## 6. Indirect communication

En fortsättning på interprocesskommunikation och remote invocation är indirekt kommunikation. Indirekt kommunikation syftar på kommunikationen mellan två enheter som saknar direkt koppling till varandra. Där det finns en eller flera mellanliggande enheter som vidarebefordrar eller processar meddelandet påväg till dess faktiska destination. Att sändaren inte behöver känna till mottagaren kan ske i både tid och rum.

*Space uncoupling* innebär att sändare är fysiskt sett inte på samma plats som mottagaren. T.ex. en server som skickar ut events som läses av den som är intresserad. *Time uncoupling* syftar däremot på att processerna som pratar med varandra inte nödvändigtvis “lever” samtidigt. En process producerar (producer), sen avslutas, och därefter startar en process som konsumerar detta (consumer). Indirekt kommunikation kan ha en eller båda dess egenskaper. Har en kommunikation både time coupling och space coupling kan den användas för t.ex. remote invocation eller interprocesskommunikation. Kommunikation som har egenskapen time uncoupling kan jämföras med vanlig *asynkron kommunikation*, men skillnaden är att asynkron kommunikation fortfarande kräver att de kommunicerande parterna har överlappande livstids under tiden de kommunicerar.

Gruppkommunikation är en annan aspekt av indirekt kommunikation, och liknar IP-multicast. Däremot syftar gruppkommunikation på ett högre abstraktionslager som kan baseras på multicast, men också liknande nätverk. Gruppkommunikation lägger till mycket värde i form av grupphantering, feldetektering och pålitlighet (bland annat ordningen på paketen som kommer fram till flera olika ställen) genom koordination. Ett praktiskt exempel på gruppkommunikation är *JGroups* som är en implementation skriven i Java.

Ett vanligt system som använder indirekt kommunikation är ett *publish-subscribe-system* (även *distributed event-based system* i det sammanhanget). Detta går ut på att låta en enhet producera någon form av meddelande (*publisher*) som andra enheter är intresserade av (*subscribers*). Den som producerar är inte nödvändigtvis intresserad av vem som är intresserad av meddelandet, utan agerar endast efter att någon enhet eventuellt är intresserad och konsumerar det. Ett publish-subscribe-system finns ofta bland annat i finansiella system för t.ex. förändring av en akties värde. En naiv implementation av ett sådant system kan vara att placera den producerande enheten central och låta alla konsumenter abonnera på denna. Denna lösning skulle dock sakna skalbarhet då den centrala enheten kan överbelastas. En bättre lösning är att låta ett nätverk av producent finnas, som delar på lasten och agerar som en stor producerare. För att bygga på ännu längre finns också peer-to-peer lösningar där alla noder i nätverket agerar producent och konsument.

Ytterligare en del i indirekt kommunikation är *distribuerade meddelandeköer*. Medan gruppkommunikation var en till flera-kommunikation, kan meddelandeköer ses som en abstraktion för en till en-tjänst, då tekniken går ut på att en sändare placerar meddelanden i en kö, och en enskild mottagare plockar ut dem. Distribuerade meddelande köer uppnår därmed både space- och time uncoupling. Den programmatiska sida av meddelande köer kan övergripligt implementeras på tre sätt, blockerande mottagning, icke-blockerande mottagning och genom att meddela att något kan läsas från kön. Vi implementation av ett sådant kösystem kan en naiv och småskalig lösning vara att ha en centraliserad köhanterare som processer pratar med för att placera, polla eller plocka ut ett meddelande. Nackdelen med ett sådant system är samma som för ett publish-subscribe-system i att det skalar dåligt. En bättre lösning är att låta ett distribuerat nät av enheter tillsammans skapa en köhanterare. Ett praktiskt exempel av detta är IBMs WebSphere MQ som definierar *queue managers* som de som hanterar meddelandeköerna.

## 16. Transactions and concurrency control

En transaktionen är en serie händelser som sker i en grupp, och där målet är att alla eller inga av dem utförs - odelbar. En transaktion ska behålla systemet i ett giltigt stadie efter den är utförd. Detta innebär att servern som utförs en transaktion måste kunna återställa stadiet av de påverkade objekten om något skulle krascha under tiden transaktionen utförs. Några händelser i en transaktion för en enkelt banksystem skulle kunna vara:

*Transaction begin  
deposit 200*

*withdraw 100*

*deposit 100*

*Transaction end*

Fortsättningsvis är det viktigt att allt eller inget-principen även följs om servern själv skulle krascha - *failure atomicity*. En transaktion ska också stödja *isolation*, vilket syftar på att dess mellanliggande stadie (stadiet under tiden den utförs) inte är synligt för andra processer - den ska utföras som alla instruktioner inom den är en enda stor atomär operation. *ACID* (*atomicity, consistency, isolation* och *durability*) syftar på de egenskaper en transaktion ska uppnå.

*Concurrency control* syftar på hur flertrådig användning av samma dataset hanteras. Transaktioner används för att lösa en hel del problem som uppstår i samband med att flera olika processer försöker komma åt och eventuellt skriva till samma mängd data. Två vanliga problem är *lost update problem* och *inconsistent retrieval*. Det första syftar på när en uppdatering av data går förlorad för att en annan process hämtat stadiet innan den första processen hinna skriva tillbaka.   
Exempel:   
***A*** *hämtar data*

***B*** *hämtar data* ***B*** *modifierar data*

***B*** *skriver data*

***A*** *modifierar data*

***A*** *skriver data*

I detta fall går Bs ändring av data förlorad, då B hämtade efter A, och skrev före.

Ett concurrency control-system behöver också ha strategier för att hantera *granulariteten* av låsbara objekt (beskrivs nedan) för att inte skapa flaskhalsar när många objekt låses bara för en transaktion behöver skriva till några få.   
  
Det andra problemet syftar på hur en process hämtar data under tiden den modifieras av en annan process. Om transaktioner efterfrågar eller modifierar samma data behöver de köras seriellt (en efter den andra) för att motverka ovan beskrivna fel.

*Recoverability* syftar på hur en server kan återhämta stadiet efter något gick fel, eller bara ifall den uttryckligen avbröts, under utförandet av en transaktion. Med detta följer några vanliga problem, t.ex. *dirty read*. Detta inträffar då både As och Bs transaktioner modifierar samma data relaterat till dess nuvarande värde. Om A utförs först, och sedan vill avbryta transaktionen (abort transaction), efter att Bs transaktion utförs, kommer B egentligen ha läst från ett värde som inte fanns.

Exempel:

***A*** *läser data****A*** *modifierar data till current + 10*

***B*** *läser data*

***B*** *modifierar data till current + 10*

***A*** *avbryter*  
  
I detta fall är Bs “current” av data inte längre giltigt, en *dirty read*, då den berodde på As modifiering som nu avbrytits.

För att implementera transaktioner seriellt ifall de utför operationer på samma resurs används *lås*. Detta förhindrar två processer att modifiera samma resurs samtidigt genom att endast låta en process köras i taget. Ett *låsschema* kan implementeras på olika sätt som beskriver hur ett lås ska tas och återlämnas. *Strikt tvåfaslåsning* bygger på tvåfaslåsning där lås tas i första fasen (*growing phase*) och återlämnas i den andra fasen (*shrinking phase*). Strikt tvåfaslåsning definierar att alla lås som tas i första fasen hålls ända tills transaktionen avbryts eller avslutas.

Ett vanligt problem för lås är s.k. *deadlocks*, vilket syftar på två eller fler processer som väntar på varandra att blir färdiga. Exempel:

***A*** *låser objekt* ***O1*** *för att modifiera den****B*** *låser objekt* ***O2*** *för att modifiera den*

***A*** *behöver tillgång till* ***O2*** *men måste vänta på att* ***B*** *återlämnar låset*

***B*** *behöver tillgång till* ***O1*** *men måste vänta på* ***A*** *återlämnar låset*

(A och B väntar nu på varandra)

En naiv lösning av deadlocks är att låsa samtliga objekt när någon modifiering ska göras. Fastän detta löser deadlocks är det opraktiskt att låsa all tillgång då de försämrar prestandan genom att reducera samtidighet. Andra metoder för att istället detektera deadlocks är att timeout eller genom undersökning av en *wait-for-graf*. Detta är en graf som lagrar vilka andra transaktioner en transaktion väntar på.

En s.k. *optimistisk approach* eller *optimistisk concurrency control* är en lösning som ämnar att lösa nackdelar med tvåfaslåsning, såsom risken för deadlocks. Tekniken baseras på faktumet att i de i flesta applikationer är chansen låg att två transaktioner samtidigt försöker komma åt samma objekt. Därmed bör de sällan uppkommande felen hanteras med någon sorts konflikthanterare i stunden. Fördelen över tvåfaslåsning är bland annat bättre utnyttjande av samtliga transaktioner, då väntan på lås blir mindre. Det kräver dock validering innan skrivning sker, så att eventuellt överlappande transaktioner kan upptäckas.

## 17. Distributed transactions

En påbyggnad på vanliga transaktioner är *distribuerade transaktioner*, dvs. transaktioner som utför operationer på distribuerade objekt i ett distribuerat system. Detta kommer med många utmaningar, bland annat hur atomicitet implementeras (flera enheter behöver då meddelas om att transaktionen t.ex. avbryts. Distribuerade transaktioner behöver också lösa problemet med samtidighet (eng. concurrency) över flera olika enheter. Varje server hanterar sin egen samtidighet, men måste koordinera med andra servrar för att kunna serialisera transaktioner globalt i systemet.

En *koordinator* kan används för att koordinera förfrågade transaktioner. En klient skickar en förfrågan om att börja en transaktion och tilldelas ett *transaction ID* som är unikt för det systemet. En koordinator ska dock inte hjälpa till mer än att just koordinera - det är upp till varje server (*participant*) som äger objektet att hålla koll på samtliga objekt involverade i transaktionen. Det som är fördel med ett sådant system är att koordinatorn kan få en överblick på vilka servrar som är involverade i transaktionen.

Precis som i transaktioner på en server innebär atomicitet för transaktioner att allt eller inget utförs. För att upprätthålla denna princip för distribuerade system finns olika protokoll, t.ex. *one-phase atomic commit protocol*. Detta går ut på att låta koordinatorn skicka ut information om att t.ex. transaktionen ska avbrytas till alla dess deltagare, och fortsätta göra så tills att koordinatorn fått bekräftelse att de mottagit informationen. Ett annat protokoll är *two-phase commit protocol* som möjliggör för deltagare i transaktionen att rösta om den ska utföras eller avbrytas, och där målet är att nå konsensus.

Deadlocks var ett problem som kunde lösas med timeouts i transaktioner på en server. Men i distribuerade transaktioner är det opraktiskt att försöka komma upp med godtyckliga timeouts på de olika servrarna i systemet. Istället används en större wait-for-graf som kan detektera deadlocks i hela systemet. Denna kan genereras av de lokala wait-for-graferna och hanteras av en central server.

## Reflektion

Transaktioner var någonting som berördes under projektet i andra året, databas-transaktioner. Det var småskaligt och lätt att förstå vad som hände och varför. När det kommer till distribuerade transaktioner verkar det vara mer intressant i hur de samspelas i ett stort system, då det verkar vara en stor utmaning.