1. Ett distribuerat system är ett system där delar av hårdvaran eller mjukvaran befinner sig på olika datorer och kommunicerar med meddelanden över ett nätverk. För att det ska kallas för ett distribuerat system finns enligt författaren några krav. Systemet måste stödja exekveringen av flera program samtidigt på de olika datorerna som bygger upp systemet. Det ska inte finnas ett behov av en global klocka som synkroniserar tid över datorerna i systemet. De ska i stället förlita sig helt och hållet på meddelanden som ankommer. Sista kravet är att fel i program och datorer i systemet ska kunna ske oberoende med resten, alltså att om en dator är nere ska resten av systemet fortsätta fungera.

Alla stora tjänster på nätet är uppbyggda med distribuerade system. Sökningar på webben är ett bra exempel på vad distribuerade system används till då de har en extremt stor mängd information att läsa och skriva till. Deras funktioner skulle helt enkelt inte vara möjliga på den skalan de finns i utan distribuerade system. Andra användningsområden är ehandel, betalningstjänster, sociala medier, onlinespel, storskalade medicintekniska produkter, utbildning, logistik, forskning, mm.

Distribuerade system håller på att förändras på grund av en del trender:

- Bättre täckning med internetåtkomst med mobila enheter och wifi, mm.
- Fler typer av enheter används, som mobiler, laptops, inbyggda system som vitvaror, fordon, etc. Dessa nya typer av enheter har generellt sett en högre grad av mobilitet vilket innebär att en och samma enhet ofta används på olika ställen, på olika sätt.
- Distribuerade multimediakällor som videoströmning och nedladdning via olika tjänster blir mer populärt. De kräver hög ”quality of service” då tjänsterna ofta blir meningslösa om de inte har tillräcklig kvalitet.
- Software-as-a-service och platform-as-a-service blir vanligare. Infrastruktur ses mer och mer som en tjänst och nya sätt att utveckla för dessa virtuella och skalbara plattformar utvecklas.

Delade resurser och hur de ska spridas över flera datorer är centralt inom distribuerade system. Man använder sig därför av de olika webb-protokollen för att interagera mellan servers.

En stor svårighet med distribuerade system är att de ska vara plattformsoberoende. Tjänsten skall alltså kunna fungera på klienter med olika nätverk, hårdvaror, operativsystem, programmeringsspråk, och separata implementationer. Hur öppen mot andra utvecklare tjänsten ska vara är ett till problem som behöver tas upp, till exempel om andra ska få ansluta sig mot tjänsten, och vilka krav som ställs på de då. Säkerheten är också central i ett väl fungerande system. Det ska alltså inte gå att stoppa data som håller på att skickas mellan servrar och läsa innehållet. För både säkerhet och QOS är det viktigt att tjänsten alltid är tillgänglig. Skalbarheten är alltså viktig så att tjänsten lätt ska kunna uppgraderas vid ökat antal användare, samt kunna temporärt spinna upp fler instanser vid en spik i användning. Felhantering ska ske på ett bra sätt, ex. loggning, och det ska alltid finnas backups om ett system skulle gå ner. Samtidighet måste räknas med vid utveckling så lås och relevanta säkerhetsåtgärder används.

2. Modeller används för att representera fysiska delen av system, hur arkitekturen sätter ihop de samt hur själva systemet/tjänsten ska fungera. Det fysiska lagret skiljer sig väldigt mycket, speciellt nu när många mobila enheter och inbyggda system nått marknaden. De fysiska servrarna har också gått över till mer virtualiserade och teoretiska ”pooler” med infrastruktur.

Modellen för systemets arkitektur kan delas upp i fyra delar.

- Vad och vilka ska kommunicera med systemet
- Hur ska de kommunicera (på vilket sätt)
- Vad för roller de olika delarna av systemet ska ha
- Vart de olika delarna ska ligga fysiskt

De olika sätten kommunikation kan ske skulle kunna vara kommunikation direkt mellan processerna, exempelvis mellan sockets. Det kan även vara ”remote invocation”, alltså att man ”kallar” en metod från en annan dator via interfaces och lite tricks i koden. Det går även att kommunicera mindre direkt med request/reply som till exempel HTTP.

Andra indirekta metoder är ”publish-subscribe”, där en källa information släpps (publish) och användare kan abonnera sig på informationen för att få ändringar. Gruppvis kommunikation används ofta om det är flera användare som ska ha exakt samma information samtidigt.

Client-server modellen är den allra vanligaste och oftast enklaste sättet att implementera distribuerade system. Modellen är byggd på att det finns en eller flera servrar som innehar data, och en eller flera klienter som vill få tag på serverns data. Den används av alla vanliga webbsidor med request/response modellen.

Peer-to-peer är väldigt lik client-server modellen men istället för att ha en server används klienternas datorer som servrar. Man kan därför avlasta tjänstens hårdvara och lägga mer av kraven på användarna. Ex. BitTorrent.

Ett problem med dessa modeller uppstår när man vill öka antalet servrar, för att t.ex. öka antalet möjliga användare. Då finns det proxy servers som agerar som ett mellanlager för att kunna sprida ut användarna över de olika servrarna samt ”cacha” delar av informationen som återkommer ofta och därmed spara bandbredd.

Ett återkommande mönster inom distribuerade system är ”layering”, där ett s.k. middleware används för att förenkla heterogeniteten mellan de olika typer av datorer. I det mönstret brukar man även dela upp tjänsten i tre lager: presentation (interagerar med användaren), logik (bearbetar informationen), data (sparar och hämtar data). AJAX eller liknande protokoll används ofta för att hålla informationen mellan presentation och logik lagren uppdaterad, och därmed skapa mer av en applet än en hemsida. Här skiljer sig även ”thin client” och ”thick client” där thin clients innebär att bearbetningen sker på en server istället för användarens dator. Ex. Google Docs vs Microsoft Word.

Modeller kan även skapas för att modellera hela systemet. Man bör då ha hänsyn för faktorerna: interaktion, felhantering, säkerhet vid skapandet av modellen.

4. Kommunikation mellan processer är en central del i ett fungerande distribuerat system. Kommunikationen sker oftast i form av meddelanden, där meddelandet kan skickas eller tas emot. Det kan antingen vara synkront (blockerar allt, väntar på meddelandet) eller asynkront (fortsätter jobba och tar emot meddelandet när det kommer). Meddelanden skickas från en IP-adress och en port till en eller flera IP+port.

Abstraktionen för detta är sockets. Sockets finns oftast i antingen UDP eller TCP beroende på om prestanda eller pålitlighet föredras.

Ett problem som kan uppstå vid överföring av data med hjälp av dessa protokoll är hur informationen ska vara strukturerad. Antingen måste ett protokoll etableras i förväg, eller så behöver innehållet konverteras över till rätt format vid ankomst. Exempel på ett sådant protokoll är XML. RMI och RPC kan också förenkla det här steget genom att abstrahera det till ”vanlig” kod.

XML, till skillnad från serialiserade objekt/RMI, RPC är kodat på ett sätt som gör det möjligt för människor att läsa och ändra innehållet i meddelanden.

Det finns vissa fall där parvis kommunikation inte är smidigast. Multicast finns då för att skapa ”grupper” av användare som alla ska få samma information. Man kan då gå med i gruppen, lämna gruppen, eller förbli i den och ta emot data som kommer från sändaren. Utöver att vara smidigare för utvecklaren innebär multicast att bandbredd sparas på de olika länkarna mellan sändaren och mottagarna. Paketen delas bara upp när det behövs för att de ska åka rätt. Ett exempel på användningsområde för multicast är videosamtal/ip-telefon med flera personer.

Ett annat sätt att optimera kommunikation över nätverket är virtualiserade nätverk. Exempelvis Skype som har skapat ett nätverk av användare och mappat ut vissa datorer som är starka nog att agera som en enklare server.

5. Request/reply fungerar likt http, med en viss procedur vid varje meddelande:

- Klient skickar förfrågan
- Servern accepterar förfrågan
- Klient skickar meddelande
- Server skickar svar

Det finns även flera olika typer av operation som kan skickas så servern kan ta emot och svara på meddelanden smidigare samt sätta begränsningar för säkerhetens skull.

Remote procedure call (RPC) är en grundläggande version av remote invocation. Det var RPC som på 80-talet satte igång tankarna av distribuerade system. RPCs utformning kan dock ses som ganska klumpig. Till exempel förlitar den sig extremt mycket på interface. Det innebär att utvecklingen blir ganska smidig, men det kan vara oklart om något ändras på andra sidan.

RMI programmeringens lösning på det tidigare nämnda problemet med rå överföring via sockets. Likt RPC fungerar det genom att ”länka ihop” kodbaserna på vardera sida. RMI är dock en nyare lösning på samma problem. En nackdel med RMI är att data skickas i form av objekt. Objekten som skickas behöver då serialiseras och deserialiseras, samt infogas i programmet på mottagarsidan.

Java RMI är Javas version av det här protokollet. Med den kan man t.ex. skicka request/reply meddelanden. De fungerar genom att koden på båda sidorna kallar vissa funktioner som skickar, tar emot eller frågar den andra datorn att köra en viss metod. För att köra Java RMI finns ett par saker som programmet behöver ha:

- Server och klientprogram ska finnas definierade, samt att båda programmen måste ”initialiseras” innan de kan börja kalla metoder över RMI. Görs lämpligt i main eller liknande funktion.
- Factory-metoder behövs för att skapa upp de objekt som dyker upp från andra datorn över RMI. Det måste finnas en factory-metod för alla RMI-delade objekt.
- En *binder* som används för att hålla koll på namnen av objekten som skickas över
- Flertrådig hantering av inkommande RMI invocations på serversidan. Oftast används 1 tråd per invocation för att säkerställa att inget annat låser sig under tiden att till exempel filer laddas eller data bearbetas.
- En *activator* som håller koll på vilken server och URL på den för varje objekt som finns i klientens dator.
- Lokal lagring för objekten, antingen i minnet kopplat med en form av garbage collector eller en mer permanent lagring. För att spara på minne i ett system med permanent lagring kan objekten ”passiveras” och därmed flyttas till hårddisken.

I Java RMI finns även inbyggd garbage collection. Genom att hålla en lista på alla referenser på objekt som finns aktiva på klientdatorn kan servern kasta de objekten som inte längre används. Det finns därför ”gömda” metoderna *addRef* och *removeRef* för att möjliggöra dessa funktioner.