Lecture 6: Road Map

Material from BHG

- Concurrency Control
- A Model for Database
- Transaction Execution
- Distributed Recovery
- Atomic Commitement
- 2-Phase Commit
- 2PC with DT log
- Alternative Topologies for 2PC
Concurrency Control [BHG]

Concurrency Control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and potentially interfere with each other. It is closely related to Recovery, which is the activity of ensuring that software and hardware failure do not corrupt persistent data. Both problems arise in the design of hardware and operating, real time, communication, and database systems.

The main component of the (abstract) database model is that of transaction: an execution of a program that accesses a shared data. The goal of concurrency control is to guarantee that transactions execute atomically, namely, that no transaction interferes with others, and that a transaction either commits, and then its effects become permanent, or aborts and has no effects.
A Model for Database

A database consists of a set of named data items, each having a value. E.g., suppose from and to are two accounts stored on different machines. Then to transfer $x$ in between accounts, the DBS can execute:

1. $t_1 := \text{Read(from)}$  \ \ \ \ //check $t_1 \geq x$
2. $t_2 := \text{Read(to)}$
3. Write (from, $t_1 - x$)
4. Write (to, $t_2 + x$)

obviously, 1 and 2 can be performed concurrently, and similarly 3 and 4. But, 1 and 2 should be performed before 3 and 4 respectively. Also, \{1,2,3,4\} should be performed as a single atomic instruction, w/o interference of other transactions. Thus, when another transaction accesses from and to, it should either see both as before, or after, the transfer of $x$, but not in the midst (e.g., after 1 and 3 are executed, but not 4.)
Transaction Execution

The scheduler is in charge of deciding which site performs which operations in which order. Its goal is to ensure:

- **Serializability.** The effect of each execution on the DBS is as if transactions are performed *serially*.

- **Recoverability.** Guaranteeing that DBS contains effects of all committed transactions and none of aborted transactions.

- **Cascadelessness.** Ensuring the each transaction reads only values that are written by committed transactions. (Each read operation must delay until all transactions that have previously issued writes commit or abort; this also ensures recoverability). OR

- **Strictness.** Each read and write operation performed only when all transactions that have previously issued writes (to same variables) commit or abort. This guarantees both recoverability and cascadelessness.
Distributed Recovery

Assume no replication of data. There is a unique DM and scheduler in charge of each data item. Each distributed transaction \( T \) has a home site where it is issued. \( T \) submits the operations to the TM at the home site; the TM at the home site then forwards the operations to other sites: each read and write of \( x \) are forwarded to the site where \( x \) is stored, and processes by the DM at that site as if \( T \) were local.

The case of commit/abort is different; the TM at \( T \)'s home site should submit the commit/abort to all sites were \( T \) accessed data, which is considerably more difficult!

In addition, while the TM at the home site may send commit, the scheduler may decide to abort. Moreover, some (but not all) sites involved in the transaction may fail, and the transaction should abort.
Atomic Commitment

Consider a transaction \( T \) involving \( S_1, S_2, \ldots, S_n \). W.l.o.g., assume that \( S_1 \) is the \( T \)’s home site. Before the TM at \( S_1 \) sends \( \text{commit}(T) \) to all sites, it must make sure that scheduler and all DMs are ready (and willing) to \text{commit}.

(A scheduler should be ready to commit if \( T \) satisfies recoverability at the site. The DM should be ready to commit if all values written by \( T \) are on stable storage; it’s not consulted if it only issues reads in \( T \).)

An \textbf{Atomic Commit Protocol} (ACP) is an algorithm for coordinator (home site) and participants at whose termination they all \text{commit} or \text{abort}. Initially, they each have a \text{vote}, \text{yes} or \text{no}.

What are the differences between ACP and BA???
ACP: Requirements

- **AC1.** All processes that decide, reach same decision. (Agreement.)
- **AC2.** Decisions are irreversible.
- **AC3.** Commit is only reached if all processes vote yes.
- **AC4.** Commit is always reached if all processes vote yes and there are no failures.
- **AC5.** If at some point all failures are repaired and no failure occurs for sufficiently long, all processes eventually decide. (Termination.)

**AC5** deals with the issue that under some failure patterns, ACP may cause processes to become blocked. Note that its possible that all processes vote yes and decide abort. Also, processes that votes no can trivially decide abort. A process that votes yes is uncertain until it finds out what the decision is. For example, if it is disconnected from all other processes. (In this case, it is “blocked” until faults are repaired.)
2-Phase Commit (2PC)

Assume no failures!

Round 1: The coordinator send VOTE-REQ

Every process that receives VOTE-REQ prepares answer; if it’s NO, it decides ABORT and stops.

Round 2: All whose vote is YES send it to coordinator.

If coordinator gets (YES) votes from all, and its vote is YES, it decides COMMIT, else it decides ABORT.

Round 3: The coordinator sends its decision to all.

Everyone that receives the coordinator’s decision, decides on the same value (and stops.)

The two phases are the voting phase (rounds 1-2) and the decision phase (round 3, or 2-3 for coordinator.) Processes are uncertain between the end of round 1 and the end of round 3. The coordinator is never uncertain. AC1–AC4 are trivially satisfied. As to AC5 . . .
Towards AC5 – Timeouts

Time-outs are possible at:

- **Round 1:** if a process awaiting VOTE-REQ time-outs, can decide ABORT.

- **Round 2:** if the coordinator times-out while awaiting votes, it can just decide ABORT (and send ABORT to every YES-voter.)

- **Round 3:** if a process \((p)\) times-out while uncertain, it cannot unilaterally decide; it must invoke a TERMINATION PROTOCOL.

**Termination Protocol:** \(p\) can wait until it can communicate with coordinator. This satisfies AC5, but may take unnecessarily long time. Alternatively, \(p\) may try to communicate with \(q\) who knows decision (assuming \(p\) knows id of all participants.)

A process \(q\) getting \(p\)'s request, may either send it decision (if it knows it), or if it hasn’t yet voted, decide ABORT, or just not be able to help \(p\). The worst case now is when \(p\) fails to communicate with any process that is not uncertain. In that case, it can just wait for communication with coordinator to resume.
Towards AC5 – Recovery

If when $p$ recovers it remembers it pre-failure state, and this state is not “uncertain”, then $p$ can recover independently.

Otherwise, $p$ cannot distinguish between a scenario where all others voted YES and the decision is COMMIT, and a scenario where some voted NO (or not voted at all) and the decision is ABORT. Thus, $p$ is exactly like a process that times-out at round 3.

To remember its state (in case of failure), $p$ keeps a DT log in a failure-resistent memory.
Coordinator 2PC with DT log

send VOTE-REQ to all *
write START-2PC on DT log *
await YES/NO from all
on timeout {
    $P_Y :=$ set of processes who voted YES
    write ABORT in DT log; *
    send ABORT to every process in $P_Y$ *
    return }
if all and coordinated voted YES {
    write COMMIT on DT log
    send COMMIT to all }
else {
    $P_Y :=$ set of processes who voted YES
    write ABORT on DT-log *
    send ABORT to every process in $P_Y$ }
return
Regular Participant 2PC with DT log

wait for VOTE-REQ from coordinator
  on timeout { write abort in DT log; return }
if vote is YES then begin
  write YES on DT log
  send YES to coordinator
  wait for decision
    on timeout run termination protocol
  if decision is COMMIT then write COMMIT on DT-log
  else write ABORT on DT-log
end
else begin
  write ABORT on DT-log *
  send NO to coordinator *
end
return
Termination Protocol

Initiator:

\textbf{start}: send \texttt{DEC-REQ} to all

wait for \texttt{decision} from any process

on \texttt{timeout} goto \texttt{start}

if \texttt{decision} is \texttt{COMMIT} then write \texttt{COMMIT} on DT log

else write \texttt{ABORT} on DT log

return

Responder:

wait for \texttt{DEC-REQ} from any process

if \texttt{ABORT} in DT-log then send \texttt{ABORT} to initiator

else if \texttt{COMMIT} in DT-log then send \texttt{COMMIT} to initiator

else if \texttt{YES NOT} on DT-log then

\{

\begin{itemize}
  \item decide \texttt{NO}
  \item write sc \texttt{abort} on DT-log
  \item send \texttt{ABORT} to initiator and coordinator
\end{itemize}

\}

else skip;

return
Comments on 2PC with DT

- A process decides when it writes COMMIT/ABORT on DT log.
- When a process decides, the DM can execute the transaction.
- For recovery, each process inspects its DT log.
- 2PC is resilient to site and communication failures (even to network partitioning.)
- 2PC is subject to blocking: A process may be blocked if it times out while uncertain.
- In the absence of failures, 2PC requires three rounds. In the presence of failure, two additional rounds may be needed (however, blocked processes may take arbitrarily long to invoke the termination protocol.)
- In absence of failures, 2PC requires $3n$ messages. In the presence of failures, additional $\mathcal{O}(n^2)$ may be required.
Alternative Topologies for 2PC

**Decentralized 2PC.** Coordinator broadcasts its vote to all participants (piggybacking VOTE-REQ on its vote.) If coordinator’s vote is **YES**, then everyone that receives it and has a **YES** vote broadcasts its vote. Anybody that either received **NO** or has a **NO** vote, just decides **ABORT**. The others, wait until they receive all votes to decide.

Communication complexity: $O(n^2)$. Time complexity: 2 rounds.

**Linear 2PC.** Processes are arranged in a line,

$$P_1, \ldots, P_n$$

with the coordinator being $P_1$. For $i = 1, \ldots, n - 1$, $P_i$ sends **YES** to $P_{i+1}$ if its vote is **YES** and it had received no **NO** votes, and **NO** otherwise. Processes that receive **NO** decide **abort** (but continue to pass messages.) If $P_n$ receives **YES** and its own vote is **YES**, it starts sending back **commit**.

Communication complexity: $O(2n)$. Time complexity: $2n$ rounds.
Summary of 2PCs

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Messages</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized 2PC</td>
<td>$3n$</td>
<td>3</td>
</tr>
<tr>
<td>Decentralized 2PC</td>
<td>$n^2 + n$</td>
<td>2</td>
</tr>
<tr>
<td>Linear 2PC</td>
<td>$2n$</td>
<td>$2n$</td>
</tr>
</tbody>
</table>

Note that all entries in the table refer to the case of the protocol running without failures. Failures cause the Termination Protocol to be invoked, which causes extra time and messages.

Hybrid approaches are also possible. For example, in Linear 2PC we can let $P_n$ broadcast the decision. The number of messages will not change, but the number of rounds will be reduced to $n + 1$. 