## 1. Characterization of distributed systems

Ett distribuerade system kan övergripligt definieras som datorer (hårdvara och mjukvara) som över nätverk kommunicerar med varandra via meddelanden för att tillsammans koordinera givna uppgifter.

Fördelarna med ett sådant system är bland annat att flera datorer kan exekvera samtidigt i ett stort samarbete och att detta sker på ett smidigt sätt, då de inte kräver en gemensam synkronseringsklocka (det sker i form av meddelanden istället). Fortsättningsvis erbjuder ett distribuerat system isolerade fel då datorerna arbetar för sig.

Distribuerade system används i flera av dagens industrier för att utnyttja ovanstående fördelar. Ett uppenbart exempel är en sökmotor (Web search) som balanserar lasten över flera olika datorer i ett nätverk. Noterbart för sökmotor är just hur Google har behövt designa ett väldigt sofistikerat distribuerat system för att hantera all dess inkommande trafik (i form av Google-sökningar), och som anses som det mest komplexa distribuerade systemet.

Finansiella system är också en stor del i ämnet, då dessa distribuerade system har annorlunda krav. Ett finansiellt distribuerat system kräver bland annat bättre pålitlighet (exempelvis för börsen och aktievärden).

Distribuerade system kan även ses och användas som en tjänst för användare. Flera företag levererar redan idag tjänster på ett sätt som elektricitet och vatten erbjuds, exempelvis virtualisering av operativsystem.  
  
Fortsättningsvis förs idén av en distribuerad tjänst vidare till en vision av att lagring av data, körning av applikationer osv. sker via internet. Detta kallas då *molnet* och idén är att dessa ska kunna ersätta lokala enheter, både dess lagring och applikationer, och därmed kunna komma åt en stor mängd tjänster oavsett enhet. Övergripligt är *molnet* ett distribuerade system, datorkluster.

Ett distribuerat system ska som designmål vara skalbart, vilket innebär att om med tiden fler förfrågningar sker per tidsenhet ska systemet kunna förbättras för att klara av den nya lasten. För att uppnå skalbarhet behöver man tänka på att kontrollera kostnaden på de fysiska resurserna och prestanda förluster. Ett distribuerat system ska kunna expanderas om det så behövs - t.ex. om en filserver kan hantera 20 användare, ska två stycken kunna hantera 40 användare. Fortsättningsvis ska det inte introducerar flaskhalsar när systemet växer, därmed ska ett system sträva efter att undvika centraliserade system (likt det gamla DNS-systemet). Sammanfattningsvis är en ideal skalbarhet den som inte kräver någon mjukvaru förändring.

Felhantering i ett distribuerat system måste också hanteras varsamt då fel kan uppstå individuellt på datorer i systemet - om det t.ex. inträffar fel på en dator som tvingar stopp kommer alla andra datorer fortfarande köra. Felen måste alltså ens *detekteras* för att kunna hanteras. Däremot i ett stort system skulle det vara opraktiskt att försöka hantera alla fel som uppkommer, därmed bör klienter designas på ett sätt som tolererar en viss mängd fel.

## 2. System models

För att designa och beskriva ett distribuerat system kan man använda sig av olika typer av modeller. Dessa modeller existerar i olika nivåer för att abstrahera på olika sätt.

*En fysisk modell* kan anses som den mest konkreta modelltypen, då den beskriver kompositionen av det fysiska datorerna som ingår i systemet. En fysisk modell ska även svara på vilka nätverksteknologier som ingår i det distribuerade systemet. Historisk sedan början av distribuerade system har kravet på den fysiska modellen förändrats mycket. Tidiga distribuerade system ingick det i nästan helhet statiska datorer, men i de kontemporära system ingår det en del mobila enheter, som kräver ytterligare stöd för bland annat uppteckning.

*Arkitekturmodeller* beskriver lite mer abstrakt (eller översiktligt) hur ett system är uppbyggt. De behandlar frågor som *vilken kommunikation paradigm ska användas?* (interprocess, remote osv.) och *vilka entiteter ska kommunicera i systemet?* (objekt, komponenter osv.) Arkitekturmodeller är därmed mer beskrivande och kan ses som en ritning av ett distribuerat system. Fortsättningsvis ses även roller och ansvar över i en arkitekturmodell och då besvaras frågor såsom *vem är sändare och vem är mottagare?* (om klient-server används). Sist ska även en arkitekturmodell besvara hur de olika delarna i dess “ritning” kopplas till den fysiska modellen. Vilka/vilken fysisk dator ska användas för en server samt vilka fysiska datorer ska ansvara för en tjänst osv.

Fortsättningsvis behandlar en *grundläggande modell* mer specifika aspekter i ett distribuerat system. Istället för att beskriva datorer, översikt och roller, besvarar en grundläggande modell explicita och specifika delar såsom:

* Interaktion  
  Tekniska aspekter hur processer pratar med varandra - ska det ske asynkront? vilken bandbredd sker kommunikation över?
* Felhantering  
  Hur fel förväntas kunna inträffa och var. Hur fel ska hanteras när de inträffar. Olika typer av fel måste hanteras på specifika sätt. T.ex. *omission failure* (som innebär att en process inte kunde slutföra sin uppgift) resulterar ofta i att processen kraschar. Den typen av fel bör då hanteras genom att detektera de med hjälp av timeouts (en förbestämd maxtid för hur länge en process får ta att utföra en uppgift).
* Säkerhetsrisker  
  Hur ska kommunikation mellan objekt ska ske för att det ska vara säkert - vilka rättigheter en sändare behöver ha för att få ingå i en konversation med eller starta en process hos en specifik mottagare. Detta är särskilt viktigt för distribuerade system där det lagras personlig data såsom e-post eller bankuppgifter. Det finns många typer av säkerhetsrisker, t.ex. kan en mottagare använda *spoofing* som innebär att denna uppger sig att vara någon den inte är (en server måste kunna verifiera sändaren av ett meddelande). För att hantera alla risker kan ett distribuerat system implementera krypterad kommunikation, autentisering och säkra kommunikationskanaler.

## 4. Interprocess communications

Kommunikation inom ett distribuerat system kan delas in i lager. En typ av lager är middleware-lagrena vars uppgift är att hantera problemen med heterogena system. Ett av middleware-lagerna är interprocess communication. Det lagret beskriver hur processer kommunicerar med varandra i form av meddelanden.

Till grunden är kommunikation via meddelanden en serie av *send* och *receive* - en process skickar ett meddelande och en annan process tar emot meddelanden på destinationen. Detta kan ske såväl synkront som asynkront, vilket innebär att sändaren kan välja att vänta på att meddelandet kommer fram (synkront, blocking) eller fortsätta med andra saker under tiden meddelandet skickas (asynkront). Varje meddelande som skickar har information om de olika adresserna nödvändiga för den kommunikationen - inkluderar internetadress, portnummer osv. Som start- och slutpunkt för två kommunicerande processer används en *sockel*. Det är en abstraktion för en sorts dörr där meddelanden kan skickas och tas emot. Vidare används antingen TCP eller UDP beroende på om pålitlighet eller prestanda prioriteras.

Data som skickas är i många fall i form av en datastruktur representerad i ett visst programmeringsspråk. När denna data väl ska skickas behöver den konverteras till en lång rad bitar (*marshalling*) (språk med pekare och referenser behöver serialisera den faktiska data). Den serialiserade data byggs sedan upp igen av mottagare (*unmarshalling*). Till detta följer en mängd utmaningar och saker att tänka på, t.ex. använda little endian eller big endian, hur flyttal representeras.

I Java kan objekt serialiseras genom att implementera gränssnittet Serializable (Java RMI). Detta gör att objektet kan säkert skickas till (Java plattar ut objektet till en rad bitar) eller anropas hos en annan process som bygger upp det hos sig. Java hanterar även då ifall ett objekt innehåller referenser till andra objekt, isåfall serialiseras dem också (taget att även den klassen implementerar Serializable).

I XML (Extensible Markup Language) används en form av textkodning när t.ex. ett objekt ska delas med en annan process. En XML-textfil innehåller informationen uppdelad i markup-taggar som enkelt kan läsas av hos mottagaren. XML-formatet liknar HTML-formatet som används för web. Fördelen med ett format som är textbaserat är att det underlättar felsökning när något går fel.

När kommunikation ska ske till många användare från en avsändare är parvis kommunikation ineffektivt. Genom att dela in mottagare i en grupp och kommunicera en-till-flera kan hastigheten öka. Detta kallas multicast och kan ske både från server till klienter och klient till servrar. Just klient till flera servrar kan ses som en redundans för klienten när någon form av förfrågning görs från en klient och flera servrar bär på informationen.

## 5. Remote invocation

För att processer ska kunna kommunicera i ett distribuerat system används RMI (Remote Method Invocation). RMI kan ske i formen *request-reply* som innebär att en process skickar en förfrågning till en annan process, och mottagaren ger tillbaka en respons som reflekterar hur det gick. Request-reply kan vara både synkron (vilket innebär att avsändare väntar på svar innan något annat görs, blocking) och asynkron beroende på vilket typ av förfrågning som görs.

Request-reply protokollet baseras på tre primitiva kommunikationsoperationer; *doOperation, getRequest* och *sendReply*, och kan själv matcha en förfrågan från *doOperation* med ett svar. Request-reply fungerar likt annan internettrafik på så sätt att TCP eller UDP används (med samma för- och nackdelar) samt att timeouts kan används på en förfrågning. Protokollet är också designat för att undvika onödigt jobb - protokollet specificerar att meddelanden som skickas ska kunna identifieras. Detta kan användas av mottagare för att förkasta förfrågningsduplikat.

RMI är nära relaterat till RPC (Remote Procedure Call) men har anpassats till en värld av objekt, distribuerade objekt. En metod som anropas kan alltså potentiellt existera i en annan process. Vidare tillåter RMI att parametrar till en metod inte behöver skickas som värde, utan också även som referens - något som är användbart när objekt är stora och komplexa. RMI kopplas till *distributed object model* som är en förbättring på objektmodellen för att göra den mer passande för distribuerade objekt. I denna modell finns *remote object references* som syftar på en referens till ett klass instans i en annan process och *remote interfaces* som syftar på en uppsättning av metoder som går att utföra på det givna distribuerade objektet.

Distribuerad skräphantering fungerar liknande som det gör för ett språk som själv implementerar det, t.ex. Java. Det som skiljer en vanlig skräphantering för ett programmeringsspråk och distribuerad skräphantering är att uppsamlaren måste ta hänsyn till objektreferenser i andra processer. För att uppnå detta används den lokala uppsamlaren i samarbete med den distribuerade uppsamlaren. När distribuerade objekt ska samlas upp är det viktigt att ta hänsyn till *race conditions* med anropen till referensräknaren - en lösning på detta i Java är att placera en temporär referens i listan med aktiva objektreferenser.

Fortsättningsvis hanteras felhantering för distribuerade objekt likt den för lokala objekt. I Java finns t.ex. *RemoteException* som används som exception-bas när ett distribuerat objekt behöver kasta ett exception.