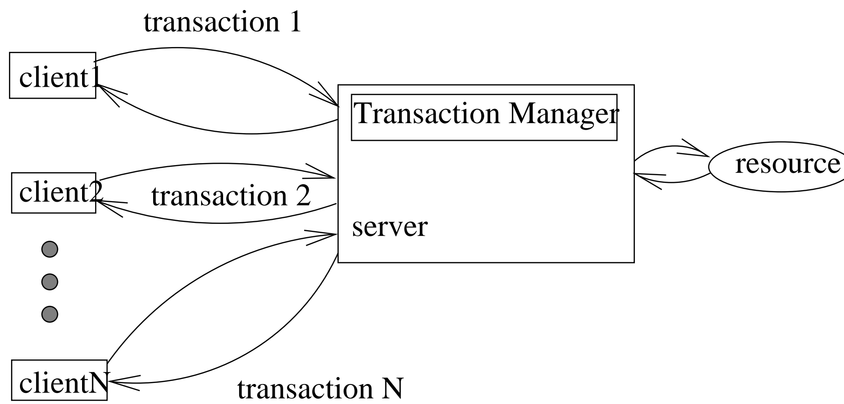
12 Distributed Transactions

- Models for distributed transactions
- Attaining distributed commitment
- Distributed Concurrency Control

12.1 Single Server Transactions



- Till now, transactions have referred to multiple clients, single server.
- How do we have multiple clients interacting with multiple servers? eg complicated funds transfer involving different accounts from different banks?
- Generalise transactions to distributed case...

12.2 Distributed Transactions

12.2.1 Distributed Transaction Requirements

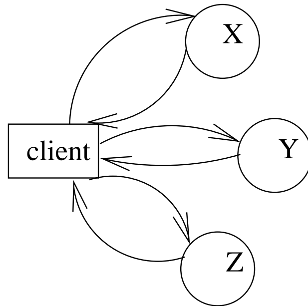
General characteristics of distributed systems

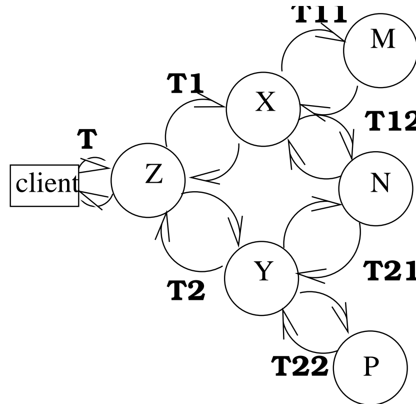
- Independent Failure Modes
- No global time
- Inconsistent State

Need to consider:

- how to achieve distributed commitment (or abort)
- how to achieve distributed concurrency control

12.2.2 Models





Simple Distributed model

Nested Transaction

- If client runs transactions, then each transaction must complete before proceeding to next
- If transactions are nested, then transactions at same level can run in parallel
- Client uses a single server to act as *coordinator* for all other transactions. The coordinator handles all communication with other servers

Question: What are the requirements of transaction ids?

12.3 Atomic Commit Protocols

- Distribution implies independent failure modes, ie machine can fail at any time, and others may not discover.
- If *one phase commit*, client requests commit, but one of the server may have failed - no way of ensuring durability
- Instead, commit in 2 phases, thus allowing server to request abort.

12.3.1 2 Phase Commit

- One *coordinator* responsible for initiating protocol.
- Other entities in protocol called *participants*.
- If coordinator or participant unable to commit, all parts of transaction are aborted.
- Two phases

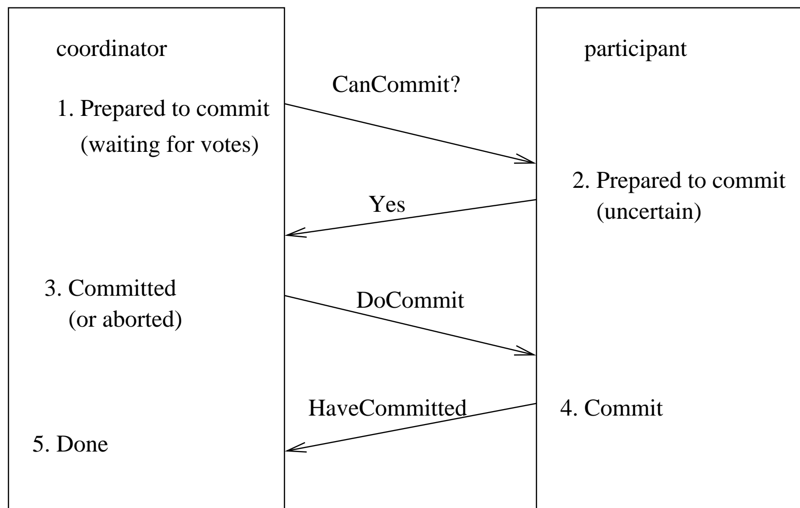
  Phase 1 Reach a common decision

  Phase 2 Implement that decision at all sites

2 Phase Commit Details

1. **Phase 1** The coordinator sends a *Can Commit?* message to all participants in transaction.
2. Participants reply with vote *yes* or *no*. If vote is *no* participant aborts immediately.

3. **Phase 2** Coordinator collects votes including own:
    (a) If all votes are *yes*, coordinator commits transaction and sends *DoCommit* to all participants.
    (b) Otherwise transaction is aborted, and coordinator sends *abortTransaction* to all participants.
4. When a participant recieves *DoCommit*, it commits its part of the transaction and confirms using *HaveCommited*



2 Phase Commit Diagram Note:

- If participant crashes after having voted to commit, it can ask coordinator about results of vote.
- Timeouts are used when messages are expected.
- Introduces new state in transaction Prepared to commit.

12.4 Distributed Concurrency Control

12.4.1 Locking

- Locking is done per item, not per client.
- No problems generalising to multiple servers...
- ...except in dealing with distributed deadlock
- Same techniques as usual, but interesting dealing with distributed deadlock detection.

12.4.2 Optimistic Concurrency Control

- Need to worry about distributed validation
- Simple model of validation had only one transaction being validated at a time - can lead to deadlock if different cordinating servers attempt to validate different transaction.

- Also need to validate in correct serialisable order.
- One solution is to globaly only allow one transaction to validate at a time.
- Other solutions is to validate in two phases with timestamp allocation - local, then global to enforce ordering.

12.4.3 Timestamping

- If clocks are approximately synchronised, then timestamps can be
  $< localtimestamp, coordinatingserverid >$
  pairs, and an ordering defined upon server ids.

12.5 Summary

- Nested Transactions are best model for distributed transactions
- Two Phase Commit protocol suitable for almost all case
- Distributed Concurrency control is only slightly more diffcult than for single server case