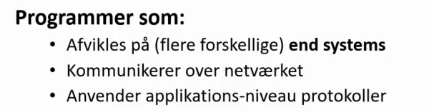
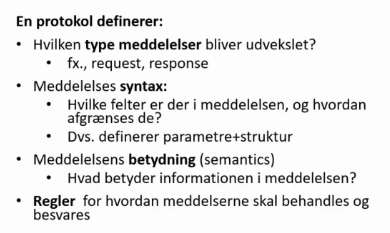
**Applikationslagsprotokoller**



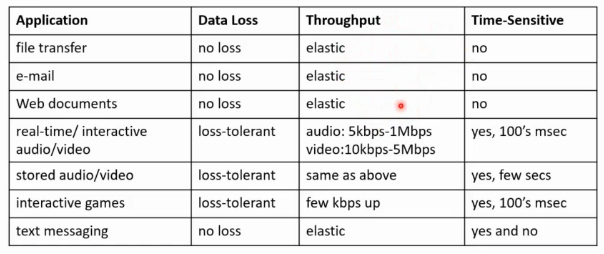
Eksempler kan være: Discord, GitHub, File transfer protokol(FTP), dropbox...osv.

En *applicationslagsprotokol* understøtter applikationer (*HTTP, SMTP, DNS* osv...)



Komponenter som snakker sammen kaldes *processer* (et kørende program på en end-host). Der findes roller til processer som f.eks *client/server modellen*. En process kan også have tildelt flere roller. En process kan både modtage og sende data hvilket styres gennem *sockets.* (lidt ligesom en dør til transportlaget). Processer navngives ved *IP-adressen* og et *portnummer*. Servers ”lytter” på en port den har bestemt. **Nogle portnumre er ”velkendte” eller ”reserveret” porte, f.eks HTTP som bruger port 80, HTTPS bruger 443 og DNS bruger 53. Disse velkendte porte kan ikke bruges i webapplikationer.**

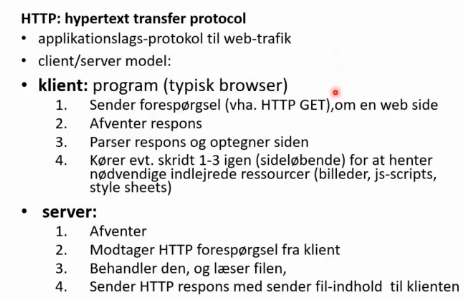
Nogle applikationer bruger kommunikationskrav:



Der er også sikkerhedskrav som: Kryptering, data integritet (der er ikke ændret på dataen) pg authenication (modparten er den som den giver sig ud for at være).

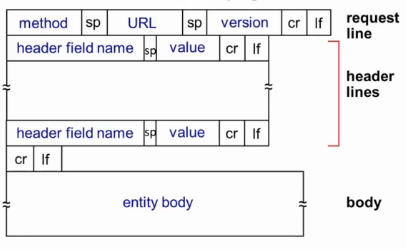
**HTTP**

Eksempel på et simpelt HTTP scenarie:



det behøves ikke være en klient, bare en eller andet end-host. Serveren kan også sende filer f.eks. et database look-up.

En *HTTP-forespørsel* består af en header(som sendes som tekst)



(Dette er et request, et repons er samme format)

Method = GET, POST, PUT osv (metode navnet)

URL = URL som metoden anvendes på

Version = Protokol versionen

**HTTP reponstid**

Her bruger vi *Round Trip Time (RTT)*, den tid det tager at sende en pakke og får et svar tilbage. HTTP repons tid kræver 1 RTT for at opsætte TCP og 1 til fil transmissionstiden.

*HTTP Responstid = 2\*RTT + File transmission time*

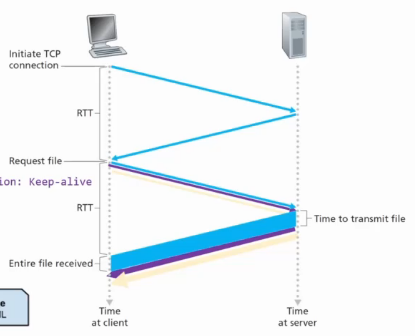
Hvis man skal indlæse en hjemmeside som har et HTML dokument, Stylesheet, scriptfil og et jpg, ville dette tage *4 \* HTTP responstid = 4 \* (2\* RTT) = >8 \* RTT.* **Der kræver et HTTP respons for hver objekt der skal hentes.** Advanceret hjemmesider kan sende parallel forbindelser (f.eks 1 til at hente HTML og en til at hente linked objekter (billeder, scripts osv).

Dette er meget hurtigere for klienten, men der er krævende for serveren.

**Persistent HTTP**

Her sparer man en RTT per HTTP respons, fordi serveren holder forbindelsen åben (altså der skal ikke oprettes en ny TCP forbindelse). Dette sker med Connection: Keep-alive. HTTP/1.1 bruger ”pipelining” hvor forespørgelserne kan blive sendt i rap, uden at vente på svar (samlebåndspricip).

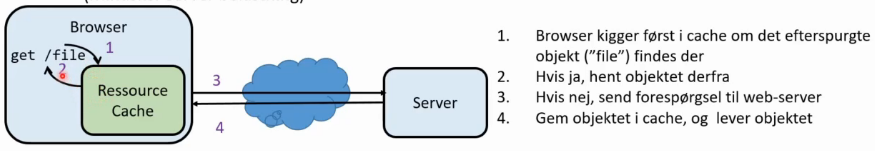
Eksemplet fra før ville være *HTTP repons pipelining = 2 RTT + File transmission + server overhead*



Dette løser problemet. Det er hurtigere for klienten og mindre krævene for serveren fordi den kun skal opretholde en TCP forbindelse per klient. HTTP/2 gør dette hurtigere ved at sende indhold til klienten uden forespørgsel.

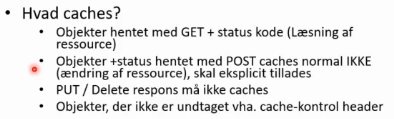
**HTTP Caching**

Her catcher man en kopi af forespurgte objekter (ligesom i en CPU med L1-L3 caches). Web-caching gør svartiden hurtigere og mindske server-belastning.



*HTTP cache control* styrer hvor længe kopien er gyldig og hvorvidt det kan caches. F.eks bliver styling ikke ændret meget, hvilket gør det godt at cache, men noget som dynamiske database opslag ændres ”tit” og er mindre godt at cache. (bliver styret af HTTP med ”*max-age=x*” hvor x er antal sekunder hvor klienten må bruge en cache, hvis det cached objekt er for gammelt skal man opdatere cachen). HTTP beskriver også hvornår objektet er blevet ændret, hvis objektet er blevet opdateret, så skal cachen også opdateres (*last-modified: dato & if-modified-since*).

Cache control kan bestemme om objekter er *no cache* (indholdet skal valideres af serveren), *no-store*(objektet må ikke caches, der skal ske en ny forespørgsel hver gang) og *private/public* cache (privat må kun gemmes i klientens cache)

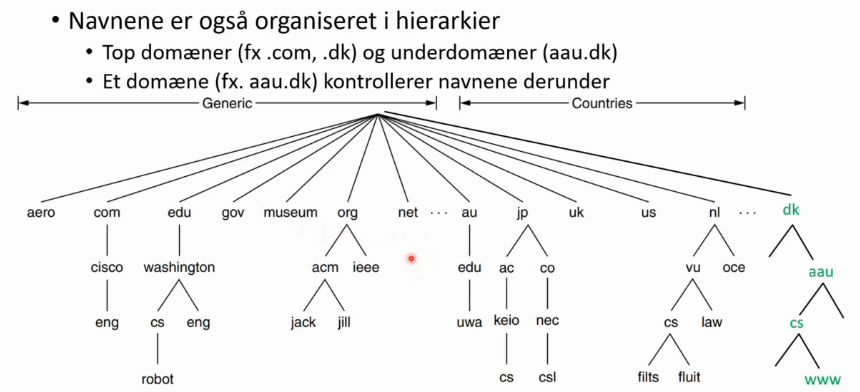


*HTTP cache control* benytter også betinget forespørgsler med *ETAGS*. Dette bruges til at validere objektet i cachen (altså er det cached objekt den samme som det objekt serveren har). Dette implementeres som en *hash-streng.* Man kan også bruge en *HTTP Proxy* som bruges som en fælles cache mellem alle clienter (placeret hos en ISP). HTTP proxies har også andre brug som: Load balancing af en server farm, indholds filter, anonymizer, compression/cryptering. **Dette virker ikke med HTTPS pga. signaturen ikke normalt gemmes.**

**Domain name system (DNS)**

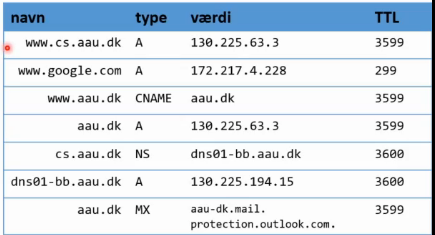
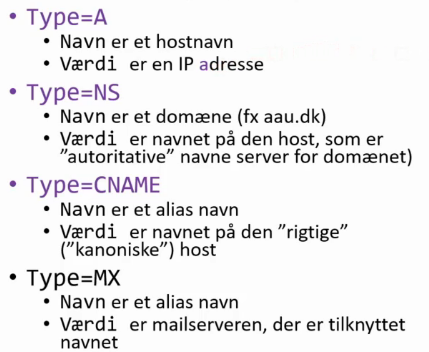
**Formålet med DNS**

DNS’ formål er at oversætte hostnavne til nummeriske værdier (fra www.whatever.dk til en IP). DNS er en *Distribueret Hierarkisk Database*, man uddeler belastning ud på mange servere i et hierarki. Dette undgår *single-point-failure* som sørger for at hvis en maskine går ned, så går hele webservicen ikke ned. DNS bruger *UDP* på port 53 (*velkendt port for DNS*). Hierarkier er opsat på følgende måde:



DNS benytter sig også af dataposer, som består af 4 ting: *Navn, type, værdi og TTL* (time to live).

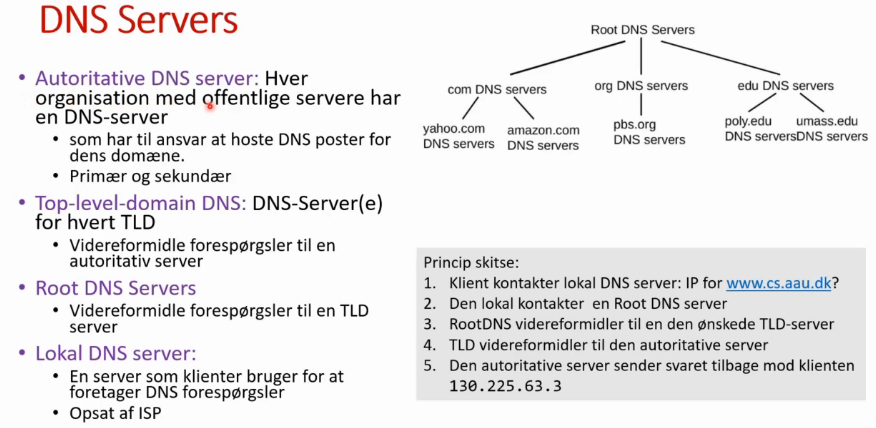
Typerne er beskrevet således:

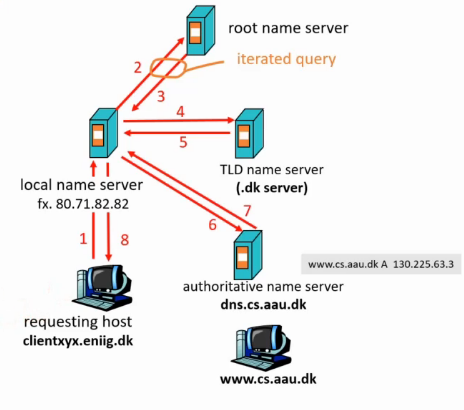
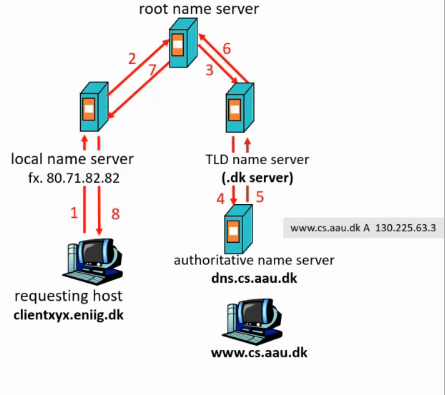


Her kan man se www.cs.aau.dk har *type A*, hvilket betyder at værdien er en *IP-adresse*, time to live er et antal sekunder. Mere om TTL i DNS-caching.

**DNS server forespørgelser**

DNS servere er også opbygget i hierakier, her er der tale om 4 forskellige ”niveauer”:

Dette procedure kald i ”princip skitse” er defineret både *rekursivt og iterativt*:



**DNS caching**

Når en name server for kendskab til en *DNS-post* gemmer den det i en *lokal cache* så der ikke skal bruges flere server kald på den samme forespørgsel. *DNS-caching* fjerner meget belastning fra serverne, men hvis hosten af hjemmesideren ændre IP-adresse er de cached adresser ikke gyldige. Her bruges Time to live (TTL) som er det antal sekunder som DNS-serveren holder på informationen i sin cache (en ”bedst før dato”).

*Root servers* består af **13 globale servers navngivet a-m.root-server.net**. Root servers er en logisk server som dækker over mange tusind fysiske servers. Der er mange fysiske servers fordi de altid bliver kaldt af *local-name-serveren*. *Flaskehalse* og *single-point-failures* bliver undgået med de mange fysiske servere og *DNS-caching mekanismen*.

**Opdatering af DNS**

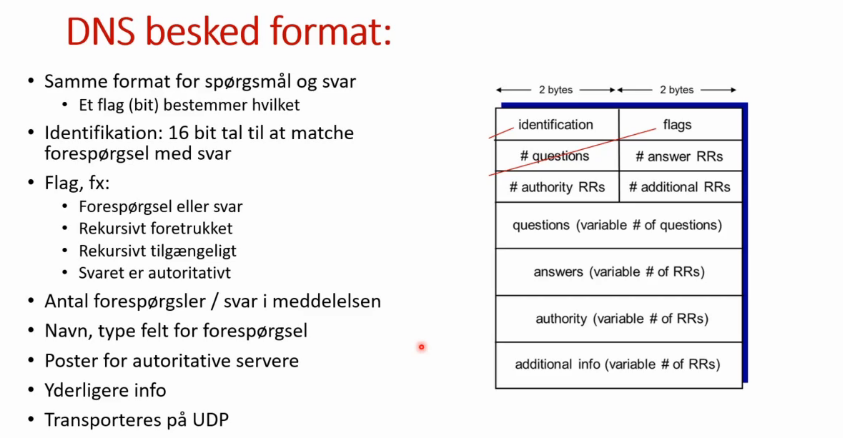
Hvis man skal registrer et domæne skal det købe det (hvis det er ledigt) og få det registreret hos en registrator (dk-hostmaster.dk). Her skal der også være minimums information som: ønsket domæne, name server og IP-adressen. Dette information bliver sat ind i et Top Level Domain server for .dk.

Opdatering af poster styres gennem et management interface hos en system admin eller en dynamisk-DNS.

**DNS sikkerhed**

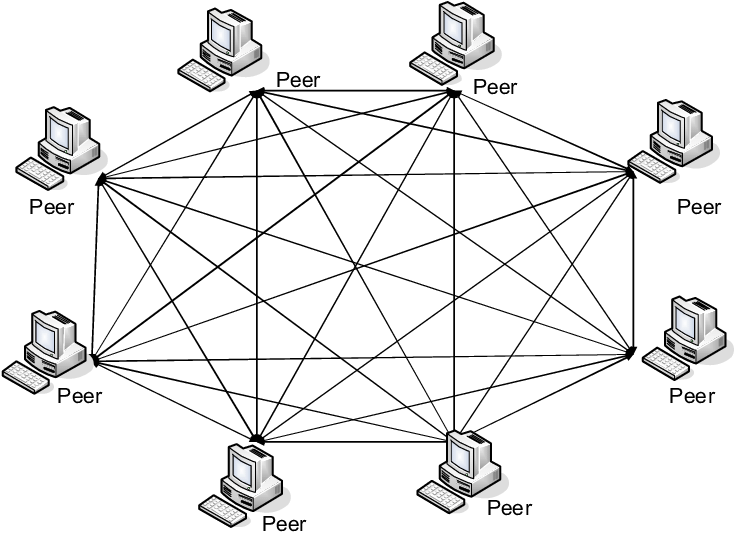
DNS kører over UDP som ukrypteret og er derfor \*lidt\* sårbar overfor diverse angreb. Spoofing, cacheforgiftning (lægge falske oplysninger i DNS-cachen og omdiregere brugere til et forkert website) desuden skal dette ske udenom caching-systemet. Dette er teknisk svært at udføre, men ikke umuligt. Dog er et DOS-angreb ikke muligt da der eksitere mange fysiske globale servers af root servers og på lavere niveauer er adresserne typisk cachet. Et DDOS angreb er kun muligt når der er snakke om store angreb. Der findes også DNSSEC (DNS Secure) hvor serverne ikke tager imod forespørgelser fra hvem som helst.

**DNS værktøjer**



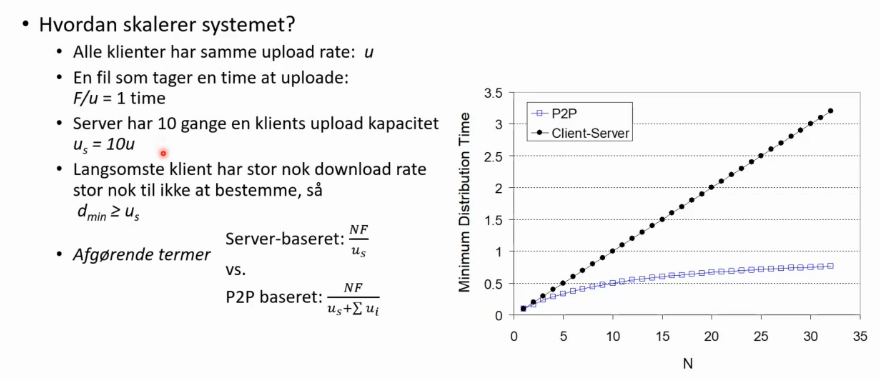
**Peer to peer (P2P)**

Peer to peer er et alternativ til client-server modellen som benytter sig af end-host forbindelser. De er et system af slutbruger (dog nogle gange servers). End-hosts bidrager med resourcer som lager, beregningskraft eller netværkskapacitet og det virker derfor som et ”noget-for-noget princip” system. P2P systemer ejes ikke af nogen, så derfor er det meget svært at lukke systemet ned. På grund af der ikke er en centraliseret server er oppetiden også bedre end client/server modellen. Eksempler på P2P-systemer kan være: BitTorrent/uTorrent, Skype og TOR. Typiske anvendelser er fil-distribution, men kan også anvendes til andre ting.



P2P bliver brugt i ulovlige sammenhæng (som Napster, Piratebay og mange andre server indexe). Disse indexe er ulovlige, fordi de henviser til gratis filer med ophavsret.

P2P systemer er bedre til fildistribution end client/server modellen, fordi der ikke er mulighed for en flaskehals fra serveren. Hvis serveren skal uddele filer til mange end-hosts fordi serveren skal uploade mange kopier da clienterne skal downloade en hver. P2P skal kun uploade en fil til en end-host, som andre end-hosts kan downloade, hertil hjælper alle peers med upload-kapacitet som gør systemet bedre på stører skala (med nogle undtagelser).



Problemet med peer to peer opstår ved at end-hosts er dynamiske og melder sig til og fra konstant. Klienter vil hellere nyde end yde, hvilket ikke giver en ”fair” belastning over alle klienterne. Det er heller ikke sikkert at klienterne har en fast IP (hvis de bruger et NAT’ed netværk)