DIS - Transaktioner 1 - Two-General's Problem, two-phase commit protocol, Clocks

Noter

Fejlhåndtering - server går ned

- Transaktioner skal som udgangspunkt være persisteret, altså gemt et sted, så de kan genskabes om nødvendigt.
- Hvis en server ikke svarer, så kan der anvendes en timeout, som opgiver at få svar efter noget tid.
- Ergo: et distribueret system skal som minimum kunne håndtere forsinkelser (latency) og nedbrud (failures).
- Evnen til at håndtere og udbedre fejl er essentiel.

Two-phase Commit Protocol

Generelt set er det problemet med denne her. Typisk problemet når vi taler distribueret transkationer – vi kan ikke altid regne med at alle andre servere er klar.

Billede af en coordinator, der spørger om andre servere er klar, hvor de alle giver en response - alle skal være klar, før commitet forekommer

Hvis alle melder tilbage med grønt lys, så bliver transaktionen gennemført (**commit**).

Hvis ikke alle melder positivt tilbage, så bliver transaktionen afbrudt (**abort/rollback**)

Hvis en deltager har meldt positivt tilbage og ikke hører mere fra lederen, så er deltageren i en **uncertain state**.

Den form for koordinering er et eksempel på **konsensus** i et distribueret system. **Konsensus** dækker over at hvis man har forskellige servere, så skal de også være enige om hvad der står på dataen

A fundamental problem in distributed computing is to achieve overall system reliability in the presence of a number of faulty processes.

This often requires processes to agree on some data value that is needed during computation.

Examples of applications of consensus include whether to commit a transaction to a database, agreeing on the **identity of a leader, state machine replication, and atomic broadcasts.**

Fase 1 - The commit-request phase
(or voting phase), in which a coordinator process attempts to prepare all the transaction's participating processes (named participants, cohorts, or workers) to take the necessary steps for either committing or aborting the transaction and to vote, either "Yes": commit (if the transaction participant's local portion execution has ended properly), or "No": abort (if a problem has been detected with the local portion), and

1. The coordinator sends a query to commit message to all cohorts and waits until it has received a reply from all cohorts.
2. The cohorts execute the transaction up to the point where they will be asked to commit. They each write an entry to their undo log and an entry to their redo log.
3. Each cohort replies with an agreement message (cohort votes Yes to commit), if the cohort's actions succeeded, or an abort message (cohort votes No, not to commit), if the cohort experiences a failure that will make it impossible to commit.

Fase 2 - The commit phase
in which, based on voting of the cohorts, the coordinator decides whether to commit (only if all have voted "Yes") or abort the transaction (otherwise), and notifies the result to all the cohorts. The cohorts then follow with the needed actions (commit or abort) with their local transactional resources (also called recoverable resources; e.g., database data) and their respective portions in the transaction's other output (if applicable).

Success
If the coordinator received an agreement message from all cohorts during the commit-request phase:

1. The coordinator sends a commit message to all the cohorts.
2. Each cohort completes the operation, and releases all the locks and resources held during the transaction.
3. Each cohort sends an acknowledgment to the coordinator.
4. The coordinator completes the transaction when all acknowledgments have been received

Failure
If any cohort votes No during the commit-request phase (or the coordinator's timeout expires):

1. The coordinator sends a rollback message to all the cohorts.
2. Each cohort undoes the transaction using the undo log, and releases the resources and locks held during the transaction.
3. Each cohort sends an acknowledgement to the coordinator.
4. The coordinator undoes the transaction when all acknowledgements have been received.

Two-General's Problem

Der skal være en, der styrer slagets gang. Det skal ikke være den samme altid - turen skal gå på skift. Der er aldrig kun én server der skal stå for alle ting

Det kaldes blandt andet for en **coordinator role**, en **log leader** eller generelt set **log election**

Husk at transaktioner skal gemmes og kunne genskabes - det gøres via en log over hvad der er sket.

For at koordinere har vi skabt det, der hedder en **two-phase commit protocol**. #

Quorum

- Et quorum er et udtryk for et givent flertal, der skal til for at et distribueret system kan overleve nedbrud.
- Et quorum på to tredjedele betyder at et system med alle noder intakte kan overleve at en tredjedel af dem dør på grund af systemfejl eller lignende.
- Et quorum kan samtidig anvendes til at løse konflikter omkring data i systemer, der er **eventually consistent**.

Eventually consistens
Betyder at data bliver konsistense over tid - MobilePay

Clocks
Leslie Lamport

Kausalitetsbegrebet Leslie Lamport
idéen om begivenheder, der er indtruffet før hinanden a --> b

- Han omtaler clocks som en logisk konstruktion, der tæller når en værdi bliver ændret.
  Hvis a --> b, så C(a) < C(b)
- Han bruger begrebet clocks, eller urer – en logisk konstruktion(et værktøj) der tæller op når en værdi bliver ændret
  Hvis A kommer for B, så er tiden af A sket før tiden af B

- Brugen af clocks gør det muligt at opstille en samlet rækkefølge over begivenheder i et distribueret system.
  a c b, eller
  Ci(a) < Cj(b) eller Ci(a) = Cj(b) og Pi ≺ Pj
- Når man har de her clocks, så gør det muligt at opstille en samlet rækkefølge over begivenheder i et DIS.
  Det skriver man generelt set med A (dobbelt pil) til B.

- Når der er opstillet en samlet rækkefølge for begivenheder i et distribueret system, så kan hver node vedligeholde en liste over hvilke ressourcer, der er ledige hvornår - og sørge for at sende forespørgsler på ressourcer når der er behov for det.
- **Det kræver dog at alle noder kender til hinanden i det distribuerede system**

En node(server) kan få adgang til en ressource når den har en forespørgsel på en ressource i sin liste, og den forespørgsel kommer før andre forespørgsler i henhold til den samlede rækkefølge. (Jeg kan godt få adgang til en ressource nu hvis der ikke er andre der har bedt om det)
a c(c = dobbelt pil) b.

Problemet er at en begivenhed kan være vigtig, men hvis den ankommer for sent, så honoreres den reelle rækkefølge ikke.Så måske er det en god idé at anvende **rigtige ure** i stedet for tilnærmede.
Fly eksempel - Sidder i fly, mens en anden sletter

Vector Clocks
I stedet for en værdi, der stiger løbende, så kan der anvendes rigtige ure til at afgøre den reelle rækkefølge af begivenheder i et distribueret system.

Strong Clock Condition
Strong clock condition optræder når forsinkelser ikke ændrer i en faktisk rækkefølge og altså bliver indregnet korrekt.
One of the mysteries of the universe is that it is possible to construct a

system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition.