Distributed Transactions

- Synchronizing the updates and reads of data in a distributed system.
- Atomic Transactions
- Two Phase Commit

Transactions

Example

- If we want to move $\$1\,000$ from one bank account A to another account B we can have the following code:
  - $a \leftarrow read(\mathrm{A})$
  - $b \leftarrow read(\mathrm{B})$
  - $a \leftarrow a - 1\,000$
  - $b \leftarrow b + 1\,000$
  - $write(a,\mathrm{A})$
  - $write(b,\mathrm{B})$
- There might arise several problems when this is executed:
  - A crash might occur:
    - before the write statements: nothing will be changed
    - between the write statements: $\$1\,000$ will disappear into the abyss
    - after the write statements: this is no problem

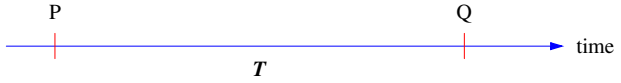
Transactions (cont.)

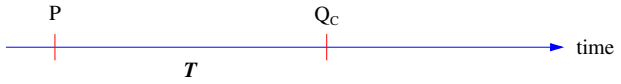
- The transaction performer might change its mind and stops the transaction abort or rollback:
  - The accounts A and B must be set to the original values.
  - Some other transaction might have read a new value and uses it for new calculations: $\$1\,000$ might disappear or emerge from nowhere. dirty read
- The solution is to use an atomic transaction.

Atomic Transaction

Indivisible: is completely performed or not at all.

Isolation: the result of concurrent atomic transactions should be as if they were performed in a certain order, not necessarily known from the beginning.





T: transaction composed of several events
P: precondition, holds before ***T***
Q: post condition, holds after ***T*** if ***T*** is correctly performed
Q_C: crash condition, holds after ***T*** if ***T*** is interrupted (crash)

Atomic Transaction (cont.)

- Transaction ***T*** atomic $\Rightarrow Q_C = (P \vee Q)$.

$\Rightarrow$

- **momentarily** transition from P to Q
  - *commit*
- *abort* (*roll back*) keeps P
- A crash before *commit* must leave the system in state P
- A crash after *commit* must leave the system in state Q
- *commit* is an acknowledgement to the client user that the transaction will hold "forever".
- ***T*** restartable if it has not made *commit*

Transaction

Definition

A sequence of

- *read*
- *write*
- *end* ≡ *commit*
- *abort* or *rollback*

that is sent by a Client program to a data storage system.

- all or none of the *write* commands should be performed
- no other transaction should see intermediary data (no dirty reads)

ACID properties

- Härder and Reuter, 1983
- A transaction must obey the following properties:
  - Atomicity — all or nothing
  - Consistency — take the system from one consistent state to another consistent state, e.g. in a banking system it might be that the sum of all accounts is constant.
  - Isolation — no illegitimate influence among different transactions.
  - Durability — the write operations of a committed transaction must hold for the future.

Nested Transactions

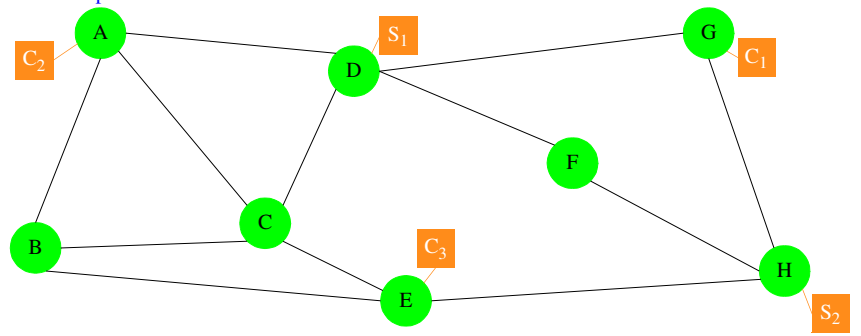
- Nested Transactions are transactions composed of other transactions, subtransactions.
  - subtransactions at one level may run concurrently with other subtransactions at the same level. This could be done on different servers.
  - subtransactions can commit or abort independently.
- Commit rules:
  - A transaction may commit or abort only after its child transactions have completed.
  - When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A decision to abort is final.
  - When a parent aborts, all of its subtransactions are aborted, even if they have committed provisionally.
  - When a subtransaction aborts, the parent can decide whether to abort or not.
  - If the top level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted.

Atomic Transactions - solutions

- Pessimistic
  - Keep out other transactions from the involved data.
  - Locks
  - two-phase commit
- Optimistic
  - Start transaction directly
  - When changing the data first check that the involved data has not been changed.
    - If changed the transaction must be restarted.
    - Possibly there might have to be a cleanup.
  - non-blocking solutions
  - when there is low probability for concurrent access of the same data.

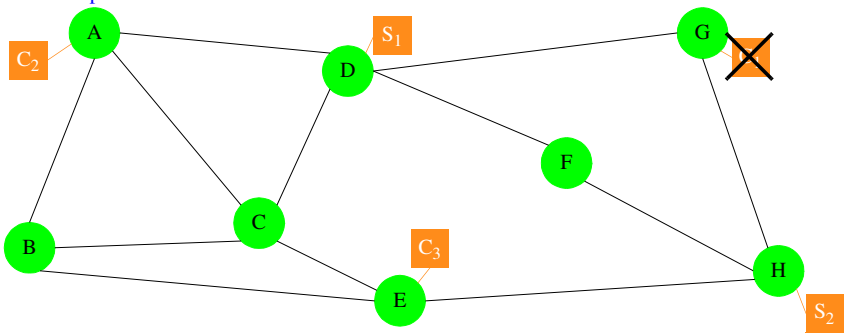
Transactions in Distributed Systems

- There can be partial crashes
  - Some clients/servers crashes while others continue to work.
  - Traditional solutions for crash recovery does not work.
  - use two-phase commit



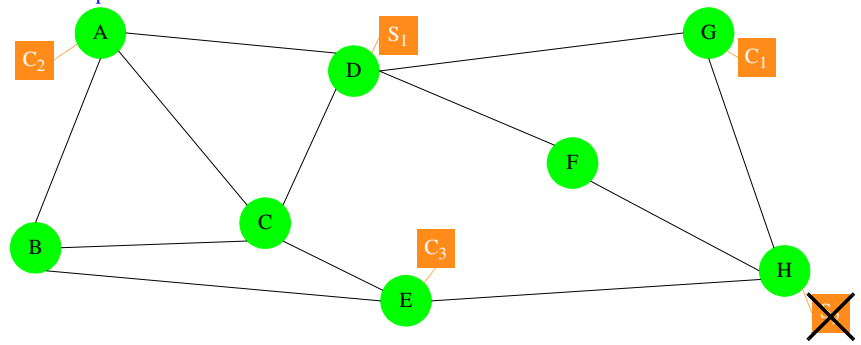
Transactions in Distributed Systems

- There can be partial crashes
  - Some clients/servers crashes while others continue to work.
  - Traditional solutions for crash recovery does not work.
  - use two-phase commit



Transactions in Distributed Systems

- There can be partial crashes
  - Some clients/servers crashes while others continue to work.
  - Traditional solutions for crash recovery does not work.
  - use two-phase commit



two-phase commit

- phase 1:
  - Lock transaction data.
  - perform reads and writes in normal order **but**
  - write on an intention list instead of changing the corresponding memory addresses.
  - The intention list should be stored on stable storage,
    i.e. memory that survives a processor crash, e.g. a hard disk.
- *commit* - the intention list is made valid
- phase 2:
  - write to the memory addresses according what has been stored on the intention list.
  - Then the locks can be released.

two-phase commit (cont.)

- If after a crash the intention list
  - is marked as valid the transaction was committed before the crash occurred.
    - The write operations on the intention list are performed
    - then the locks are released.
  - is not marked as valid the transaction was not committed before the crash occurred.
    - The locks are released.
    - The transaction might be restarted.

Distributed two-phase commit Protocol

- phase 1 (voting phase):
  - The coordinator sends a *canCommit* request to each of the transaction participants.
  - When a participant receives a *canCommit* request it replies with its vote, *Yes* or *No*, to the coordinator.
    If it answers *No* it aborts its part of the transaction.
    If it answers *Yes* it puts its new data on an intention list on stable storage.
- phase 2 (completion phase):
  - The coordinator collects the votes including its own:
    - If all votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* message to all the participants.
    - Otherwise the coordinator sends a *doAbort* message to all the participants that voted *Yes*.
  - A participant that voted *Yes* is waiting for a *doCommit* or *doAbort* message.
    - If it is a *doCommit* message it marks the intention list as valid and sends a *haveCommited* message to the coordinator.
      Then it copies the data on the intention list to the corresponding storage.
    - If it is a *doAbort* message it removes the intention list.

three-phase commit

- Takes care of the problem that the coordinator might crash.