Deling av data
Transaksjoner

INF5040

Foreleser: Olav Lysne

Introduksjon

- Tjenere kan tilby samtidig aksess fra klienter til de objekter/data tjenerne innkapsler
  - fler-trådede tjenere => behov for synkronisering mellom tråder
- Tjenere kan tilby persistent lagring av objekter/data
  - => behov for feiloppretting etter at en tjener-prosess har feilet
- Klienter har behov for å utføre sekvenser av tjener-operasjoner som en udelelig enhet
  - => atomiske transaksjoner
- En tjeneste kan være basert på flere samarbeidende tjenere
  - => distribuerte transaksjoner

Ønskede egenskaper (til transaksjoner)

- Alt-eller-ingenting
  - *feil-atomisitet*: effekten er atomisk selv om tjeneren feiler
  - *varighet*: etter at en tjener har bekreftet at transaksjonen er vellykket utført, kan resultatet ikke gå tapt, selv om tjeneren skulle feile like etterpå
    - => resultatet må sikres på et permanent medium (disk)
    - => dataverdiene som representerer resultatet må være *gjenopprettbare*

Ønskede egenskaper (til transaksjoner)

- Isolasjon
  - midlertidige resultater av en transaksjon må ikke være synlige for andre transaksjoner
  - => behov for synkronisering (samtidighetskontroll)
  - Seriell utførelsesorden
    - sikrer isolasjon, men seriell utførelse normalt ikke akseptabelt
  - Serialiserbar utførelsesorden ("serial equivalence")
    - resultatet av en samtidig/flettet utførelse av transaksjonene må være det samme som en seriell utførelse av dem
    - algoritmer for samtidighetskontroll sikrer dette

Avslutning av transaksjon

- *Commit punkt* for transaksjonen T
  - alle operasjoner i T som aksesserte tjenerens database, er vellykket utført
  - effekten av operasjonene er permanent lagret (typisk i en logg)
- Vi sier at transaksjonen T er "sikret" (engelsk: commited)
  - tjenesten (eller databasesystemet) har "forpliktet" seg
    - resultatene av T er permanent lagret i databasen

Transaksjonell tjeneste

- Tilbyr aksess til ressurser via transaksjoner
  - samarbeid mellom klient og transaksjonelle tjener
- Operasjoner til transaksjonelle tjenester

OpenTransaction() → TransId
CloseTransaction(TransID) → {commit, abort}
AbortTransaction (TransID) → {}

Samtidighet

- Samtidighet: fletting av operasjoner fra forskjellige transaksjoner
  - bedre system utnyttelse
  - kortere responstid
- fletting av operasjoner kan bl.a. gi problemer som
  - problemet med tapt oppdatering
  - problemet med midlertidig oppdatering ("dirty read")
  - problemet med ukorrekt analyse (summeringsproblemet)

Problemet med tapt oppdatering

x: database element
T1: x = x + 1000
T2: x = x + 50

Flettet utførelse			Verdi i databasen
T1:	read(x)	⇐	500
	x = x +1000		
T2:	read(x)	⇐	500
	x = x + 50		
T1:	write(x)	⇒	1500
T2:	write(x)	⇒	550

Oppdateringen utført av T1 forsvinner

Problemet med midlertidig oppdatering ("dirty read")

X: database element
T1: x = x + 1000
T2: x = x + 50

Flettet utførelse			Verdi i databasen
T1:	read(x)	⇐	500
	x = x +1000		
	write(x)	⇒	1500
T2:	read(x)	⇐	1500
	x = x + 50		
T1:	abort T1		
T2:	write(x)	⇒	1550

T2 baserer oppdateringen på en temporær verdi av x.
Transaksjonen som produserte denne verdien aborteres
=> feil i utførelsen av T2

Problemet med ukorrekt analyse

T1: overfør 100 fra A til B
T2: beregn A + B

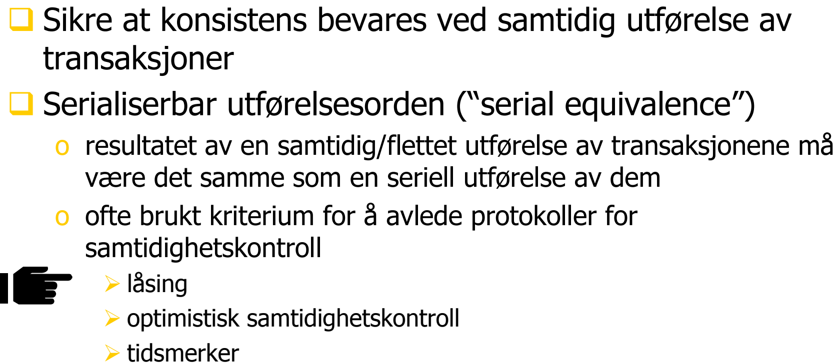
Flettet utførelse (schedule)

T1:	read(A)
	read(B)
	A=A-100
	write(A)
T2:	read(A)
	read(B)
	sum= A + B
T1:	B=B+100
	write(B)

T2 ser en delvis oppdatert database der ny verdi til A leses,
mens gammel verdi til B leses.

Samtidighetskontroll

- Sikre at konsistens bevares ved samtidig utførelse av transaksjoner
- Serialiserbar utførelsesorden (“serial equivalence”)
  - resultatet av en samtidig/flettet utførelse av transaksjonene må være det samme som en seriell utførelse av dem
  - ofte brukt kriterium for å avlede protokoller for samtidighetskontroll



    - låsing
    - optimistisk samtidighetskontroll
    - tidsmerker

Recovery/gjenoppretting

- *Ansvar til tjener*: Sikre at alle operasjoner i en transaksjon blir fullstendig utført og resultatet permanent lagret. Dersom det oppstår en feil under utførelsen, må ingen operasjoner være utført.
- *Recovery*: Fører tjenerens database tilbake til en tilstand der ingenting av transaksjonen er utført
- Utføres når feil oppstår
- En transaksjon som avbrytes ved `Abort` må ikke ha effekt på andre samtidige transaksjoner

Feiloppretttbarhet (recoverability)

- Situasjon som ikke er feiloppretttbar:
  - En transaksjon har utført `commit` etter at den har sett effekten av en transaksjon som seinere utfører `abort` ("dirty read")
- Hindre ikke-feiloppretttbarhet ved "dirty read"
  - forsink utførelsen av `commit` inntil også alle andre transaksjoner denne transaksjonen har sett effekten av, også har utført `commit`.

Problemet med midlertidig oppdatering ("dirty read")

X: database element
T1: x = x + 1000
T2: x = x + 50

Flettet utførelse			Verdi i databasen
T1:	read(x)	⇐	500
	x = x +1000		
	write(x)	⇒	1500
T2:	read(x)	⇐	1500
	x = x + 50		
T2:	write(x)	⇒	1550
T1:	abort T1		

T2 baserer oppdateringen på en temporær verdi av x ("dirty read").
Transaksjonen som produserte denne verdien aborteres
=> feil i utførelsen av T2: **ikke feiloppretttbar!!**
=> T2 må forsinke sin commit inntil T1 har terminert

Problemet med midlertidig oppdatering: cascading aborts

X: database element
T1: x = x + 1000
T2: x = x + 50

Flettet utførelse		Verdi i databasen
T1:	read(x) ⇐	500
	x = x +1000	
	write(x) ⇒	1500
T2:	read(x) ⇐	1500
	x = x + 50	
	write(x) ⇒	1550
T1:	abort	

T2 baserer oppdateringen på en temporær verdi av x og venter med å utføre `commit`. Transaksjonen som produserte denne verdien (T1) aborteres
=> feil i utførelsen av T2

=> T2 må aborteres

Dersom andre transaksjoner har sett T2's temporære verdier
=> disse må også aborteres

Denne situasjonen kalles **cascading aborts**

Hindre **cascading aborts:** Transaksjoner kan kun lese data objekter fra transaksjoner som allerede har utført `commit`.

Prematur `write`

- Skriving av dataverdier til databasen som er basert på effekten av andre transaksjoner som ikke har utført `commit` ennå.
  - P.g.a. måten recovery ofte implementeres på ("before images") kan "prematur write" føre til feil i utførelsen
  - Tiltak: forsink write inntil andre tidligere transaksjoner som har oppdatert samme dataobjekt, har enten utført `commit` eller abortert

Strikt transaksjonsutførelse

- Utførelse av en samling transaksjoner som ikke inneholder "dirty read" eller "prematur write"
  - må generelt forsinke read og write operasjoner for å oppnå dette
  - forsinker inntil alle transaksjoner som tidligere har utført write på de samme dataobjekt har enten utført `commit` eller abortert

Oppsummering:
Ønskede egenskaper til transaksjoner

- *Atomisitet*: Alt eller ingenting utføres
- *Konsistens*: Sikre at dataene benyttes lovlig. Generelt ansett å være programmererens ansvar
- *Isolasjon*: En transaksjon gjør *ikke* sine oppdateringer synlig for andre transaksjoner før den har utført "commit". Sikres av metoden for samtidighetskontroll
- *Varighet:* (engelsk: durability) Når en transaksjon har utført "commit", må dens effekt i databasen aldri tapes pga en seinere feil. Sikres av recovery metoden
- Kalles samlet ACID egenskapene ...

Samtidighetskontroll
basert på låsing

INF 5040

Foreleser: Olav Lysne

Samtidighetskontroll

- Skal sikre isolasjon mellom samtidige transaksjoner
- Grunnlaget for samtidighetskontroll: Serialiserbarhetsteori
- Seriell utførelsesorden
  - sikrer isolasjon, men seriell utførelse normalt ikke akseptabel
- Serialiserbar utførelsesorden
  - resultatet av en samtidig/flettet utførelse av transaksjonene må være "ekvivalent" med en (eller annen) seriell utførelse av dem

Testing av serialiserbarhet

- De fleste serialiserbarhetskontroll metoder tester ikke for serialiserbarhet (for komplekst).
- I stedet for, utvikler *regler* (protokoller) som garanterer at man får serialiserbare flettede utførelser
- Når et sett med regler utvikles, må man *bevise* at disse gir serialiserbare flettede utførelser
- Eksklusiv låsing
  - mest utbredte protokoller for samtidighetskontroll er basert på denne teknikken

Lås-abstraksjon

- Et lås er et *token* som indikerer at en prosess aksesserer en ressurs i en bestemt modus
- Minimale lås-modi: read, write
- Lås brukes til å indikere til andre prosesser den nåværende bruk av ressursen

Prinsipp låsing

- Tjeneren forsøker å sette lås på ethvert dataelement som brukes av operasjonene i en klients transaksjoner
- Dersom dataelementet allerede er låst av en annen klient, må anropet vente inntil dataelementet er låst opp.
- Bruk av låsing kan føre til vranglås (deadlock)
  - et par av transaksjoner har hver låst et dataelement den andre har behov for å aksessere

Eksempel låseprotokoll

Tid ↓

Transaksjon T		Transaksjon U	
Operasjoner	Lås	Operasjoner	Lås
OpenTrans			
saldo := A.Read()	lås A		
A.Write(saldo-4)			
		OpenTrans	
		saldo := C.Read()	lås C
		C.Write(saldo-3)	
saldo := B.Read()	lås B		
		saldo := B.Read()	vent på at T låser opp B
B.Write(saldo+4)		.	
CloseTrans	lås opp A,B	.	
		.	lås B
		B.Write(saldo+3)	
		CloseTrans	lås opp B,C

To-fase låsing (2PL)

- I 2PL
  - en transaksjon kan ikke be om en ny lås etter at den har frigitt en lås
  - gir to faser (vokse- og minke-fasen)
- Mest populære samtidighetskontroll teknikk. Brukt i:
  - RDBMSer (Oracle, Ingres, Sybase, DB/2, etc.)
  - ODBMSer (O2, ObjectStore, Versant, etc.)
  - Transaksjonsmonitorer (CICS, etc.)

Strikt to-fase låsing (2PL)

- Alle lås holdes inntil transaksjonen utfører `commit` eller aborterer
- Sikrer strikt utførelsesorden av transaksjoner
  - hindrer "dirty read" og "premature write"

Lås kompatibilitet

- To typer lås: *read-lock*, *write-lock*
- En "lock manager" innvilger lås
- Innvilgelsen avhenger av kompatibiliteten mellom den ønskede låstype, og de allerede innvilgede lås
- Kompatibilitet defineres i en kompatibilitetsmatrise:

Lås kompatibilitet for et dataelement		*Ønsket lås* Read	Write
	Ingen	OK	OK
Allerede innvilget lås	Read	OK	Vent
	Write	Vent	Vent

Låsekonflikter

- Et forespurt lås kan ikke innvilges hvis en inkompatibel lås allerede er innvilget til en annen samtidig prosess
- Alternative tilnærminger for å handtere konflikter:
  - Tvinge prosessen som ønsker låsen til å vente inntil låsen som forårsaket konflikten er frigjort.
  - Informere prosessen eller tråden om at låsen ikke kan innvilges

Implementasjon av lås

- Lock manager
  - separat modul i tjener-programmet som innvilger lås
- Datastrukturer
  - en tabell av lås. Info per lås:
    - (liste av) trans.id til transaksjonen(e) som har låsen
    - identifikator til dataelementet (f.eks. objektreferanse)
    - låstype
    - condition variabel
      - lock manager *signaliserer* denne når låsen frigis
      - prosessen eller tråden som venter på denne, vil startes
- Operasjoner
  - `lock` (og noen ganger `try_lock`)
  - `unlock`

Vranglås

- 2PL kan føre til at prosesser venter på hverandre for å frigjøre lås
- Vranglås må oppdages av lock-manager
  - "venter-på" graf
    - noder er transaksjoner
    - kanter er "venter-på" relasjoner
    - vranglås => løkke i "venter-på" grafen
- Vrangløs må løses opp i ved å abortere en eller flere av de involverte prosesser
  - et problem her er i hindre "utsultning"
- Dette krever at det utføres "undo" på alle read/write operasjoner som disse prosessene allerede har utført

Låsegranularitet

- 2PL kan anvendes på ressurser av enhver granularitet
- Høy grad av samtidighet med liten låsegranularitet
- Liten granularitet => høyt antall lås
  - Kan medføre signifikant låse-overhead
- Avveining mellom grad av samtidighet og låse-overhead
- Hierarkisk låsing er et kompromiss

Hierarkisk låsing

- Brukes i forbindelse med ressurser som kan modelleres som "containere" for andre, mindre objekter
  - filer (inneholder poster)
  - mengde eller sekvens (av objekter)
- En lås på en foreldrenode har samme effekt som å sette lås på alle nodens barnenoder
- Reduserer antall lås når blandet låsegranularitet er påkrevet

Hierarkisk låsing

- Nye låsemodi
  - intention read (IR)
  - intention write (IW)
- Intention lock
  - settes på sammensatte objekt
  - signaliserer til andre transaksjoner som ønsker å sette en lås på hele det sammensatte objektet at en annen transaksjon allerede har lås på noen av objektene inneholdt i det sammensatte objektet
- Gir mer effektiv forvaltning av lås

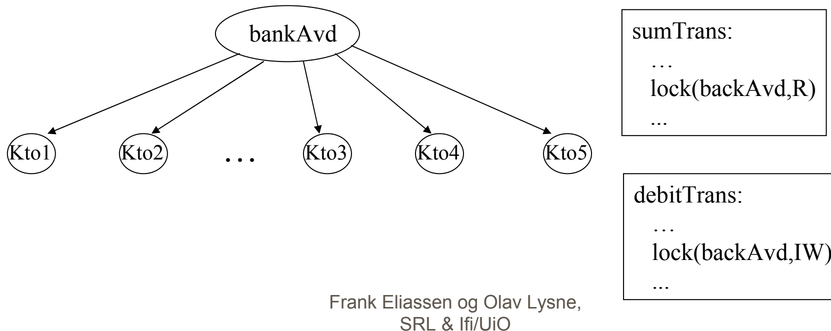
Hierarkisk låsing II

- Låsekompatibilitet

	IR	R	IW	W
IR	OK	OK	OK	Vent
R	OK	OK	Vent	Vent
IW	OK	Vent	OK	Vent
W	Vent	Vent	Vent	Vent

Eksempel hierarkisk låsing

- summeringstransaksjon settes R-lås på bankAvd-objektet
- Debiter-transaksjon prøver å sette IW-lås på samme objekt (og må vente)
- => summeringen vil ikke bli forstyrret & kun to lås satt



Oppsummering

- Samtidighetskontroll koordinerer aksess til delte data og objekter og sikrer isolasjon
- To-fase låsing sikrer konfliktserialiserbarhet
- Strikt to-fase låsing sikrer mot "dirty reads" og "premature writes".
- Hierarkisk låsing effektiviserer implementasjon av låsforvaltning for store mengder objekter.
- CORBA Concurrency Control Service tilbyr både to-fase låsing og hierarkisk låsing for objekter som er implementert utenfor databasesystemer.

Distribuerte transaksjoner

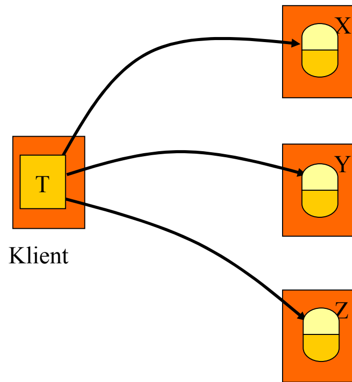
INF5040

Foreleser: Olav Lysne

Distribuert transaksjon

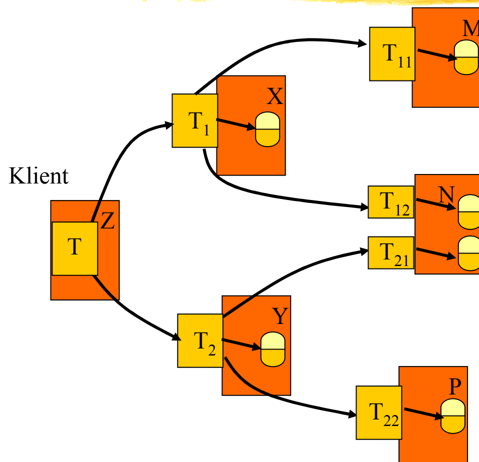
- En klient transaksjon som anroper operasjoner i flere forskjellige tjenere
- To grunnleggende former
  - Enkle distribuerte transaksjoner
  - Nestede distribuerte transaksjoner

Enkel distribuert transaksjon



Karakteristika:
- en tjener av gangen eksekverer
- ingen samtidighet mellom sub-transaksjoner

Nestet distribuert transaksjon



Karakteristika:
- tjener kan anrope operasjoner hos andre tjenere
- hver transaksjon struktureres som en mengde nestede transaksjoner
- sub-transaksjoner på samme nivå kan eksekvere samtidig
- dersom en sub-transaksjon feiler kan en alternativ sub-transaksjon startes i stedet for

Komponent roller

- Distribuerte systemkomponenter som er involvert i en transaksjon kan ha rollen som:
- Transaksjonell klient
- Transaksjonell tjener
- Koordinator

Koordinator

- Koordinator spiller en nøkkelrolle ved *forvaltningen* av transaksjonen
- Koordinator er komponenten som handterer begin/commit/abort transaksjonsanrop
- Koordinator allokerer system-globale entydige transaksjonsidentifikatorer
- Koordinator inkluderer nye tjenere i transaksjonen (`Join` operasjon) og holder oversikt over alle deltakere
- Første tjener som klienten kontakter (ved anrop av `OpenTransaction`), blir koordinator for transaksjonen

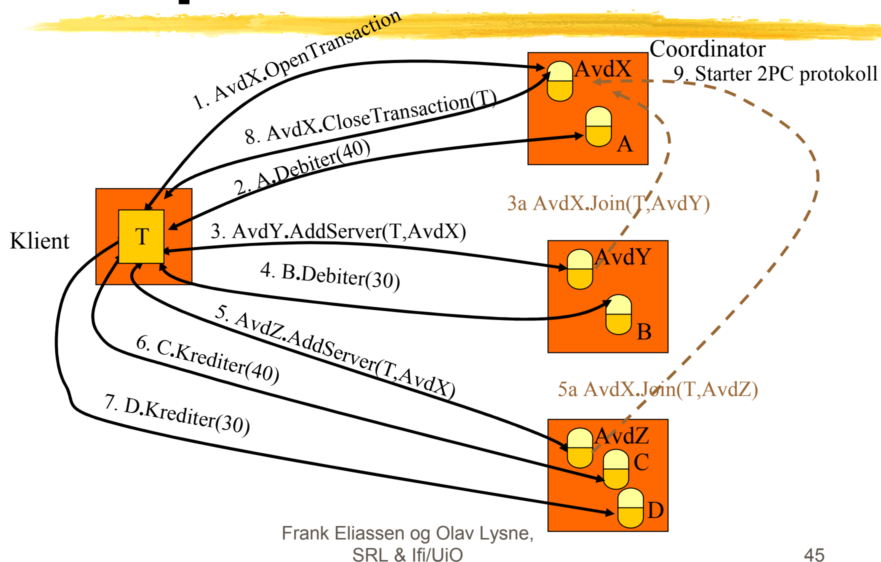
Transaksjonell tjener

- Enhver komponent med en ressurs som aksesseres eller modifiseres under transaksjonell kontroll
- Transaksjonelle tjenere må kjenne sin koordinator
  - via parameter i `AddServer` operasjonen
- Transaksjonelle tjenere registrerer sin deltakelse i transaksjonen hos en koordinator
  - ved anrop av `Join` operasjon i koordinator.
- Transaksjonelle tjenere må implementere en transaksjonsprotokoll (two-phase commit - 2PC)

Transaksjonell klient

- Ser transaksjonen kun gjennom koordinator
  - anroper tjenester i koordinator
    - Open Transaction
    - CloseTransaction
    - AbortTransaction
- Implementasjonen av transaksjonsprotokollen (2PC) er transparent for klienten

Eksempel



2-PC protokoll

- Flere autonome distribuerte tjenere
  - For at transaksjonen skal avslutte med commit, må alle de transaksjonelle tjenerne kunne utføre commit
  - Hvis en eneste tjener ikke kan utføre commit på sine oppdateringer, må alle tjenerne utføre abort
- En-fase protokoll ikke tilstrekkelig
  - tillater ikke en tjener å gjøre unilateral abort
    - som f.eks. ved deadlock oppklaring
- Behov for to faser
  - fase en: stemmegiving
  - fase to: fullførelse

Fase en: stemmegiving

- Koordinator spør alle tjenere om de er i stand til (og villige til) å utføre commit (`CanCommit?(T)` anrop)
- Tjenerne svare:
  - **Ja**: vil utføre commit hvis koordinator ber om det, men tjeneren vet ikke ennå om den faktisk vil utføre commit
    - bestemmes av koordinator
  - **Nei**: tjenerer utfører umiddelbart abort på sine operasjoner
- Herav, tjenere kan unilateralt utføre abort, men kan *ikke* unilateralt utføre commit

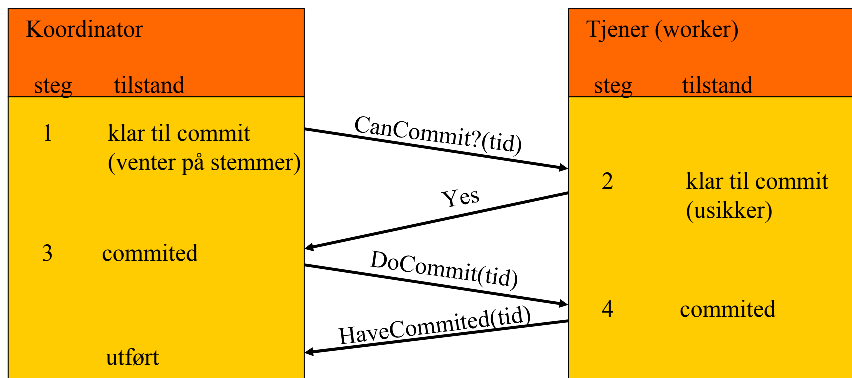
Fase to: fullførelse

- Koordinator samler inn alle stemmer fra tjenerene, inklusive seg selv, og avgjør å utføre
  - commit, dersom alle stemte **Ja**
  - abort, dersom noen stemte **Nei**
- Alle stemmegivere sendes
  - `DoCommit(T)` anrop, dersom avgjørelsen er commit
  - `AbortTransaction(T)` anrop, ellers
- Tjenerene bekrefter `DoCommit(T)` straks de har utført commit
  - tilbakekall ("backcall") `HaveCommited(T)` til koordinator

Tjenerens usikkerhetsperiode

- Periode der en tjener må være i stand til å utføre commit, men har ennå ikke gjort det
- Vanligvis kort periode
  - tiden koordinator bruker for å motta og prosessere stemmer
- Feilsituasjoner kan forlenge denne perioden, som kan forårsake problemer
- En tjener kan anrope `GetDecision(T)` hos koordinator for å vite utfallet av stemmegivingen
  - f.eks. etter timeout hos tjeneren

2PC protokollen



Recovery i 2PC

- Feiling før start av 2PC resulterer i `AbortTransaction`
- Feiling av koordinator *før* `DoCommit` anropene resulterer i abort
- Etter dette punktet, vil koordinator gjenta alle anropene på `DoCommit` ved restart
- Dersom en tjener feiler
  - før stemmegiving, aborterer den ved restart
  - *etter* stemmgivingen, anroper den `GetDecision(tid)` ved restart
  - *etter* å ha utført commit, anroper den `HaveCommitted(tid)` igjen ved restart

Oppsummering

- To-fase commit
  - fase en: stemmegiving
  - fase to: fullføring
- CORBA Transaction Service
  - implementerer to-fase commit
  - krever ressurser som er "transaksjonsoppmerksom"
- Transaksjoner og EJB
  - programmatic & declarative transactions
  - container-støtte for distribuerte transaksjoner
    - basert på CORBA OTS og X/Open XA protokoll
  - EJB container/server implementerer Java Transaction API (JTA) og Java Transaction Service (JTS)
- Utvidede transaksjonsmodeller & OASIS BTP
  - B2B transaksjoner (recovery ved kompenserende aksjoner)