fails
        ii. Consistent: Does not violate violate database restrictions after completion
            1. Examples of constraints: unique keys, primary keys → should be no duplicates
        iii. Isolated: Results of incomplete transactions are not visible to other transactions
        iv. Durable: After the transaction completes (committed) it cannot be lost
2. Distributed commit
    a. A bunch of computers are cooperating on some task, e.g. bank transfer
    b. Each computer in the system has a different role - data
    c. 

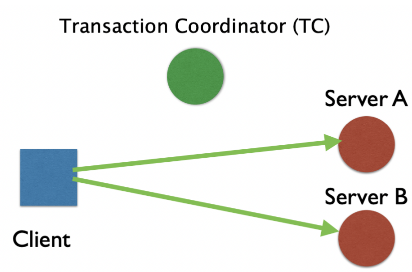
    d. Transfer:
        src : x -= 10, dst: y+= 10
    e. Need atomicity: all ops execute or none (do not want corrupted bank system)
    f. Challenges: failures, concurrency, performance (efficiency) → even if there are challenges such as failures, we need atomicity (all operations executing)
    g. Raft → build better replicated state machine since it is a consensus algorithm
    h. Cannot use raft to build this protocol
        i. Servers are not replicas → they hold different account unlike raft which requires identical logs
        ii. In raft, all replicas execute same command to keep the same data
    i. Need another protocol
3. The idea
    a. First tentative changes
    b. Later commit or undo (abort)

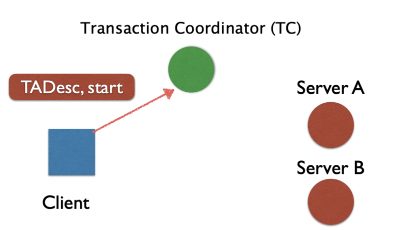
c.
```
reserve_handler(u, t):
  if u[t] is free:
    temp_u[t] = taken -- A TMP VERSION
    return true
  else:
    return false
```
```
commit_handler():
  copy temp_u[t] to real u[t]
```
```
abort_handler():
  discard temp_u[t]
```

d. Use temporary objects → in the future, we want to undo the changes sometimes (similar to GFS)

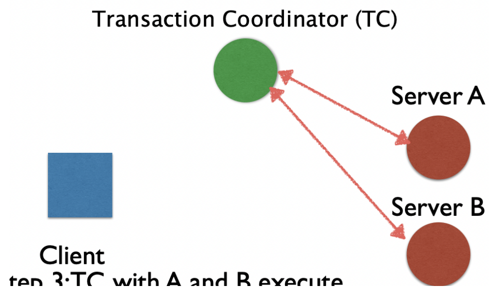
e. Also needs a single entity (leader) that decides whether to commit it prevents any chance of disagreement && leader communicates with the client

f.


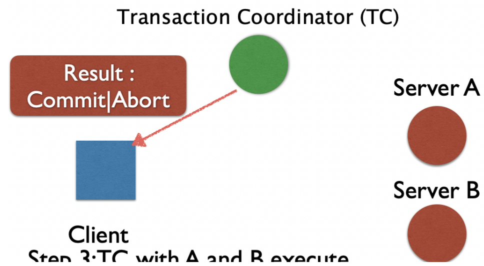
g. Steps:
   i. Client still sends RPCs to A and B


   ii. On end_transaction, client sends “go” to TC (the transaction should be all or nothing)


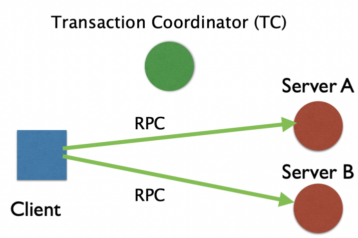
   iii. TC with A and B execute distributed commit protocol (the transaction should be success or error)



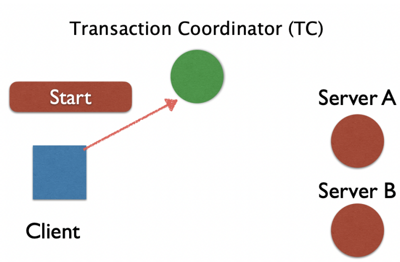
iv. Result gets sent back to the user on whether it was committed or aborted



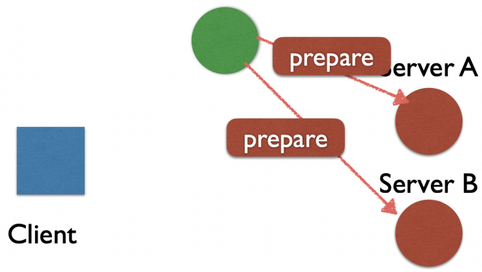
4. Properties we want
    a. Correctness
        i. If any commit, none abort
        ii. If any abort, none commit
    b. Performance: (since doing nothing is correct…abort)
        i. If no failures, and A and B can commit, then commit
        ii. If failures, come to some conclusion ASAP
        iii. We're going to develop a protocol called "two-phase commit" (2PC)
    c. Used by distributed databases for multi-server transactions
5. 2PC no failures
    a. These RPCs contain read-write operations included in the transactions



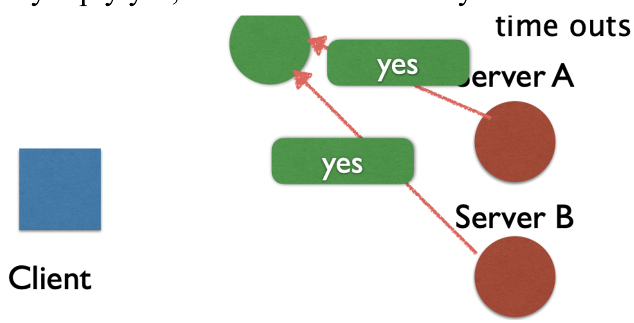
    b. Client sends start to the coordinator



c. Coordinator starts first phase of the protocol: prepare phase
    i. Coordinator sends RPC and waits for response (response is yes or no) depending on whether the servers are willing to commit the transactions

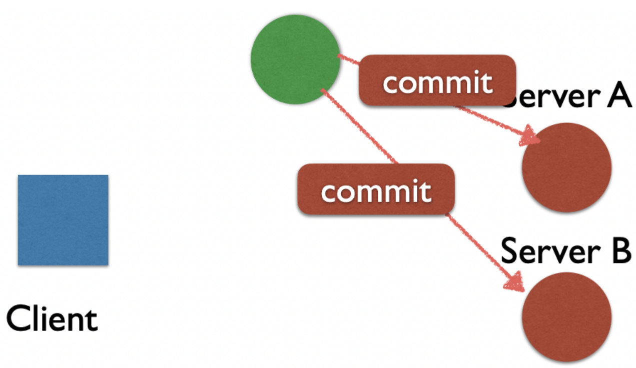


d. If they reply yes, which means that they have not crashed, no time outs, etc



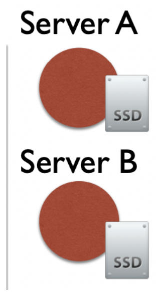
e. A and B respond, saying whether they're willing to commit

```
If  A:"yes" && B:"yes" {  send "commit" }
else { send "abort" }
```



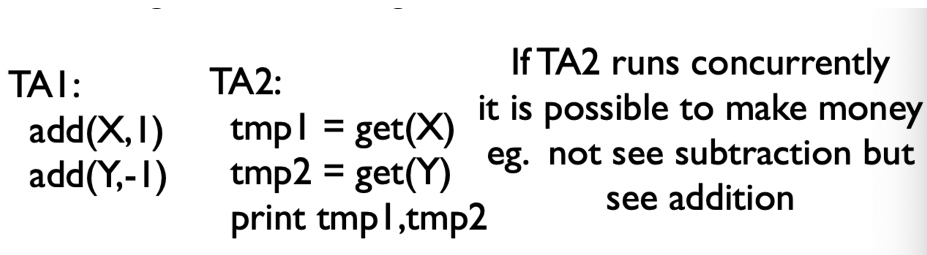
f. Servers replace real data according to the temporary data created by transactions
g. A/B commit if they get a commit message. I.e they actually modify the user's account
h. Why is this correct so far?
    i. Neither can commit unless they both agreed
    ii. But crucial that neither changes mind after responding to prepare. Not even if failure! → even if there are failures, they need to both agree or disagree

6. 2PC with Failures
    a. What about failures?
        i. Network broken/lossy/slow
        ii. Server crashes
    b. What is our goal wr.t. Failure?
        i. Resume correct operation after repair (most important is that it is correct → it is all or nothing) → efficiency is not a big issue
        ii. Recovery, not availability
    c. Single symptom: timeout when expecting a message
    d. Where do hosts wait for messages?
        i. TC waits for yes or no response to prepare message
        ii. A and B wait for prepare messages and commit/abort messages
7. Terminating
    a. TC timeout for yes/no (scenarios in 2 phase protocol)
        i. Abort → ignore the transactions sent by client
        ii. TC has not sent any "commit" messages. So TC can safely abort, and send "abort" messages
    b. One of the servers, B, timeout for prepare (doesn't receive prepare message)
        i. Abort
        ii. A/B timeout while waiting for prepare from TC have not yet responded to prepare, so TC can't have decided commit
        iii. So A/B can unilaterally abort
        iv. Respond "no" to future prepare
    c. B timeout for commit/abort and B voted NO
        i. Abort (Cannot apply temporary data)
        ii. If B voted "no", it can unilaterally abort
        iii. At least one server doesn't reply or does "abort" → coordinator cannot send out commit messages
    d. B timeout for commit/abort and B voted YES
        i. If B voted "yes". Can unilaterally abort? No!
        ii. TC might have gotten "yes" from both, and sent out "commit" to A, but crashed before sending to B
        iii. So then A would commit and B would abort: incorrect
        iv. If B voted "yes", it must "block" and must repeatedly ask the coordinator what to do (wait for TC decision)
        v. By blocking, it becomes unavailable
8. 2PC with failures
    a. What if B crashes and restarts?
        i. If B sent "yes" before crash, B must remember! → if not, result inconsistency

        ii. Can’t change to “no” and abort after restart
        iii. Since TC may have seen previous yes and told A to commit
    b. Participants must write persistent (on-disk) state:
        i. 

        ii. B must remember on disk before saying “yes”, including modified data
        iii. If B reboots, disk says “yes” but no “commit”, B must ask TC
        iv. If TC says “commit”, B copies modified data to real data
    c. What if TC crashes and restarts?
        i. If TC might have sent “commit” or “abort” before crash, TC must remember
        ii. And repeat that if anyone asks (i.e. if A/B/client didn’t get msg)
        iii. TC can’t change its mind since A/B/client may have already acted
9. 2PC
    a. This protocol is called “two-phase commit”
        i. All hosts that decide reach the same decision
        ii. No commit unless everyone says “yes”
        iii. TC failure can make servers block until repair
    b. Has a bad reputation regarding availability
    c. Slow because of multiple phases/ message exchanges
        i. Locks are held over the prepare/commit exchanges; blocks other transactions
        ii. TC crash can cause indefinite blocking, with locks held
    d. Thus, usually used only in a single small domain (e.g. not between banks, not between airlines, not over wide areas)
    e. Better transaction schemes are an active area of research
10. Question
    a. Does 2PC and raft solve the same problem?
    b. No
        i. Raft is the replicated state machine that have operations throughout all servers while 2PC has servers that stores different account and executes different commands
        ii. Use raft to get high availability by replicating, i.e., to be able to operate when some servers crash, the servers all do the same thing
            1. Raft servers are more available

            a. Even if there are some failures, we can still operate
            b. Relies on majority (does not wait for all servers → wait for just majority of servers reply)
        2. 2PC
            a. If at least one server is down, the whole protocol is lost
    iii. Use 2PC when each participant does something different – And all of them must do their part
    iv. Raft does not ensure that all servers do something since only a majority have to be alive

11. Concurrency
    a. What about concurrent transactions?
        i. We usually want concurrency control as well as atomic commit
        ii. Eg. transfer along with audit - sum
        iii. 

12. Serializability
    a. Not to be confused with linearizability
    b. Correctness condition for transactions
    c. Concurrent execution of transactions must be equivalent to a serial execution
        i. If no equivalent serial execution exists, a transaction have read results of other incomplete transactions
        ii. If serial order respects the real time the transactions were committed by the client, then
            1. Strict serializability

13. Linearizability
    a. A strong consistency model
    b. Correntness condition for R/W operations
    c. Raft supports linearizable semantics
    d. Clients cannot read stale data