6. Indirekt kommunikation kan användas för att lösa många problem inom datalogi. Fördelarna med det är att det inte finns någon direkt koppling mellan sändare och mottagare. Exempelvis behöver inte en sändare veta om ID:t på mottagaren (Space uncoupling), samt att sändaren och mottagaren inte behöver existera under samma tid för att kommunicera (Time uncoupling). Det här används när det finns en stor risk att användare kopplas till och från ofta. En nackdel med indirekt kommunikation är sämre prestanda.

Indirekt kommunikation behöver inte alltid vara både indirekt i tid och utrymme. Många är bara indirekta på ett av de här sätten. Exempelvis är IP-multicast indirekt i utrymme men direkt i tid. Det indirekta i multicast härstammar från hur sändaren skickar meddelanden. Det görs mot hela multicast-gruppen och inte enskilda användare. Den behöver alltså inte ens veta om vilka medlemmar som finns i gruppen. Alla användare måste dock existera i samma tid för att kunna få meddelandet. Liknande fall förekommer för publish-subscribe system.

Det finns en viktig skillnad mellan asynkron kommunikation och indirekt kommunikation i tid. För en asynkron kommunikation behöver båda parterna existera samtidigt, men de kan fortsätta göra annat under tiden att meddelande skickas/svar väntas på.

Gruppkommunikation är som beskrivet ovan en stor del av den indirekta kommunikationens användningsområde. Gruppkommunikation används för att:

- Skicka ut information till ett stort antal klienter
- Stödja applikationer för samarbete där varje event från en användare måste propageras till alla andra användare, exempelvis onlinespel
- Stödja användandet av flera servrar som sändare för replikerad data
- Stödja användandet av lastbalancerare.

Programmering för gruppkommunikation kan vara så enkelt som följande: group.send(message). Meddelandet skickas då grupperat tills en router behöver “splittra” på den för att meddelandet ska åt olika håll. En grupp kan antingen vara öppen eller stängd, alltså kan alla skicka meddelanden i gruppen, eller så kan bara vissa. Överlappande eller icke-överlappande grupper beskriver om gruppens medlemmar kan vara med i flera grupper eller bara en.

Implementeringen av gruppkommunikation kan vara ganska avancerad. Pålitlighet levereras genom att ha krav för integritet och valideringar. Det är alltså säkert att klienten får meddelandet, och att meddelandet ser ut som det gjorde för sändaren.

Publish subscribe system är den mest använda metoden av indirekt kommunikation. Det kan användas för att:

- Publicera information, exempelvis priser på aktiemarknaden
- Live feeds som livestreaming, RSS, mm.
- Många fler användningsområden som behöver strömmar av information.

Ett tradingföretag skulle kunna använda sig av publish subscribe på följande sätt:

En server på företaget tar konstant emot uppdateringar från externa källor (ex. Börsen). De ses som ett event, och skickas sedan inom publish-subscribe systemet för varje aktie som uppdateringen berör. Sedan får aktiehandlaren som är intresserad av just den aktien får ett event med uppdaterad info.

Bra egenskaper för ett publish-subrsibe system är att den ska vara heterogen, alltså fungera med många olika typer av enheter. Ex. IoT, mobila enheter, hemmaservrar/IoT-hubbar. De ska även jobba asynchront, alltså inte sitta och vänta på nästa event, utan snarare jobba på och ta emot det när det väl dyker upp. Det går att särskilja de olika eventen med hjälp av olika “intressen”. De kan kallas för channel-based, topic-based, content-based, type-based.

Det finns olika sätt att sända event i ett publish subscribe system.

- Flooding: Alla event skickas till alla noder. Noden själv sorterar igenom de och hittar vilka den har abbonerat på.
- Filtering: Event skickas bara vidare när det finns en abbonerad användare i den riktningen
- Rendezvous: Ett nätverk av filtering noder

Message queue system är ännu ett slag av indirekta kommunikationssystem. Det kan användas t.ex. för att skicka transaktioner till en databas. En egenskap med meddelandena är att de är persistenta, alltså försvinner de inte tills de behandlats. I meddelandeköer brukar det även vara möjligt att hantera meddelandena med enklare operationer innan de ens absorberats av mottagaren.

En tuple space är en form av delad plats där alla användare kan skicka in eller hämta ut data.

16 Transaktioner används för att se till så att en server förblir i ett konsekvent tillstånd även efter en krash. Detta implementeras genom att antingen gå igenom med hela transaktionen eller inget av den.
Det går att implementera synkroniserad överföring av data utan transaktioner. Man använder då lås och atomiska operationer för att se till att allt sker eller inget. Fel som kan uppstå med transaktioner är (failure model): problem med utskrift till fil, serverkraschar, försenade meddelanden.

Egenskaper av en transaktion är att den är fri från störningar av andra operationer, samt att all-or-nothing implementeras. Exempelvis vid en banköverföring måste både ”-” och ”+” operationen ske i samma transaktion för att inte pengar ska försvinna eller motsatsen.

All or nothing kan defineras med två egenskaper: *failure atomicity* (atomicitet bevaras även vid krash) och *durability* (alla förändringar som skett vid en transaktion ska sparas på ett permanent sätt, ex. hårddisk). En minnesregel för egenskaper av transaktioner är ACID: **A**tomicity, **C**onsistency, **I**solation, **D**urability.

Concurrency control är viktig för att se till så transaktionerna blir rätt. Det finns många ”standard” problem som kan uppstå om man inte verkställer bra concurrency control. Exempelvis Lost update problem och Inconsistent retrievals problem. Ett begrepp som kan användas för att beskriva om transaktioner kan köras paralellt är serial equivalence. För att uppnå det behöver de båda kunna köras samtidigt och få ut samma svar som om de kördes en efter en. Operationer som kan skapa konflikter behöver köras i rätt ordning. Ex. read+write, write+write. Dock är read+read ingen konfliktskapande operation. Detta kan uppnås genom att sätta lås på de operationer som skapar konflikter. Servern behöver kunna backa om en transaktion ”abortas”. Problem som Dirty Reads kan då uppstå.

En nästlad transaktion är en som innehåller ”barn”, alltså sub-transaktioner som alla behöver utföras fullständigt innan förälder-transaktionen kan räknas som färdig. Om förälder-transaktionen ”abortas” måste även alla barn göra det också.

För att se till att delad data inte läses och skrivs fel används lås. De leder till att en resurs bara finns tillgänglig för en i taget. Problemet med lås är att Deadlock kan uppstå. Det är därför viktigt att låshanteraren placerar operationerna i rätt ordning för att detta inte ska uppstå. De kan därför märka när en deadlock håller på att uppstå, stoppa en av transaktionerna och köra den senare. Ifall något verkligen fastnar finns timeouts på låsen där transaktionerna som fastnat abortas.

Allt ovan är dock byggt på att försöka se till att problem inte ska uppstå, även i situationer där inget hade hänt. En annan syn på det hela är optimistisk concurrency control där man istället försöker lösa de få felen som uppstår. Transaktionen sker därför i tre faser: Arbetsfasen, där allt skrivs och uppdateras; valideringsfasen där konflikter och fel hittas; uppdateringsfasen där uppdateringarna blir permanenta. Prestanda sparas med hjälp av den här metoden för att de transaktioner som bara läser information inte behöver verifieras.

För att se till att operationerna sker i rätt ordning använder de sig av Timestamps. De innehåller inte riktig tid utan är i stället bara en pseudo-tid som indikerar vilka transaktioner som ska köras innan.

Optimistiska metoden fungerar bäst i situationer där få krockar uppstår, annars kan det bli mycket arbete för servern att fixa problemet. Exempelvis en databas som mest ska läsas från kan köras med optimistiska metoden. Exempel på tjänster som använder optimistiska metoden är Dropbox, Google workspace, Wikipedia.

Pessimistiska metoden fungerar bäst när många krockar kan uppstå, men har därför också lägre prestanda generellt sett. Pessimistiska metoden används till exempel för MySQL, mm.

17 Distribuerade transaktioner är transaktioner som behöver skicka/hämta information från flera olika servrar. Likt vanliga transaktioner går det att göra distribuerade nästlade transaktioner. Delar av transaktionen kan då anropa noder som i sin tur anropar andra noder. För att se till att alla delar av transaktionen utförs utses en koordinator, som brukar vara noden som klienten skickat transaktionen till.

För att säkerställa atomicitet med distribuerade transaktioner brukar *Atomic commit protocols* användas. *Commit*:en kan ske i en eller två faser. I en enfas-commit är det bara en nod som bestämmer om ändringen sker eller inte. En tvåfas-commit använder istället ett röstningssystem. Alla deltagare måste rösta om ändringen ska ske eller inte. Om en enda röstar att den inte ska ske får inte ändringen gå igenom. För att få rösta för commit behöver noden ha permanent lagring redo för den uppdateringen. Den kan inte ändra sig om något skulle gå dåligt efter att commit röstats. Den som håller i röstningen är koordinatorn. En nackdel med two-phase-commit är att prestandan inte är särskilt bra. Det finns protokoll som förbättrar prestandan men då skickas också fler meddelanden. Proceduren för nästlade distribuerade transaktioner med two-phase commit fungerar ungefär på samma sätt. Det är viktigt att se till så hela trädet med transaktioner *abort*:as om en av de har gjort det.

Ett annat sätt att lösa two-phase-commit är att rösta genom det hierarkiska trädet av transaktioner. Varje subtransaktion blir alltså en egen koordinator för de transaktionerna under sig. När alla de under röstat rapporterar subtransaktionen uppåt vad resultatet blev. Något som är gemensamt för båda commit-protokollen är att timeouts måste användas. Kommer inget svar inom en viss tid måste transaktionen avbrytas. Lås på distribuerade transaktioner fungerar som lokala lås, förutom att låset bara kan släppas när hela transaktionen (och alla sub- och sub-sub transaktioner, etc.) är commitad. Timestamps blir något mer komplicerat då klockorna på servrarna sällan är helt synkroniserade. Lösningen är att försöka synka de men också ha felhantering med ex. delays eller offsets.

Deadlocks i distribuerade transaktioner kan också uppstå. Vanliga algoritmer för att lösa deadlocks försöker hitta cirklar i vilka processer som väntar på vad. Ex. A väntar på B, B väntar på C, C väntar på A. Den distribuerade varianten använder sig istället av något som liknar P2P system, där varje nod försöker känna till grannarnas wait-for, etc.

För att uppfylla kravet om *durability* är det viktigt att de resurser som uppdaterats efter en transaktion förblir det, även vid en systemkrasch. Därför kan en recovery file användas för att spara de senast uppdaterade resurserna. Ett annat sätt att lösa problemet av en eventuell krasch är loggning. Istället för att spara absoluta värden som i recovery file sparas istället en kedja med uppdateringar med pekare till vad som uppdaterats, förmodligen i en form av recovery file. För att återskapa resursen kan då loggen följas och samma uppdateringar utföras på datan. Det är viktigt att ha med information om pågående rösningar så att inget ”antas” vara commitat om det kanske inte blev det innan kraschen.

Reflektion

Kul att få en bredare synvinkel på det som togs upp i databaskursen. Det verkar som att mycket av det som implementeras av databassystemen också kan användas för andra former av distribuerade resurser. Det verkar även som en stor optimering att köra multicast istället för unicast med relativt liten skillnad kodmässigt, det hade nog varit något att ha med i 1:ans projektarbete.