

Martin Damhus



GRUNDBOG I

DATALOGI



www.datalogi.systime.dk

SYSTIME >

Grundbog i datalogi

© 2008 Martin Damhus og Systime A/S

Kopiering og anden gengivelse af dette værk eller dele deraf er kun tilladt efter reglerne i gældende lov om ophavsret, eller inden for rammerne af en aftale med COPY-DAN.
Al anden udnyttelse forudsætter en skriftlig aftale med forlaget.

Ekstern redaktion: Elisabeth Husum

Omslag:

Udarbejdet af Systime A/S

Omslagsillustrationer:

iStock Photo, © Andrey Solovyev og Eric Hood

Sat med Minion pro 10/13

Sats: Anne Marie Kaad

1. e-bogudgave 2008

ISBN-13: 978-87-616-2281-5

(ISBN-10: 87-616-2281-8)

Bogens website: www.datalogi.systime.dk

Trykt udgave:

Trykt hos:

Clemenstrykkeriet A/S, Århus

Printed in Denmark 2008

1. udgave, 1. opdag

ISBN 13: 978-87-616-2042-2

(ISBN 10: 87-616-2042-4)



Systime website viser, at der findes materialer
til produktet på internettet.
Se betingelser på www.systime.dk

SYSTIME >

Skt. Pauls Gade 25

DK-8000 Århus C

Tlf.: 70 12 11 00

www.systime.dk

INDHOLD

INDHOLD 3

FORORD 9

KAPITEL 1:

COMPUTEREN 11

- 1.1. processor (cpu) og hukommelse (ram) **12**
- 1.2. hardware-opbygning **13**
- 1.3. opbygning af ram **14**
- 1.4. harddisk og andet lager **16**
- 1.5. opsummering af sprogbrug **17**
- 1.6. filer **18**
- 1.7. nøgleord **19**

KAPITEL 2:

PROCESSOR OG MASKINSPROG 20

- 2.1. processor **21**
 - 2.1.1. afvikling af instruktioner **22**
 - 2.1.2. alu **26**
 - 2.1.3. registre **29**
- 2.2. von neumaann-princippet **30**
- 2.3. instruktionsafvikling – et eksempel **32**
- 2.4. maskinkode og assemblerkode **34**
- 2.5. fima – et fiktivt assemblersprog **35**
 - 2.1.4. plus og minus **36**
 - 2.1.5. indlæs og skriv **38**
 - 2.1.6. hop-instruktioner **39**
- 2.6. højniveausprog **41**
- 2.7. nøgleord **42**

KAPITEL 3:

OPERATIVSYSTEM OG KERNE 43

3.1. kerne **46**

 3.1.1. i/o og afbrydelser **46**

 3.1.2. kernetilstand og brugertilstand **48**

3.2. lageradministration **51**

3.3. processer **54**

 3.3.1. processtilstande og skedulering **56**

 3.3.2. samtidighed og synkronisering **58**

 3.3.3. asynkron beskedudveksling **60**

3.4. bootstrapping **61**

3.5. nøgleord **64**

KAPITEL 4:

INTERNETTET – I BRUGERPERSPEKTIV 65

4.1. klient og server **66**

4.2. websider og webadresser **67**

 4.2.1. markup og fremvisning **68**

 4.2.2. dynamiske websider (server-side scripting) **69**

 4.2.3. client-side scripts **70**

4.3. protokoller **72**

4.4. http **74**

 4.4.1. request og response **74**

 4.4.2. sessions **78**

 4.4.3. cookies **79**

4.5. emails **81**

4.6. chat og instant messaging **83**

4.7. fildeling **83**

4.8. domæner, ip-adresser og dns **85**

4.9. nøgleord **87**

KAPITEL 5:

NETVÆRK – LAGDELT KOMMUNIKATION 88

5.1. internettets fysiske opbygning **88**

5.2. lagdelt kommunikation **91**

 5.1.1. post: kommunikation med tre lag **91**

 5.1.2. netværk: kommunikation med fem lag **93**

 5.1.3. tjenester og grænseflader **95**

 5.1.4. protokoller og protokolstak **96**

5.3. applikationslag **97**

5.4. transportlag	98
5.4.1. proces-adressering over netværk	98
5.4.2. pålidelig og upålidelig kommunikation	100
5.4.3. pålidelig kommunikation vha. kvitteringer	102
5.5. netværkslag	105
5.5.1. routing	106
5.5.2. ekstra information	108
5.5.3. offentlige ip-adresser	109
5.5.4. nat og lokale ip-adresser	109
5.6. data-link-laget	110
5.7. det fysiske lag	112
5.8. et eksempel til at illustre det hele	114
5.9. nøgleord	119

KAPITEL 6:

PROGRAMMERING 120

6.1. programmer som algoritmer	121
6.2. data, information og modellering	123
6.2.1. analog og digital repræsentation	126
6.3. programmeringssprog	126
6.3.1. syntaks og semantik	127
6.3.2. typesystem	130
6.3.3. programbiblioteker	133
6.4. oversættelse og fortolkning	134
6.4.1. højniveau- og lavniveausprog	134
6.4.2. oversættelse	132
6.4.3. fortolkning	136
6.4.4. opsummering: implementation af programmeringssprog	139
6.5. paradigmer indenfor programmeringssprog	140
6.6. imperativ programmering	141
6.6.1. variable og udtryk	142
6.6.2. lister	144
6.6.3. metoder og udtryk	145
6.6.4. kontrol-flow	148
6.6.5. kontrolstruktur: forgrening	149
6.6.6. kontrolstruktur: løkker	151
6.7. objekt-orienteret programmering	152
6.7.1. hvad er et objekt?	153
6.7.2. klasser	155
6.7.3. nedarvning	157
6.8. nøgleord	158

KAPITEL 7:

DATABASER 159

7.1. datamanipulation	160
7.2. datamodel og skema	161
7.3. datamodel: e/r-modellen	162
7.3.1. entiteter og entitetsklasser	163
7.3.2. attributter	164
7.3.2.1. simple og sammensatte attributter	164
7.3.2.2. en-værdi-attributter og flerværdi-attributter	165
7.3.2.3. attribut-værdien NULL	165
7.3.3. kandidatnøgler og primærnøgler	165
7.3.4. relationer og kardinaliteter	166
7.3.5. e/r-diagrammer	169
7.3.6. e/r-modellering	171
7.3.7. eksempel: e/r-modellering	172
7.4. relationelle databaser	175
7.5. fra e/r-model til tabeller i relationel database	176
7.6. integritet af database	179
7.7. normalformer	181
7.7.1. første normalform (1NF)	183
7.7.2. anden normalform (2NF)	183
7.7.3. tredje normalform (3NF)	185
7.8. databegrænsninger	186
7.8.1. fremmednøgler	188
7.8.2. domæne-begrænsninger og tupel-begrænsninger	190
7.8.3. eksplisitte begrænsninger	192
7.9. opsummering: sikring af databasens integritet	192
7.10. backup og tilgængelighed	193
7.11. nøgleord	193

KAPITEL 8:

DIGITALE MULTIMEDIER 194

8.1. grafik	195
8.1.1. pixels og skærmfremvisning	196
8.1.2. farvemodeller	197
8.1.3. billedkvalitet	198
8.1.4. bitmap-billeder	200
8.1.5. vektorgrafik	203
8.1.6. udprint af grafik	205
8.1.7. tabsfri komprimering af grafik	205
8.1.8. grafikkomprimering med tab	208

8.2. lyd	208
8.2.1. optagelse af lyd i digitalt format	209
8.2.2. afspilning af digital lyd	210
8.2.3. komprimering af lyd	211
8.3. video	213
8.4. 3d-grafik	214
8.5. multimedier over netværk: streaming	215
8.5.1. streaming af lagret data	217
8.5.2. live streaming	218
8.5.3. real time-interaktion	219
8.5.4. udfordringen: best effort	219
8.6. nøgleord	223

KAPITEL 9:

IT-SIKKERHED **224**

9.1. it-systemer og aktører	224
9.2. mål med it-sikkerhed	225
9.2.1. fortrolighed, integritet og tilgængelighed	226
9.2.2. trusler, sårbarheder og modmidler	227
9.2.3. privacy	228
9.2.4. uafviselighed	230
9.3. sårbarheder og trusler	230
9.3.1. velmenende personer (brugere)	230
9.3.2. ondsindede personer	231
9.3.3. kodeord	232
9.3.4. software	232
9.3.5. malware	233
9.3.6. spam	236
9.3.7. kommunikation over netværk	236
9.3.8. trådløse netværk	239
9.4. modmidler	240
9.4.1. autorisation	241
9.4.2. adgangskontrol	241
9.4.3. kryptografi	243
9.4.4. antivirus-software	244
9.4.5. firewalls	244
9.4.6. intrusion detection-systemer (ids)	245
9.4.7. spamfiltre	246
9.4.8. awareness	246
9.4.9. fysisk sikkerhed	246
9.4.10. sporbarhed	246

9.5. kryptografi	247
9.5.1. eksempel: kryptering à la cæsar	248
9.5.2. klartekst, ciffertekst og nøgle	248
9.5.3. symmetrisk kryptering	249
9.5.4. asymmetrisk kryptering	250
9.5.5. aftale af symmetrisk nøgle: diffie-hellman	253
9.5.6. autentificering med random challenge	254
9.5.7. data-integritet med hashing	255
9.5.8. digital signatur	257
9.5.9. pki	259
9.5.10. kryptografisk opsummering	260
9.6. realisering af it-sikkerhed	260
9.6.1. omkostninger	261
9.7. nøgleord	263

KAPITEL 10:

PROJEKTARBEJDE I DATALOGI **264**

10.1. arbejdsproces	264
10.1.1. brainstorm	266
10.1.2. analyse og kravspecifikation	266
10.1.3. modellering og design	267
10.1.4. implementation	268
10.1.5. fejlretning (debugging)	269
10.1.6. test	269
10.1.7. dokumentation	270
10.1.8. brugervejledning	270
10.2. brugergrænseflader	271
10.2.1. brugervenlighed	271
10.3. software-arkitektur	273

STIKORD **283**

ILLUSTRATIONER **292**

FORORD

Grundbog i datalogi er en grundbog til valgfaget Datalogi C på de fire gymnasiale uddannelser STX, HF, HHX og HTX. Bogen dækker de krav, læreplanen stiller.

Mit ønske med *Grundbog i datalogi* har været at skabe en datalogibog, der både er fagligt seriøs og sjov at læse – for såvel gymnasieelever som computerentusiaster, der vil vide lidt mere om, hvad der rører sig bag skærm og kabinet. Mit håb er, at man vil finde bogen teoretisk, men ikke tør. Med figurer, analogier og farverige eksempler skulle det til tider tunge stof gerne blive spiseligt for alle.

STRUKTUR

Bogen er i to dele: En trykt og en webbaseret. Groft sagt findes det teoretiske stof i den trykte del, mens ”alt det sjove” – praktiske opgaver i programmering, så man får godt med bits op under neglene – foregår på webdelen. Webdelen er altså ikke et supplement, men en fuldt integreret del af *Grundbog i datalogi*.

BOGEN

Den trykte del af *Grundbog i datalogi* forudsætter intet andet end læse- og lærerlyst.

Bogen dækker helt grundlæggende teori indenfor

- ◆ den fysiske opbygning af computer og netværk,
- ◆ processor og maskinsprog,
- ◆ operativsystemer,
- ◆ netværk, herunder internettet og lagdelt kommunikation,
- ◆ programmering,
- ◆ databaser,
- ◆ digitale multimedier,
- ◆ IT-sikkerhed,
- ◆ projektarbejde i datalogi.

Bogen er stort set *teknologineutral* – på den måde kan teoretisk stof eksemplificeres med netop de teknologier, der fx passer til et specifikt temaforløb, opfylder elevernes ønsker eller harmonerer med en lærers præferencer.

Kapitler kan læses i det udvalg og den rækkefølge, der passer til det enkelte undervisningsforløb. Vær dog opmærksom på, at

- ◆ indenfor et enkelt kapitel forudsætter et afsnit ofte de foregående afsnit,
- ◆ kapitel 2 og kapitel 3 forudsætter begge kapitel 1,
- ◆ kapitel 5 er lettest at forstå, hvis man har læst (dele af) kapitel 4.

Ved hvert kapitels afslutning findes en liste med nøgleord, der opsummerer de væsentligste begreber fra kapitlet.

WEBDELEN

Webdelen af *Grundbog i datalogi* omfatter

- ◆ forslag til temaforløb, der dækker læreplanens krav til et pensum; hvert temaforløb er fuldt dækkende i forhold til læreplanen og kan vælges af lærer og elever i årets første timer,
- ◆ praktiske, teknologispecifikke opgaver i varierende sværhedsgrader,
- ◆ kapitel- og temabaserede quizzere med evaluering,
- ◆ projektkatalog med forslag til datalogifaglige og tværfaglige projektforløb,
- ◆ stof og referencer til den særligt interesserende elev,
- ◆ datalogisk ordbog.

Webdelen findes på www.datalogi.systime.dk.

FEJL OG KOMMENTARER

Grundbog i datalogi er i sin førsteudgave, og der tages forbehold for de fejl, der utvivlsomt vil forekomme. Alle henvendelser vedrørende fejl, mangler eller sløseri – ja, enhver kommentar – modtages med kyshånd på martindamhus@gmail.com.

TAK TIL

Jeg vil gerne benytte lejligheden til at rette en særlig tak til følgende 3 personer:

- ◆ min hustru, Kira, for vidtstrakt tålmodighed med mig under skrivningen,
- ◆ gymnasielærer Marianne Terp for kritiske og konstruktive kommentarer til råmanuskriptet,
- ◆ datalogilærer Elisabeth Husum for uundværlig faglig og pædagogisk kon-sulentbistand på manuskriptet og dets disposition.

Endvidere tak til: Ture Damhus, Michael Friis, Camilla Bruhn Poulsen, Michael Sejr Schlichtkrull og Mathilde Harder Schousboe.

KAPITEL 1

COMPUTEREN

Et godt første spørgsmål i faget datalogi lyder “Hvad er en computer?”.

ORDFORKLARING

COMPUTER

Elektronisk maskine, der kan *modtage input*, enten løbende eller som forprogrammering, *behandle data* og *meddele resultater*.

Når du skal forstå en computer, løber den allerførste skillelinie mellem hardware og software.

En computer er hardware, der styres af software.

ORDFORKLARING

HARDWARE

De fysiske komponenter i en computer.

Følgende er eksempler på hardware: processoren, hukommelse, tastatur, skærm, harddisk, printer, grafikkort, bundkort, netkort, DVD-drev.



@Attributed to VistaIcons.com.

Fig. 1.1. Eksempler på hardware.

ORDFORKLARING

SOFTWARE

Programmer.

Software er for eksempel internet-browsere, tekstbehandlingsprogrammer og filmafspilningsprogrammer – men også operativsystemer, sms-ordbøger, GPS-systemer og elektro-niske automatpiloter til rutetrafik med fly.



Fig. 1.2. Eksempler på software.

Dette kapitel beskæftiger sig med hardware, mens kapitlerne 2 – 10 omhandler software.

1.1. PROCESSOR (CPU) OG HUKOMMELSE (RAM)

En computers opgave er basalt set at afvikle programmer. Det gøres med to enheder,

- ♦ *processor*: afvikler programmer,
- ♦ *hukommelse*: opbevarer de programmer, der er under afvikling.

Tilsammen kan man kalde processoren og hukommelsen for “computerens hjerne”.

Processoren kaldes også for *CPU'en* (engelsk: *central processing unit*). Programafvikling er faktisk bare *behandling af data* – det kaldes også *dataprocessering*, deraf ordet “processing” i navnet CPU. Vi hører meget mere om dataprocessering i kapitel 2.

Hukommelsen kaldes også for *RAM'en* (*random acces memory*).

1.2. HARDWARE-OPBYGNING

En computer er fysisk opbygget af processor, hukommelse samt yderligere tilkoblede enheder (engelsk: *devices*).

ORDFORKLARING

ENHED

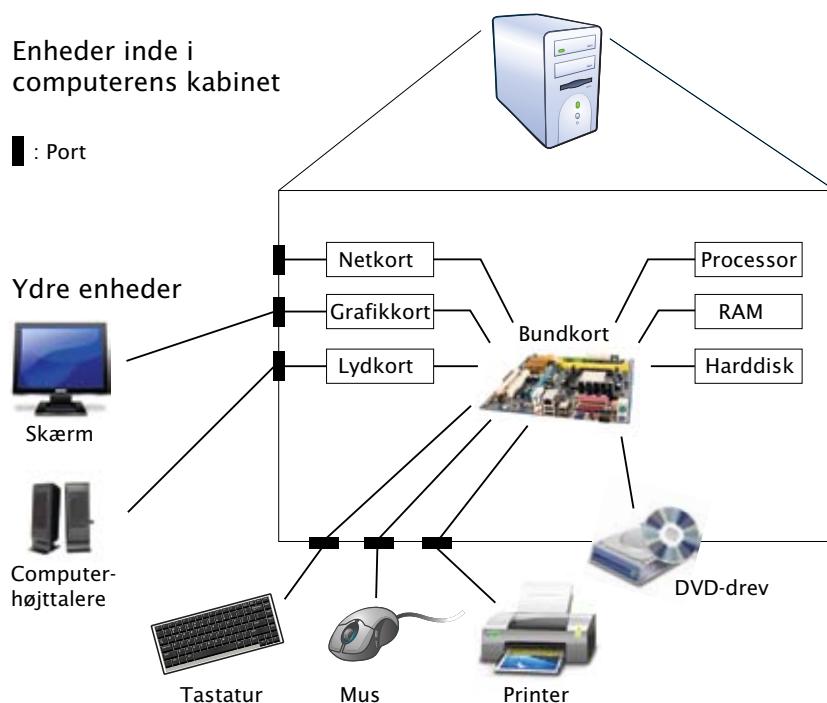
En enkelt hardware-komponent.

EKSEMPEL

ENHEDER

Mus, tastatur, skærm, diskdrev, printer, netkort, hukommelse og processor er alle eksempler på enheder.

Alle enheder er forbundet til computerens *bundkort*.



@tribute vistaicons.com.

Fig. 1.3. Computerens opbygning: Enheder koblet på bundkortet .

Ydre enheder er enheder, der er uden for selve computeren, såsom tastatur, mus eller skærm. Ydre enheder er også elektronik. Ydre enheder tilkobles særlige porte på computeren, der er forbundet til bundkortet.

Via ledninger på bundkortet kan enhederne med elektriske signaler snakke med processoren og hukommelsen. Ledningerne kaldes tilsammen *bussen*.

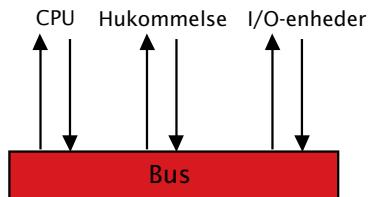


Fig. 1.4. Bus.

ORDFORKLARING

I/O-ENHEDER

En I/O-enhed er en enhed, der ikke er processor eller hukommelse.

Tænker man på processor og hukommelse som computerens hjerne, er I/O-enhederne "computerens krop".

Forkortelsen I/O står for Input/Output. Enhver I/O-enhed kan en eller begge af følgende ting:

- ◆ formidle beskeder til andre enheder (levere input),
- ◆ modtage beskeder fra andre enheder (modtage output).

EKSEMPEL

MUS, SKÆRM OG NETKORT ER I/O-ENHEDER

Mus, tastatur, skærm, diskdrev, printer og netkort er alle eksempler på I/O-enheder. En mus formidler bruger-bevægelser som signaler (input) til processoren, men modtager ikke noget output. En skærm modtager beskeder (output) fra processoren om, hvad den skal vise på displayet, men leverer ikke noget input. Et netkort formidler data modtaget over et netværk (input) og modtager data, der skal sendes over netværk (output).

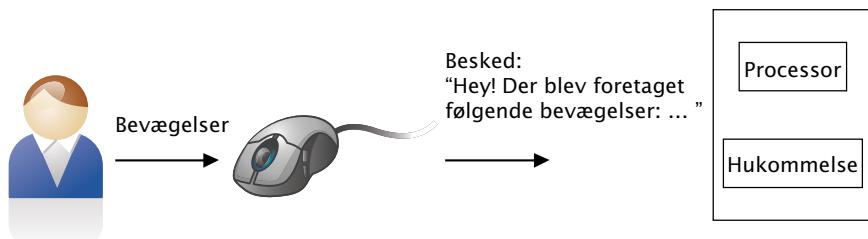


Fig. 1.5. En mus formidler beskeder fra omverdenen til processor og hukommelse.

ADVARSEL

INPUT/OUTPUT-FORVIRRING

Hvis en enhed A sender en besked til en anden enhed B, er beskeden på samme tid et output fra A og et input til B. En besked kan derfor ikke siges at være enten "et input" eller "et output". Man hælder dog alligevel til at sige, at en mus er en input-enhed (selvom den outputter beskeder, der fungerer som input til processor og hukommelse), og en skærm er en output-enhed (selvom skærmen modtager input fra processor og hukommelse).

Hver enhed kan betjenes gennem et sæt af specifikke kommandoer, dvs. kommandoer, der er helt specielt beregnet til lige netop den pågældende enhed. Operativsystemet på computeren vil gerne kommunikere med enheden ved hjælp af mere generelle kommandoer. Operativsystemet på computeren kommunikerer derfor med hver enhed gennem enhedens *driver*, en software-stump, der oversætter mellem operativsystemets generelle kommandoer og enhedens specifikke kommandoer.

1.3. OPBYGNING AF RAM

Computerens hukommelse, RAM'en, er den eneste lagerenhed, som CPU'en kan kommunikere direkte med.

ORDFORKLARING

HUKOMMELSE (RAM)

Enhed, der opbevarer programmer under afvikling samt data, som disse programmer behøver.

Hukommelsen er opbygget som en fortløbende række af celler, der alle har samme størrelse og indeholder data. Hver celle har en adresse. Adressen angives med et tal. Data i celler repræsenteres med tal. Således indeholder hver celle et tal. I en computer er alle talværdier *binære*, dvs. opbygget af symbolerne 0 og 1. Symbolerne 0 og 1 kaldes *bits*.

ADRESSE	DATACELLES VÆRDI
000	1001 0100 0111 1011 1110 1110 0011 1000
001	1101 1100 0101 1001 0010 0000 1101 0111
002	0011 0100 0000 1001 0100 0100 1110 0111
...	...
999	1000 0100 0111 1111 0111 0111 0101 1000

Fig. 1.6. En hukommelse med 1.000 adresser kan tegnes som et skema med 1.000 rækker. Hver datacelle indeholder her et 32 bit langt binært tal.

TIP

KÆRT BARN MANGE NAVNE

Computerens hukommelse kaldes også for internt lager, primært lager, main memory, primary storage og, som nævnt, RAM.

En hukommelse har en *controller*, der styrer adgangen til hukommelsen. Vil man *læse* værdien i en bestemt datacelle, fortæller man controlleren cellens adresse, hvorefter controlleren udleverer den angivne celles værdi. Vil man *skrive* i en bestemt celle, fortæller man controlleren adressen på den celle, der skal skrives i, samt hvad der skal skrives i cellen; herefter sørger controlleren for at overskrive cellens dataindhold.

TIL SÆRLIGT INTERESSEREDE

BITS, BYTES OG BINÆRE TAL

Inde i en computer foregår al databehandling med de fornævnte binære tal, hvor der kun er de to cifre 0 og 1. Det er praktisk for computeren, fordi de to cifre blot svarer til to forskellige fysiske tilstande. Fx kan en computer repræsentere cifrene 0 og 1 med to elektriske signaler, der har forskellig spænding. På **WWW** kigger vi nærmere på bits, bytes og de binære tal.

1.4. HARDDISK OG ANDET LAGER

Det ville være ideelt, hvis alle de programmer og alt det data, vi ønsker at arbejde med, kunne opbevares i hukommelsen permanent. Det er dog umuligt – af to grunde:

- Hukommelsen er tit *for lille* til at kunne huse alle programmer og alt data på én gang.
- Når der slukkes for strømmen til computeren, *slettes* hukommelsen.

Computere er derfor udstyret med *sekundært lager*, fx i form af harddisk. En harddisk kan rumme masser af data, og data slettes ikke, hvis strømmen til den slukkes. Til gengæld har CPU'en ikke direkte adgang til harddisken, så data fra harddisk skal indlæses i hukommelsen, før CPU'en kan arbejde på det.

Læsninger og skrivninger til harddisk tager væsentlig længere tid end læsninger og skrivninger til hukommelsen. En vigtig del af en effektiv computer er derfor et system til *lager-administration*, som sørger for at mindske trafikken mellem harddisk og hukommelse.

ADVARSEL

RAM OG HARDDISK ER IKKE DET SAMME

Før du læser videre er det en god ide at være helt på det rene med, at RAM og harddisk er to forskellige enheder, der tjener hver sit formål.

Harddisk er ikke det eneste sekundære lager. Der er også eksterne lagermedier som fx CD'er, DVD'er, USB-nøgler, eksterne harddisk eller backup-servere.

WWW

LÆSNING OG SKRIVNING AF DISK

På [WWW](#) finder du en kort forklaring af, hvorfor hukommelsen er hurtig og harddisken langsom.

TIL SÆRLIGT INTERESSEREDE

LAGERTYPER

Lagertyper, der slettes, når strømmen slukkes, kaldes volatile (engelsk for "flygtig"). RAM er volatil. Det er elektrisk flash-hukommelse også. Lagertyper, hvor data bevares, selvom strømmen slukkes, kaldes ikke-volatile. Harddisk er ikkevolatile. Nogle typer lager kan både skrives i og læses fra, men visse typer kan kun læses fra – sådanne typer lager kaldes ROM (read only memory) . En CD-ROM er fx en CD, hvis indhold ikke kan overskrives.

1.5. OPSUMMERING AF SPROGBRUG

Vi opsummerer nedenfor nogle vigtige forkortelser og ord. I resten af bogen regnes med, at du ved hvad "CPU", "RAM" og "lager" betyder.

ORD ELLER FORKORTELSE	BETYDER
CPU	Processor
RAM	Hukommelse
Lager	RAM, harddisk eller andet lagermedie
Primært lager	RAM
Sekundært lager	Harddisk, CD'er, USB-nøgler, mv.

Fig. 1.7. Opsummering af sprogbrug.

1.6. FILER

Data lagres som *filer*. Det kan være billedfiler, videoklip, tekstdokumenter eller adresdebøger til emailprogrammer. Også programmer lagres i filer. Filer opbevares som regel over længere tid på fx en harddisk. Hver enkelt fil har et *filnavn* og indeholder en mængde information, der på en eller anden måde hænger sammen.

Filer har ofte en *filtype* (også kaldet et *filformat*), der signalerer filens indholdstype og den måde, indholdet er lagret på – der er fx typer af tekstfiler og musikfiler. En fil repræsenteres – alt efter filtype – konkret som en organiseret samling af enten enkelte *bits* (0'er og 1'er), *bytes* (klumper á 8 bit), tekstrækker eller noget helt fjerde. En tekstfil er fx typisk organiseret som en samling af linier, der hver indeholder et antal bogstaver. Filtypen fastlægger altså en bestemt *struktur* for filen.

WWW

FILTYPER

På **WWW** finder du en liste af almindelige filtyper med henvisninger til, hvor du kan læse mere om de enkelte filtyper.

Gennem computerens *filsystem* kan man oprette, slette, læse, redigere, omdøbe og flytte filer. Filsystemet oversætter handlinger udtrykt på et menneskevenligt, abstrakt plan (såsom "slet fil", "se filstørrelse" eller "åbn fil") til konkrete handlinger på maskinniveau (fx at tilgå bestemte adresser i hukommelsen, indlæse bestemte harddisk-områder i hukommelse eller skrive bestemte dataværdier i bestemte harddisk-områder).

1.7. NØGLEORD

- ◆ Computer
- ◆ Hardware
- ◆ Software
- ◆ Processor
- ◆ Hukommelse
- ◆ CPU
- ◆ RAM
- ◆ Enhed
- ◆ Bundkort
- ◆ Bus
- ◆ Ydre enhed
- ◆ I/O-enhed
- ◆ Harddisk
- ◆ Primært lager
- ◆ Sekundært lager
- ◆ Fil

KAPITEL 2

PROCESSOR OG MASKINSPROG

En computer kan *afvikle programmer*. Det er denne evne der gør, at vi kan sætte computere til at styre fly, lave vejrudsiger eller vinde i skak over verdensmestrene. Der er programmer til kontoropgaver, mobiltelefoni, emails og lagerstyring. Computerprogrammer er dermed overalt.

Et *program* er en liste af instruktioner, vi gerne vil have computeren til at udføre. Tilsammen udtrykker instruktionerne den opgave, computeren skal løse. Computeren afvikler programmet ved at udføre instruktionerne i listen i den rigtige rækkefølge.

ORDFORKLARING

INSTRUKTION

En ordre til computeren om at udføre en enkelt (typisk meget lille) opgave.

ORDFORKLARING

PROGRAM

Organiseret liste af instruktioner, der ved udførelse får computeren til at løse en bestemt (større) opgave.

For at afvikle et program på en computer må vi formulere hver enkelt instruktion i programmet, så computeren forstår den. Til gengæld skal computeren udføre hver instruktion, den forstår, korrekt, og udføre instruktionerne i listen i den rigtige rækkefølge.

EKSEMPEL

ISSPISENDE COMPUTER

En (lidt tåbelig) computer forstår kun de to ordrer "Spis en is" og "Fortæl, hvor mange is du har spist", men ikke ordenen "Spis rå løg". Vi har skrevet følgende to programmer:

Program 1:	Program 2:
Spis en is	Spis en is
Spis en is	Spis rå løg
Fortæl, hvor mange is du har spist	Fortæl hvor mange is, du har spist

Hvis vi nu beder computeren om at afvikle program 1, vil den spise én til is, og så svare: "2". Hvis vi beder computeren om afvikle program 2, vil den først spise en is, hvorefter den vil møde instruktionen "Spis rå løg", som den ikke forstår. Hvad så?! – Mon ikke computeren går i stå? Den spiser i hvert fald ingen rå løg, når den ikke forstår orden.

Programmer behandler data. For eksempel behandler tekstbehandlingsprogrammer tekstmapper og filmafspilningsprogrammer afspiller film.

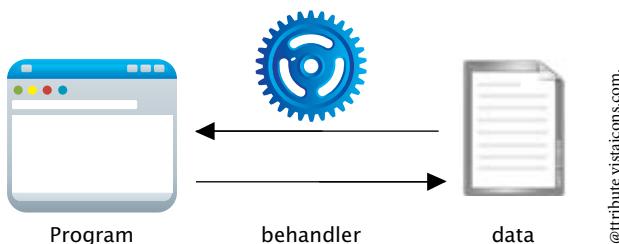


Fig. 2.8. Programmer behandler data.

På den måde er et program som en bageopskrift: Der er ingredienser (data), og forklaringer på, hvad der skal gøres med ingredienserne og i hvilken rækkefølge (instruktioner).

2.1. PROCESSOR

En computer, der er i gang med at afvikle et program, skal hele tiden vide, hvilken instruktion der er den næste.

Derfor består programafvikling af to ting:

- ◆ at udføre instruktioner,
- ◆ at holde regnskab med, hvilken instruktion der næste gang skal udføres.

Det er computerens *processor*, der varetager begge dele. Processoren bestemmer, hvad computeren *gør* og hvornår.

ORDFORKLARING

PROCESSOR (CPU)

Den hardware-del i computeren, der varetager afvikling af instruktioner. Processoren kaldes også CPU'en (central processing unit).

Processoren indeholder en regneenhed og en kontrolenhed. Groft set står regneenheden for at *udføre instruktioner*, mens kontrolenheden fortæller regneenheden, *hvilke instruktioner* der skal udføres. Kontrolenheden indlæser instruktionerne fra computerens hukommelse. Regneenheden kaldes også for ALU'en (en forkortelse for *arithmetic-logical unit*).

ORDFORKLARING

ALU

Processorens regneenhed.

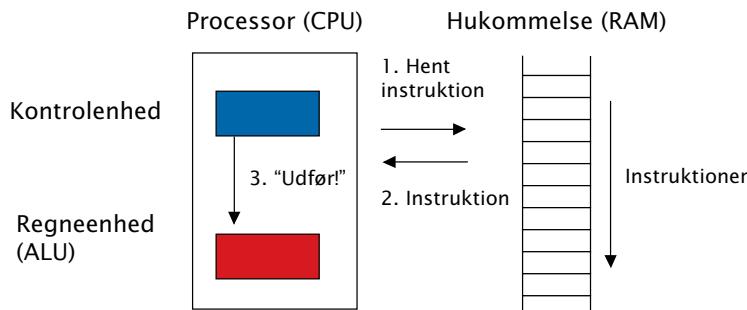


Fig. 2.9. Afvikling af instruktioner. Processor (CPU), kontrolenhed, regneenhed (ALU) og hukommelse (RAM).

Inden du læser videre, kan det være nyttigt at genopfriske begreberne *hukommelse*, *adresse* og *datacelle* fra afsnit 1.3.

2.1.1. AFVIKLING AF INSTRUKTIONER

Når computeren er tændt, afvikler processoren hele tiden instruktioner, én ad gangen. Det er forskelligt fra processortype til processortype, præcis hvilke instruktioner der kan udføres.

ORDFORKLARING

INSTRUKTIONSSÆT

En processors instruktionssæt er de instruktioner, processoren kan udføre.

Når vi vil lave et program til en bestemt processor, skal programmet altså bestå udelukkende af instruktioner fra den processors instruktionssæt.

EKSEMPEL

ISSPISENDE COMPUTERS INSTRUKTIONSSÆT

Den isspisende computers processors instruktionssæt består af de to instruktioner "Spis en is" og "Fortæl, hvor mange is du har spist".

I en rigtig processor indeholder instruktionssættet typisk:

INSTRUKTIONSTYPER	FORMÅL
Aritmetiske	Behandler tal: plus, minus, gange, division, m.fl.,
Logiske	Behandler logiske udtryk: AND , OR , NOT , m.fl.,
Indlæs/skriv	Indlæser fra eller skriver i computerens hukommelse
Hop	Hopper til en ny instruktion

Fig. 2.10. Typer af instruktioner. Aritmetik betyder "regning med tal".

ANALOGI

INSTRUKTIONER ≈ LEGOKLODSER

Det ser måske ikke ud som om man kan komme særligt langt med sådanne simple instruktioner. Men skinnet bedrager: Det er med instruktioner som med legoklodser. Selv om man kun har nogle bestemte typer legoklodser til rådighed, kan man stadig bygge så store slotte det skal være. Tilsvarende kan komplekse programmer opbygges udelukkende med disse simple instruktioner.

Det er altså ikke nogen begrænsning, at instruktionerne er få og simple. Det er heller ikke en begrænsning at skulle holde sig til et bestemt instruktionssæt – bare det indeholder et fornuftigt udvalg af instruktioner fra hver af de fire kategorier ovenfor.

Som de fire kategorier af instruktioner viser, består processorens arbejde basalt set i at hente data i hukommelsen, arbejde på det, og gemme det i hukommelsen igen.

Møder processoren en hop-instruktion, hopper den til et andet sted i programmet (en ny adresse), hvorefter den fortsætter med at hente/bearbejde/gemme data.

Processoren afvikler instruktioner ved at gennemløbe en simpel to-trins-cyklus, kaldet Fetch/Execute (*fetch*: engelsk for “hent”, *execute*: engelsk for “udfør”). For at gøre dette behøver processoren

- ♦ en *programtæller*, et tal, der er lig med den adresse, næste instruktion skal hentes fra,
- ♦ en *instruktionsholder*, der indeholder den instruktion, der er ved at blive udført.

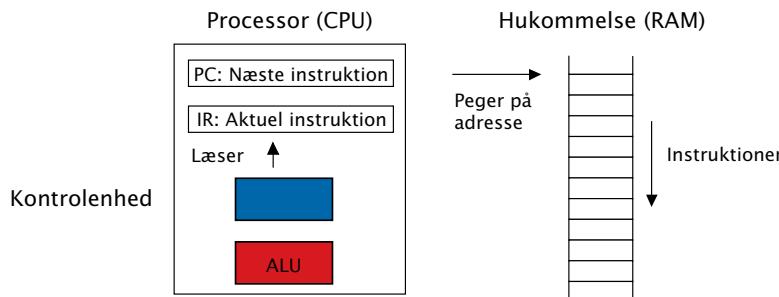


Fig. 2.11. Programtæller (PC) og instruktionsholder (IR).

Programtælleren kaldes også **PC** for *program counter* (og ikke – som i andre sammenhænge – “personal computer”). Instruktionsholderen kaldes også **IR** for *instruction register*. Ordet “register” skyldes, at den aktuelle instruktion og programtællerens værdi begge gemmes i såkaldte *registre* – særlige lagerceller i processoren. Vi hører mere om registre i afsnit 2.1.3.

FETCH/EXECUTE**1. Fetch: Instruktion hentes**

- a. Kontrolenheden finder adressen på næste instruktion ved at kigge på programtælleren (**PC**-registret læses).
- b. Instruktionen fra den pågældende adresse i hukommelsen gemmes i instruktionsholderen (i **IR**-registret).
- c. Programtælleren sættes til at indeholde adressen på den næste instruktion (talværdien gemt i **PC**-registeret tælles 1 op).

2. Execute: Instruktion udføres

- a. Processorens kontrolenhed kigger i instruktionsholderen og finder ud af, hvilke udregninger instruktionen kræver udført.
- b. Kontrolenheden beordrer regneenheden at udføre de nødvendige udregninger.
- c. Regneenheden fortæller udregningernes resultat til kontrolenheden.
- d. Kontrolenheden reagerer passende på svaret (den kan fx gemme et resultat i et register eller ændre programtælleren, hvis instruktionen var en hop-instruktion).

Når et execute-trin er afsluttet, er processoren klar til at hente næste instruktion. Faktisk holder processoren aldrig pause med at udføre denne fetch/execute-cyklus. Den gentages igen, og om igen, og om igen – indtil der bliver slukket for strømmen. En processor er derfor altid i gang med at udføre en instruktion.

I afsnit 2.3 tager et vi detaljeret eksempel på et forløb efter fetch/execute-cyklen.

Et gennemløb af en fetch-execute-cyklus kaldes en *clock-cyklus*, fordi det er et ur, indbygget i computerens hardware, der bestemmer, hvornår næste fetch-execute skal udføres. Det er nødvendigt at styre cyklerne med et ur, da alle dele af processorhardwaren skal være synkroniserede. *Clock-frekvensen* for en computer er det antal clock-cykler, som computeren gennemløber på eet sekund. En computer med clock-frekvensen 2 GHz (gigahertz= 10^9 pr. sekund) afvikler altså *2 milliarder instruktioner i sekundet*.

TIL SÆRLIGT INTERESSEREDE

ARKITEKTURER OG CLOCK-CYKLER

En computer, der udfører én instruktion pr clock-cyklus, siges at have enkelt-cyklus-arkitektur. Man kan dog forøge computerens hastighed væsentligt ved i stedet at benytte en anden arkitektur.

Med en fler-cyklus-arkitektur splittes udførelsen af hver instruktion ud på flere clock-cykler. Til gengæld kan hver enkelt clock-cyklus' varighed afkortes.

Med en pipelinet arkitektur splittes instruktioner over flere clock-cykler, og processoren udfører flere instruktioner delvist overlappende (men stadig i rækkefølge). En simpel pipeline med 2 trin kunne for eksempel begynde at udføre fetch for instruktion nr 2, når instruktion nr 1 sættes i gang med trinnet execute.

Det kræver indsigt i processorarkitektur at forstå fler-cyklus-arkitekture og pipelinede arkitekturer til bunds. Ideen bag begge er, at én clock-cykel kan inddeltes mere detaljeret end blot i de to trin "fetch" og "execute". På **WWW** finder du en kort forklaring af arkitekturerne og henvisninger til steder, hvor du kan læse mere.

Alle moderne processorer er pipelinede.

2.1.2. ALU

ALU'en kan foretage udregninger, men har ingen hukommelse. Derfor er den som en glemsom lampeånd. Uanset hvilket regnestykke, vi fortæller lampeånden, regner den svaret ud uden at blinke og uden at vi behøver at bekymre os om, hvordan den fandt svaret. Men sekundet efter har lampeånden glemt, hvad den blev bedt om at gøre, og hvad konsekvensen af handlingen var. Tilsvarende kan processorens kontrolenhed ses som den magiske lampes ejer, der utrætteligt får sin lampeånd til at udføre den ene ordre efter den anden.

EKSEMPEL

ALU-ARITMETIK

ALU'en kan udføre følgende aritmetiske udregninger: $2+3$, $4 \cdot 17$, $22 - 8$, $36/6$.

Alle tal og resultater i ALU-udregninger er hele tal. Derfor er udregninger som $36/5 = 7,2$ umiddelbart et problem. I dette kapitel antager vi, at alle udregninger med division er udregninger, hvor divisionen går op.

WWW

KOMMA-TAL

På **WWW** gennemgår vi kort, hvordan ALU'en håndterer komma-tal.

Udregningerne i eksemplet ovenfor kan alle udtrykkes med 3 input, nemlig 2 tal og 1 operation:

UDREGNING	OPERATION	OPERAND 1	OPERAND 2	ALU-SVAR
2+3	ADD	2	3	5
4 • 17	MULT	4	17	68
22-8	SUB	22	8	14
36/6	DIV	36	6	6

Faktisk kan alt, hvad vi gerne vil have ALU'en til at gøre, udtrykkes med en serie af udregninger på formen "1 operation og 2 operander" (operanderne er de værdier, operationen "opererer" på).

EKSEMPEL

ALU-ARITMETIK I FLERE SKRIDT

Vi vil have ALU'en til at udregne $(4+17-9)/2$. Dette regneudtryk indeholder 4 tal og 3 operationer, så vi må omforme de nødvendige udregninger til en serie af regnestykker, der hver indeholder 2 tal og 1 operation:

UDREGNING	OPERATION	OPERAND 1	OPERAND 2	ALU-SVAR
4 + 17	ADD	4	17	21
ALU-svar - 9	SUB	21	9	12
ALU-svar / 2	DIV	12	2	6

Denne udregningsmåde svarer til, hvordan vi selv kunne bestemme tallet $(4+17-9)/2$ med hovedregning.

Rent teknisk er ALU'en en elektronisk komponent med 3 indgående ledninger (A, B og C) og hvorfra 2 ledninger udgår (D og E). Vi taler derfor for eksempel om, at ALU'en kan "modtage input på A", når et inputsignal sendes på ledningen A. Set udefra ser ALU'en ud som skitseret:

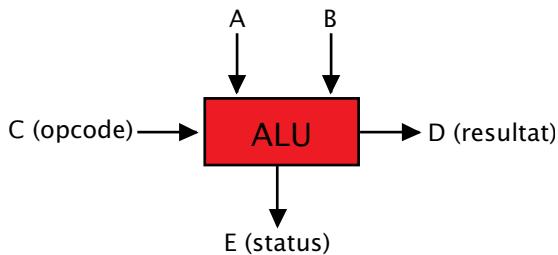


Fig. 2.12. ALU.

Når ALU'en modtager input på ledningerne A, B og C, beregner den resultatet af operationen C anvendt på operanderne A og B. Hvis udregningerne udføres uden problemer, outputtes det korrekte svar på D sammen med status OK på E. Signalet på linien C kaldes tit for "opcode" for at antyde, at det er den kode, der fastlægger operationen.

I de ovenstående eksempler outputter ALU'en OK på statuslinien E. Imidlertid går der nogle gange noget galt, når ALU'en skal udføre operationen C på operanderne A og B. Hvis det sker, skal kontrolenheden vide, at beregningen er gået galt – og at det, der outputtes på resultat-linien D, er forkert. ALU'en meddeler dette ved at sætte status E til "ikke OK", og typisk angives også en fejlkode, sådan at kontrolenheden kan se, hvad der gik galt.

EKSEMPEL

DIVISION MED 0

Modtager ALU'en A = 5, B = 0 og C = DIV, vil ALU'en outputte noget vrøvl på resultatlinien D og en besked i stil med "ikke OK, fejlkode: Division med 0" på statuslinien E. ALU'en har her forsøgt at dividere 5 med 0, hvilket ikke giver mening.

EKSEMPEL

OVERFLOW

ALU'en kan kun håndtere begrænsede talstørrelser på de to operand-ledninger og på resultat-ledningen. Ledningen vil typisk kunne transportere data på størrelse med en datacelle i hukommelsen. I denne bog skal ALU-input og ALU-output derfor maksimalt have 8 cifre.

Hvis resultatet af en udregning overskridt begrænsningen, kan ALU'en ikke udføre udregningen korrekt. Prøver vi give ALU'en A = 99.999.999, B = 20 og C = ADD, er det korrekte svar (100.000.019) større end den øverste grænse på 99.999.999.

Situationen kaldes aritmetisk *overflow*. I dette tilfælde vil ALU'en outputte noget vrøvl på resultatlinien D og en besked i stil med "ikke OK, fejlkode: Overflow" på statuslinien E.

TIL SÆRLIGT INTERESSEREDE

LOGISKE OPERATIONER

ALU'en håndterer også de *logiske operationer*, der er nævnt i Figur 2.10. Her er operan-
derne er sandhedsværdier. Du finder en eksempler på logiske instruktioner og ALU'ens
håndtering af dem på [WWW](#).

2.1.3. REGISTRE

CPU'en læser sine instruktioner fra RAM'en (husk Figur 2.9). I den løbende afvikling af
instruktioner har CPU'en dog brug for at holde hus med flere ting, bl.a.

- ◆ hvilken instruktion udføres netop nu
- ◆ hvad er den næste instruktion, der skal udføres
- ◆ værdier, hentet i hukommelsen (læst), som skal bruges til beregninger
- ◆ beregnede værdier, der skal skrives i hukommelsen
- ◆ beregnede værdier, der skal regnes videre på ved næste instruktion
- ◆ hvilke adresser i hukommelsen, der skal skrives til eller læses fra

Derfor har processoren sine egne lagerenheder, der kaldes registre. Her kan disse oplys-
ninger løbende noteres.

ORDFORKLARING

REGISTER

Lagerenhed inde i processoren.

I typiske processorer har de fleste registre samme størrelse som én datacelle i hukom-
melsen. På den måde kan data fra en celle i hukommelsen indlæses i et register uden at
fyldes for meget eller for lidt. Det er mange gange hurtigere for processoren at læse fra et
register end at læse fra hukommelsen – data fra hukommelsen skal først transporteres til
et register, før det kan læses.

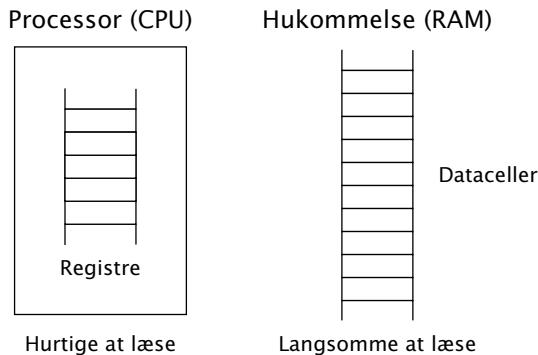


Fig. 2.13. *Registre*.

De fleste registre benyttes til at lagre beregningsdata, men to særlige registre er vigtige at bemærke:

ORDFORKLARING

PC-REGISTERET

I PC-registeret lagres programtælleren.

ORDFORKLARING

IR-REGISTERET

I IR-registeret lagres den aktuelle instruktion.

2.2. VON NEUMANN-PRINCIPPET

Vi har set, at processoren

- ◆ indlæser *instruktioner* fra hukommelsen,
- ◆ *indlæser* data fra hukommelsen.

Instruktionerne kan lagres i hukommelsen, fordi de repræsenteres som data (mere præcist: som binære tal). Både instruktioner og data lagres altså som tal i RAM'en. Kigger man på en bestemt lagercelle i hukommelsen, kan man blot se værdien gemt på den pågældende adresse, men ikke skelne instruktioner fra data.

En given datacelle kan altså godt på et tidspunkt blive indlæst som kode og på et andet tidspunkt som data! Først i det øjeblik indholdet af datacellen indlæses i processoren, afgøres det, om talværdien i cellen skal opfattes som kode eller som data:

- ◆ Datacellens indhold opfattes som en instruktion, hvis processoren er i fetch-trinnet af fetch/execute, og PC-registeret fortæller den processoren, at den pågældende datacelle indeholder næste instruktion.
- ◆ Datacellens indhold opfattes som data, hvis datacellen indlæses som følge af en indlæs-instruktion.

Vi kunne derfor sagtens opleve et forløb som det på figuren:

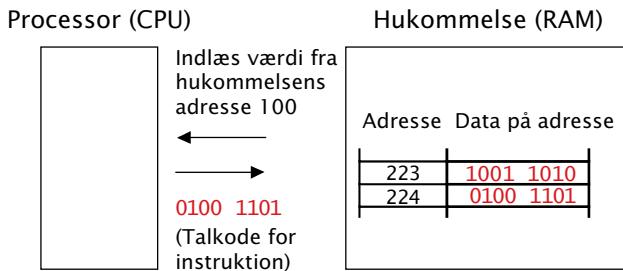


Fig. 2.14. Instruktioner kan opfattes som data. Her indlæses talkoden for en instruktion i CPU'en som taldata ved en indlæs-instruktion.

Dette princip for lagring af programkode kaldes for *von Neumann-princippet*.

ORDFORKLARING

VON NEUMANN-PRINCIPPET

1. Både data og instruktioner repræsenteres som tal.
2. Talrepræsentationerne af data og instruktioner lagres i samme lager (hukommelsen).

Processoren kan indlæse tal fra hukommelsen og dermed indlæse både data og instruktioner. Processoren kan skrive talværdier i hukommelsen og dermed ændre både data og programinstruktioner.

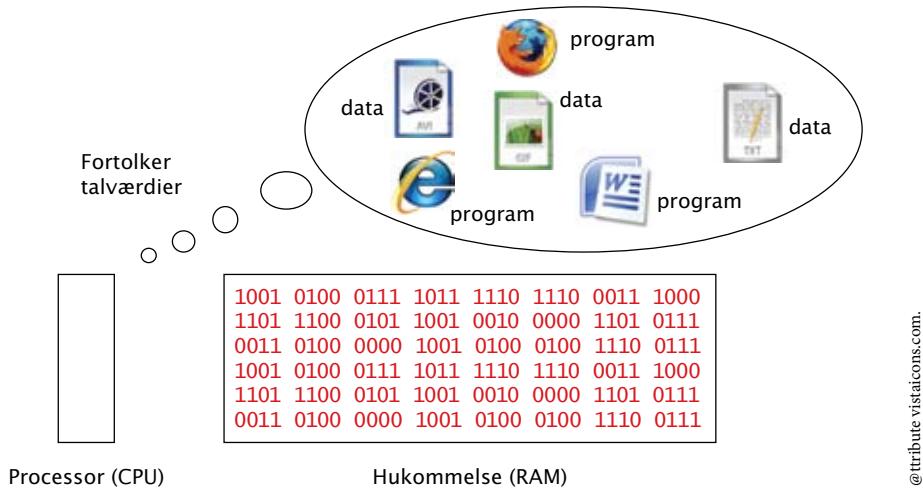


Fig. 2.15. Med von Neumann-princippet er både programmer og data i samme hukommelse samtidigt. Begge dele er lagret som tal. Det er op til CPU'en at fortolke talværdier som enten data eller program-instruktioner.

Med von Neumann-princippet kan computeren have både programdata og flere forskellige programmer i hukommelsen samtidig. Det er der fordele i:

- Computeren kan hurtigt skifte fra et program til et andet – den skal blot begynde at udføre instruktioner fra en anden startadresse.
- Computeren kan hurtigt hente data ind og ud af programmer – det er meget hurtigere at hente data fra hukommelsen end fra fx en harddisk.
- Programkode (instruktioner) kan bruges som input for andre programmer – det er fx praktisk i forbindelse med *oversættelse* (se afsnit 6.4).

Alle moderne computere følger von Neumann-princippet.

Ideerne i von Neumann-princippet blev første gang formuleret af den ungarsk-født amerikaner, videnskabsmanden John von Neumann (1903-1957), efter de flestes mening et geni. Hans forskning var revolutionerende indenfor talrige felter såsom matematik, kvantemekanik og statistik – og altså også datalogi.

2.3. INSTRUKTIONSAFVIKLING – ET EKSEMPEL

Vi tager et forsimplet eksempel på instruktionsafvikling via fetch/execute-cyklen i en tænkt maskine. Vi er særligt interesserede i at

- ♦ følge med i indholdet af **PC**,
- ♦ følge med i indholdet af **IR**,
- ♦ forstå von Neumann-princippet – hvordan instruktioner og data er to sider af samme sag.

Vi forestiller os at der uover registrene **PC** og **IR** er 4 dataregistre, som vi kalder **REG0**, **REG1**, **REG2** og **REG3**. I hvert dataregister kan lagres én talværdi (på 8 bit).

Lad os derfor forestille os et start-setup, hvor **PC**, **IR** og hukommelse ser ud som følger og dataregistrene alle indeholder talværdien **0000 0000**. Du skal ikke bekymre dig om, hvorfor talværdierne fortolkes som de viste instruktioner – sådan er det bare i dette eksempel.

PC	IR	FORTOLKNING SOM INSTRUKTION:
000
RAM		
ADRESSE	TALVÆRDI LAGRET PÅ ADRESSE	
000	0101 0000	Indlæs data fra adresse 000 i hukommelsen til registeret REG1 .
001	0001 1000	Læg indholdet af REG1 sammen med indholdet af REG2 og gem resultatet i REG0 .
002	1000 0011	Skriv indholdet af REG0 på adresse 003 i hukommelsen.
003	0000 0000	...
...

Nu påbegyndes fetch/execute. Det kan være en god ide at følge eksemplet med papir og blyant ved din side. Der sker følgende:

Fetch # 1: Da **PC** har værdien 000, indlæses instruktionen fra adressen 000 til **IR**. Det er instruktionen med talrepræsentation **0101 0000**. **PC** tælles 1 op og har nu værdien 001.

Execute # 1: Da **IR** indeholder talværdien **0101 0000**, skal der “indlæses data fra adresse 000 i hukommelsen til registeret **REG1**”. På adresse 000 i hukommelsen ligger talværdien **0101 0000**, og denne værdi lagres nu i **REG1**. Bemærk, at instruktionen her *indlæser sig selv som dataværdi*.

Fetch # 2: Da **PC** har værdien 001, indlæses instruktionen fra adressen 001 til **IR**. Det er instruktionen med talrepræsentation **0001 1000**. **PC** tælles 1 op og har nu værdien 002.

Execute # 2: Da **IR** indeholder talværdien **0001 1000**, skal “indholdet af **REG1** lægges sammen med indholdet af **REG2** og gemmes i **REG0**.” ALU'en aktiveres for at fuldføre beregningen. **REG1** indeholder talværdien **0101 0000** og **REG2** indeholder værdien **0000 0000**. Resultatet af regnestykket er **0101 0000**, og det lagres i **REG0**.

Fetch # 3: Da **PC** har værdien 002, indlæses instruktionen fra adressen 002 til **IR**. Det er instruktionen med talrepræsentation **1000 0011**. **PC** tælles 1 op og har nu værdien 003.

Execute # 3: Da **IR** indeholder talværdien **1000 0011**, skal “indholdet af **REG0** skrives på adresse 003 i hukommelsen.” **REG0** indeholder talværdien **0101 0000**, og dette tal lagres på adressen 003 i hukommelsen.

Efter disse tre fetch/execute-trin ser hukommelsen ud som følger:

RAM	
ADRESSE	TALVÆRDI LAGRET PÅ ADRESSE
000	0101 0000
001	0001 1000
002	1000 0011
003	0101 0000

PC har på dette tidspunkt værdien 003 og **IR** indeholder værdien **1000 0011**. Registrene **REG0** og **REG1** indeholder begge værdien **0101 0000**, mens **REG2** og **REG3** indeholder værdien **0000 0000**.

2.4. MASKINKODE OG ASSEMBLERKODE

Processoren forstår kun instruktioner, som den læser i lageret. Disse instruktioner er lagret som tal.

ORDFORKLARING

MASKINKODE

Processor-instruktioner i talformat.

Maskinkode afhænger altså af instruktionssættet. Maskinkode er besværlig at forstå for mennesker, fordi instruktionerne er lagret som binære talkoder:

```
.....  
1001 0100 0111 1011 1110 1110 0011 1000  
1101 1100 0101 1001 0010 0000 1101 0111  
0011 0100 0000 1001 0100 0100 1110 0111  
.....
```

Fig. 2.16. Maskinkode for en 32-bit processor. Hver linie med 8×4 bit = 32 bit svarer til én instruktion.

Assemblerkode er et tekstbaseret sprog, der koder instruktioner på en måde, der mere læsbar for mennesker. Hver linie assemblerkode kan oversættes direkte til én maskinkode-instruktion. Assemblerkode afhænger også af instruktionssættet.

ORDFORKLARING

ASSEMBLERKODE

Maskinkode lagret i et format, der er læsbart for mennesker.

Et særligt program, der kaldes en assembler, oversætter assemblerkode til maskinkode.

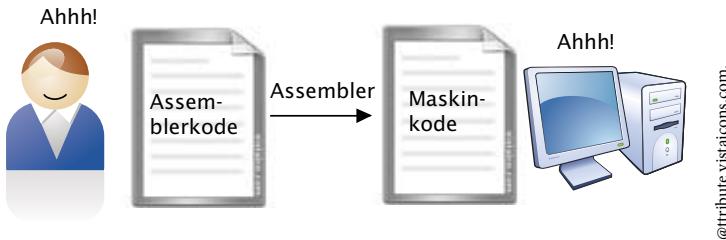


Fig. 2.17. (De fleste) mennesker kan lide assemblerkode. Computere kan lide maskinkode.

ADVARSEL

I daglig tale siger folk tit "assembler" i stedet for "assemblerkode", og de bruger under tiden ordet "maskinkode" også om de talkoder, vi kalder maskinkode her i bogen.

2.5. FIMA – ET FIKTIVT ASSEMBLERSPROG

Vi skal nu se på FIMA, der er et assemblersprog for en fiktiv maskine (FIMA står for fiktiv maskine-assembler). Den fiktive maskine vil vi omtale (lidt slapt) som *FIMA-maskinen*. Hukommelsen i FIMA-maskinen har 10.000 adresser, nummereret fra 0 til 9.999. På hver

adresse er der lagret et 8-cifret decimal-tal, der kan have fortegnet + (plus) eller fortegnet – (minus). Datacellernes værdier ligger altså altid mellem **-99.999.999** og **+99.999.999**. Vi undlader at skrive + foran positive værdier.

ADVARSEL

FIMA-MASKINEN BRUGER IKKE BINÆRE TAL

Selvom en ægte computer lagrer al data i 2-talssystemet, lagrer FIMA-maskinen for nemheds skyld al data i 10-talssystemet.

Udover de 2 registre til afvikling af instruktioner, **PC** og **IR**, har FIMA-maskinen 4 registre til at indlæse data i: **R0**, **R1**, **R2** og **R3**.

FIMA-MASKINENS REGISTRE

PC	IR	R0	R1	R2	R3
----	----	----	----	----	----

WWW

FIMA-SIMULATOR OG FLERE FIMA-INSTRUKTIONER

På **WWW** kan du eksperimentere med at skrive og køre programmer i FIMA. FIMA-instruktionssættet er også mere omfattende end nævnt her i bogen. Du finder de resterende instruktioner på **WWW**.

I de følgende afsnit kigger vi på FIMA-maskinens instruktionssæt.

2.1.4. PLUS OG MINUS

For at kunne foretage udregninger skal plus og minus være med i FIMA-instruktionssættet:

FIMA-INSTRUKTIONER: PLUS OG MINUS		
INSTRUKTION	ASSEMBLERKODE	BESKRIVELSE
ADD	ADD reg1 reg2 reg3	Lægger indholdet af to registre reg1 og reg2 sammen. Resultatet af udregningen gemmes i registeret reg3 .
SUB	SUB reg1 reg2 reg3	Trækker indholdet af register reg2 fra indholdet af reg1 . Resultatet af udregningen gemmes i registeret reg3 .

Som vi kan se, har begge instruktioner 3 operander. Det minder os om formatet på de regnestykker, vi beder ALU'en udregne. Bemærk, at de aritmetiske operationer kun kan udføres på data i registrene!

EKSEMPEL

EN LINIE ASSEMBLERKODE MED OPERANDER

Følgende linie assemblerkode

ADD R0 R0 R1

har de tre operander **R0**, **R0** og **R1**. Hvis vi afvikler kodelinien, vil indholdet af registeret **R0** blive lagt sammen med indholdet af selvsamme register. Resultatet af regnestykket gemmes i registeret **R1**.

Et fornuftigt program fortæller processoren, hvornår det er færdigt. Vi behøver altså følgende instruktion:

FIMA-INSTRUKTION: STOP		
INSTRUKTION	ASSEMBLERKODE	BESKRIVELSE
STOP	STOP	Afslutter programmet.

EKSEMPEL

STOP-INSTRUKTION

Antag, at registrene **R0** og **R1** indeholder tallene 1 og 3, mens registrene **R2** og **R3** indeholder 0. Hvis vi kører assemblerprogrammet

ADD R0 R1 R2	\\\ 1 + 3 = 4 gemmes i R2
ADD R0 R2 R1	\\\ 1 + 4 = 5 gemmes i R1
SUB R1 R2 R3	\\\ 5 - 4 = 1 gemmes i R3
STOP	\\\ Afslut program

vil registrene bagefter have ændret indhold. **R0** indeholder 1, **R1** indeholder 5, **R2** indeholder 4, og **R3** indeholder 1:

R0	R1	R2	R3
1	5	4	1

Vi har ovenfor benyttet *kommentarer* i assemblerkoden (tekst efter to skråstreger). Det ændrer ikke koden, men gør den lettere at læse for mennesker.

2.1.5. INDLÆS OG SKRIV

Processoren skal kunne læse og skrive i hukommelsen:

FIMA-INSTRUKTIONER: INDLÆS OG SKRIV		
INSTRUKTION	ASSEMBLERKODE	BESKRIVELSE
LOAD	LOAD addr reg1	Indlæser indholdet af den datacelle i hukommelsen, der har adresse addr , i registeret reg1 .
STORE	STORE reg1 addr	Gemmer indholdet af registeret reg1 på adressen addr i hukommelsen.

Program # 1 på side 39 er et eksempel brug af på **LOAD** og **STORE**.

Et program opererer på data. Det er ofte praktisk at indlæse dele af det data, programmet skal bruge, inden programmet startes. Det gøres ved at tildele bestemte talværdier til bestemte adresser i hukommelsen. Derfor starter FIMA-assemblerprogrammer med en dasektion, hvor data skrives i hukommelsen (data skal så bruges senere). Det gøres med en række linier på formen

MEMWRITE **addr** **num**

der skriver værdien af tallet **num** på adressen **addr** i hukommelsen. **MEMWRITE** står selvfølgelig for “memory write”. Fx skriver

MEMWRITE **100** **9**

værdien **9** på adressen **100** i hukommelsen. Dasektionen markeres med en “**DATA:**”-linie. Selve kode-sektionen kommer bagefter og markeres med en “**CODE:**”-linie.

For at illustrere brugen af **LOAD** og **STORE** (og **MEMWRITE**), betragter vi følgende simple program. Det indlæser to talværdier fra hukommelsen, lægger dem sammen, og gemmer dem i hukommelsen igen. Vores program indlæses, så første instruktion ligger på adresse **0** i hukommelsen.

PROGRAM # 1	
ADRESSE	KODELINIER OG KOMMENTARER
000	DATA: \\ Databladsen starter
001	MEMWRITE 44 -11 \\ Værdien -11 skrives på adressen 44
002	MEMWRITE 45 2 \\ Værdien 2 skrives på adressen 45
003	CODE: \\ Kode-afsnit starter
004	LOAD 44 R1 \\ Værdien på adresse 44 (-11) \\ indlæses i register R1
005	LOAD 45 R2 \\ Værdien på adresse 45 (2) \\ indlæses i register R2
006	ADD R1 R2 R3 \\ (-11)+2 = -9 gemmes i R3
007	STORE R3 46 \\ Indholdet af register R3 (-9) \\ gemmes på adressen 46
008	STOP \\ Stop program

Når vi har kørt programmet, vil hukommelsen se ud som følger:

HUKOMMELSE	
ADRESSE	INDHOLD AF DATACELLER
000 – 008	Kodelinierne i vores program
...	...
044	-11
045	2
046	-9
...	...

2.1.6. HOP-INSTRUKTIONER

De sidste instruktioner, vi skal bruge, er hop-instruktioner til at hoppe rundt i programmet. Med hop-instruktioner kan programafviklingen styres – programlinier kan *overspringes* eller *gentages* – i stedet for blot at blive udført en for en i den rækkefølge, de er anført. Det er hop-instruktioner der gør det muligt for programmer *handle ud fra betingelser* ("at tage beslutninger").

Hop-instruktioner virker ved at sætte værdien i **PC**-registeret – programtælleren – til den adresse, der skal hoppes til.

FIMA-INSTRUKTIONER: UBETINGET HOP OG BETINGET HOP		
INSTRUKTION	ASSEMBLERKODE	BESKRIVELSE
JUMP (ubetinget hop)	JUMP addr	Sætter indholdet af PC-registeret til værdien addr.
JUMPZERO (betinget hop)	JUMPZERO reg1 addr	Sætter indholdet af PC-registeret til værdien addr, hvis indholdet af registeret reg1 er 0.

Lidt forklaring til de to instruktioner:

- ◆ **JUMP**: Processoren henter i hvert fetch/execute-runde instruktionen på den adresse, der er lagret i registeret **PC**. Hvis **PC**-registeret bliver overskrevet med en ny adresse, fortsætter processoren derfor fra den nye adresse i næste runde. **JUMP** kaldes et ubetinget hop, fordi instruktionen får processoren til at hoppe uanset hvad.
- ◆ **JUMPZERO** (står for “*Jump if zero*”): Processoren hopper kun til instruktionen på den angivne adresse, hvis indholdet af register 1 er 0 (hvis **reg1** = 0). **JUMPZERO** kan derfor bruges til at vælge, om der skal hoppes. Det kaldes et betinget hop.

Vi ændrer nu Program # 1 fra før til et FIMA -program, der benytter alle typer af instruktioner:

PROGRAM # 2	
ADRESSE	KODELINIER OG KOMMENTARER
000	DATA: \\\ Datablokken starter
001	MEMWRITE 44 1 \\\ Værdien 1 skrives på adressen 44
002	MEMWRITE 45 2 \\\ Værdien 2 skrives på adressen 45
003	CODE: \\\ Kode-sektion starter
004	LOAD 44 R1 \\\ Værdien på adresse 44 (1) \\\ indlæses i register R1
005	LOAD 45 R2 \\\ Værdien på adresse 45 (2) \\\ indlæses i register R2
006	ADD R1 R1 R1 \\\ Indholdet af R1 fordobles og gemmes igen i R1
007	SUB R1 R2 R3 \\\ Forskellen mellem register R1 og R2 \\\ lagres i R3
008	JUMPZERO R3 6 \\\ Hvis indholdet af register R3 er 0, \\\ hoppes til adressen 6
009	STOP \\\ Stop program

Kan du regne ud, hvad programmet gør?

2.6. HØJNIVEAUSPROG

Som vi har set, forstår en CPU kun maskinkode. Kode, det er svært for mennesker at skrive programmer i, fordi den består af tal. Assemblersprog giver maskinkode-instruktionerne symbolske navne såsom **ADD**, **LOAD** og **JUMP**. Disse navne er lettere at huske for mennesker.

Det er dog stadig stadig meget omstændeligt at skrive bare nogenlunde avancerede programmer i assemblerkode. En linie assemblerkode svarer typisk til én maskininstruktion og udfører derfor noget ekstremt simpelt – som at lægge to tal sammen. Enkle handlinger fylder ofte utallige linier assemblerkode. Koden bliver derfor uoverskuelig. Det tager lang tid at skrive alle disse linier, og risikoen for at lave fejl på bare en af linierne er stor.

Blandt andet derfor skriver en moderne programmør som oftest kode i et såkaldt *højniveausprog*, der har indbygget nogle kommandoer, som er mere kraftfulde end assembler-sproget – det kan være

- ◆ *avancerede funktioner*, fx “åbn fil”, “kontakt webserver” eller “vis aktuelt klokkeslæt på skærmen”,
- ◆ *avancerede datastrukturer*, fx lister eller tekststrenge,
- ◆ *avancerede kontrolstrukturer*, fx forgreninger og løkker.

Fortvivl ikke, hvis du ikke ved hvad lister, tekststrenge, forgreninger og løkker er – vi ser meget mere til højniveausprog i kapitel 6. Pointen her er blot, at en programmør er mere effektiv, hurtigere og mere fejlfri, når han programmerer i højniveausprog. Han skal nemlig skrive færre linier kode end med assemblerkode – det gør programmeringen overskuelig. Højniveausprog ligger desuden i notation og udformning tæt på menneskelige sprog og måder at tænke på. Det forekommer derfor oftest mere naturligt at udtrykke en opgave i højniveausprog end som lister af assemblerinstruktioner.

Det eneste problem med højniveausprog er, at en CPU ikke forstår dem. En CPU forstår jo kun maskinkode. Derfor er højniveausprog udstyret med programmer, der omsætter højniveauprogramkode til *lavniveaukode* (assemblerkode eller maskinkode) . Vi ser nærmere på hvordan det sker i afsnit 6.4.

Højniveausprogene bruges til at skrive det meste avancerede software i – fx operativsystemer, internetbrowsere, tekstbehandlingsprogrammer og banksystemer. På den måde har vi en lagstruktur:

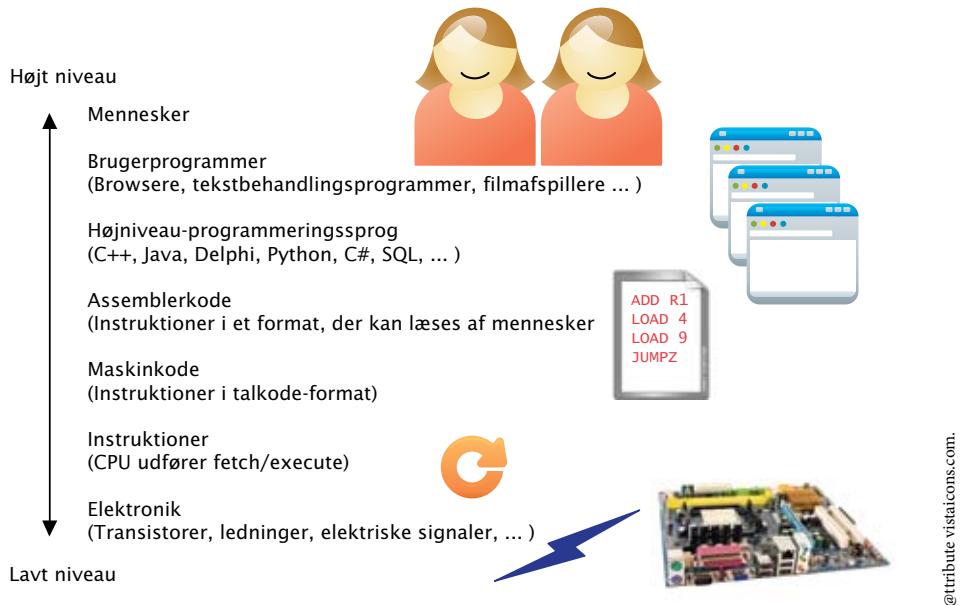


Fig. 2.18. Lagstruktur. Fra et brugernært, maskinuafhængigt, højt niveau af abstraktion til et bruger-fjernt, maskinbundet, lavt niveau af abstraktion.

2.7. NØGLEORD

- ◆ CPU
- ◆ Hukommelse
- ◆ Adresse
- ◆ Program
- ◆ Instruktion
- ◆ Instruktionssæt
- ◆ Fetch/Execute-cyklus
- ◆ ALU
- ◆ ALU-operation
- ◆ Overflow
- ◆ Register
- ◆ Registrene PC og IR
- ◆ Instruktioner repræsenteres som tal i hukommelsen
- ◆ Von Neumann-princippet
- ◆ Maskinkode
- ◆ Assemblerkode
- ◆ Højniveausprog

KAPITEL 3

OPERATIVSYSTEM OG KERNE

Som computerbruger ønsker man, at computeren er *hurtig* og *let* at bruge.

Et *operativsystem* (også kaldet et *styresystem*) hjælper med at opfylde disse ønsker – ved at være en slags mellemmand mellem brugere, brugerprogrammer og computerens hardware-dele. Figur 3.19 illustrerer dette forhold.

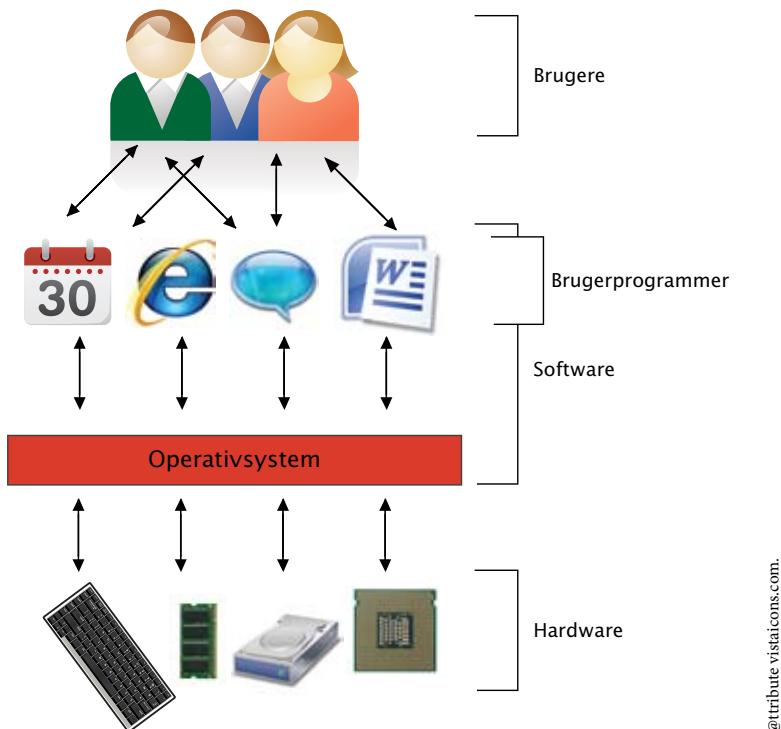


Fig. 3.19. Operativsystem – abstrakt set.

Figur 3.19 viser også hvorfor man undertiden kalder kombinationen af operativsystem og hardware for en *platform*: Brugerprogrammer "kører ovenpå" platformen. Brugerprogrammer er fx internetbrowsere, tekstbehandlingsprogrammer og spil.

I dette kapitel kigger vi på operativsystemets specifikke roller. Operativsystemet hjælper ikke kun med brugbarhed og brugervenlighed – det kan også sikre, at

- ◆ computeren kan køre flere programmer samtidigt,
- ◆ computeren er stabil og sikker,
- ◆ computeren kan deles mellem flere forskellige brugere med forskellige rettigheder og indstillinger.

Et operativsystem gør ikke bare livet skønt for computerbrugere – det hjælper også programører ved at

- ◆ gøre computeren let at skrive programmer til.

Operativsystemet funger nemlig som grænseflade mellem programmer og hardware. Programmer, der kun kommunikerer med operativsystemet, er uafhængige af den underliggende hardware. Det betyder at et program ikke skal skrives om, selvom det flyttes til en maskine med en anden hardware-sammensætning (og samme operativsystem).

EKSEMPEL

OPERATIVSYSTEMER

Der er mange operativsystemer, fx: serien af Windows-systemer, Solaris, Linux, Unix, MacOS, DOS, FreeBSD.

TIP

KÆRT BARN MANGE NAVNE

Man forkorter ofte "operativsystem" til "OS".

Set fra brugersynspunkt er operativsystemet det vindue, hvorigennem de programmer, vi ønsker at bruge, betjenes.

Set fra computerens synspunkt er operativsystemet et kontrolprogram, der kontrollerer al kommunikation mellem programmer. Operativsystemet er ideelt set også det eneste program, der "snakker med hardwaren". Et program, der ønsker at snakke med et andet program eller med et stykke hardware, spørger operativsystemet om lov, og det er operativsystemet, der bestemmer på hvilket tidspunkt og hvordan kommunikationen kan foregå. Samlet kan man sige, at operativsystemet står for at *administre* computersystems mange *systemressourcer*:

SYSTEMRESSOURCE		BRUGES AF ET PROGRAM TIL FX
HARDWARE-RESSOURCER	CPU-tid (clock-cykler)	At blive udført
	Lager (RAM)	Gemme mellemregninger midlertidigt
	Lager (Harddisk)	Gemme data over længere tid, fx som filer
	Anden hardware (I/O-enheder)	Vise ting på skærmen Modtage bruger-indtastninger Sende ting afsted over internettet (via netkortet)
SOFTWARE-RESSOURCER	Mulighed for at kommunikere med et andet program	Samarbejde med andre programmer

Fig. 3.20. Systemressourcer.

Administrationen består i at *allokere og deallokere systemressourcerne* (at allokere betyder at tildele).

ANALOGI

REGERING OG POLITI

Et operativsystem er som et lands regering. En regering skaber og kontrollerer rammerne for, hvordan fornuftige ting kan foregå (folk kan gå på arbejde, lave fritidsaktiviteter, der værnes om miljøet, mv.). Regeringen sørger også for at fordele landets ressourcer rimeligt mellem indbyggerne. Sidst, men ikke mindst, sørger regeringen for – med lovgivning og politi – at forhindre kriminalitet.

Tilsvarende bestemmer et operativsystem over alle systemressourcer, fordeler dem rimeligt mellem programmer, der ønsker at bruge dem, og skaber rammerne for at programmer trygt og sikert kan afvikles samtidigt på computeren – uden at eventuelle programmer med fejl i eller ondsindede programmer (såsom virus) kan gøre skade.

Vi er vist klar til en opsummering:

ORDFORKLARING

OPERATIVSYSTEM

Grundlæggende program installeret på en computer. Operativsystemet administrerer systemressourcer – blandt andet adgang til og brug af hardware. Programmer snakker med hinanden og med hardwaren via operativsystemet.

3.1. KERNE

Den centrale del af operativsystemet kaldes kernen. Hvor operativsystemet kan siges at være en platform for brugerprogrammer, kan kernen siges at være en platform for resten af operativsystemet. I hele dette kapitel vil vi kun komme ind på de af operativsystemets opgaver, som løses i kernen.

ORDFORKLARING
KERNE Den del af operativsystemet, der administrerer systemressourcer.

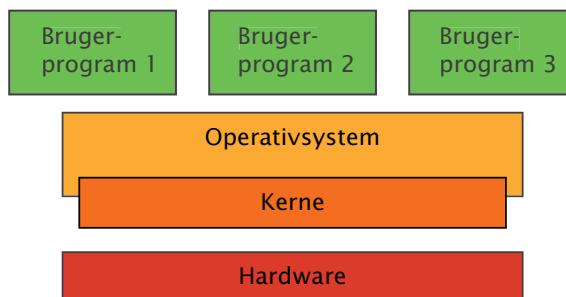


Fig. 3.21. Kernen er et program. Kernen er det laveste abstraktionslag i computeren, der er implementeret i software. Kernen er den del af operativsystemet, der kommunikerer direkte med hardwaren.

Når computeren startes, gøres kernen funktionsklar ved en kompliceret proces, der kaldes bootstrapping – mere herom i afsnit 3.4. Herefter foretager kernen sig ikke noget på egen hånd. Kernen reagerer derimod på to slags eksterne begivenheder:

- ◆ *systemkald* fra software (programmer),
- ◆ *afbrydelser* fra hardware.

Vi skal nu se, hvad disse betegnelser dækker over.

3.1.1. I/O OG AFBRYDELSER

I/O-enheder kan ikke forudsige, hvornår de bliver aktiveret. Det gælder fx et tastatur. Computeren kan sagtens være tændt, uden at vi taster løs. Vi forventer dog stadig, at der sker noget i det øjeblik, vi beslutter os for at trykke på en tast. Er computeren gået på pauseskærm, forventer vi, at pauseskærmen forsvinder når musen flyttes. Har vi holdt pause fra et tekstbehandlingsprogram, forventer vi, at bogstaver dukker op på skærmen, straks vi genoptager skrivningen.

Computeren kan altså ikke på forhånd planlægge efter, hvornår dens enheder bliver aktiveret.

Når en I/O-enhed bliver aktiveret, signalerer den det til processoren ved at sende en meddeelse over systembussen. Processoren kan så reagere passende på beskeden, fx stoppe pauseskærmen eller vise bogstaver på skærmen.

ORDFORKLARING

AFBRYDELSE

At en I/O-enhed bliver aktiveret og sender et signal til processoren for at fortælle det.

Hændelsen kaldes en afbrydelse, fordi hardwareenheden spørger om lov til at afbryde CPU'en i det arbejde, den ellers var i gang med. Den meddelelse, enheden sender til CPU'en, kaldes en *interrupt request* (IRQ). Når CPU'en bliver afbrudt, overgiver den kontrollen til en interrupt handler, en komponent i operativsystemet, der håndterer afbrydelser.

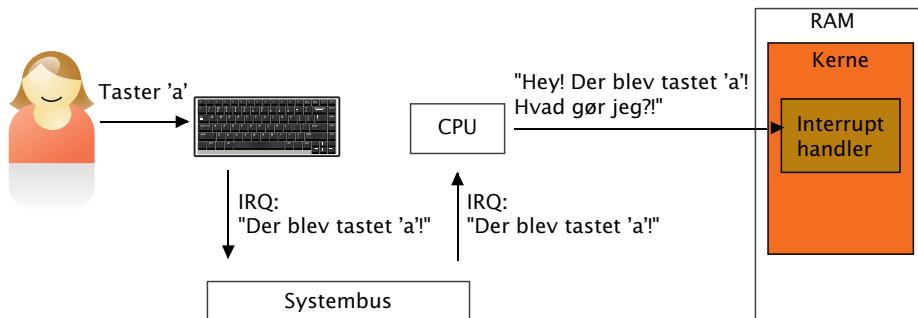


Fig. 3.22. Interrupt request (IRQ) og interrupt handler.

Interrupt handleren sørger allførst for at gemme de oplysninger, der skal bruges for at CPU'en senere (når afbrydelsen er blevet håndteret) kan genoptage sit arbejde uden problemer.

Dernæst håndterer interrupt handleren afbrydelsen. Hvis der blev fx blev trykket på tasten "a", omsætter interrupt handleren signalet fra tastaturet til talkoden for "a" (den kunne fx være 00111110) og meddeler talkoden til CPU'en. CPU'en kan så reagere passende på tastetrykket. Hvis brugeren fx er i gang med et tekstbehandlingsprogram, sendes beskeden om, at der blev trykket "a", videre til tekstbehandlingsprogrammet. Tekstbehandlingsprogrammet noterer sig, at der blev trykket "a", og kan herefter reagere på tastetrykket i overensstemmelse med sin programkode, fx sørge for at vise symbolet "a" på skærmen.

Når det program, som brugeren har henvendt sig til med sit "a", med sikkerhed er blevet informeret om, at der blev tastet "a", lader interrupt handleren CPU'en fortsætte sit oprindelige arbejde.

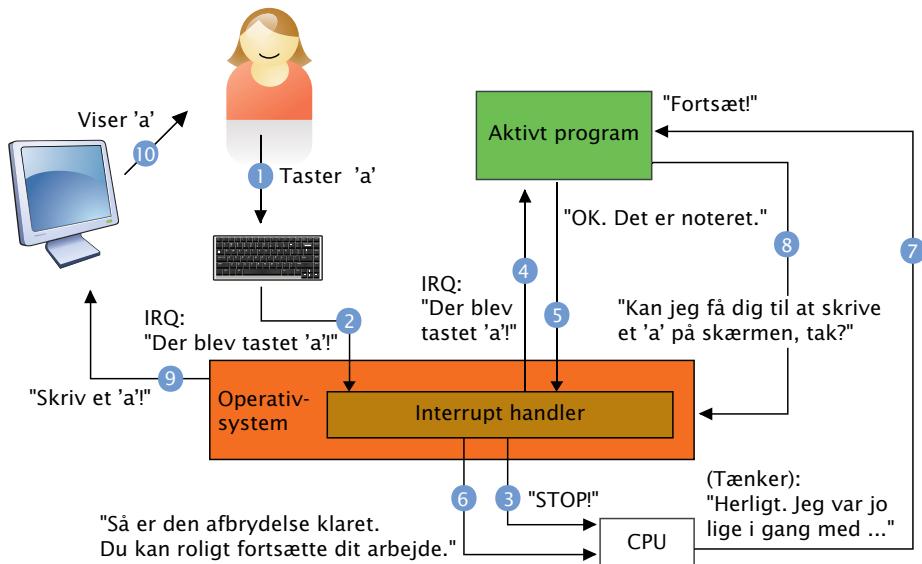


Fig. 3.23. Forløbet fra en bruger taster 'a' på tastaturet mens et tekstbehandlingsprogram er åbent og indtil skærmen viser 'a'. (Selvom der er 10 trin, er denne figur stadig en forsimpling! Fx er hele Figur 3.22 indeholdt i trin 1 og 2 ovenfor).

3.1.2. KERNETILSTAND OG BRUGERTILSTAND

Et operativsystem tillader flere brugere at bruge systemet samtidigt og tillader flere programmer at køre samtidigt. Det er derfor afgørende, at programmer – ved en fejl eller af ond vilje –

*ikke kan komme til at slette eller ændre i andre programmers data eller kode
eller tage kontrol over hardwaren.*

Førhindrer man ikke dette, har man et situation, man kunne kalde et "utilregneligt, nedbrudsplaget hacker-paradis" hvor ethvert resultat, et program har beregnet, *kunne* være forkert, og fjendtlige personer har fri adgang til at kopiere digitale signaturer og læse netbanken, stjæle private emails og lave afpresning eller kort og godt ødelægge ferievideoer og familiealbummet. Selve meningen med at have en computer til at hjælpe sig skrumper derfor voldsomt i denne situation.

Et operativsystem sørger derfor for at afværge situationen. Det sker ud fra filosofien “med magt følger ansvar”, et argument, der kan bruges i politik og altså også indenfor operativsystemer. Operativsystemet opdeler typisk maskinkode i to typer instruktioner (husk at processoren konstant afvikler instruktioner, en for en, som er lagret i RAM'en):

- ◆ *Privilegerede instruktioner*: Kun operativsystemet må udføre disse instruktioner!
- ◆ *Ikke-privilegerede instruktioner* (opfindsomt navn!): Alle programmer må udføre disse instruktioner.

Privilegerede instruktioner har magten til at udføre kraftfulde operationer, der kan få fatale konsekvenser for computeren, hvis de bruges forkert eller ondsindet. Det kræver særlige rettigheder (særlige privilegier) at udføre en privilegeret instruktion, deraf navnet. Programfunktioner, der kræver privilegerede instruktioner, kaldes *kritiske* funktioner.

Ikke-privilegerede instruktioner kan ikke skade computeren i nævneværdigt omfang ved misbrug eller forkert brug.

Figur 3.25 på side 51 opilater typiske privilegerede og ikke-privilegerede instruktioner.

Det er CPU'en, der udfører instruktioner. Hver gang CPU'en møder en privilegeret instruktion, er den på vagt og spørger sig selv: “Har jeg lov at udføre denne instruktion?” Det kommer jo an på, om det er tilfældigt brugerprogram, der beder om at få instruktionen udført, eller operativsystemet. For at skelne mellem de to situationer, har CPU'en to tilstande: *kerneltilstand* og *brugertilstand*.

ORDFORKLARING

KERNELTILSTAND

CPU-tilstand, hvor alle instruktioner må afvikles – også privilegerede.

ORDFORKLARING

BRUGERTILSTAND

CPU-tilstand, hvor kun ikke-privilegerede (ufarlige) instruktioner må afvikles.

Operativsystems kerne kører i kerneltilstand, mens resten af operativsystemet kører i brugertilstand. Alle andre programmer kører i brugertilstand. CPU'en aflæser sin tilstand i et særligt status-register. Status-registeret har plads til 1 bit. Er bitten sat til 1, køres i kerneltilstand, er den sat til 0, køres i brugertilstand. Det er *kun* operativsystems kerne, der kan skifte status-bitten fra 0 til 1.

TIL SÆRLIGT INTERESSEREDE

SIKKERHEDSNIVEAUER

Med opdelingen i bruger- og kernetilstand siges systemet at have 2 sikkerhedsniveauer. I visse systemer opererer man med en finere inddeling, fx i 4 eller flere sikkerhedsniveauer. Jo højere sikkerhedsniveau, jo flere rettigheder. Til gengæld har kun meget få programmer adgang til at afvikle under højeste sikkerhedsniveau.

Undertiden kan brugerprogrammer have brug for at benytte kritisk systemfunktionalitet. Det gøres gennem *systemkald*.

ORDFORKLARING

SYSTEMKALD

At et brugerprogram med begrænsede rettigheder beder kernen om at udføre en systemfunktion.

Når et program foretager et systemkald, overdrages kontrollen til kernen. Kernen kontrollerer, om den funktion, programmet vil have udført, er tilladelig. Hvis den er det, udfører kernen opgaven for programmet, hvorefter kontrollen gives tilbage til programmet. Beder programmet om noget åbenlyst fjendtligt eller uhensigtsmæssigt (fx "slet harddiskens indhold"), undlader kernen at udføre funktionen – det resulterer i en programfejl.

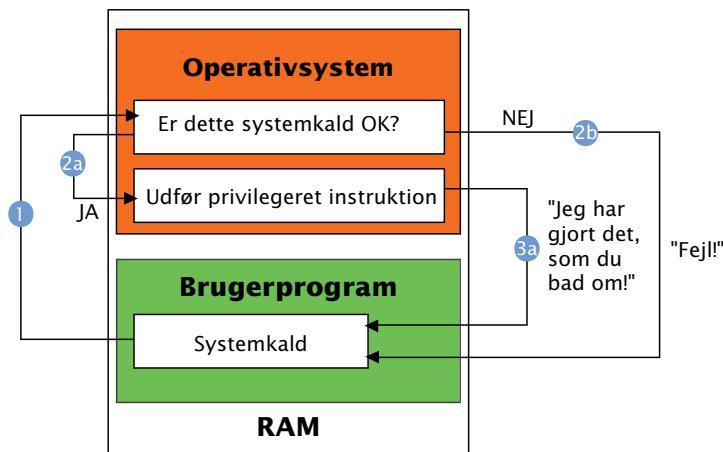


Fig. 3.24. Systemkald.

Et systemkald kaldes undertiden for en software-genereret afbrydelse, fordi CPU'en skal skifte tilstand fra brugertilstand til kernetilstand for at udføre den ønskede funktion. Ker-

netilstand kaldes på engelsk “supervisor mode” eller “monitor mode”, fordi kernen *overvåger* alle systemkald.

Hvis et program, der er under afvikling, aldrig foretager systemkald – eller værre endnu: pga en programmeringsfejl ender i en *uendelig løkke* (udfører de samme ting igen og igen uden at kunne stoppe) – hvordan får operativsystemet så kontrollen igen?

Svaret er, at CPU'en har en indbygget *timer*, der afbryder ethvert program, der har kørt i i for lang tid. “For lang tid” defineres fra CPU til CPU, men det kunne fx være hvert millisekund (=1000 gange i sekundet). Ved et sådant timeout giver CPU'en kontrollen til operativsystemet.

PRIVILEGEREDE INSTRUKTIONER	IKKE-PRIVILEGEREDE INSTRUKTIONER
I/O-instruktioner	-
Skrivning i eller læsning af andre processers data eller kode	Skrivning i og læsning af egen data eller kode
Indstil timeren	-

Fig. 3.25. Eksempler på privilegerede og ikke-privilegerede instruktioner.

3.2. LAGERADMINISTRATION

Når et brugerprogram kører, har det brug for plads i hukommelsen til

- ◆ (dele af) selve programkoden
- ◆ midlertidige data
- ◆ beregninger

EKSEMPEL

TEGNEPROGRAM

Når et tegneprogram kører, har det brug for plads i hukommelsen til selve tegneprogrammet (programkoden), alle de billedfiler, der arbejdes på (midtildige data), samt til at foretage operationer på billedfilerne, fx skyggelægninger, farveskift mv. (beregninger).

Da lagerpladsen i hukommelsen er en *kritisk systemressource*, er det operativsystemet, der administrerer den. Derfor er brugerprogrammer nødt til at bede om den plads, de har brug for, gennem systemkald. Lageradministration er den OS-opgave, der består i at fordele den begrænsede plads i hukommelsen mellem mange aktive programmer og nødvendig (midtildig) data.

Operativsystemet kan tildele lagerpladsen i hukommelsen til programmerne *dynamisk*. Det vil sige, at OS ikke på forhånd behøver vide, hvor meget plads et program vil bruge – og at et kørende program løbende kan få ekstra plads tildelt, om nødvendigt. En tildeling af en portion hukommelse til et program kaldes en *allokering*. Man siger, at operativsystemet udfører *dynamisk lagerallokering*.

Udover at tildele pladsen i hukommelsen, så alle kørende programmer har plads nok, skal OS også sørge for, at de enkelte programmer ikke ved en fejl kommer til at benytte lagerområder, som andre kørende programmer har fået tildelt. Det er både for at undgå uforudsigelig og fejlagtig opførsel og for at sikre computeren mod, at ondsindede programmer med vilje ødelægger data eller kode i andre kørende programmer. Hvert kørende program får derfor tildelt sit eget *adresserum*.

ORDFORKLARING

ADRESSERUM

Et kørende programs adresserum er den samling adresser i hukommelsen, et kørende programmet kan læse og skrive i. OS sikrer, at de enkelte adresserum er isoleret fra hinanden.

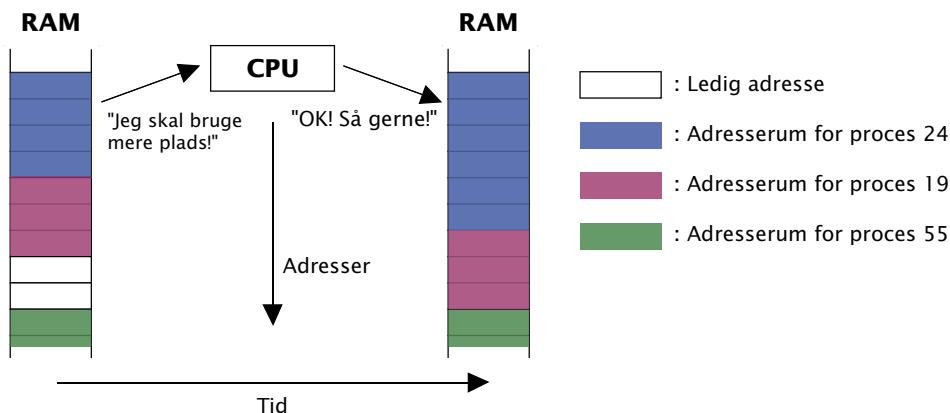


Fig. 3.26. Adresserum og dynamisk lagerallokering. Adresserummets størrelse og fysiske placering kan løbende ændres med dynamisk lagerallokering. Her omrokes fx proces 19 for at give ekstra plads til proces 24. (Kørende programmer kaldes processer, og vi hører mere om dem i afsnit 3.3).

TIL SÆRLIGT INTERESSEREDE

LOGISK OG FYSISK ADRESSERUM

Man taler om fysiske og logiske adresserum. En proces' *fysiske adresserum* er de faktiske adresser, processen har til rådighed. En proces' *logiske adresserum* er processens "egen opfattelse af adresserummet". En proces regner altid med at have et sammenhængende adresserum, der starter på adresse 0.

En proces med behov for et adresserum på 1000 adresser regner fx med, at dens adresserum strækker sig fra adresse 0 til adresse 999. CPU'en sørger for, at processens adressering virker korrekt – også selvom det logiske adresserum ikke svarer til det fysiske. Processen kan fx i den fysiske hukommelse have adresserummet fra adresse 1500 til 2499 eller endda et fysisk sammenhængende adresserum, såsom adresserne 0 – 499 og 1500 – 1999.

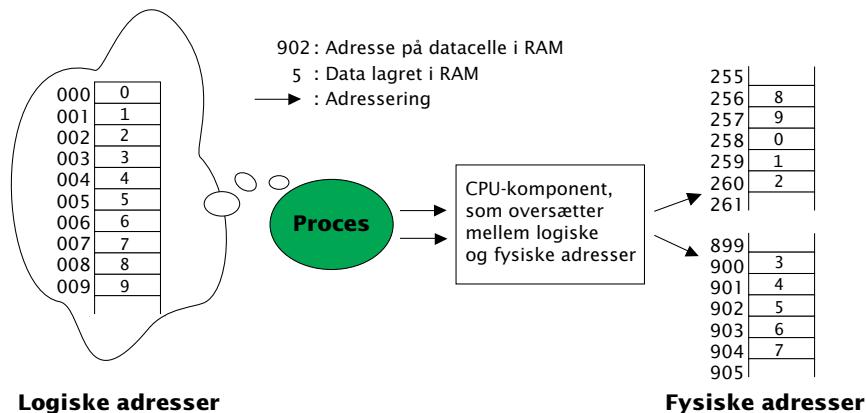


Fig. 3.27. Logisk og fysisk adresserum.

TIL SÆRLIGT INTERESSEREDE

VIRTUEL HUKOMMELSE

Ikke alene kan OS levere logiske adresserum til processer. OS kan også simulere en meget større RAM, end der fysisk er til stede. Hvis en kørende proces skal bruge plads, men alt er optaget, *swapper* (med et sigende ord) OS simpelt hen (dele af) andre processer, der i det øjeblik "ikke er aktive", ud på harddisken og frigør derved plads i RAM. Når den ud-swappede proces på et tidspunkt skal aktiveres, *swappes* den ind i RAM igen – evt. med andre ud-swapninger til følge. Alt dette står OS for, uden at processerne behøver bekymre sig om hvordan. Den udvidede pladsmængde i RAM, som et OS med denne teknik simulerer, kaldes *virtuel hukommelse*. Prisen for at bruge virtuel hukommelse er langsommelighed – at swappe processer ind og ud fra harddisken er tidskrævende.

fortsættes...

TIL SÆRLIGT INTERESSEREDE

VIRTUEL HUKOMMELSE (FORTSAT)

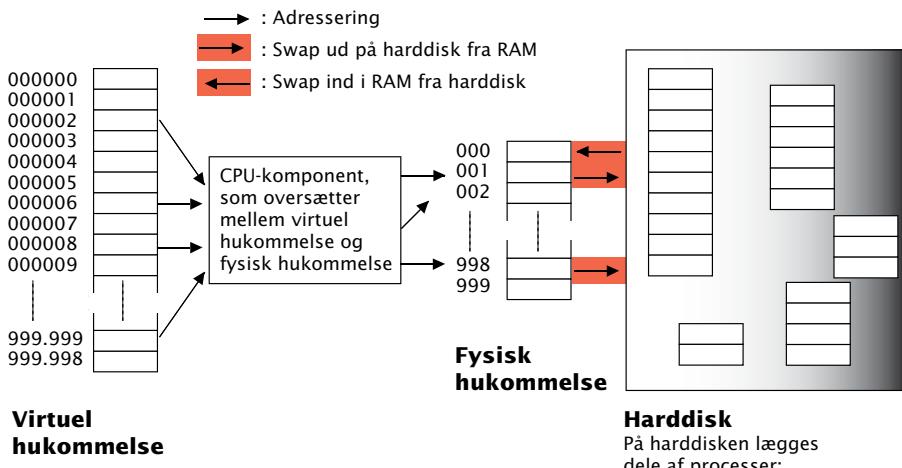


Fig. 3.28. Den virtuelle hukommelse er meget større end den faktisk tilstedevarende, fysiske hukommelse.

TIL SÆRLIGT INTERESSEREDE

FILSYSTEMER

For at lagre data (filer) på en harddisk kræves overvejelser, der ligner måden CPU'en håndterer problemstillingen "fysisk/logisk adresserum". Et *filsystem* er en speciel komponent (et samarbejde mellem harddisk og OS), der indeholder information om, *hvor-dan* filer læses og skrives, og *hvor* filerne fysisk er på harddisken. Hver gang disken tilgås, aktiveres filsystemet.

En sammenhængende fil kan sagtens lagres på flere sammenhængende diskområder, ligesom en proces kan have et fysisk usammenhængende adresserum. Fænomenet kaldes fragmentering. Filsystemet sørger for, at de separate fil-fragmenter stadig fortolkes som én fil. En meget fragmenteret disk er langsom. Derfor skal en disk af og til defragmenteres.

3.3. PROCESSER

Man skelner mellem programmer og *processer*. Ved et program forstås simpelt hen den mængde kode, der udgør programmet. Ved en proces forstår man et program, der er under afvikling.

ORDFORKLARING

PROCES

Program, der er indlæst i hukommelsen og under afvikling.

Programmer er *passive*, mens processer er *aktive*. En proces rummer, uddover programkode, information om sin tilstand, bl.a om hvor den er nået til i udførelsen af sin kode.

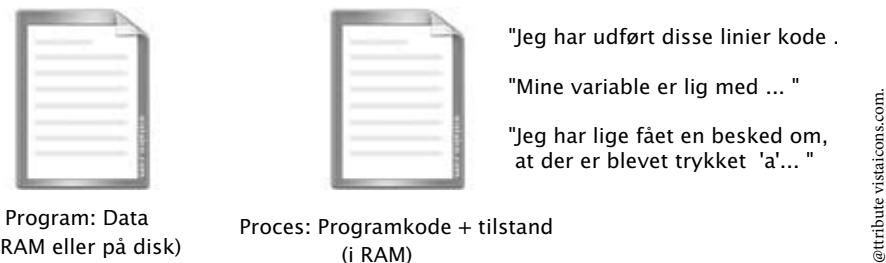


Fig. 3.29. Program og proces.

I beskrivelsen af processer bruger vi en gennemgående bageanalogi: Et program er som en bageopskrift. Trin for trin står der, hvad bageren skal gøre. Hvor programmet er opskriftens tekst, altså et stykke papir, der ikke ændrer sig, er den tilhørende proces selve det at udføre bageopskriftens anvisninger.

En proces kan sagtens indlæse andre processer i hukommelsen. Et kørende program kan derfor undertiden bestå af flere processer.

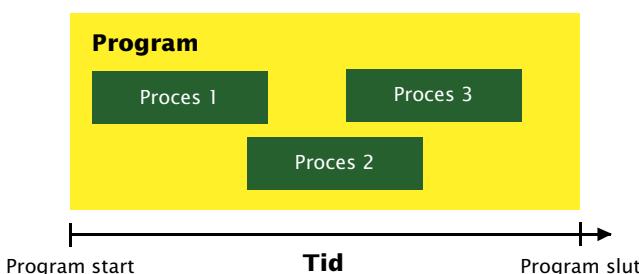


Fig. 3.30. Program med flere processer.

I en bagesituation kan man sagtens forestille sig, at der skal både røres æg ud i dejen og fremstilles glasur; to helt uafhængige processer, men begge nødvendige for at udføre det samlede program. Et program kan opfattes som en samling af processer, der skal udføres, eventuelt i en bestemt rækkefølge.

To processer kan også repræsentere to kopier (*instanser*) af samme program, fx to instanser af et kontorprogram til regneark, startet af to forskellige brugere.

Vi husker fra forrige afsnit, at et program under afvikling har brug for plads i hukommelsen til (dele af) selve programkoden, midlertidige data og udførelse af beregninger. Med procesbegrebet kan vi præcisere dette: Programmet har brug for plads til sine processer, og hver proces har brug for kunne operere på midlertidige data og foretage beregninger.

For at koordinere deres fælles arbejde skal processer kunne “snakke sammen”. Lad os sige at to processer, en glasur-ansvarlig og en bage-ansvarlig, sammen står for en kage med glasur. De har tydeligvis brug for at koordinere deres handlinger, så den glasur-ansvarlige ikke påfører glasuren før kagen er bagt færdig.

FORMÅL MED PROCES-KOMMUNIKATION	
Korrekt rækkefølge	Opgaver skal udføres i den rigtige rækkefølge i forhold til hinanden.
Informationsdeling	Arbejde udført af en proces skal kunne overdrages til en anden (fx resultatet af en beregning).
Effektivitet	Opgaver, der fordeles som (evt. uafhængige) delopgaver på flere kommunikerende processer, kan udføres hurtigere.
Overskuelighed	Program kan opdeles på en overskuelig måde i logisk sammenhængende processer med hver sin delopgave.

Fig. 3.31. Proces-kommunikation.

Somme tider kaldes processer, der slet ikke har brug for at samarbejde med andre, for isolerede processer. I bage-analogien kunne det være en proces, der blot pudser vinduer i baggrunden, helt uafhængigt af kagebagningen. Eller slet ikke har med bagningen at gøre.

3.3.1. PROCESTILSTANDE OG SKEDULERING

Programafvikling sker ved, at hvert program løbende indlæser processer i hukommelsen. Processoren sørger for at afvikle de indlæste processer. Processoren kan kun afvikle én instruktion fra én proces ad gangen. Der kan imidlertid sagtens være mange processer indlæst på én gang.

Processerne skiftes til at få udført en række instruktioner. Når en proces får udført instruktioner, siges den at være i tilstanden *running* (kørende). Når en proces er *running*, er alle andre aktive processer “sat på pause”. Når en proces har fået afviklet alle sine instruktioner, siges den at terminere (afslutte), og dens tilstand skifter til *terminated*.

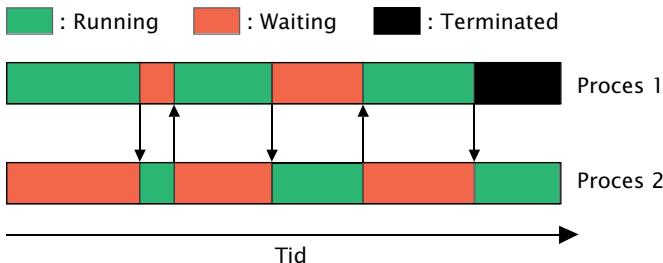


Fig. 3.32. Et afviklingsforløb med 2 aktive processer, der skiftes til at vente på hinanden.

En proces, der er indlæst i hukommelsen, men som ikke er i gang med at blive afviklet, er enten *ready* (klar) eller *waiting* (ventende). En klar proces er parat til at få udført instruktioner, så snart CPU'en giver tid til det. Processer i ventetilstand er ikke parat til få udført instruktioner, men venter på noget uden for processen selv, for eksempel brugerindtastninger eller at en anden proces beregner et resultat, der skal bruges. Når det, en ventende proces venter på, indtræffer, skifter processen tilstand til klar.

Det er CPU'en, der bestemmer, hvordan den fordeler sin tid mellem processer, der er *ready*.

ORDFORKLARING

PROCESSKIFT

At CPU'en skifter fra at udføre instruktioner for en proces A, der er *running*, til at udføre instruktioner for en anden proces B, der er *ready*. Ved proces-skiftet skifter A tilstand til *ready*, og B skifter tilstand til *running*.

En proces er som regel i tilstandene *ready* og *waiting* mange gange, før den er helt udført.

ORDFORKLARING

SKEDULERING

At CPU'en fordeler sin tid mellem processer, der er *ready*, efter et mønster, der både er effektivt og sikrer, at alle processer "kommer til".

Ordet skedulering kommer af det engelske "scheduling", der betyder planlægning. Skeduleringen bestemmes med en særlig algoritme, skeduleringsalgoritmen, der kan varierere fra CPU til CPU. En ting ved vi dog fra tidligere: Hvis en proces kører "for længe", afbryder CPU'en timer processen og skifter til operativsystemet.

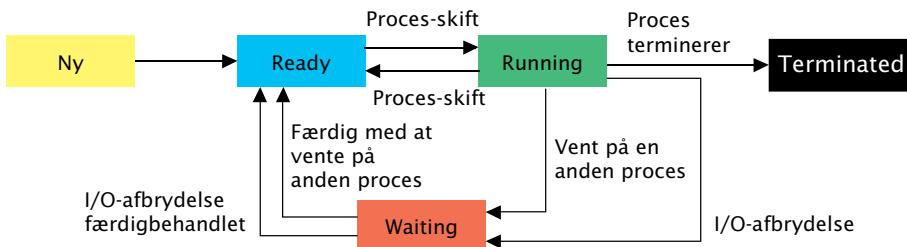


Fig. 3.33. De mulige processtilstande.

En proces, der er under afvikling, er i en af de 3 tilstande *ready*, *running* eller *waiting*. Når processoren bliver afbrudt af hardware, afbrydes den proces, der er *running*, og der skiftes til en interrupt handler.

Processkiftene sker så svimlende hyppigt, at det for mennesker ligner, at alle processer kører samtidigt. Fx virker det naturligt at høre musik fra et musikafspilningsprogram, samtidigt med at man downloader en film fra nettet med et fildelingsprogram, skriver på en sjov datalogirapport i et tekstbehandlingsprogram og chatter med vennerne via et chatprogram. I virkeligheden skiftes disse 4 kørende processer – og en lang række andre (OS-)processer – til at afvikle instruktioner, en proces ad gangen.

3.3.2. SAMTIDIGHED OG SYNKRONISERING

For at kunne samarbejde med hinanden må samarbejdende processer afvikles ”samtidigt” og have mulighed for at kommunikere.

ORDFORKLARING

SAMTIDIGHED

At flere processer afvikles på samme tid og har mulighed for at samarbejde.

Samtidighed giver problemer, som vi illustrerer via programmet ”Bageprogram # 1” nedenfor. Processer kommunikerer via beskeder. For processer gælder, at

$$\text{kommunikation} = \text{beskedudveksling}$$

Vi vender straks tilbage til bageriet for at illustrere dette princip. Her er processerne P og Q er ansvarlige for hhv. kagebagning og glasur-påføring. Processerne sættes til at afvikle. Vi kan ikke vide, hvilken proces CPU'en først sætter til at afvikle. Hvis vi derfor koder P og Q som følger

BAGEPROGRAM # 1	
P = BAGEANSVARLIG	Q = GLASURANSVARLIG
KODE:	KODE:
<Lav kagedej>	<Påfør glasur>
<Bag kage>	

er koden utilstrækkelig, fordi vi risikerer følgende (forkerte!) afviklings-sekvens:

P: <Lav kagedej>

Q: <Påfør glasur>

P: <Bag kage>

I denne afviklingssekvens bliver glasuren påført den ubagte kagedej og derefter bagt – med en gyselig “kage” som resultat. P og Q er altså nødt til at koordinere, at P bliver helt færdig, inden Q tager over. Det kaldes *synkronisering*.

ORDFORKLARING

SYNKRONISERING

At to processer afstemmer rækkefølge for udførelse af fælles opgaver ved at udveksle beskeder.

Beskedudveksling foregår med de to *primitiver* **Send** og **Modtag**:

- ◆ **Send(A, besked1)**: Sender beskeden **besked1** til processen A.
- ◆ **Modtag(B, besked2)**: Afventer, at processen B afsender beskeden **besked2**.

Udstyrer vi P og Q med primitiver for at koordinere deres bageindsats, kunne deres kode se ud som følger:

BAGEPROGRAM # 2	
P = BAGEANSVARLIG	Q = GLASURANSVARLIG
KODE:	KODE:
<Lav kagedej>	<Modtag(P, "Kage klar til glasur")>
<Bag kage>	<Påfør glasur>
<Send(Q, "Kage klar til glasur")>	

Her er eneste mulige afviklings-rækkefølge følgende:

P: <Lav kagedej>
P: <Bag kage>
P: <Send(Q, "Kage klar til glasur")>
Q: <Modtag(P, "Kage klar til glasur")>
Q: <Påfør glasur>

Det er fordi Q ikke foretager sig noget, før end P sender beskeden “**Kage klar til glasur**”.

3.3.3. ASYNKRON BESKEDUDVEKSLING

I eksemplet “Bageprogram # 2” ovenfor er der tale om *synkron* beskedudveksling: Q venter med at udføre <Påfør glasur> til en besked modtages. Processer kan også kommunikere asynkront.

Som et eksempel på asynkron beskedudveksling ønsker vi nu, at glasur-processen Q udnytter tiden til at vaske op, mens P bager kagen færdig. Vi omkoder derfor Q med en ny type **Modtag**. Den nye type **Modtag** hedder **AsynkronModtag**, og når Q møder en **AsynkronModtag**, kan Q godt fortsætte sit arbejde, hvis der ikke umiddelbart modtages nogen besked.

BAGEPROGRAM # 3
Q = GLASUR-ANSVARLIG
KODE
GENTAG <Vask op> INDTIL <AsynkronModtag(P, "Kage klar til glasur")> DEREFTER <Påfør glasur>

Både **Send** og **Modtag** kan laves både synkront og asynkront:

- ◆ *Synkron send*: Afsender blokerer (“sættes på pause”), indtil modtager har fået sin besked.
- ◆ *Synkron modtag*: Modtager blokerer, indtil besked modtages.
- ◆ *Asynkron send*: Afsender fortsætter ufortrødент afvikling efter afsendelse af besked.
- ◆ *Asynkron modtag*: Når modtager når til det sted i kodden, hvor **Modtag**-prioritivet er, modtages enten en gyldig besked, eller også fortsættes ufortrødент uden modtagelse af besked.

Synkron kommunikation kaldes også blokerende, mens asynkron kommunikation kaldes ikke-blokerende.

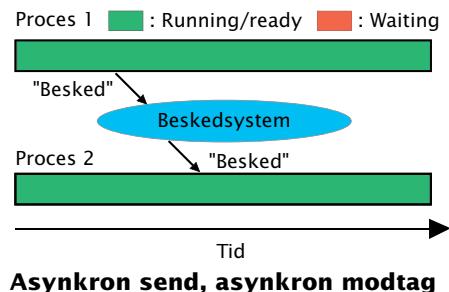
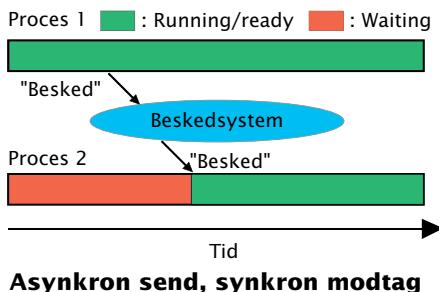
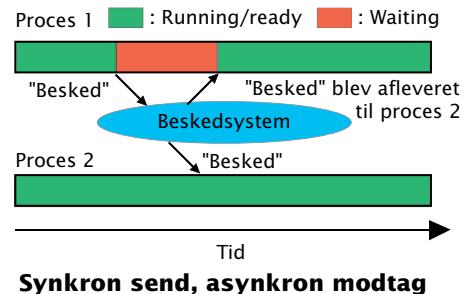
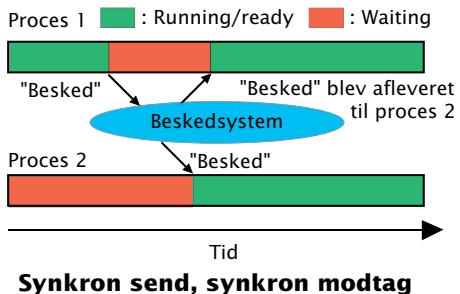


Fig. 3.34. Synkron og asynkron beskedudveksling.

WWW

TRÅDE

Processer kan med fordel opdeles i mindre bidder, der deler hovedprocessens adresserum og midlertidige data. Sådanne bidder kaldes tråde. I sammenhænge, hvor hastighed spiller en rolle, fx lyd og film, kan det være særligt effektivt at opdele en proces i tråde, der afvikles samtidigt. Vi ser nærmere på tråde på **WWW**.

3.4. BOOTSTRAPPING

Når en computer skal starte – fx efter at have været slukket, eller efter en genstart – skal operativsystemet startes. Som vi ved, læser computeren sine instruktioner fra hukommelsen, og hukommelsen ryddes, når computeren slukkes. Et tilsyneladende paradoks – hvordan skal computeren starte uden instruktioner at udføre?

Løsningen er at starte computeren uden brug af hukommelsen: Det kaldes at *boote*.

ORDFORKLARING

BOOTSTRAPPING, BOOTE OG BOOTSEKVENS

Opstartsforløbet, fra computeren tændes, til operativsystemet er indlæst og startes, kaldes bootstrapping. Man siger også, at computeren booter. De skridtvise operationer, computeren udfører, mens den booter, kaldes for bootsekvensen.

“Bootstrapping” refererer oprindeligt til, at computeren “trækker sig selv op ved støvlestroppe” (*boot strap* betyder støvlestrop).

Når computeren booter, udføres allerførst instruktioner fra et særligt lager *uden for hukommelsen*, nemlig på en bestemt chip på bundkortet, kaldet bootchippen. Computeren er bygget sådan, at straks strømmen tilsluttes, startes det program, der er lagret på bootchippen. Programmet kaldes *bootstrap-programmet* og er forholdsvis simpelt (men ikke så simpelt som at binde snørebånd). Bootstrap-programmet har 3 trin:

1. tjekke, at hardware-enheder fungerer,
2. *initialisere* hardware (at initialisere betyder at “klargøre”),
3. indlæse kernen i hukommelsen og starte den.

Hvert trin afhænger af, at det foregående trin blev fuldført fejlfrit. For eksempel kan kernen ikke indlæses i hukommelsen, hvis hukommelsen er defekt eller forkert initialiseret. Trin 1 kaldes ofte for Power-On Self-Test (forkortet: POST).

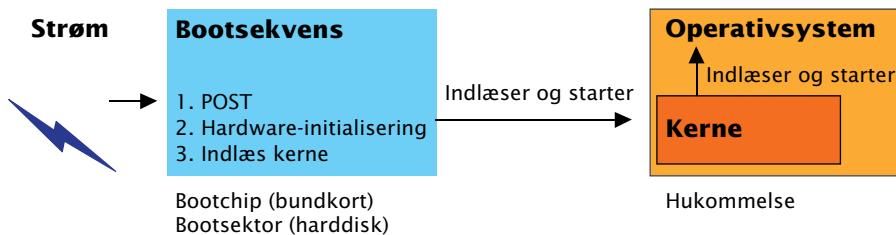


Fig. 3.35. Computerens opstartsforløb fra slukket til tændt.

Figuren herunder uddyber de tre trins funktion:

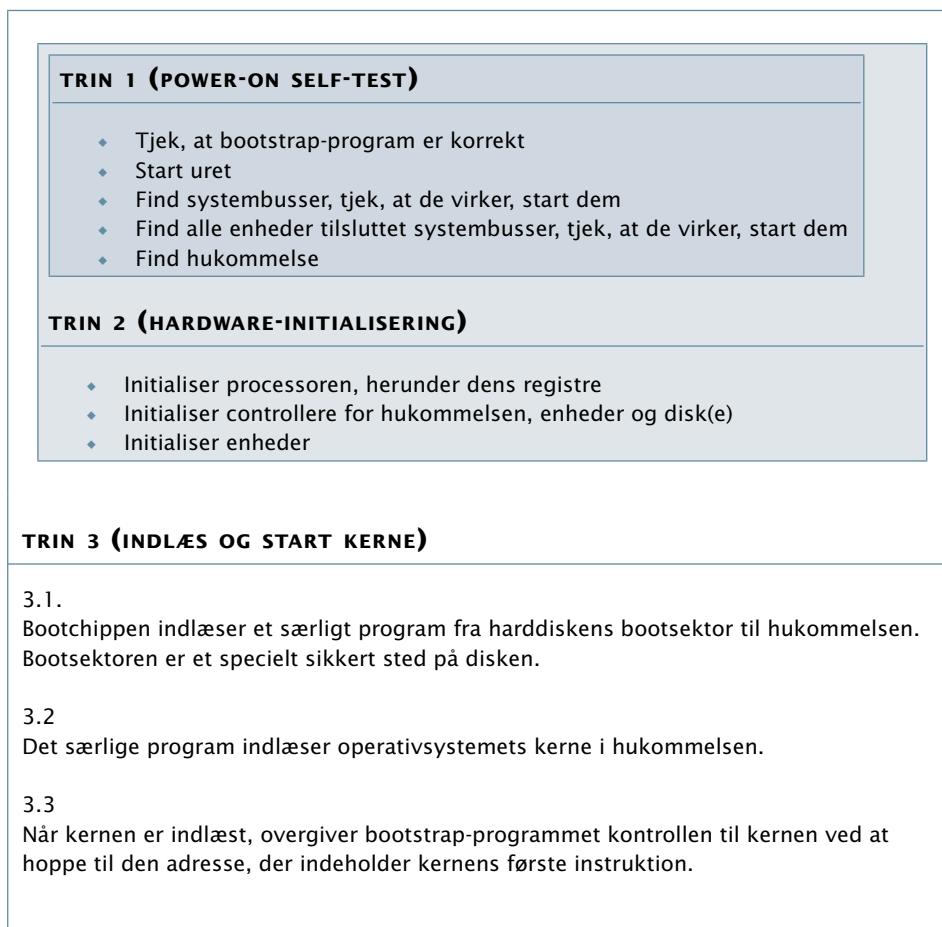


Fig. 3.36. Bootstrapping i tre trin.

Når kernen først er indlæst, sørger den for at indlæse resten af operativsystemet.

ADVARSEL

De enkelte opgaver under trin 1 og trin 2 udføres i forskellig rækkefølge alt efter bootstrap-program. Trin 3 udføres altid i samme rækkefølge. Begrebet "Power-On Self-Test" bruges nogle gange om dele af trin 2.

Hvis bootstrap-programmet ikke fungerer korrekt, kan computeren ikke starte. Derfor lagrer man typisk en del af bootchippens kode i read-only memory (ROM) . På den måde ødelægges bootstrap-programkoden ikke ved en utilsigtet overskrivning eller pga. virus.

Normalt ønsker man, at computeren har mulighed for at boote på forskellige måder. Det giver bl.a. følgende fordele:

- ◆ *Flere operativsystemer og boot fra flere medier:* Hvis man har flere operativsystemer installeret, kan man som bruger ofte vælge, hvilket OS der skal startes (mellem bootsekvensens trin 2 og trin 3). Man kan endda vælge at boote fra andre medier end harddisken, fx en CD, en USB-nøgle eller via et lokalnetværk.
- ◆ *Boot-kodeord:* Man kan beskytte computeren mod ubudne gæster med et kodeords-tjek inden indlæsning af kernen.
- ◆ *Brugerdefinerede indstillinger:* Som bruger kan man ændre og gemme en ønsket bootsekvens.

Hvis hele bootstrap-programmet, inklusive bootsekvens-indstillinger, var lagret i boot-chip-ROM, ville bootchippen skulle udskiftes hver gang bootsekvensen skulle ændres. Derfor gemmer man bootsekvens-indstillingerne i et særligt skrivbart lager på bundkortet (typisk såkaldt flashmemory). Dette lager slettes ikke når strømmen slukkes.

Tilsammen kaldes bootchippen og boot-programmet på disken for bootloaderen. På PC'er er hovedkomponenten i bootloaderen det såkaldte Basic Input Output System, BIOS, der i hovedtræk varetager trin 1 og 2 af bootsekvensen.

3.5. NØGLEORD

- ◆ Operativsystem
- ◆ Systemressource
- ◆ Kerne
- ◆ Afbrydelse
- ◆ Systemkald
- ◆ Kernetilstand og brugertilstand
- ◆ Proces
- ◆ Program
- ◆ Proces vs. program
- ◆ Adresserum
- ◆ Procestilstande: Ready, Running, Waiting, Terminated
- ◆ Processskift
- ◆ Samtidighed
- ◆ Synkronisering
- ◆ Synkron og asynkron kommunikation
- ◆ Bootstrapping

KAPITEL 4

INTERNETTET – I BRUGERPERSPEKTIV

Hvad er internettet? Spørgsmålet lyder måske ikke så svært, men det er svært at give et kort svar på. Vi venter derfor til kapitel 5 med at indkredse teknisk hvad internettet er. Derimod er det ikke svært at forklare, *hvad vi bruger internettet til* – fx:

- ◆ Emails
- ◆ Informationssøgning
- ◆ Sociale fællesskaber
- ◆ Fildeling
- ◆ Chat
- ◆ Computerspil
- ◆ Online-shopping
- ◆ Internet-telefoni

Fra din computer kommunikerer du med én eller flere computere via internettet vha. fx en *web browser* (normalt bare *browser*).

ORDFORKLARING

BROWSER

Program til fremvisning af og interaktion med internet-indhold.

EKSEMPEL

BROWSERE

Programmer som Internet Explorer, Firefox, Safari og Opera.

Skal du mere end bare at besøge websider, bruger du andre programmer, fx spil eller chatprogrammer. De arbejder sammen – over nettet – med programmer hos den kammerat, du spiller computerspil eller chatter med.

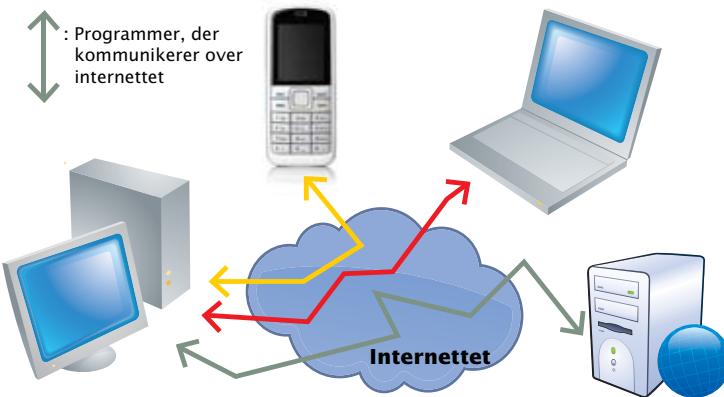


Fig. 4.37. Kommunikation over internettet vha. programmer.

4.1. KLIENT OG SERVER

Lad os sige, at vi vil åbne websiden på adressen www.eksempel.dk. Vi indtaster adressen i browseren, der så kontakter den webside-server, der svarer til adressen. Serveren er en internet-tilkoblet maskine, der altid er tændt og klar til at modtage forespørgsler fra browserne. Når browseren forespørger indholdet af www.eksempel.dk, sørger serveren for at udlevere indholdet. Denne enkle to-trins-kommunikation består af forespørgsel (*request*) og svar (*response*).

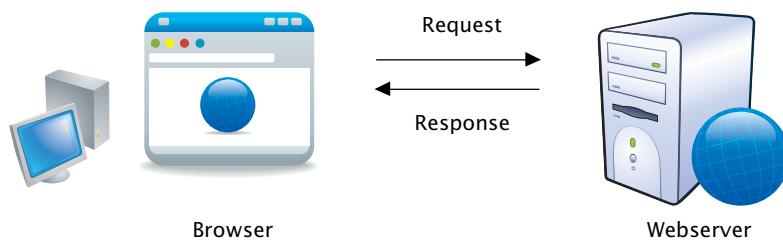


Fig. 4.38. Request og response.

Browseren og webserveren kommunikerer vha. den såkaldte *HTTP-protokol*, som vi kommer nærmere ind på i afsnit 4.3.

Helt generelt kommunikerer programmer over nettet ved at udveksle beskeder. Forespørgsler og svar er eksempler på beskeder.

Request-response-forløbet i en kommunikation mellem browser og webserver er et eksempel på et generelt designmønster indenfor datalogisk arkitektur:

ORDFORKLARING

KLIENT-SERVER-PRINCIPPET

En model med to aktører: *klient* og *server*. Klienten forespørger tjenester hos serveren. Serveren svarer ved at levere de forespurgt services. Klienten er aktiv – den, der tager initiativ til en forespørgsel. Serveren er passiv – den svarer blot på forespørgsler.

I tilfældet “browser og webserver” er browseren klient hos webserveren. Browsere er HTTP-klienter. I almindelighed er klienter brugerprogrammer, der kører på fx en PC, mens servere typisk er større programmer, der kører på en væsentligt kraftigere maskine. En maskine kan godt have flere serverprogrammer kørende. Servermaskiner betjenes kun sjældent direkte mennesker, og har typisk hverken skærm, tastatur eller mus.

EKSEMPEL

KLIENTER OG SERVERE

Webservere, spilservere, printservere og filservere er alle eksempler på servere. Webserveren tilbyder adgang til webindhold. Dens klienter er typisk browsere. Spilserveren tilbyder adgang til at spille et online-spil. Dens klienter er spilprogrammer. Printserveren tilbyder adgang til en eller flere printere. Dens klienter er tit tekstbehandlingsprogrammer, der beder om at få printet dokumenter.

Filserveren tilbyder adgang til en samling filer. Dens klienter særlige programmer, der kan hente filerne til brugerens computer.

Serveren er tit til tjeneste for mange klienter samtidig. Klienten er typisk i kontakt med en bruger, mens serveren typisk ikke er det. I klient-server-situationer vil vi ofte referere til “brugeren” og dermed underforstå, at der sidder et menneske, brugeren, og styrer klientens forespørgsler.

4.2. WEBSIDER OG WEBADRESSER

En *webside* er en type “netværksdokument” – et dokument med fx tekst, billeder, links og film, der ligger tilgængeligt på et netværk – fx internettet. Er man tilkoblet netværket, kan man se websiden. En webside er det samme som en *hjemmeside*. Websider identificeres med *webadresser*.

ORDFORKLARING

WEBADRESSE OG URL

En webadresse på en webside er et stykke tekst, der fastlægger hvor på netværket, websiden findes. En webadresse kaldes med et teknisk udtryk for en *uniform resource locator (URL)*.

EKSEMPEL

WEBSIDER OG WEBADRESSER

Følgende er eksempler på URL'er til websider, du måske kender: www.google.com, www.hotmail.com, www.myspace.com, www.folketinget.dk.

En webside er et eksempel på et *websted* eller et *web site* (der løst sagt betyder “et sted på internettet” – et site kan også være andet end lige præcis en webside).

ORDFORKLARING

WEBSERVER

Maskine, der gør en webside tilgængelig på et netværk – fx internettet. Websiden er tilgængelig via sin URL.

4.2.1. MARKUP OG FREMVISNING

En webside er anderledes end et traditionelt tekstdokument: Det er en fil skrevet i en særlig kode, kaldet *markup*. En browser fortolker markup-koden og sørger for en fremvisning, mennesker kan forstå – med links, tekst, knapper, menuer mv.

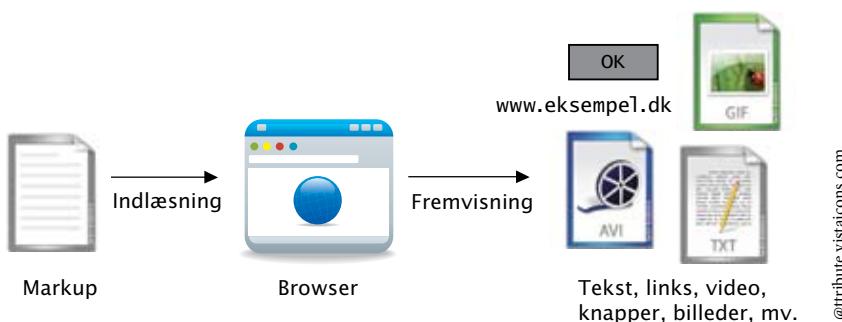


Fig. 4.39. Fremvisning af markup-dokument.

WWW

HTML

På **WWW** eksperimenterer vi med markup-sproget HTML.

4.2.2. DYNAMISKE WEBSIDER (SERVER-SIDE SCRIPTING)

Markup-kode har begrænsninger – det kan typisk *ikke tage input*. Hvis man som website-forfatter skriver en færdig website med markup-kode, vil denne website fremvises på samme måde altid. Det er sjældent ønskeligt.

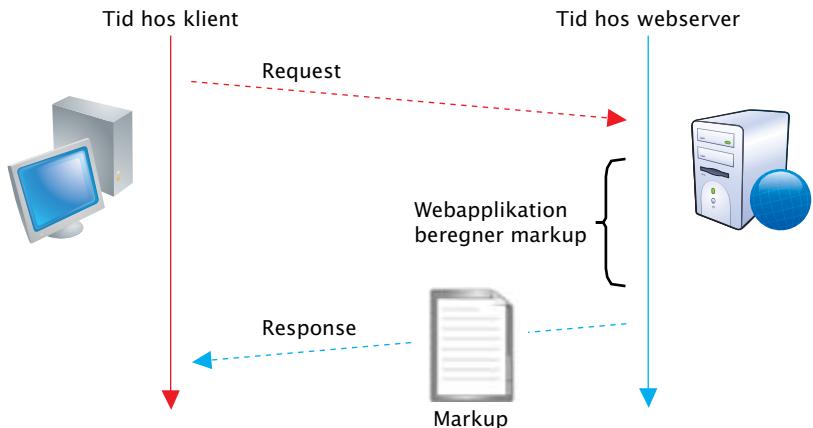
Løsningen på markup-begrænsningerne er at lave en dynamisk website.

ORDFORKLARING

DYNAMISK WEBSITE

Website, hvis konkrete indholdsdele først bestemmes, når den forespørges.

En dynamisk website er ikke lagret som (ren) markup-kode. Når en klient forespørger siden, sørger et program på webserveren for at beregne det aktuelle indhold af websiden, formulere det som markup-kode, og sende svaret til klienten. Programmet kaldes for *webapplikationen*.



@attribute.vistaicons.com.

Fig. 4.40. Dynamisk website beregner markup ved hver forespørgsel.

Dynamiske websider kan altså bruges til at vise forskellige udgaver af websiden i forskellige situationer. Det giver mulighed for

- ♦ interaktive websider, der fremviser forskellige ting på baggrund af brugers indtastninger og museklik,
- ♦ bruger-afhængige websider, der fremvises forskelligt alt efter bruger,
- ♦ at programmere websider, der er lette at vedligeholde.

De beregninger, webapplikationen foretager, er lagret som *scripts*.

ORDFORKLARING

SCRIPT

Lille stykke kode, der ved at bruge funktioner fra programmer kan udføre opgaver, fx beregne markup-kode eller åbne popup-vinduer.

WWW

SCRIPT

Det kan svært at forstå, hvad et script er og hvorfor scripts er smarte, før man har prøvet selv at skrive et script. På **WWW** eksperimenterer vi med at skrive scripts.

Scripts kan bruge funktioner fra alle typer programmer. De bruger typisk den webapplikation, de er en del af, men kan også fx trække på programmer, der kan hente information frem fra en database. Scripts, der udføres på en webserver, kaldes *server-side scripts*.

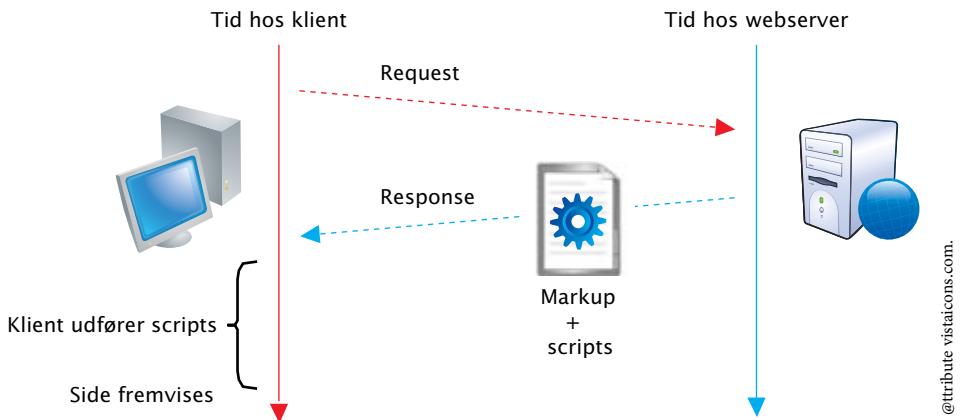
ADVARSEL

SCRIPTS ER IKKE PROGRAMMER

Et program kan køre helt uafhængigt af andre programmer. Et script er ikke selvstændigt: Et script benytter et program til at udføre sine opgaver.

4.2.3. CLIENT-SIDE SCRIPTS

Ligesom serveren foretager beregninger til dynamiske websider, kan klienten foretage beregninger, der bruges til at fremvise en side. Det gøres også med scripts. Klienten henter de nødvendige scripts sammen med resten af siden og udfører dem derefter selv. Hvis en klient udfører scripts, kaldes de *client-side scripts*.



@attribute vistaicons.com.

Fig. 4.41. Client-side scripts.

Client-side scripts bruges til at håndtere beregninger og interaktion, der ikke kræver konstant kontakt til serveren. Et client-side script kan i principippet gøre de samme ting som almindelige brugerprogrammer, men man benytter dem primært til at producere webside-effekter, der ligger uden for rammerne af markup. Rent teknisk er der nemlig en række begrænsninger for, hvad markup-kode kan få en computer til at gøre. Her følger nogle få eksempler på, hvad client-side scripts kan, som typisk markup ikke kan:

- ◆ interaktion i form af en dialog-boks, fx
 - ◆ ”Vil du læse om dialogbokse? Klik ’Ja tak’ eller ’Skrid, dialog-boks.’”
- ◆ reagere på andre ting end klik på link og knapper, fx
 - ◆ åbne en ny side, blot muse-pilen holdes over et sted på siden,
 - ◆ tjekke brugerindtastninger mens de bliver testet,
- ◆ levere dynamisk indhold.

WWW

EVENTS

At client-side scripts kan reagere på ”andre ting end klik på link og knapper” dækker over, at de kan reagere på såkaldte *events* (begivenheder). Det er for eksempel en event, at musen flyttes ind i et bestemt skærmområde, eller at bestemte knapper på tastaturet trykkes ned. På **WWW** ser vi eksempler på client-side scripting med events.

4.3. PROTOKOLLER

Ovenfor har vi allerede nævnt HTTP-*protokollen*. Men inden vi går i detaljer med HTTP, må vi vide hvad en protokol er? Det viser sig at være et helt centralt spørgsmål, hvis man vil forstå internettet og kommunikation med eller mellem maskiner overhovedet. Protokoller bruges nemlig overalt, hvor maskiner skal kommunikere med andre maskiner eller med mennesker. Netop fordi maskiner er maskiner, er de nødt til at have nøjagtige instrukser om, hvordan "samtaler" skal forløbe. De kan nemlig ikke, som mennesker, selv opfinde indslag til en samtale eller forstå "ukendte" indslag.

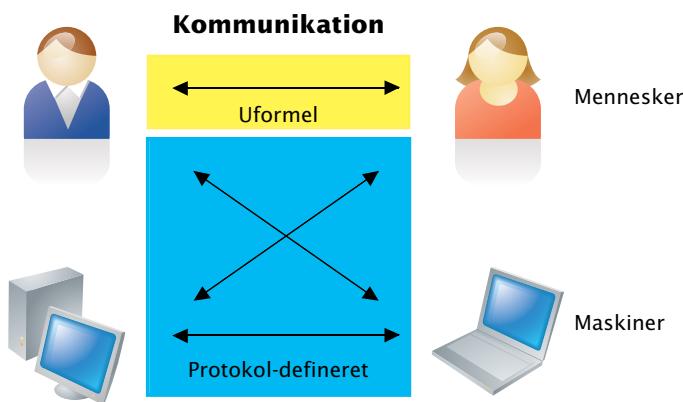


Fig. 4.42. Protokoller definerer kommunikation med og mellem maskiner.

I internetsammenhæng er protokollerne den "lim", der binder internettet sammen. Og HTTP er langt fra den eneste protokol.

ORDFORKLARING

PROTOKOL

En protokol definerer hvilke beskeder, der kan udveksles mellem to eller flere kommunikende programmer, og i hvilke rækkefølger disse beskeder kan udveksles. Protokollen fastlægger også hvad der skal gøres i tilfælde af, at en bestemt besked modtages eller afsendes.

En protokol er altså kort og godt et regelsæt for hvordan, programmer 'snakker sammen' – en slags sprog.

EKSEMPEL

EN SPØRGSMÅL-SVAR-PROTOKOL TIL KLASSETIMER

Du har datalogitime, og din lærer kværner løs om protokoller. Pludselig stopper han og spørger ud i klassen: "Spørgsmål?" (en besked der sendes af læreren og modtages af alle elever). Du rækker hånden op (en besked fra dig til læreren). Læreren kigger smilende: "Ja...?" (en besked til dig om at fremsætte spørgsmålet). Du stiller spørgsmålet (en besked til læreren), som læreren hører (modtager) og derefter svarer på (sender et svar).

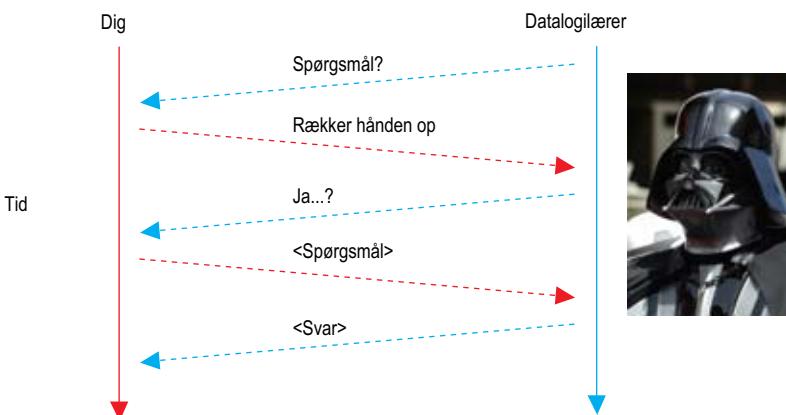


Fig. 4.43. Spørgsmål-svar protokol.

Det vigtigt for kommunikationen i klasselokalet, at både eleverne og læreren er enige om protokollen. Hvis for eksempel en elev i klassen ikke forstår dansk, ikke rækker hånden op eller kun vil stille biologi-spørgsmål (svarende til, at denne elev bruger en anden protokol), ville læreren og eleven ikke meningsfuldt kunne udveksle beskeder.

I denne bog stifter vi bekendtskab med en række forskellige protokoller. Hver er designet til et bestemt formål, og protokollerne er dermed ganske forskellige. Der er for eksempel forskel på, hvilket sprog, emailservere bruger, og hvilket sprog, netværkskort bruger.

Uanset protokollernes forskellighed, kan et forløb med udveksling af beskeder efter en bestemt protokol ofte tegnes som et tidsforløb med to aktører (som i Figur 4.43).

ADVARSEL

PROTOKOL OG PROTOKOL-SOFTWARE

I daglig tale siges ofte "protokol" også om protokol-softwaren – et særligt ekstra-program, der kan sende og modtage beskeder over protokollen (men ikke andet).

4.4. HTTP

Kommunikationen mellem en browser og en webserver foregår over HTTP-protokollen. Browseren sender en forespørgsel, hvorefter webserveren svarer. Du har sikkert lagt mærke til, at webadresser (der kan ses i fx browserens adressebjælke) ofte er på formen

<http://www.etellerandet.nogetmere.com>

Her angiver det foranstillede `http://`, at det er HTTP-protokollen, der bruges til at hente indholdet på webadressen www.etellerandet.nogetmere.com.

ORDFORKLARING

HTTP

HTTP er en forkortelse for *hyper text transfer protocol* ("protokol til overførsel af hypertext"). HTTP definerer hvordan HTTP-klienter (browsere) kommunikerer med HTTP-servere (webservere).

4.4.1. REQUEST OG RESPONSE

HTTP-protokollen har to beskedtyper: *request* (forespørgsel) og *response* (svar). Vi illustrerer beskedtyperne med et eksempel:

EKSEMPEL

REQUEST OG RESPONSE

Lad os forestille os, at en browser vil indlæse en webside, skrevet i markup-sproget HTML. Websiden har URL'en www.eksempel.dk/index.html. Den indeholder to billeder samt forklarende tekst. Teknisk består websiden af

- ◆ en HTML-fil, `index.html`, der indeholder teksten og "sammenklistrer" tekst og billeder,
- ◆ billedfilen `billede1.jpg`, der forestiller en croissant,
- ◆ billedfilen `billede2.jpg`, der forestiller en liflig bagel.

fortsættes...

EKSEMPEL

REQUEST OG RESPONSE (FORTSAT)

Der sker nu følgende:

1. Webserveren lytter afventede på sin forbindelse til internettet: "Mon der er nogle browsere, der har lyst til at kontakte mig?"
2. Browseren kontakter webserveren og sætter dermed en forbindelse op mellem computeren og webserveren.
3. Browseren sender en HTTP-forespørgsel til webserveren over forbindelsen. Forespørgslen indeholder bl.a. information om, at det er HTML-filen `index.html`, der efterspørges.
4. Webserveren modtager forespørgslen, henter filen `index.html` frem, pakker den sammen i et HTTP-svar og sender svaret til browsen over forbindelsen.
5. Broweren modtager `index.html` over forbindelsen og opdager, at `index.html` refererer til de to billedfiler.
6. Broweren laver to nye HTTP-forespørgsler, hvor billederne `billede1.jpg` og `billede2.jpg` forespørgses. Trinnene 2 – 5 gentages for hvert af de to billeder.
7. Broweren har nu alt hvad den skal bruge for at fremvise websiden. Den kontakter derfor ikke serveren yderligere. Efter et stykke tid afbryder serveren forbindelsen. Det kaldes et (*connection*) *timeout*.

Vi illustrerer trinnene:

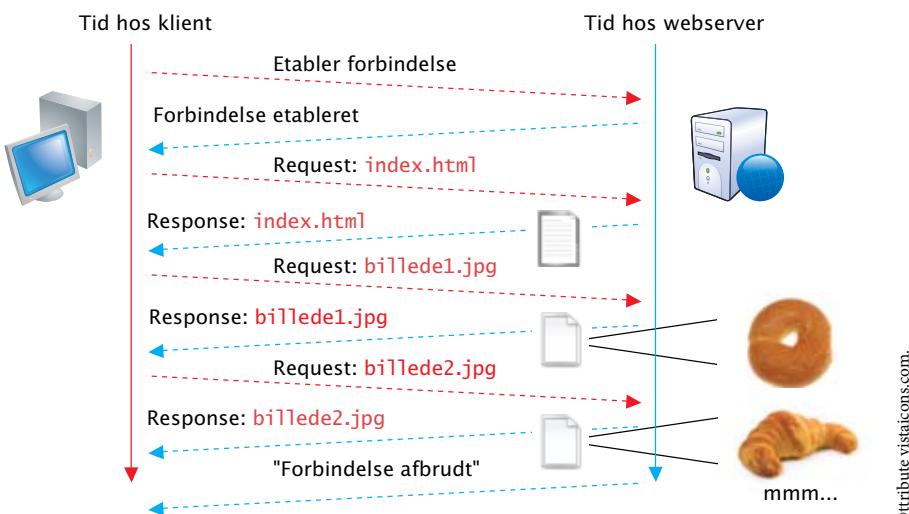


Fig. 4.44. Forespørgsel om en webside over HTTP. Websiden består af tre filer: HTML-filen `index.html` og to billedfiler, `billede1.jpg` og `billede2.jpg`.

Der findes to grundlæggende slags HTTP-forespørgsler, nemlig ”Giv mig indholdet på den medfølgende URL, tak” (**GET**) og ”Giv mig indholdet på den medfølgende URL, tak, her er de ekstra informationer du, kære server, skal bruge” (**POST**):

- ◆ **GET:** Forespørger webindhold.
- ◆ **POST:** Sender information til serveren samtidig med, at webindhold forespørges. Bruges ofte hvis serveren skal have indtastninger i felter, brugeren har udfyldt.

(**POST** kommer af det engelske ord ”post”, der bl.a. kan betyde at sætte en meddeelse op på en opslagstavle).

GET og **POST** kaldes forespørglens *metode*. Der er andre metoder end **GET** og **POST**, men dem kommer vi ikke ind på her.

EKSEMPEL

HTTP-FORESPØRGSEL MED METODEN GET

```
GET /index.html HTTP/1.1  
Host: www.eksempel.dk  
User-Agent: Mozilla/4.0
```

”**HTTP/1.1**” angiver den version af HTTP-protokollen, browseren gerne vil bruge, mens ”**User-Agent: Mozilla/4.0**” angiver browser-typen.

EKSEMPEL

HTTP-FORESPØRGSEL MED METODEN POST

```
POST /login.html HTTP/1.1  
Host: www.eksempel.dk  
User-Agent: Mozilla/4.0
```

username=oleolsen&password=oleolsenersej

Her logger Ole Olsen ind på en særlig loginside, **login.html**, efter at have angivet sit brugernavn, Ole Olsen, og sit kodeord, **oleolsenersej**. Bemærk, at data (nemlig brugernavn og kodeord) medsendes til serveren med metoden **POST**.

Generelt tager HTTP-forespørgsler formen

```
<Metode> <Sti til objekt> HTTP/<version>  
Host: <URL til webside, der indeholder objekt>  
<Yderligere information om selve forespørgslen>  
<Data relateret til forespørgslen>
```

HTTP-svar har også et helt bestemt udseende:

```
HTTP/<version> <Statuskode> <Status-besked>
<Information om svaret: størrelse, filtype, tidspunkt for afsendelse fra serveren, mv.>

<Svar: Det forespurgte objekt (data)>
```

Statuskoden og statusbeseden angiver med et tal og en tekstbesked, om forespørgslen gik godt eller ej.

HTTP-STATUSBESKEDER		
STATUSKODE	STATUSBESKED	FORKLARING
200	OK	Forespørgslen lykkedes. Serveren svarer med det forespurgte objekt.
400	Bad Request	Serveren kunne ikke forstå forespørgslen.
403	Forbidden	Serveren forstod forespørgslen, men nægter at udføre den, fordi browseren ikke har lov til at se indholdet.
404	Not Found	Det forespurgte objekt findes ikke.
500	Server Error	En uforudset fejl på serveren forhindrer forespørgslen i at blive besvaret.

Fig. 4.45. Nogle almindelige HTTP-statuskoder og tilhørende statusbeskeder.

EKSEMPEL

HTTP-SVAR

Herunder er et eksempel på en vellykket forespørgsel af et HTML-dokument:

HTTP/1.1 200 OK

Date: Fri, 8 Feb 2008 13:19:21 GMT

Content-Type: text/html

Content-Length: 1416

```
<html>
<body>
Velkommen til www.eksempel.dk.
Her finder du et billede af en croissant og en bagel. Mums.
...
</body>
</html>
```

4.4.2. SESSIONS

Som udgangspunkt er HTTP en *tilstandsløs* protokol. Det vil sige, at en server ikke via HTTP kan opdage, om en række forespørgsler kom fra samme klient. En “samtale” over HTTP ”husker” ikke en udveksling af forespørgsel og svar, når først den er overstået.

Udfordringen for det tilstandsløse HTTP er at sætte en server i stand til at huske og skelne sine klienter.

ANALOGI

MÆRKELIGT VÆRTSHUS

Serverens problem med, at HTTP er en tilstandsløs protokol, kan måske forstås bedre vha. en analogi. Lad os forestille os et (lidt) mærkeligt værtshus. Ekspedienten (=serveren) står bag disken med ryggen til kunderne (=klienterne), vendt mod hylderne med drikkevarer. Kunderne skiftes til at rejse sig, gå op til disken og råbe en bestilling (=en forespørgsel) til ekspedienten, som derefter giver kunden det ønskede uden på noget tidspunkt at vende sig om og kigge på ham. Ekspedienten kan ikke høre forskel på de enkelte kunder.

Hvis nu ekspedienten modtager tilræbene ”En fadøl, tak!” og ”En fadøl, tak!”, leverer ekspedienten to gange en fadøl til en kunde – men aner ikke, om det i virkeligheden er den samme kunde, der har fået to fadøl. Hvis ekspedienten modtager bestillingen ”Det samme som sidste gang!” er han i problemer – hvem spørger, og hvad fik vedkommende sidste gang?

EKSEMPEL

INTERNETBUTIK

Gartner-Georg og Kokke-Karen shopper i en internetbutik fra hver deres computer med hver deres browser. I løbet af de to online-indkøbsture sker der mange udvekslinger af HTTP-beskeder (hver gang en vare lægges i en indkøbskurv, er det en forespørgsel). Hvis internetbutikken bruger HTTP uden at tage højde for tilstandsløsheden, kan serveren risikere at glemme indholdet af Georgs kurv mellem hans forespørgsler. Serveren risikerer også at blande Georgs og Karens forespørgsler sammen, så Georg får spæk-brætter og Karen får plæneklipper.

Løsningen på problemet er at medsende ekstra information i hver forespørgsel og svar. Derved kan serveren opretholde en *session* med browseren.

ORDFORKLARING

SESSION

At en klient og en server vedligeholder en forbindelse over længere tid og begge ”husker” den information, der allerede er blevet udvekslet (= forbindelsens tilstand).

Første skridt på vejen til at lave en session er sessions-ID'et. Sessions-ID'et er et særligt tal, serveren og browseren bliver enige om fra start af (det sker ved at serveren bestemmer tallet og oplyser det til klienten). Hver gang browseren sender en HTTP-besked til serveren, sendes sessions-ID'et med. På den måde kan serveren skelne mellem de enkelte klienter, fordi de har hver deres sessions-ID.

Server og klient udveksler tit sessions-ID via *cookies*.

4.2.3. COOKIES

Det er tit praktisk for både en webside og dens besøgende at websiden kender lidt til sine gæster og deres præferencer. Det kan desuden være smart, at websiden kan huske sine besøgende fra besøg til besøg.

ORDFORKLARING

COOKIE

Et stykke tekst, sendt fra en server til en browser, der derefter sender den uændret tilbage hver gang serveren forespørges noget. På den måde kan serveren genkende klienten.

ANALOGI

COOKIES ≈ POST-ITS

Du er førstegangs-kunde i boghandlen Bøger A/S. Idet du træder indenfor, sætter en ansat i døren en lille post-it på din ryg. På sedlen står et unikt kundenummer (dit sessions-ID) samt butikkens navn. Hvis du bare kigger rundt i butikken og så går igen, behøver butikken ikke skrive noter på sedlen, når du går. Men hvis du fx køber en bog om heste eller golf, så noterer butikken disse oplysninger (samtidigvis mere, såsom dit navn eller kreditkortoplysninger) på post-it'en.

Næste gang du besøger butikken, genkender den ansatte straks sedlen på din ryg – der står jo "Bøger A/S" på den. Den ansatte læser videre på sedlen, at du kan lide heste og golf. Han henvender sig til dig og siger "Hejsa. Godt at se dig igen. Vi har fået nye hestebøger siden sidst og en kalender med golf-billeder. Hvis du vil købe, så bare sig til – du behøver ikke engang opgive dine kreditkortoplysninger igen."

Cookiens *værdi* – dvs. cookiens tekstdhold – bestemmes af serveren ved første kontakt mellem klient og server. Værdien sendes til klienten i en HTTP-besked. Cookien gemmes som en fil på brugerens computer. Browseren angiver cookien i sine forespørgsler til serveren. Serveren gemmer brugerinformation knyttet til cookien.

EKSEMPEL

COOKIES

Ole køber ind i internet-boghandlen Bøger A/S Online, der bruger cookies. Da Ole åbner butikkens side, sender serveren et HTTP-svar med følgende linie kode:

Set-Cookie:USER_ID=12345678;

Browseren gemmer **USER_ID** og værdien **12345678** i en særlig fil – en *cookie* – på Ole Olsens computer. Ole klikker nu lystigt rundt på Bøger A/S Online. Hver gang browseren sender en forespørgsel til serveren, kigger den i cookie-filen først og ser, at den skal sende en linie af formen

Cookie:USER_ID=12345678;

med til serveren. Selvom serveren ikke nødvendigvis kender Oles navn, kan den alligevel holde styr på brugeren med ID-nummer 12345678 – og dermed for eksempel på varer i denne brugers indkøbskurv.

Alle cookies forsynes med en levetid. Når levetiden udløber, slettes cookien. Levetiden kan variere fra få minutter til flere år – afhængigt af anvendelse.

4.5. EMAILS

Emails kender vi alle. De er hurtige og billige (eller gratis!) at sende, evt. til mange modtagere. Modtagerne kan tjekke dem, når det passer dem (emails er *asynkron kommunikation*, jf. afsnit 3.3.3). En klokkeklaar succes. Spørgsmålet er – hvordan virker de?

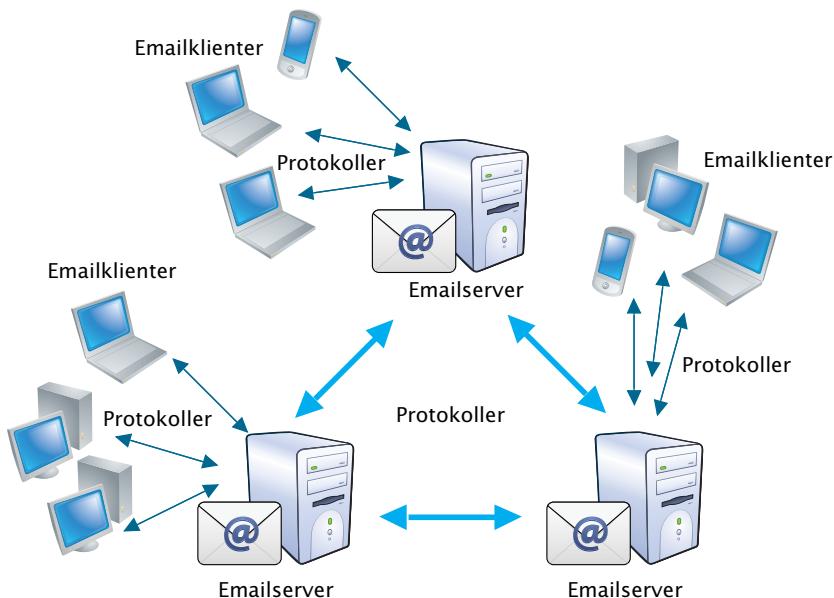


Fig. 4.46. Internettets email-system består af 3 dele: emailklienter, emailservere og protokoller til udveksling af email.

Emailklienter er de programmer, man bruger til at sende og modtage emails med. En emailserver er en slags centralt posthus, hvor brugerne har hver deres private postkasse.

Lad os igen se på Ole, der denne gang vil sende en email til Maria:

1. Ole skriver sin email, vha. sin emailklient eller et andet program på hans computer.
2. Oles emailklient sender emailen til Oles emailserver, der putter emailen i sin udbakke.
3. Oles emailserver forsøger at sende emailen fra udbakken til Marias emailserver.
4. Hvis det lykkes, modtager Marias emailserver emailen, der herefter opbevarer den, indtil Maria får lyst til at kontakte sin server for at få de nyeste emails.
5. Hvis det ikke lykkes at kontakte Marias emailserver selv efter mange forsøg, får Ole til sidst sin email tilbage med besked om, at levering ikke kunne ske.

Som det fremgår, sker der altså i alt tre transmissioner, før emailen er fremme (to, hvis Ole og Maria har samme emailserver):

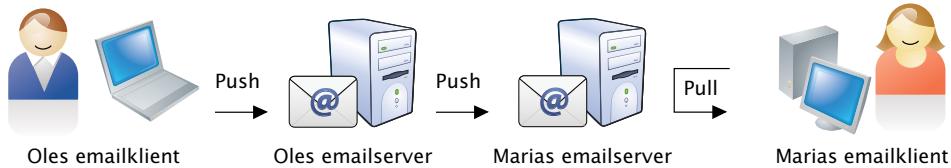


Fig. 4.47. Transmission af email sker i tre trin.

Til de to første transmissioner bruges typisk protokollen SMTP (forkortelse for “simple mail transfer protocol”). SMTP er en såkaldt *push*-protokol, fordi afsender bestemmer, hvornår emailen skal ”*skubbes* afsted”. Det er klart, at SMTP ikke kan bruges til den tredje transmission, da emailserveren ikke ved, hvornår Maria har sin computer tændt. Derfor skal tredje transmission bruge en *pull*-protokol, altså en protokol, hvor Maria selv bestemmer, hvornår hun vil ”*trække*” emailen fra emailserveren til sin computer. Som pull-protokol bruges ofte POP3, IMAP eller HTTP.

Push og pull er eksempler på et generelt server/klient-koncept:

ORDFORKLARING

PUSH OG PULL

Lad os forestille os en server, en klient og en information. Har serveren informationen, kan klienten få den ved at lave et *pull*.

- ♦ Pull: Klienten kontakter serveren og beder om informationen.

Har klienten informationen, kan klienten give den til serveren ved at lave et *push*.

- ♦ Push: Klienten kontakter serveren og giver informationen.

Bemærk, at når en emailserver kontakter en anden for at overbringe en email, så optræder den første emailserver som SMTP-klient, mens den anden optræder som SMTP-server. Emailservere er altså selv klienter hos de servere, de viderestiller emails til.

Lad os se på, hvordan Maria kan lave et email-pull vha. protokollen POP3:

ANALOGI

MODTAGE EMAIL ≈ SAMTALE

Emailklient: "Banke, banke på!"

Emailserver: "Det er posthuset '**POP3.EMAIL-SERVICE-PROVIDER.COM**', goddag! Hvis' email ønsker De?"

Emailklient: "Marias email, tak. Jeg er hendes emailklient, forstår du."

Emailserver: "Bevis det."

Emailklient: "Ok. Kodeordet er '**MariaErSej**'."

Emailserver: "Det er godtaget. Der er kommet tre emails i alt siden sidst. De kommer gennem kablet til dig nu."

[Flump, flump, flump] : Tre emails flumper gennem kablet og lander hos Maria.

Emailklient: "Tre meddelelser – tak for det. Jeg modtog dem alle. Du kan slette dine kopier nu."

Emailserver: "Selv tak. En fornøjelse. Jeg har nu slettet kopierne. På gensyn."

4.6. CHAT OG INSTANT MESSAGING

Emails er ikke den eneste kommunikationsform på internettet. Det er også udbredt at chatte i et chatforum eller via et *instant messaging*-program (instant betyder "øjeblikkelig"). Disse to typer kommunikation er, i modsætning til emails, som oftest *real-time* og *synkron*, hvilket betyder at de kommunikerende parter (afsender og modtager) er aktive på samme tid.

Der er mange chat-protokoller. Nogle er klient/server-protokoller, hvor de kommunikende parter begge logger på en chat-server og udveksler beskeder ved at *pushe* dem til serveren, der så videresender til rette modtager. Andre chat-protokoller etablerer en direkte forbindelse mellem de kommunikerende parter, der så kan udveksle beskeder uden kontakt med servere.

4.7. FILDELING

Fildeling er et vidt begreb: "at flere forskellige brugere deler adgang til en samling filer". Adgangen kan være begrænset til at hente en kopi af filerne eller række til at redigere i og slette filerne. På den måde kan enhver offentligt tilgængelig webside opfattes som en fildeling: Alle har jo adgang til at hente de filer, websiden er opbygget af. Webservere er et eksempel blandt mange på, at filer kan deles via en fil-server.

Vi vil her kun beskæftige os med *peer-to-peer*-fildeling, forkortet P2P.

ORDFORKLARING

P2P-FILDELING

Fildeling direkte mellem brugere – uden brug af centrale fil-servere.

Navnet "peer-to-peer" kommer af, de fildelende brugere kaldes peers (kammerater).

EKSEMPEL

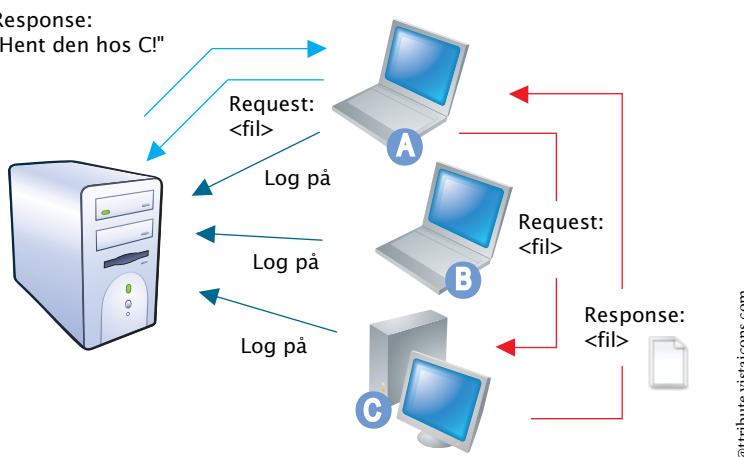
P2P-FILDELING

Maria har fildelingsprogrammet **HeltLovligt** på sin mobiltelefon. Hun vil gerne downloade en musikfil, mere præcis Haydns oratorium "Skabelsen". Maria starter programmet, der kontakter **HeltLovligt**-serveren, som hurtigt svarer tilbage med en liste de **HeltLovligt**-brugere, der har en kopi af "Skabelsen" til deling. Maria vælger nu Oles PC på listen, fordi det valg giver den korteste download-tid. En forbindelse bliver herefter etableret direkte fra Marias mobil-telefon til Oles PC, og en kopi af "Skabelsen" suser fra Oles musiklager til Maria.

Skulle Ole slukke sin PC midt i overførslen, finder **HeltLovligt**-programmet straks en anden **peer** at hente resten af filen fra.

Mens Maria henter sin klassiske musik, kan det sagtens tænkes at Sonja, en tredje **HeltLovligt-peer**, der er tilkoblet internettet med en håndholdt kalender, henter nogle af Marias filer med hit-musik.

Bemærk, at selvom der ikke er centrale fil-servere involveret i P2P, er der dog stadig en server involveret – nemlig den server, der koordinerer kommunikationen mellem fildelings-klienterne. Endvidere optræder en forespørgende peer som klient hos den peer, der har den ønskede fil. Denne peer optræder til gengæld som server. Med P2P er alle peers altså servere hele tiden, og klienter hver gang de forespørger filer.



@attribute visaticons.com.

Fig. 4.48. P2P-fildeling. Peers logger på fildelings-programmets centrale server. Hvis en klient forespørger en fil hos serveren, fortæller serveren hvilken peer der har filen.

4.8. DOMÆNER, IP-ADRESSER OG DNS

I det foregående har vi antaget, at de opkoblede computere (servere og brugercomputere) uden videre kunne kontakte hinanden. Teknisk foregår dette ved hjælp af *IP-adresser*, som du måske har hørt om? *DNS* – Domain Name System – hjælper med at få fat på de IP-adresser, der skal bruges.

ORDFORKLARING

DOMÆNE

Et internet-domæne, eller blot et domæne, er et navn, der identificerer en server på internettet. Domænenavnet er en del af serverens webadresse. Domænenavnet er unikt på hele internettet.

EKSEMPEL

DOMÆNENAVN

HTML-websiden med URL www.eksempel.dk/index.html har domænet www.eksempel.dk.

ORDFORKLARING

IP-ADRESSE

En IP-adresse består af 4 tal mellem 0 og 255, adskilt af punktummer, fx 87.172.255.0. Computere tilkoblet internettet har hver en *unik IP-adresse*. Det er IP-adressen, computeren bruger til at finde vej til hinanden over internettet.

Når du åbner en webside med en browser, indtaster du websidens webadresse, fx www.eksempel.nogetmere.dk. Før browseren kan hente websiden, skal denne webadresse oversættes til en IP-adresse. Det sker via DNS-systemet, der kan oversætte mellem domæner og IP-adresser.

Når nu man har IP-adresser, hvad skal man så med domæner? Svaret er, at vi mennesker har nemmere ved at huske domænenavne end IP-adresser – ligesom det er nemmere at huske en persons navn end vedkommendes postadresse. På den måde minder DNS om “De Gule Sider” – vha. en persons navn (domænet) slås personens adresse op (IP-adressen).

DNS – DOMAIN NAME SYSTEM

DNS består af:

1. Et offentligt tilgængeligt system af DNS-servere (kaldet navne-servere), der kan oversætte fra domænenavne til IP-adresser.
2. En klient/server-protokol i applikationslaget, der sætter computere i stand til at kommunikere med systemet af DNS-servere.

Server-systemet er inddelt i 3 typer: Lokale navne-servere (*local name servers*), rod-navne-servere (*root name servers*) og autoritative navne-servere (*authoritative name servers*).

Lokale navne-servere er “tættest” på klienterne og er installeret på de fleste større lokalnetværk (fx skoler, firmaer og universiteter). De kan oversætte webadresser på lokalnetværket til IP-adresser. Lokale navne-servere er kun lokalt tilgængelige.

Alle computere på internettet er registreret hos en autoritativ navne-server, der kender computerens IP-adresse og kan oversætte computerens webadresse (hvis computeren altså har en webadresse!) til dens IP-adresse. Tit fungerer en lokal navne-server også som autoritativ navne-server.

Rod-navne-servere og autoritative navne-servere er globalt tilgængelige. For at forstå den rolle, rod-navne-servere spiller, ser vi på, hvad en browser gør for at oversætte webadressen www.eksempel.dk til en IP-adresse:

1. Kontakt den lokale navne-server. Kender den lokale navne-server oversættelsen, svarer den med IP-adressen.
2. Kender den lokale navne-server ikke oversættelsen, kontakter den en rod-navne-server.
3. Rod-navne-serveren kigger på webadressen og svarer “spørg den autoritative navne-server på følgende IP-adresse: 14.12.97.199. Den server kender den IP-adresse, du søger”.
4. Den lokale navne-server kontakter den autoritative navne-server med IP-adressen 14.12.97.199 og beder om at få oversat webadressen.
5. Den autoritative navne-server svarer med den rette oversættelse (IP-adressen). Lad os sige, at IP-adressen for www.eksempel.dk er 64.233.167.99.
6. Den lokale navne-server sender den netop modtagne IP-adresse videre til browseren. Desuden gemmer den lokale navne-server (til eventuel senere brug) selv oversættelsen “www.eksempel.dk = 64.233.167.99”. Det kaldes *caching*.

Browseren kan nu kontakte www.eksempel.dk ved at bruge IP-adressen.

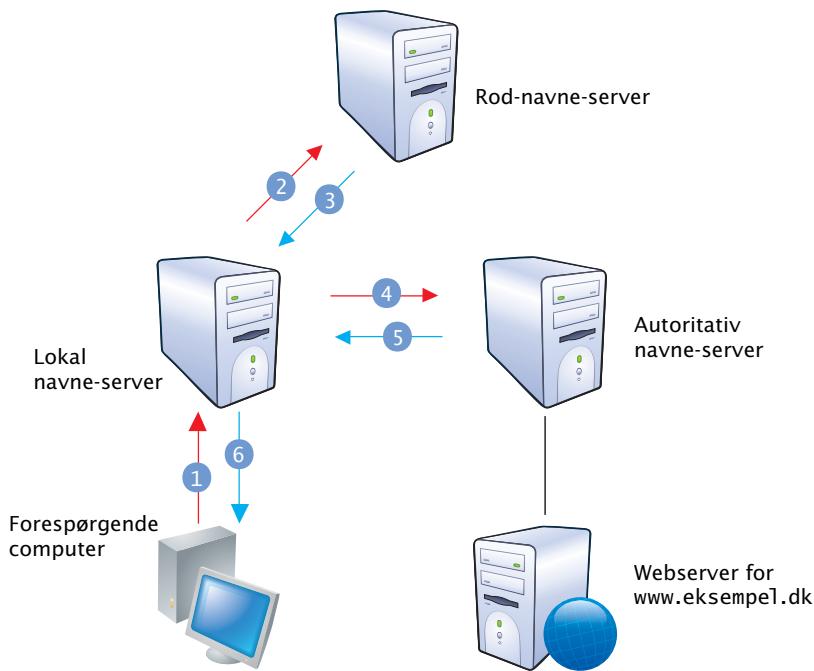


Fig. 4.49. DNS-opslag.

Vi kommer til at høre mere om IP-adresser, men det er vigtigt at vide, at IP-adresser ændres af og til. Derfor gemmer hverken lokale eller autoritative navne-servere deres oversættelser mere end et par dage ad gangen.

4.9. NØGLEORD

- ◆ Internettet
- ◆ Browser
- ◆ Webserver
- ◆ Request og response
- ◆ HTTP
- ◆ Webside
- ◆ Webadresse og URL
- ◆ Klient og server
- ◆ Protokol
- ◆ Domæne
- ◆ IP-adresse
- ◆ DNS

KAPITEL 5

NETVÆRK – LAGDELT KOMMUNIKATION

Internettet kan forstås ud fra de programmer, som brugere kommunikerer over internettet med. Det så vi i kapitel 4. I dette kapitel skal vi kigge på

- ◆ internettets fysiske opbygning,
- ◆ hvordan programmer faktisk kan kommunikere over et netværk.

Dermed besvarer vi spørgsmålet "Hvad er internettet?".

Vi bruger tit engelske ord, dels fordi mange tekniske betegnelser ikke har en god dansk oversættelse, dels fordi det gør det lettere at læse engelske tekster om netværk i fremtiden.

5.1. INTERNETTETS FYSISKE OPBYGNING

Internettet er et offentligt tilgængeligt, verdensomspændende "netværk af netværk". Det udgøres af en mængde mindre computernetværk, der er koblet sammen i et stort. De mindre netværk ejes af både firmaer, skoler og universiteter, myndigheder og privatpersoner i alle lande. De mindre netværk har ofte offentlige og private dele. De private dele kaldes *intranet* og medregnes ikke som en del af internettet.

ORDFORKLARING

LAN OG WAN

Et lokalnetværk eller et LAN er et mindre netværk, der er samlet lokalt, fx i en lejlighed eller bygning. LAN står for *local area network*. Et WAN er et større netværk, der er spredt over flere geografiske områder, fx flere bygninger rundt omkring i en by, eller flere forskellige lande. WAN står for *wide area network*.

Med denne terminologi kan man sige, at internettet er et WAN, opbygget af en lang række LANs og WANs, der er indbyrdes forbundne.

På hardware-niveau består internettet groft sagt af fire typer komponenter: *Slutsystemer*, *servere*, *routere* og *data-links*.

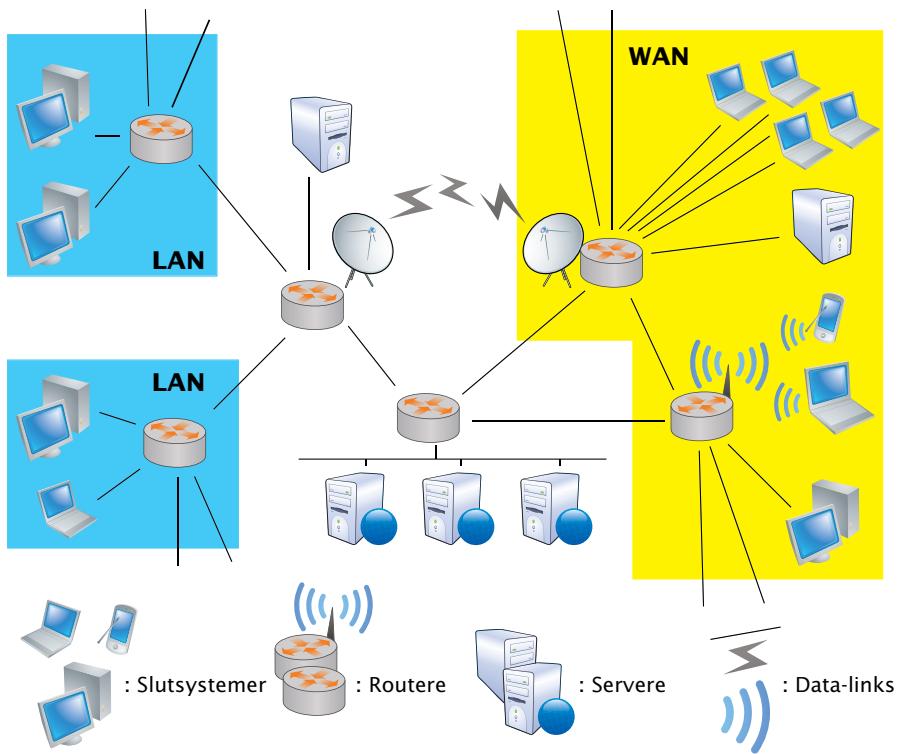


Fig. 5.50. Internettets hardware-opbygning.

Slutsystemerne er de internet-opkoblede maskiner, der betjenes direkte af mennesker, fx computere eller mobiltelefoner. Mennesker "får internettet at se" gennem slutsystemerne. Slutsystemer og servere kaldes tit under et for *værter*, da disse maskiner tit "er værter" for programmer, der kommunikerer over internettet.

ORDFORKLARING

ROUTER

En router er en særlig netværksmaskine, der er tilkoblet en række data-links. Routeren kan viderestille indgående datatrafik på et data-link til udgående datatrafik på et andet data-link.

ANALOGI

ROUTER ≈ RUNDKØRSEL

En router er som en rundkørsel: Når datatrafik (≈ en bil) via et data-link (≈ en vej, der leder ind i rundkørslen) ledes ind i en router (≈ en rundkørsel), vælges en videre rute gennem netværket og datatrafikken ledes ud via et data-link (≈ en vej ud af rundkørslen).

Data-links er de fysiske transmissionsmedier, der forbinder maskiner – som regel kobberkabler, coaxialkabler, optiske fiberkabler eller radiobølger. Data-links bruges til at overføre data. Overførselshastigheden kaldes *båndbredde* og måles tit i megabit/sekund (Mbps). Båndbredden varierer alt efter data-link.

Slutsystemer og servere er forbundet til internettet via deres internetudbydere.

ORDFORKLARING

ISP (INTERNET SERVICE PROVIDER)

En internetudbyder kaldes også en *internet service provider* (forkortet: ISP). ISP'erne er organiseret i et hierarki:

- *tier-1 ISP*'er udbyder internet *internationalt*, primært til tier-2 ISP'er
- *tier-2 ISP*'er udbyder internet *regionalt* (ofte nationalt), primært til tier-3 ISP'er
- *tier-3 ISP*'er udbyder internet *lokalt*: primært til lokale ISP'er
- *lokale ISP*'er udbyder internet til et lokalnetværk – på en skole, i en fir-mabygning, mv.

(Det engelske ord “tier” oversættes i denne sammenhæng med “lag” eller “niveau”).

ORDFORKLARING

INTERNETTETS BACKBONE

Netværket udspændt af tier-1 ISP'erne kaldes *internettets backbone* (backbone betyder “rygrad”). Tier-1 ISP'erne er forbundet med hinanden via data-links med enorm båndbredde.

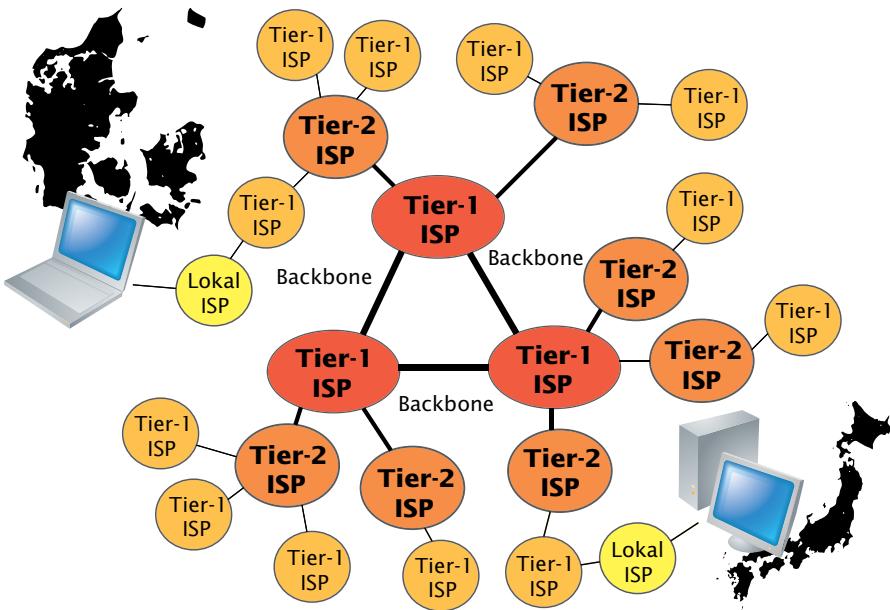


Fig. 5.51. ISP'er og backbone: Forbindelse mellem en computer i Danmark og en computer i Japan over et udsnit af internettet.

5.2. LAGDELT KOMMUNIKATION

I resten af dette kapitel gennemgår vi emnet ”kommunikation på netværk” uden hensyn til, om vi taler internettet, et LAN eller et eller andet WAN. Hovedeksemplet er dog stadig internettet.

Kommunikation på et netværk kan sammenlignes med et postsystem.

5.1.1. POST: KOMMUNIKATION MED TRE LAG

Når du vil skrive et brev til nogen, forsyner du konvolutten med frimærke, modtageradresse og afsenderadresse. Så poster du brevet i en postkasse, og derfra regner du med, at postvæsnets sørger for at levere brevet. Det er ikke din sag, om en postbil bryder sammen, eller om det bliver uvejr – når først brevet er postet, er det postvæsnets ansvar at levere det.

Postmand A tømmer nu postkassen og kører dit brev til sit posthus, Posthus A. Her læser A på din konvolut, at brevets modtager bor i det postdistrikt, som Posthus B servicerer. A skal altså bare sørge for at brevet kommer til Posthus B. Heldigvis har A hyret et fragtfirma til netop at køre post mellem Posthus A og Posthus B. Postmanden overdrager trygt dit

brev til fragtmanden – og herfra har han ligesom dig fralagt sig ansvaret – nu er leveringen af brevet til Posthus B fragtmandens ansvar. A behøver ikke spekulere på, om fragtmanden er løbet tør for benzin eller har mistet sit bykort.

Når fragtmanden ankommer til Posthus B, udleverer han dit brev til postmanden B, der herefter har ansvaret for at omdele brevet til modtagerens brevsprække.

Vi kan se, at brevkommunikationen her har tre *lag*:

- Et “brevskriver-lag” af brevskrivere (dig!) og brevmodtagere
- Et “post-lag” (postvæsnet), der tilbyder tjenesten “Fejl fri levering af post” til brevskrivelaget
- Et “fragt-lag” (fragtfirmaet), der tilbyder tjenesten “Kørsel af post mellem posthuse” til post-laget

Et brev passerer gennem lagene som illustreret på følgende figur:

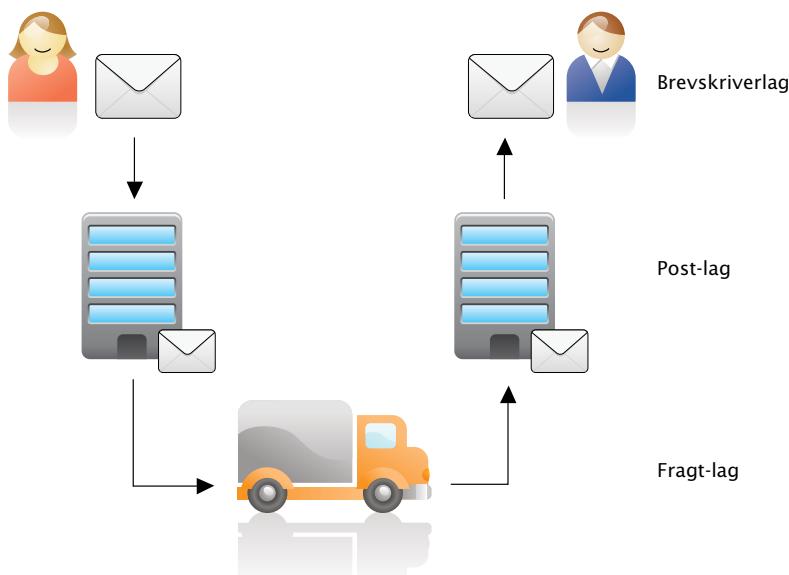


Fig. 5.52. Levering af post – kommunikation i 3 lag.

Det er vigtigt at forstå, at brevet først passerer fra øverste til nederste lag – herefter transporteres mellem posthusene – og til sidst passerer fra nederste til øverste lag.

5.1.2. NETVÆRK: KOMMUNIKATION MED FEM LAG

Netværkskommunikation er også lagdelt. Lagdelingen hjælper med at opdele det komplekse problem "at få to programmer til at snakke sammen over et netværk" i overskuelige delproblemer. Man skelner mellem 5 lag. Dette kapitel lærer dig, hvilken rolle hvert af de 5 lag spiller – foreløbig opnår vi deres navne og rækkefølge:

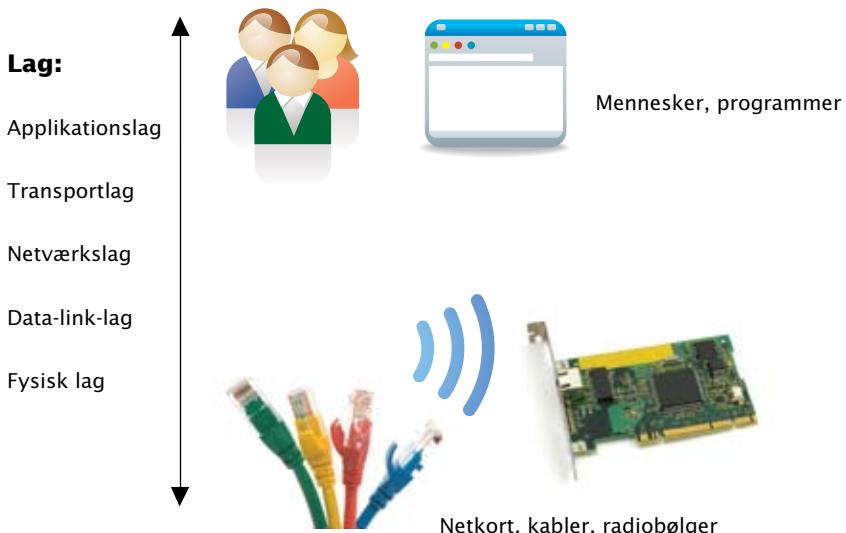


Fig. 5.53. Netværkskommunikation i 5 lag. Jo højere lag, desto mere menneskevenlig kommunikation. Jo lavere lag, desto mere maskinvenlig kommunikation.

Enhver kommunikation mellem to applikationer (=brugerprogrammer) over et netværk passerer gennem alle lagene, ligesom i analogien med postvæsnet og fragtfirmaet. Og ligesom brevskrivere benytter postvæsnet til brevlevering, gør hvert lag brug af laget nedenunder for at fungere. Fx kan man ikke sætte to programmer til at kommunikere over et netværk, hvis man ikke kan forbinde de to værter, hvorpå programmerne kører.



Fig. 5.54. Kommunikation over netværk passerer gennem alle kommunikationslag.

Figur 5.55 uddyber de enkelte lags funktion. Figuren er meget kortfattet, og det er ikke meningen, at du skal forstå den til bunds allerede nu – den er god som et start-overblik, og kan bruges som opsummering, når du har læst hele dette kapitel.

Vi husker, at en kørende applikation kaldes en *proces* – og at det er processer, der kommunikerer over netværket.

LAG	KOMMUNIKATION	OPGAVE	AFHÆNGIG AF
Applikation	Mellem processer	Forstår og reagerer på beskeder, som processer udveksler	At beskeder kan udveksles korrekt mellem processer
Transport		Sørger for at beskeder kan udveksles mellem processer. Dvs, at beskeder - ikke går tabt - når fejlfrit frem	At data kan sendes fra en computer på et netværk til en anden, men muligvis gå tabt eller blive ødelagt
Netværk	Mellem værter på netværket	Kan overbringe data mellem to computere. Data kan dog gå tabt eller indeholde fejl, når det når frem.	At data kan sendes over et data-link
Data-link	Over et data-link på netværket	Sender data fra netkortet på en maskine til netkortet på en maskine, der er direkte forbundet	At netkortet virker
Fysisk	Gennem et fysisk medie	Kan sende data gennem et fysisk data-link	Ingen ting

Fig. 5.55. Kommunikation mellem programmer over et netværk inddeltes i 5 lag. Hvert lag løser bestemte opgaver. Lavere lag har simplere opgaver og færre afhængigheder. Højere lag har mere komplekse opgaver og flere afhængigheder.

Bemærk, at hvert lag i Figur 5.55 er afhængigt af laget nedenunder. Faktisk løser det underliggende lag netop de opgaver, det overliggende lag har brug for!

5.1.3. TJENESTER OG GRÆNSEFLADER

Hvert lag “snakker med” de tilstødende lag gennem veldefinerede grænseflader. Fx er der helt bestemte måder, en applikation sender beskeder til transportlaget på. Hvert lag tilbyder laget ovenfor en tjeneste (*service*) gennem grænsefladen. Laget har en service-implementation, der hviler på tjenester fra laget nedenfor lag. Man kalder de services, et lag tilbyde laget ovenfor, for lagets *service-model*.

Ordene “service” og “tjeneste” er i denne sammenhæng ensbetydende og vi bruger begge.

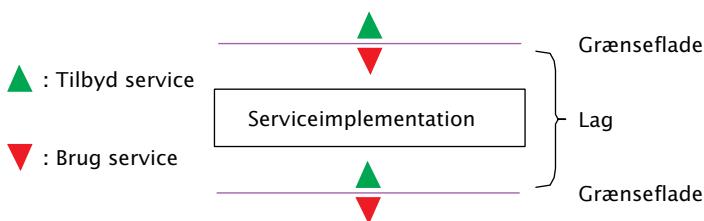


Fig. 5.56. Hvert lag har en servicemodel og en serviceimplementation. Services benyttes via en grænseflade.

Postvæsnet tilbyder fx servicen “Fejlfri levering af post” til laget ovenfor af brevskrивere. Grænsefladen til tjenesten består af postkasser. Du benytter dig som brevskriver af postvæsnets service ved at kontakte grænsefladen, dvs. ved at poste brevet i en postkasse. For at tilbyde “Fejlfri levering af post” gør postvæsnet selv brug af tjenester fra laverliggende lag – nemlig tjenesten “kørsel af post mellem posthuse” fra fragtfirmaet i fragt-laget.

Figur 5.55 kan tegnes som et hierarki af services, der hviler på hinanden. Det er en god øvelse at gøre det!

Opdelingen i lag har flere fordele:

- ◆ *Ansvarsdeling*: Hver enkelt lag er ansvarlig for nogle bestemte opgaver. Disse opgaver bekymrer de andre lag sig ikke om.
- ◆ *Modularitet*: Et lag kan udskiftes med en ny service-implementation, så længe grænsefladen overholdes. (Fx kan vi udskifte et netkort, eller posthuset kan skifte til et nyt fragtfirma).

I afsnit 10.3 kan du læse mere om fordelene ved ansvarsdeling og modularitet.

5.1.4. PROTOKOLLER OG PROTOKOLSTAK

Kommunikation mellem to forskellige lag benytter *grænseflader*. Kommunikation indenfor et enkelt lag defineres af en *protokol*. En protokol er, (som vi lærte i afsnit 4.3), et sæt retningslinier for kommunikation mellem to programmer eller maskiner. I de kommende delkapitler kommer vi ind på de enkelte lags protokoller – her giver vi blot en liste med vigtige eksempler. Du genkender protokollerne HTTP, SMTP og DNS fra kapitel 4. Du kan måske også huske begrebet “IP-adresser”? De har deres navn fra protokollen IP!

LAG	EKSEMPLER PÅ PROTOKOLLER
Applikationslag	HTTP, SMTP, DNS
Transportlag	TCP, UDP
Netværkslag	IP
Data-link-lag	ARP
Fysisk lag	Ethernet

Fig. 5.57. Eksempler på protokoller til netværkskommunikation.

EKSEMPEL

PROTOKOLLER AFHÆNGER AF HINANDEN

HTTP afhænger af TCP, forstået på den måde at HTTP benytter tjenester, som TCP tilbyder. DNS afhænger af UDP: Uden de tjenester, UDP tilbyder, virker DNS ikke. Både TCP og UDP benytter sig af tjenester, som tilbydes af IP:

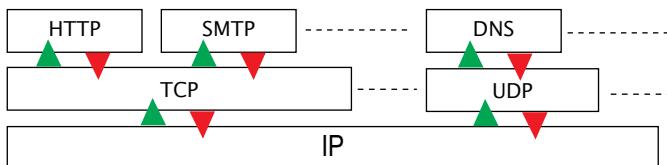


Fig. 5.58. Udsnit af internet-protokolstakken: Stabling af protokoller.

ORDFORKLARING

PROTOKOLSTAK

En samling samarbejdende protokoller kaldes tit en protokolstak. Eftersom enhver protokol afhænger af en protokol i et lavereliggende lag, kan man tænke på en samlingen af protokoller som “stablet” oven på hinanden i en stak – ligesom vi har tegnet situationen i Figur 5.58.

Den vigtigste protokolstak er *internettets* protokolstak. Den består af protokollerne i Figur 5.57 plus mange flere. Ethvert moderne operativsystem har alle protokollerne fra internettets protokolstak indbygget. Med dem kan en computer uden videre kommunikere over internettet med andre computere, der også har internet-protokolstakken installeret.

WWW

INTERNET-PROTOKOLSTAKKEN

På **WWW** kommer vi ind på de vigtigste protokoller i internettets protokolstak, såsom HTTP, SMTP, DNS, FTP, SSL, TCP, IP, m.fl.

Hvis vi skal opsummere, er internettet altså *en verdensdækkende fysisk infrastruktur, der muliggør kommunikation ved hjælp af et bestemt sæt protokoller*.

5.3. APPLIKATIONSLAG

Applikationslaget består af de protokoller, der bruges af brugerprogrammer til at kommunikere over netværk. Fx vil en browser gerne bruge protokollen HTTP til at sende forespørgsler over netværk. Altså er HTTP en protokol i applikationslaget. Emailklienter vil gerne bruge SMTP, POP3 eller IMAP til at sende emails til eller hente emails fra email-servere. Emailprogrammer vil altså gerne kommunikere via disse protokoller. Så også SMTP, IMAP og POP3 er protokoller i applikationslaget.

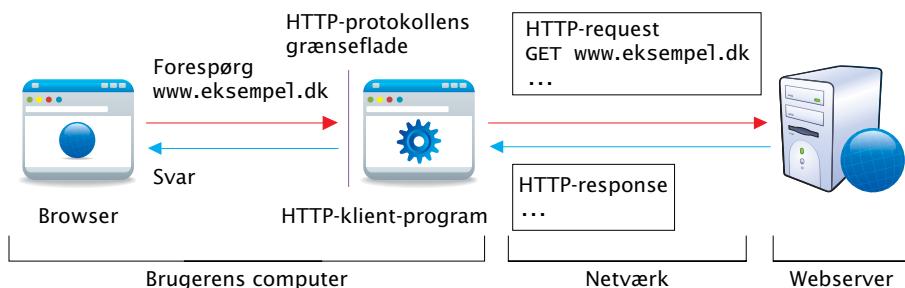


Fig. 5.59. En browser bruger HTTP-protokollen via HTTP-protokollens grænseflade. Et HTTP-program i computerens internet-protokolstak udfører den egentlige HTTP-kommunikation.

Når applikationer snakker sammen over netværk vha. en protokol i applikationslaget, bekymrer de sig ikke om, hvordan forbindelsen mellem dem er skabt. Faktisk er de også ligeglade med, at kommunikationen foregår over et netværk, bare den virker. De ser kun protokollens grænseflade.

SERVICEMODELLER FOR TRE PROTOKOLLER I APPLIKATIONSLAGET		
HTTP tilbyder	SMTP tilbyder	DNS tilbyder
at foresørge webindhold på en web-adresse med en HTTP-forespørgsel at modtage indholdet i et HTTP-svar	at sende emails til en emailserver	at oversætte domænenavne til IP-adresser

Fig. 5.60. Vi opsummerer protokollerne HTTP, SMTP og DNS ved deres servicemodeller.

5.4. TRANSPORTLAG

Transportlaget sørger for logisk kommunikation mellem processer på forskellige værter. Med logisk kommunikation mener vi, at selvom to kommunikerende processer snakker sammen over et netværk, måske endda over tusindvis af kilometers afstand, så ”føler” de to processer, at de er direkte forbundet. En proces, der vil kommunikere via HTTP, regner fx med at forespørgsler kan afsendes og svar modtages – uden overhovedet at bekymre sig om det underliggende netværk.

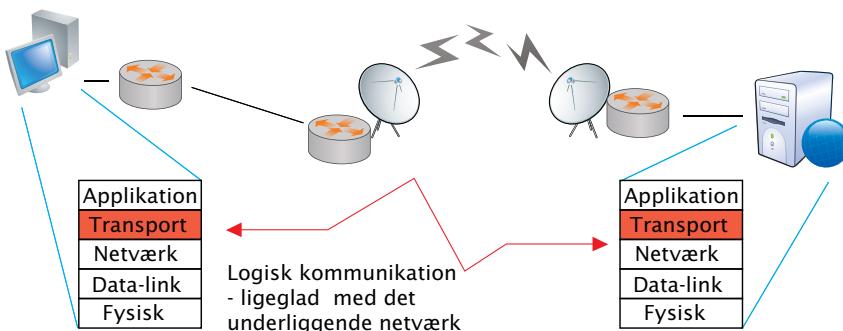


Fig. 5.61. Transportlagets funktion: Logisk kommunikation.

5.4.1. PROCES-ADRESSERING OVER NETVÆRK

Da værter typisk har mange processer, der gerne vil kommunikere over netværk, har transportlaget ansvaret for at beskeder ikke afleveres hos den forkerte proces hos modtager-værtens. At kunne lokalisere den rigtige proces kaldes *proces-adressering over netværk*.

ANALOGI

BREVSKRIVENDE BØRNEFAMILIER

I en lejlighed i København bor familien A. I en villa i Århus bor familien B. De mange børn i familierne A og B elsker at skrive breve til hinanden, men hvert barn nægter at forlade sit værelse. Hr. A påtager sig derfor at indsamle A-børnenes breve ved hvert børneværelsес indgang og poste dem i postkassen udenfor. Når der kommer post, sørger hr. A for at fordele brevene korrekt imellem børnene ved at gå en runde ved børneværelserne. I Århus har hr. B en tilsvarende rolle. Følgende hjælper med at forstå transportlagets fordelings-rolle:

Hus	≈ vært på netværk
Børn	≈ processer hos en vært
Indgang til børneværelse	≈ port
Brev	≈ besked, proces vil sende over netværk
hr. A / hr. B	≈ transportlag på vært A / vært B

Transportlaget adresserer processer over netværk ved hjælp af *porte*. En proces er tilkoblet transportlaget på en port. Hver port har et portnummer, der er et tal mellem 0 og 65536 ($=2^{16}$).

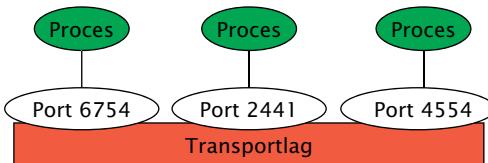


Fig. 5.62. Porte.

Protokoller i applikationslaget er tilknyttet nogle faste portnumre, som ikke udskiftes. Fx bruger HTTP altid port 80 og SMTP altid port 25. Andre portnumre står til rådighed og kan bruges af forskellige processer på forskellige tidspunkter.

Hvis en besked skal sendes fra en værts applikationslag til en andens, angiver man portnummeret for at ramme den rigtige protokol. På den måde får en HTTP-server ikke pludselig en SMTP-besked, som den ikke kan forstå.

En besked kan så finde vej til den rigtige proces over netværket, hvis den forsynes med portnummeret på processen og IP-adressen på maskinen, hvor processen kører. Hvor applikationslaget kommunikerer med beskeder (fx HTTP-beskeder), kommunikerer trans-

portlaget med *segmenter*. Segmenter er beskeddata forsynet med med afsender- og modtager-port samt varierende ekstra information.

	Modtagers portnummer	Modtagers portnummer
Segment	Ekstra information	
	Besked-data	

Fig. 5.63. Transportlags-segment.

En port er som en stikkontakt. Hvis en klient vil “snakke HTTP” med en server, skal klienten bruge “modtagers portnummer = 80” – klienten skal have det rigtige “stik” til stikkontakten.

WWW

PORTNUMRE

På **WWW** finder du en oversigt over de mest almindelige faste portnumre.

5.4.2. PÅLIDELIG OG UPÅLIDELIG KOMMUNIKATION

Transportlaget sender data via netværkslaget. Det giver umiddelbart to forhindringer:

- data kan gå tabt i netværkslaget,
- data kan blive uforudsigeligt ændret under transporten i netværkslaget.

Hvis man accepterer disse vilkår for data-transmission, får man en *upålidelig protokol*. Hvis man forhindrer fejl og tab, får man en *pålidelig protokol*. Der findes både pålidelige og upålidelige protokoller i transportlaget.

Transportlaget har altså to servicemodeller:

- *Upålidelig service*: Garanterer ikke levering af beskeder. Hvis en besked leveres, kan den have fejl – og i så fald kasseres den (svarende til, beskeden ikke blev leveret). Hvis flere beskeder afsendes efter hinanden, ankommer de, der når frem, muligvis i forkert rækkefølge.
- *Pålidelig service*: Garanterer, at beskeder leveres til modtageren uden fejl eller tab. Hvis flere beskeder afsendes efter hinanden, ankommer de i den rigtige rækkefølge.

Ingen af de to services giver garantier omkring *leveringstidspunkt*.

Fejl i leverede beskeder opdages ved at sende en besked-*checksum* med sammen med beskeden. Checksummen er et tal, afsender har beregnet på grundlag af beskedens indhold. Modtager beregner også en checksum for den modtagne besked. Hvis de to tal ikke stemmer overens, er det fordi beskedens indhold er blevet ændret siden afsendelsen – altså er der sket en fejl. I så fald kasseres beskeden.

ANALOGI

PÅLIDELIG OG UPÅLIDELIG TRANSPORT

Du skal have transporteret et nyt film-manuskript over til Søren og kan vælge mellem to transportfirmaer, Pålidelig Transport A/S og Upålidelig Transport A/S.

Vælger du Pålidelig Transport A/S, ankommer hele manuskriptet hos Søren, en side ad gangen og i den rigtige rækkefølge. Alle siderne er ubeskadigede. Pålidelig Transport A/S giver ingen garantier for, hvornår hele manuskriptet er leveret.

Vælger du Upålidelig Transport A/S, vil et ukendt antal af manuskriptets sider ankomme hos Søren, og de sider, der ankommer, ankommer muligvis i forkert rækkefølge. Måske ankommer der også nogle beskadigede sider, som Upålidelig Transport A/S kasserer.

Allerede nu virker det oplagt at spørge: Hvad skal vi dog med upålidelig kommunikation? Svaret er: Det er nemt at lave og hurtigt at bruge! En upålidelig protokol til transportlaget skal bare udvide netværkslaget – *hvor al kommunikation er upålidelig!* (jf. Figur 5.55) – med følgende to services:

- ◆ kassering af beskeder med fejl i,
- ◆ proces-adressering over netværk.

I mange situationer er det vigtigere, at data når frem hurtigt og nogenlunde i rækkefølge, end at data når frem 100 % korrekt og i 100 % korrekt rækkefølge. Det gælder fx lyd, video og spil. En smule flammer i billedet, en svag knasen i lyden eller bittesmå forsinkelser på geværkuglerne i et skydespil er ikke afgørende. Det er på den anden side klart, at pålidelig transmission altid bruges til fx emails, overførsel af webindhold og netbank-transaktioner.

En proces bruger upålidelig kommunikation ved at sende sin besked til den upålidelige transport-protokol og så ellers ”krydse fingre for” at beskeden når frem til den rette modtager (en anden proces på en anden vært).

EKSEMPEL

UDP

UDP er en upålidelig transportlagsprotokol. UDP står for *User Datagram Protocol*. (Segmenter, der afsendes af UDP, kaldes undertiden datagrammer, deraf navnet).

5.4.3. PÅLIDELIG KOMMUNIKATION VHA. KVITTERINGER

Når nu netværkslaget kan miste eller ødelægge data – hvordan laver man så pålidelig kommunikation?

Første skridt på vejen til en pålidelig kommunikation er at fortælle modtageren, at man vil til at sende data. Det sker med et tre-trins *handshake* – svarende til to mennesker giver hånd ved deres første møde.

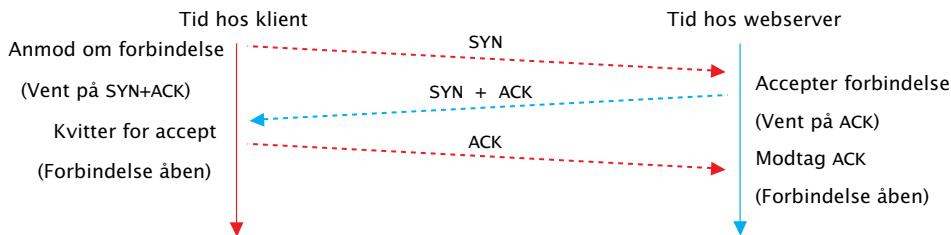


Fig. 5.64. Tre-trins handshake. SYN står for synchronize, ACK for acknowledge.

Efter et handshake ved begge parter, at der skal kommunikeres pålideligt, og de har gjort klar til denne kommunikation: Der er oprettet en forbindelse mellem afsender og modtager af data. (Pålidelig kommunikation kaldes derfor også forbindelsesorienteret).

Næste skridt på vejen: Hvad gør vi ved tab og fejl? Den eneste løsning er at *re-transmittere* data, der tabes eller ødelægges. Jo mere data, der afsendes, desto mere skal retransmitteres i tilfælde af fejl eller tab. Man vil gerne retransmittere så lidt som muligt – det giver en hurtigere data-overførsel og en mindre netværksbelastning. Bl.a. derfor opdeles store beskeder tit i mindre dele, *segmenter*, der afsendes separat. Denne opdeling kaldes segmentering. Hvis et segment tabes eller når frem med fejl i, behøver man kun retransmittere segmentet:

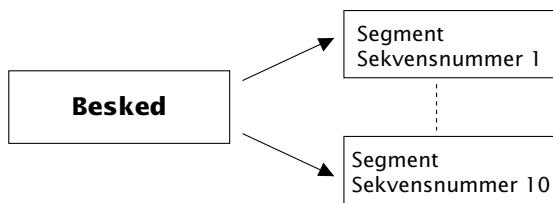


Fig. 5.65. Her segmenteres stor besked til 10 mindre segmenter, der forsynes med sekvensnumre.

Hvert segment får sit eget unikke sekvensnummer (på engelsk: *sequence number*), der identificerer segmentet og fortæller om dens placering i sekvensen af segmenter. Udover

sekvensnummer forsynes segmentet (som nævnt) med portnummer og på både afsender og modtager. Segmentering er praktisk, men det er ikke strengt nødvendigt for korrektheeden af en pålidelig transportlagsprotokol.

Sidste skridt på vejen er *kvitteringer*. Når et segment ankommer uden fejl, spørger modtager sig selv:

“Hvad er det højeste sekvensnummer blandt de segmenter, jeg har modtaget, sådan at alle segmenter med lavere sekvensnumre også er nået frem uden fejl?”

Svaret sendes som en kvittering til afsenderen, der så ved, at alle segmenter med nummer mindre end det tilsendte er nået vel frem. En sådan kvittering kaldes også et ACK.

Vi tager et eksempel for at forstå kvitteringssystemet.

EKSEMPEL

KVITTERINGER (ACKs)

Lad os sige, at modtager allerede har fået segmenterne med sekvensnumrene 1,2,3,7,8,9 og 10.

Så får modtager segmentet med sekvensnummer 4 og sender “ACK: 4” som kvittering. Dernæst får modtager segmentet med sekvensnummer 6 og sender “ACK: 4” som kvittering, fordi segment 5 stadig mangler. Når segment 5 ankommer, sendes “ACK: 10”, fordi alle segmenter op til nr. 10 nu er vel modtaget.

Modtageren bruger desuden sekvensnumrene til at sætte segmenterne sammen i den rigtige rækkefølge for at genskabe den oprindelige besked:

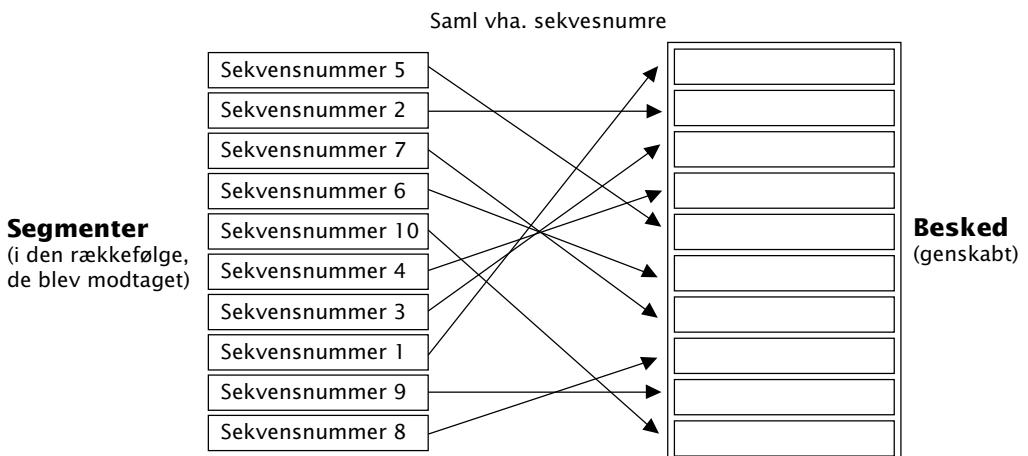


Fig. 5.66. Segmenter samles til besked vha. sekvensnumre.

Hvis der går for lang tid fra afsendelse af et segment til kvitteringen modtages, konkluderer afsenderen, at segmentet må være gået tabt eller i stykker. Segmentet re-transmitteres derfor. Ventetiden kaldes timeout-tiden.

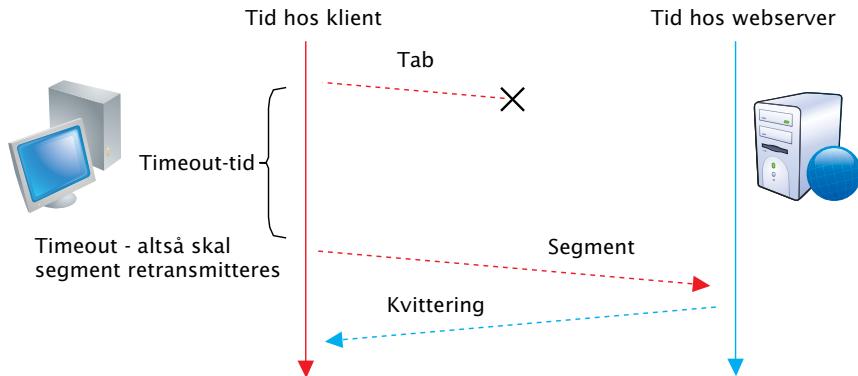


Fig. 5.67. Tab af segmenter imødegås med timeout.

Med disse mekanismer – handshake, segmentering og kvitteringer – kan pålidelig kommunikation realiseres. Skal flere beskeder udveksles over den pålidelige forbindelse, kan det ske i korrekt rækkefølge ved hjælp af sekvensnumre. Der skal selvfølgelig tages højde for situationer som:

- tab af kvitteringer,
- modtagelse af to ens segmenter,
- modtagelse af to ens kvitteringer.

EKSEMPEL

TCP

TCP er en meget udbredt pålidelig transportlagsprotokol. TCP står for Transmission Control Protocol.

Nogle argumenter for upålidelig kommunikation bunder i ulempen ved pålidelig kommunikation:

- pålidelig transmission er langsomt (pga. handshaking og retransmissioner),
- pålidelig transmission kan forårsage "tilstopning af netværket" (pga udveksling af kvitteringer og retransmissioner).

5.5. NETVÆRKSAG

Vi har set, at transportlaget tilbyder kommunikation *mellem processer* på forskellige værter på et netværk. For at levere denne service, udnytter transportlaget at netværkslaget tilbyder kommunikation *mellem værter*.

	Modtagers IP-adresse	Afsenders IP-adresse
Pakke	Ekstra information	
	Segment	

Fig. 5.68. For at sende data mellem værter forsyner netværkslaget segmenter fra transportlaget med IP-adresser på segmentets modtager og afsender. Derved fås en pakke.

Netværkslaget er en “så-godt-som-muligt-service” (på engelsk: *best effort*) . Det betyder, at netværkslaget “gør sig umage” for at levere pakker fra afsender og modtager, men *ikke giver nogen garantier*. Pakker kan blive forsinket så længe det skal være eller gå tabt. Netværkslaget er altså *upålideligt*.

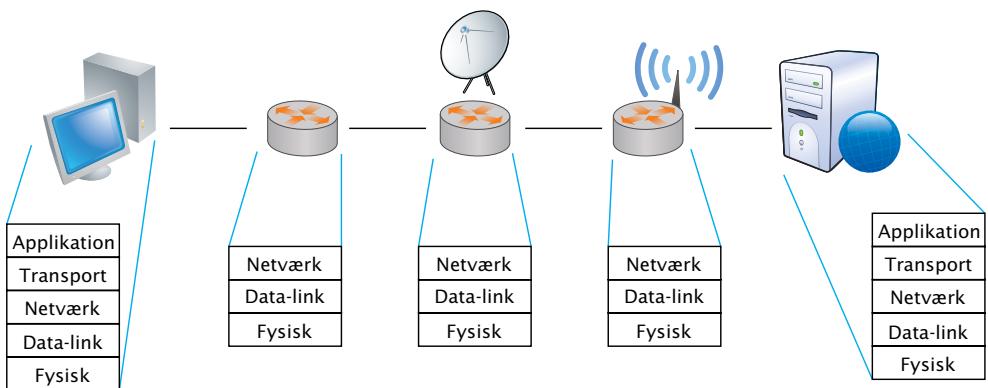


Fig. 5.69. Transportlaget findes kun på værter (computere og servere). Netværkslaget findes også på alle routere.

Lad os sige, at der er en rute mellem A og B, bestående af 6 data-links, som vist i Figur 5.70:

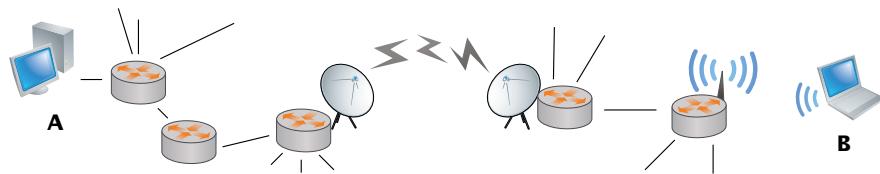


Fig. 5.70. Kommunikation mellem 2 værter over 6 data-links.

Et segment fra transportlaget kommer da fra vært A til vært B på følgende måde:

1. Segmentet forsynes med IP-adresserne på modtageren (vært B) og afsenderen (vært A). Dermed bliver segmentet til en pakke.
2. Netværkslaget på netværket router pakken gennem ruten fra Figur 5.70. Den samlede transport kaldes *routing*. Undervejs kan pakken gå tabt, blive ødelagt eller blive kasseret, hvis fx vært B ikke kan nås.
3. Hvis pakken ankommer hos vært B, udpakker netværkslaget segmentet og giver til transportlaget hos vært B.

5.5.1. ROUTING

Som du nok har gættet, er det interessante altså: Hvordan virker routingen? Først er det vigtigt at huske, at en router har et antal data-links til andre routere, og ingen router kender andre routere end sine nabo-routere:

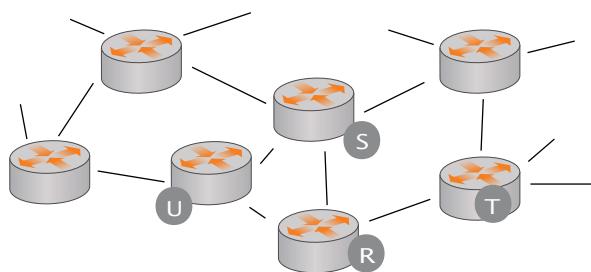


Fig. 5.71. Router-netværk. Routeren R kender kun routerne S, T og U.

Routing lykkes ved, at enhver router altid sender videre til en router, der tættere på målet. Det kaldes *forwarding*.

ANALOGI

FORWARDING ≈ VEJVISNING I RUNDKØRSEL

Vi husker, at en router er som en rundkørsel. En routers forwarding kan sammenlignes med vejvisnings-service i rundkørslen: En bil kører ind i rundkørslen via vej A. Chaufføren: "Jeg skal til Hovedgaden 11, Køge, Danmark. Kan du hjælpe mig?". Vejviseren: "Ja! Jeg kender ikke den fulde rute, men du kommer tættere på, hvis du kører ud af rundkørslen via vej F".

Til forwarding har hver router en forwarding-tabel:

HVIS IP-ADRESSE STARTER MED...	...SÅ VIDERESEND TIL NABO-ROUTER
113.4.5.XXX	U
86.99.XXX.XXX	T
200.11.11.1XX	U
130.117.01X.XXX	S
...	...

Fig. 5.72. Forwarding-tabel for routeren R fra Figur 5.71. X'er markerer vilkårlige cifre i IP-adressen.

Når en router med forwarding-tabellen ovenfor modtager en pakke med modtager-IP-adressen 86.99.191.254, kan den roligt videresende pakken til sin nabo-router T. Så sørger T for den videre routing.

Men hvordan har routeren R fået opbygget sin forwarding-tabel? Det sker ved en *routingalgoritme*, der sikrer at enhver router altid har bedst mulig information i sin forwarding-tabel. Routingalgoritmen sørger fx også for, at en router kender den korteste vej til målet, hvis der er flere mulige veje. "Korteste vej i et netværk" er en samlebetegnelse, der bl.a. indregner rutens længde i antal data-links og båndbredden på de forskellige stykker af ruten. En rute via et kabel tværs over Atlanterhavet er fx længere end en rute til nabobygningen.

På vore dages netværk er der er forskellige routingalgoritmer i brug. Vi beskriver her et eksempel på en *decentral* type, der virker ved at sende "bølger af opdatering" gennem netværket. Opdateringsbølgerne opbygger gradvist forwarding-tabellerne hos alle routere. Til sidst indeholder alle routere information nok til at kunne viderestille alle pakker.

EKSEMPEL

ROUTINGALGORITME

1. Hver router fortæller jævnligt sine nabo-routere, hvilke IP-adresser den kender en rute til, og "hvor lang" ruten er. Ruten går typisk gennem flere led.
2. Denne melding kan bevirket at en nabo-router indser: "Hov – med denne information har jeg jo en kortere vej til modtagere med disse IP-adresser?" og på denne baggrund opdaterer sin egen forwarding-tabel.
3. Når tabellen er opdateret, sender den anden router sin (opdaterede) information om router og deres længde til sine nabo-routere.
4. Trin 2 og 3 gentages, indtil ingen routere opdaterer deres tabeller længere.

Der er i alt fire tilfælde, hvor en router opdaterer sin tabel:

- ◆ Når routeren tilsluttes en netværk. Routerens tabel indeholder endnu ingenting, og routeren spørger derfor sine naboer: "Hvilke IP-adresser kender I vej til?"
- ◆ Hvis routeren får en ny nabo, der spørger "Hvilke IP-adresser kender I vej til?"
- ◆ Hvis en opdateringsbølge er i gang.
- ◆ Hvis routeren i et stykke tid ikke hører fra en nabo. I det tilfælde regnes nabo-routeren for "ikke længere aktiv" (nabo-routeren kan være blevet slukket, taget af netværket, eller være gået i stykker).

EKSEMPEL

IP

Den mest udbredte protokol i netværkslaget er IP (Internet Protocol), der har givet navn til IP-adresser.

5.5.2. EKSTRA INFORMATION I PAKKER

Feltet "ekstra information" i en pakke i netværkslaget udfyldes ofte med (bl.a.):

- ◆ *TTL (time-to-live)* : Et tælle-felt, der tælles ned for hvert data-link, pakken passerer. Når værdien 0, kasseres pakken. TTL forhindrer, at uheldige pakker (med forkert modtageradresse, hvis modtager er gået offline, mv.) rejser rundt på netværket for altid og på medvirker til at tilstoppe det fuldstændigt.
- ◆ *Checksum*: En særskilt checksum for felterne med ekstrainformation. Opdateres på ny hos hver router, pakken passerer.

- ◆ *Transportlagsprotokol:* Angiver den benyttede transportlagsprotokol. Modtageren bruger feltet til at viderestille pakken til rette protokol i laget ovenfor.

5.5.3. OFFENTLIGE IP-ADRESSE

En særlig internet-organisation varetager en fordeling af de offentlige IP-adresser, der er til rådighed, på en fair måde til regioner i verden. Der er ”kun” 4.294.967.296 (= 2^{29}) IP-adresser (IPv4) – det har vist sig at være alt for få. Alle lande, organisationer, firmaer, institutioner og undertiden privatpersoner vil gerne have en bid af kagen.

En ny version af IP (IPv6) med en ny type IP-adresser, der er flere af, har dog efterhånden vundet indpas.

5.5.4. NAT OG LOKALE IP-ADRESSE

Ofte ønsker et lokalnetværk, der er tilkoblet internettet via en eller flere routere, at have en fælles offentlig IP-adresse for alle computere i lokalnetværket. Det kan gøres med teknologien *NAT* (Network Address Translation), der giver alle værter på lokalnetværket *lokale IP-adresser*. Lokale IP-adresser er ikke offentligt tilgængelige og tildeles efter en særlig protokol på lokalnetværket. Udadtil har lokalnetværket kun én IP-adresse, og det er routerens offentlige IP-adresse. Routeren kaldes i denne situation en *gateway*, fordi den er lokalnetværkets ”hovedport” ud til internettet. Routeren er så ansvarlig for at omstille indgående kommunikation til rette lokale IP-adresse.

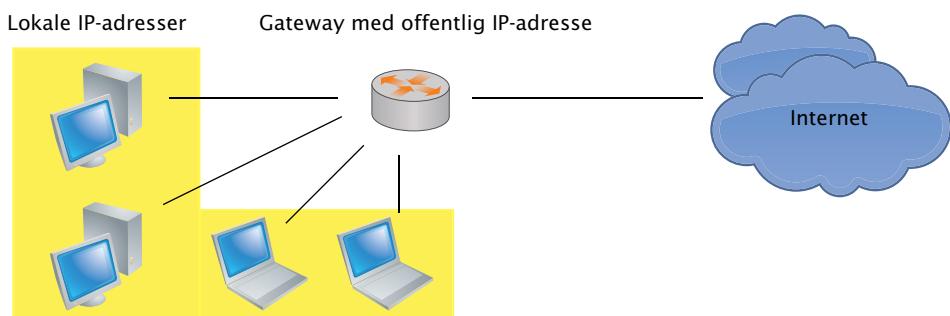


Fig. 5.73. NAT og gateway.

NAT har to primære formål:

- ◆ der bruges færre af de offentlige IP-adresser (hvilket er godt, dem var der jo for få af),

- man kan sikre lokalnetværket mod indtrængen eller virusinfektion ved at træffe sikkerhedsforanstaltninger et sted: på gateway'en. Uden NAT ville man skulle sikre hver enkelt vært på lokalnetværket.

Den enkelte computers lokale IP-adresse kan være fast eller skifte jævnligt.

5.6. DATA-LINK-LAGET

Data-link-laget er ansvarlig for at sende data over et data-link, dvs. mellem to apparater, der er direkte forbundne via et kabel eller en trådløs forbindelse. Apparaterne kan være computere eller routere.

Der er to typer data-links:

- Point-to-point*: To apparater er direkte forbundet over en "privat" forbindelse.
- Broadcast*: To eller flere apparater deles om en kanal (et fysisk medium, der kan kommunikeres over). Apparaterne kommunikerer ved at *broadcaste* på kanalen – hvis et apparat sender data på kanalen, modtager alle andre signalet.

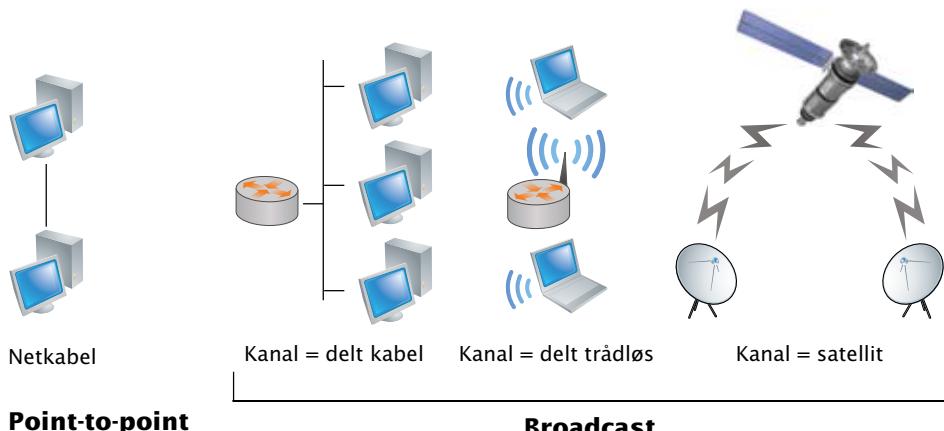


Fig. 5.74. Data-links: Point-to-point og broadcast.

Protokoller til transmission af data over et point-to-point data-link er simple, og vi kommer ikke ind på dem. I stedet kigger vi på den mere interessante situation: broadcast. Her skal protokollerne løse to problemer:

- Flere forskellige apparater tilkoblet samme kanal kan vælge at broadcaste deres data samtidig. Herved ødelægges alt sendt data, da signalerne sammenblandes – det kaldes meget sigende en *kollision*. Det svarer til, at alle gæster til en fest snakker i munden på hinanden, så ingen kan høre, hvad nogen siger.
- Som regel ønsker et apparat at sende data til en bestemt modtager, fremfor at sende til alle (som det er tilfældet med broadcast).

Protokoller til broadcast-links kontrollerer adgangen til det fysiske medium, så kollisioner kan håndteres – derfor kaldes sådanne protokoller for *MAC-protokoller* (MAC = Media Access Control). Vi ser et eksempel på kollisionshåndtering i afsnit 5.7. Her og nu kigger vi på, hvordan problem 2 løses.

Ethvert appart tilkoblet et netværk har et netkort, der er ansvarlig for faktisk at afsende og modtage fysiske signaler over netværket. Netkortet er tilkoblet computerens bundkort. Ethvert netværkskort har en unik adresse: *MAC-adressen*.

ORDFORKLARING

MAC-ADRESSE

En MAC-adresse består af 6 gange 2 hexadecimale tegn, adskilt af bindestreg. Eksempelvis er

A2-EB-72-07-C1-23

en MAC-adresse (de 16 hexadecimale tegn er 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

I modsætning til en IP-adresse kan MAC-adressen aldrig ændres – den brændes fra fabrikkens side direkte i netkortet (i ROM). MAC-adressen kaldes også den fysiske adresse, fordi den er påtrykt kortet fysisk. En særlig organisation, der tager sig af mange internetspørgsmål, sørger for at netkort-producenter ikke bruger de samme MAC-adresser, når de fremstiller netkort. Der er 2^{48} forskellige MAC-adresser (dvs. ca 140.000 milliarder) – så der er nok at tage af.

ANALOGI

PERSONNUMMER OG POSTADRESSE

En person har det samme personnummer hele livet, men kan have flere forskellige postadresser i løbet af sit liv. På samme måde har et netkort den samme MAC-adresse hele sit liv, men det skifter fx IP-adresse, hvis computeren får en ny lokal IP-adresse, skifter netværk eller netkortet installeres på en anden computer.

Data sendes over et data-link som såkaldte *frames*.

	Modtagers MAC-adresse	Afsenders MAC-adresse
Frame	Ekstra information	
	Pakke	

Fig. 5.75. Frame i data-link-laget. Framen indeholder MAC-adresserne for afsender-maskinen og modtager-maskinen.

En protokol i data-link-laget står for at fremstille frames og overgive dem til netkortet. Netkortet er derefter ansvarlig for at levere framen til rette modtager. Protokollerne i data-link-laget afhænger altså af, at netkortene virker. Vi forklarer data-link-protokollerne til broadcast vha. eksemplet ARP.

EKSEMPEL

ARP (ADDRESS RESOLUTION PROTOCOL)

Protokol til kommunikation over et enkelt data-link, baseret på en ARP-tabel i det opkoblede apparats RAM. ARP-tabellen oversætter IP-adresser til MAC-adresser.

En ARP-tabel ser i store træk ud som følger:

IP-ADRESSE	MAC-ADRESSE
88.100.21.102	A2-E8-72-07-C1-23
123.221.9.45	08-1C-A3-FF-43-44
...	...

Når ARP-protokollen modtager en IP-pakke til en IP-adresse, der allerede er i ARP-tabellen, kan den sende pakken afsted uden videre: Bare giv pakken og modtagerens MAC-adresse til netkortet. Der er imidlertid ofte situationer, hvor den rigtige IP-adresse ikke er i ARP-tabellen:

- ◆ ARP-tabellen lagres i RAM og forsvinder dermed, når computeren slukkes,
- ◆ IP-adresserne på netværket kan skifte (når det sker, slettes ”ugyldige” oversættelser mellem IP- og MAC-adresser),
- ◆ et nyt apparat kan være blevet tilkoblet netværket – og er endnu ikke i tabellen.

Derfor har ARP-protokollen på computeren A en måde at finde en efterspurgt IP-adresse:

1. Computer A's ARP-program spørger ud på netværket "Hvem har IP-adressen 88.100.21.102?". Det gøres ved *broadcast* – spørgsmålet sendes som en særlig frame til den særige MAC-adresse FF-FF-FF-FF-FF-FF. Alle netkort på samme data-link som computer A modtager frames til broadcast-modtager-adressen.
2. Alle modtagere spørger deres ovenpå liggende netværkslag – "Har jeg IP-adresse 88.100.21.102?"
3. Det netkort, hvis IP-adresse faktisk er 88.100.21.102, svarer computeren A og siger "Det er mig, og jeg har i øvrigt MAC-adresse A2-EB-72-07-C1-23".
4. Computeren A opdater sin ARP-tabel med denne nye oplysning. Derefter pakkes den IP-pakke, der ønskes afsendt, som en frame, og afsendes – via netkortet – til MAC-adressen A2-EB-72-07-C1-23.

ANALOGI

PERSONNUMMER OG POSTADRESSE (FORTSAT)

I personnummer/postadresse-analogien svarer broadcasting til at, at en (gal?) mand i firmaet Netkort A/S ráber i kantinen: "Hvad er personnummeret på den person, der arbejder i kontor A-114, 2. sal, Netkort A/S, Data Link Allé, Protokolkøbing, Danmark?" – hvorefter alle ansatte passivt gumler videre – undtagen netop den, der har kontor A-114. Hun svarer "Det er mig, og mit personnummer er 010675-2422!".

5.7. DET FYSISKE LAG

Hvordan virker et netkort? Spørgsmålet kan faktisk koges ned til: Hvordan håndterer netkort kollisioner på et data-link? Når dette spørgsmål er afklaret, ved vi hvordan data-link-laget fungerer – og så forstår vi alle 5 protokollag! En udbredt protokol til det fysiske lag er *Ethernet*-protokollen, som vi forklarer det fysiske lag udfra.

EKSEMPEL

ETHERNET

Protokol, der definerer hvordan Ethernet-netkort tilkoblet samme data-link kommunikerer med hinanden.

Teknisk foregår transmission over Ethernet som følger:

1. Et netkort vil sende en frame, og lytter derfor på sit data-link: "Er jeg mon i gang med at modtage data på mit data-link lige nu?"
2. Hvis ja, venter netkortet, til modtagelsen er færdig – og starter så forfra fra trin 1.
3. Hvis der ikke umiddelbart modtages noget, sender netkortet sin frame.

- Hvis ingen netkort sender noget samtidig, når signalet frem til de netkort, der er forbundet til samme data-link.
- Hvis andre netkort samtidigt beslutter sig for at sende en frame, opstår en kollision på data-linket. Alle signaler ødelægges, og framen skal sendes igen. Netkortet forsøger sig igen med trin 1 lidt senere (ventetiden fastsættes tilfældigt og skifter fra gang til gang).

Ethernet virker måske som en skør protokol, men den har vist sig effektiv!

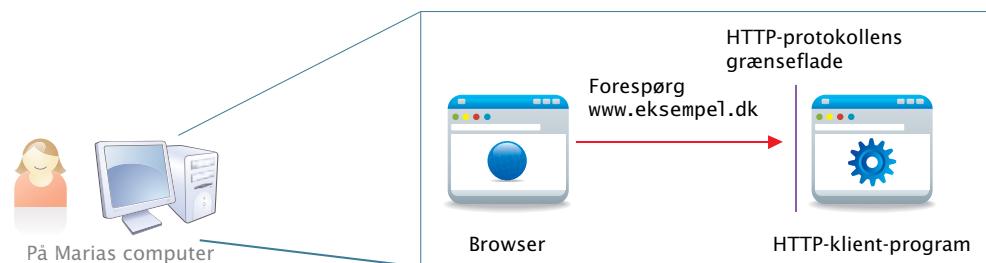
5.8. ET EKSEMPEL TIL AT ILLUSTRERE DET HELE

Lad os nu tage et eksempel til at illustrere alle trinnene i en lagdelt kommunikation. Vi forestiller os en situation, hvor Maria vil åbne websiden www.eksempel.dk med en browser på sin computer. Vi ser på hvad der sker, fra Maria indtaster webadressen i browseren, til beskeden når webserveren. Vi forenkler situationen og ser bort fra både eventuelle DNS-opslag, handshaking og diverse ekstra information, de enkelte lag påfører det data, der skal sendes.

Først indtaster Maria webadressen i browseren:



Browseren sender denne forespørgsel videre til klient-programmet til HTTP-protokollen på Marias computer. Programmet er en del af internet-protokolstakken, en samling programmer, der følger med operativsystemet på computeren:



HTTP-klient-programmet pakker forespørgslen som en HTTP-forespørgsel:

Besked i applikationslaget

```
GET /index.html HTTP/1.1  
Host: www.eksempel.dk
```

Heresfter sender HTTP-klient-programmet sin besked videre til transportlaget, der står for at transportere beskeder for applikationslaget. HTTP benytter sig af en pålidelig protokol i transportlaget, da det er vigtigt at forespørgslen når frem.

Lad os (lidt urealistisk, men hvad) forestille os at transportlaget synes, at HTTP-beskeden er så stor, at den skal segmenteres – dvs. opdeles i segmenter – for at blive sendt praktisk. Da udformer transportlaget to segmenter, med hver deres sekvensnummer, og forsyner dem med afsender- og modtager-portnummer. Da modtageren er en HTTP-server, ved transportlaget, at portnummeret skal være 80. HTTP-klient-programmet har kontaktet transportlaget gennem en port – fx 4551 – der bruges som afsender-port.

Portnumre	4551	80
Sekvensnummer	1	
Besked-data	GET /index.html HTTP/1.1	

Segment

4551	
2	
Host: www.eksempe1.dk	

Segment

Transportlaget er nu klar til at lade netværkslaget forsøge levering af de to segmenter. Netværkslaget kan route data gennem netværk ved hjælp af IP-adresser, og forsyner derfor hvert segment med IP-adresserne på afsender og modtager. Lad os sige, at www.eksempe1.dk er tilgængelig fra en webserver med IP-adressen 100.72.44.202, og at Marias bærbar har IP-adressen 21.118.0.234. Da fås to pakker:

IP-adresser	21.118.0.234	100.72.44.202
Portnumre	4551	80
Sekvensnummer	1	
Besked-data	GET /index.html HTTP/1.1	

Pakke

21.118.0.234	100.72.44.202
4551	80
2	
Host: www.eksempe1.dk	

Pakke

Netværkslaget sender nu de to pakker ned til data-link-laget. Maria er opkoblet til internettet gennem et netkabel, der løber fra hendes netkort til en router, der står i hendes stue. Data-link-laget skal sørge for at overbringe de to pakker fra Marias netkort til Marias router. Data-link-laget påklistrer derfor pakkerne de to MAC-adresser på Marias netkort (56-FF-4A-11-E3-66) og routerens netkort (02-A3-61-2D-11-66):

MAC-adresser	56-FF-4A-11-E3-66	02-A3-61-2D-11-66	56-FF-4A-11-E3-66	02-A3-61-2D-11-66
IP-adresser	21.118.0.234	100.72.44.202	21.118.0.234	100.72.44.202
Portnumre	4551	80	4551	80
Sekvensnr	1		2	
Besked-data	GET /index.html HTTP/1.1		Host: www.eksempel.dk	

Frame

Frame

Derved fås to frames. Data-link-laget sender dem til netkortet hos Maria, der herefter transmitterer dem over kablet til routeren.

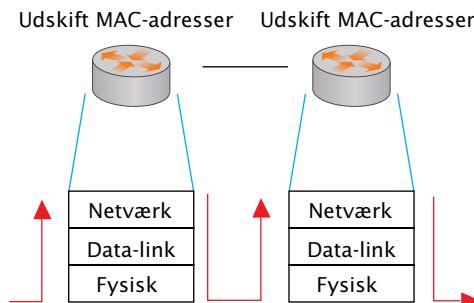
Routeren modtager de to frames. Den skræller felterne med MAC-adresserne bort og kigger på IP-adressen på modtageren. Derefter konsulterer den sin forwarding-tabel, der fortæller at pakker til IP-adresser, der starter med 100.72.xxx.xxx skal sendes til nabourteren med IP-adresse 4.68.128.213 (og MAC-adresse 54-1E-3A-D1-B0-BD). Data-link-laget hos routeren sørger nu for at sætte de rigtige MAC-adresser på. Derved fås to nye frames:

MAC-adresser	02-A3-61-2D-11-66	54-1E-3A-D1-B0-BD	02-A3-61-2D-11-66	54-1E-3A-D1-B0-BD
IP-adresser	21.118.0.234	100.72.44.202	21.118.0.234	100.72.44.202
Portnumre	4551	80	4551	80
Sekvensnr	1		2	
Besked-data	GET /index.html HTTP/1.1		Host: www.eksempel.dk	

Frame

Frame

Hver gang en frame når frem til en router, passerer den op til netværkslaget, hvor IP-adressen undersøges. Dernæst pakkes den som en ny frame med nye MAC-adresser.



Selvom de to pakker skal ende samme sted, kan deres vej gennem netværket sagtens være forskellig. Lad os sige, at det faktisk er pakken med sekvensnummer 2, der ankommer først hos webserveren. Pakken ankommer som en frame på webserver-computerens data-link ud til netværket. Data-link-laget hos webserveren fjerner MAC-adresserne og giver den resulterende pakke til netværkslaget:

21.118.0.234	100.72.44.202
4551	80
2	
Host: www.eksempel.dk	

Netværkslaget hos webserveren kigger på modtager-IP-adressen og indser “Hør hov – det er jo mig, der har IP-adresse 100.72.44.202!” Netværkslaget skræller IP-adresserne af og overgiver segmentet indeni til transportlaget (transportlagsprotokollen er angivet som ekstra information):

4551	80
2	
Host: www.eksempel.dk	

Transportlaget hos webserveren kigger på sekvensnummeret 2 og indser, at der mangler segmentet med sekvensnummer 1. Transportlaget sender derfor en kvittering af formen “ACK: 0” til Marias transportlag, der meddeler, at alle segmenter med sekvensnummer til og med 0 er modtaget korrekt – dvs. – segment nr. 1 mangler stadig. Kvitteringen skal gennem alle de samme trin som Marias forespørgsel for at nå frem, blot “i den anden retning”. Marias IP-adresse, der skal anføres på kvitteringen, stod i pakken. Lidt senere modtager transportlaget hos webserveren segmentet med sekvensnummer 1 og sender kvitteringen “ACK: 2” (“alle segmenter med sekvensnummer til og med 2 er modtaget med succes!”).

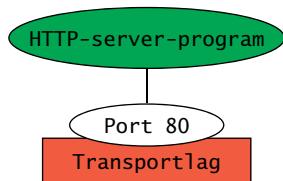
Transportlaget hos webserveren står nu med de to segmenter

4551	80
2	
Host: www.eksempel.dk	
GET /index.html HTTP/1.1	

De samles, vha sekvensnumrene, til beskeden

GET /index.html HTTP/1.1
Host: www.eksempel.dk

Modtager-portnummeret var angivet som 80. Derfor ved transportlaget, at beskeden skal sendes til HTTP-server-programmet, der altid er tilkoblet port 80:



HTTP-server-programmet modtager beskeden, genererer HTTP-svar, og sender det til transportlaget. Da transportlaget havde afsenderens portnummer, kan svaret finde vej tilbage til den rette proces (browseren på Marias computer) over netværket.

HTTP-svaret udsættes nu for alle de samme trængsler som HTTP-forespørgslen og ender til sidst med at dukke op hos Marias browser, der omsætter det modtagne til en fremvist website på Marias skærm.

Så let som at trykke på en tast (for Maria)!

5.9. NØGLEORD

- ◆ Internettets fysiske opbygning
- ◆ Slutsystem
- ◆ Router
- ◆ Data-link
- ◆ Server
- ◆ LAN
- ◆ WAN
- ◆ ISP
- ◆ Protokol
- ◆ Protokolstak og lagdelt kommunikation
- ◆ Applikationslag
- ◆ Transportlag
- ◆ Port
- ◆ Pålidelig transmission
- ◆ TCP
- ◆ Sekvensnummer
- ◆ Kvittering (ACK)
- ◆ Upålidelig transmission
- ◆ UDP
- ◆ Segment
- ◆ Netværkslag
- ◆ Pakke
- ◆ IP-adresse
- ◆ Routing
- ◆ Datalag
- ◆ Linklag
- ◆ Frame
- ◆ MAC-adresse
- ◆ Point-to-point
- ◆ Broadcast
- ◆ Fysisk lag
- ◆ Datatransmission
- ◆ Ethernet

KAPITEL 6

PROGRAMMERING

Det er sjovt at programmere! Med programmering kan du skabe dine egne maskiner, der gør lige hvad du beder dem om. Uden nogensinde at røre sav eller skruetrækker!

Programmering er også udfordrende: Kan det passe at der er fejl i programmer fra selv de største software-firmaer?! Og at selv de dygtigste programmører kan sidde i dagevis og lede efter årsagen til en program-fejl? Ja! – programmering er noget, man hurtigt kommer i gang med, men som tager lang tid at mestre.

I dette kapitel skal vi lære grundteorien omkring *at programmere* (= *at kode*).

ORDFORKLARING

PROGRAMMERING

At programmere er at skrive kildekoden til et computerprogram. Kildekoden er skrevet i et programmeringssprog.

For at forstå begrebet “programmering” skal vi altså vide følgende:

- ◆ Hvad er et program?
- ◆ Hvad er kildekode?
- ◆ Hvad er et programmeringssprog?
- ◆ Hvordan programmerer man?

Dette kapitel sætter dig i stand til at svare fyldestgørende på disse spørgsmål.

WWW

PROGRAMMERINGSØVELSER

På **WWW** er der programmeringsøvelser i alle sværhedsgrader med et udvalg af forskellige programmeringssprog.

6.1. PROGRAMMER SOM ALGORITMER

Mange computerprogrammer kan ses som *algoritmer*, der er nedskrevet i et sprog, som en computer forstår. Når maskinen afvikler programmet, udføres algoritmen. Algoritmen er en lang række af ordrer, som computeren skal udføre. Lad os se på nogle konkrete eksempler på algoritmer.

EKSEMPEL

BRØDRISTER

Det er tid til krydderboller, men brødristeren starter ikke! For at løse problemet bruger vi en algoritme:

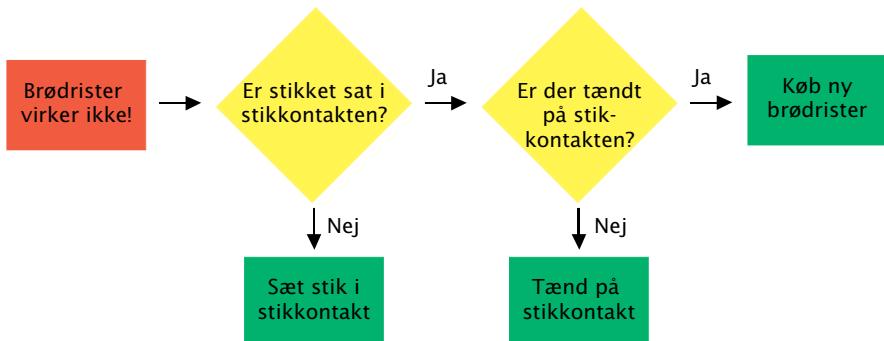


Fig. 6.76. Brødrister-algoritme.

Algoritmen er opbygget af veldefinerede skridt, og vi kan se, at det er en beslutningsprocedure, der giver et resultat (en beslutning om, hvad der skal gøres).

EKSEMPEL

GAVEINDPAKNING

Julen står for døren, og der skal pakkes gaver ind! Algoritmen til gaveindpakning tager *input* – resultatet af en gaveindpakning afhænger af, hvad det er for en gave, der bliver pakket ind.

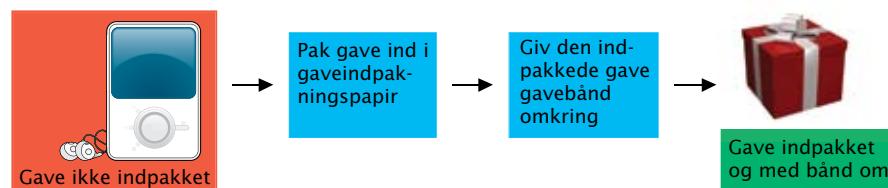


Fig. 6.77. Gave-algoritme.

EKSEMPEL

SMS-ORDBOG

En sms-ordbog er et eksempel på en mere indviklet algoritme. Vi kan tegne den som følger – uden at kende til de veldefinerede skridt, den består af:

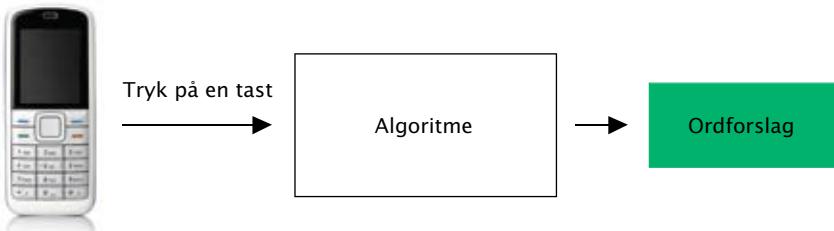


Fig. 6.78. Sms-ordbog. Trykkes 7, foreslår ordbogen “p”. Trykkes dernæst på 6, foreslår ordbogen “ro”. Trykkes til sidst 7, foreslår ordbogen “sms”.

Algoritmens resultat (ord-forslaget) afhænger tydeligvis af, hvilken input vi leverer (hvilken knap der trykkes). Algoritmen her tager altså også input.

EKSEMPEL

SUSHISPISNING

En masse stykker sushi byder sig til på et fad. De skal ned. Du bruger derfor følgende algoritme:

1. Spis et stykke sushi fra fadet.
2. Er fadet tomt? Hvis nej, gå til trin 1. Hvis ja, gå til trin 3.
3. Al sushien er spist. Slå mave med god samvittighed.

Fig. 6.79. Sushispisnings-algoritme.

EKSEMPEL

ELEKTRONIK

Algoritmer findes i næsten al elektronik – ligesom sms-ordbogen ovenfor findes i en mobiltelefon, er der er kanalvalg på TV’et, stregkodelæsere i supermarketet, bagagebåndsstyring i lufthavne, samlebånd i fabrikker og forskellige vaskeprogrammer på opvaskemaskinen.

ORDFORKLARING

ALGORITME

En *algoritme* er en metode eller procedure, opbygget af veldefinerede mekaniske eller operationelle skridt, som løser et bestemt problem. Når algoritmen aktiveres, beregner den et korrekt resultat (algoritmens *output* eller *uddata*). Algoritmen kan gives *input* (eller *inddata*) i et format defineret af algoritmen. I så fald beregnes output på baggrund af det givne input.

En algoritme kan tegnes som et *flow-diagram* (som det er gjort i Figur 6.76 og Figur 6.77) eller skrives ned som en liste af nummererede skridt (som i Figur 6.79).

For en bestemt algoritme kan der være flere korrekte svar. I så fald forlanger man typisk, at output er en af følgende:

- ◆ et eller andet korrekt svar,
- ◆ et “godt” eller det “bedst mulige” korrekte svar,
- ◆ samtlige korrekte svar.

EKSEMPEL

KORREKTE SVAR

En vejvisningswebseite kan oplyse en rute mellem to byer i Danmark. Lad os sige, at Jørgen skal fra Køge til Odense. Jørgen aktiverer websiden, som aktiverer sin indbyggede algoritme og giver den

input = {Fra: Køge, Til: Odense}

Jørgen er nu helt klart interesseret i “en god” rute: En rute over Skagen giver ikke mening, selvom den er korrekt. Der er rigtig mange ruter, der starter i Køge og slutter i Odense, og dem gider Jørgen heller ikke have beregnet.

Udover at korrekthed kræver man sædvanligvis, at en algoritme er

- ◆ *deterministisk*: giver samme resultat for det samme input,
- ◆ *stoppende*: beregner altid svaret i et endeligt antal skridt,
- ◆ *fuldstændig*: virker for alle gyldige input.

6.2. DATA, INFORMATION OG MODELLERING

Vi har i de foregående kapitler omtalt “data”. Vi giver nu en nøjere indkredsning af begrebet.

ORDFORKLARING

DATA

En formaliseret repræsentation af en mængde information. Repræsentationen kan forstås og manipuleres med gennem bestemte processer. Repræsentationen kan sigte mod menneskelig forståelse (fx trykt tekst) eller maskinbearbejdning (fx elektriske signaler).

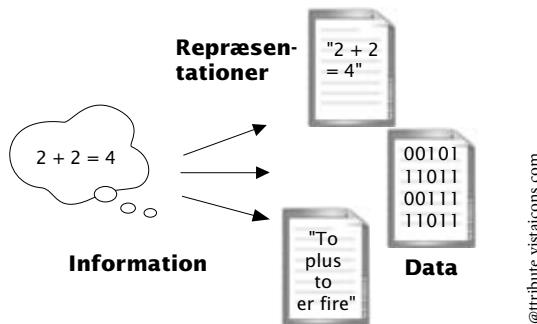


Fig. 6.80. Når information repræsenteres, fås data. Den samme information kan repræsenteres forskelligt.

Her har vi forklaret ordet "data" ud fra ordet "information". Selvom der kan gøres mange filosofiske indvendinger, vil vi her sige, at

$$\text{information} = \text{data} + \text{betydning}.$$

Data bliver til information ved at blive fortolket af en, der kan finde en betydning ved sin fortolkning. Hermed er ikke sagt, at data er meningsløst nonsens, blot at begrebet "data" ikke omfatter nogen form for menneskelig tankevirksomhed.

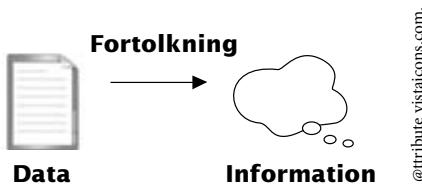


Fig. 6.81. Data fortolkes til information.

Data er håndgribeligt, fordi det *er* sin repræsentation. Information er uhåndgribeligt: det eksisterer delvist i bevidstheden hos fortolkeren. Der er derfor heller ikke én "rigtig" fortolkning af forelagt data.

EKSEMPEL

DATA VS. INFORMATION

Trykt tekst er data. Ved en visuel fortolkning (genkend bogstaver vha. synssans) og en sproglig fortolkning (læsning) forstår indholdet af det skrevne.

Trykte noder er data. Ved en visuel og musikalsk fortolkning (nodelæsning) omdannes symbolerne til forståelsen af et musikstykke.

Tale er data. Ved en auditiv-sproglig fortolkning (hør lyde, genkend dem som ord med en mening) forstår indholdet af talen.

Fortolkningerne kan sagtens resultere i forskellige forståelser.

EKSEMPEL

MASKININSTRUKTIONER LAGRES SOM TAL

I kapitel 2 lærte vi, at computeren lagrer maskininstruktioner som tal i datacellerne i RAM'en (von Neumann-princippet). Først hvis et tal der repræsenterer næste instruktion i et program indlæses i CPU'en, fortolkes det som en instruktion. I fortolkningsøjeblikket omsættes altså data (talkoden fra RAM'en) til en information (en instruktion til processoren om, hvad den skal gøre).

Vi mennesker oplever verden gennem sanseindtryk. For at kunne finde rundt i verden, må vi kunne skelne vores indtryk fra hinanden, huske dem og formidle dem. Det foregår i vores tankeverden: Vi omformer vores faktiske oplevelser fra bevidsthedsindtryk til opfattelser, af, hvad vi oplevede, som lagres i erindringen.

Hvor mennesker tænker og arbejder på informationsplan, kan maskiner kun operere på data: et program på en computer kan tage *inddata* (data, der skal reageres på eller opereres på) og skaber *uddata* (resultat). Siden programmer er til for at løse problemer for mennesker, involverer programmering en fornuftig omsætning af menneske-problemer til modeller med data-repræsentationer, som programmer kan behandle. *Modellering* er derfor en central disciplin indenfor datalogi.



Fig. 6.82. Modellering: En omsætning af virkelighed til information og information til data.

En modellering er altid en *forenkling*, hvor kun visse virkelighedsparametre repræsenteres. Sådanne parametre til modeller i programmering vælges ud alt efter relevans. For eksempel er det ikke vigtigt at kende antal græsstrå på hver fodboldbane, hvis vi vil lave en oversigt over danske fodboldstadions og deres tilskuertal fra kamp til kamp. I et bibliotek ønsker man ikke at kende til hver låners livret og helbredstilstand – kun deres lånerstatus.

6.2.1. ANALOG OG DIGITAL REPRÆSENTATION

Man skelner mellem *analog* og *digital* repræsentation af information.

Analoge repræsentationer benytter fysiske størrelser såsom spænding eller radiobølger. Traditionelt antennen-TV transmitteres fx analogt.

Digitale repræsentationer benytter et tegnsæt af symboler. Fx repræsenterer vi alle hele tal med tegnene i mængden $\{0,1,2,3,4,5,6,7,8,9,-\}$ – vi benytter tegnene som cifre i et positionssystem. Al modellering, der sigter mod computere, anvender *digitale* repræsentationer af information (*digit* er engelsk for “ciffer”). Faktisk er enhver computer-repræsentation er i sidste ende *binær* – dvs. at der blot benyttes 0 og 1 – så den er altid digital.

Digitale repræsentationer vinder indpas overalt, fordi udviklingen indenfor elektronik kan levere bedre og hurtigere resultater med digitale repræsentationer end analoge.

6.3. PROGRAMMERINGSSPROG

Et programmeringssprog er et kunstigt sprog, som man skriver programmer med. Hvor menneskesprog som dansk, engelsk eller fransk bruges til kommunikation mellem mennesker, bruges et programmeringssprog til at overbringe en computeren et sæt instruktioner.



Fig. 6.83. Programmeringssprog – måden man giver en computer ordrer på (det nytter ikke med tysk eller latin).

Programmeringssprog er for det meste tekstbaserede. Det vil sige, at mennesker skriver programmer som en tekst, der overholder formatet defineret af programmeringssproget. En tekst skrevet i et bestemt programmeringssprog, som definerer et program, kaldes programmets *kildekode* (eller bare programmets *kode*). Et program er sin kildekode.

Programmingssprog adskiller sig fra menneskelige udtryksformer ved at kræve total præcision og fuldstændighed. I en samtale mellem mennesker kan ord udelades eller underforstås. Man kan endda bruge forkerte ord eller tvetydigt kropssprog og stadig forvente at blive forstået. Computere er ikke som menneskelige tilhørere. De gør præcis, hvad de får besked på. Hvis en programmør skriver kildekode, der ikke udtrykker programmørens hensigt præcist, kan computeren ikke ”gætte” den rigtige mening.

Vi giver nu en række tekniske og opsummerende forklaringer.

ORDFORKLARING

PROGRAMMERINGSSPROG

Et programmeringssprog er givet ved en *specifikation*: et dokument, der eksakt og fuldstændigt beskriver hele sproget. Specifikationen skrives af den, der designs programmeringssproget, og læses af den, der bruger programmeringssproget til at kode med. Specifikationen fastlægger

- ◆ syntaks
- ◆ semantik
- ◆ (som regel) et typesystem
- ◆ (som regel) et standardbibliotek.

Vi kigger på syntaks og semantik i afsnit 6.3.1, typesystemer i afsnit 6.3.2 og standardbiblioteker i afsnit 6.3.3.

Et programmeringssprog er altså ”bare” en specifikation. Specifikationen kan *implementeres* på flere måder, så længe hver implementation overholder specifikationen. På den måde opdeles arbejdet med at levere et fuldt fungerende programmeringssprog i to portioner,

- ◆ *design* af sproget (at lave specifikationen),
- ◆ *implementation* af sproget (det ser vi på i afsnit 6.4).

6.3.1. SYNTAKS OG SEMANTIK

Ligesom almindelige sprog har programmeringssprog en *syntaks* og en *semantik*. I korte træk definerer syntaksen sprogets *struktur*, mens semantikken definerer sprogets *mening*.

ORDFORKLARING

SYNTAKS

Syntaksen for et programmeringssprog er et regelsæt, der definerer hvordan symboler i kildekoden skal være placeret i forhold til hinanden. Tekst, der overholder syntaksen, kaldes syntaktisk korrekt.

Syntaksen beskæftiger sig slet ikke med kildekodens mening, blot hvordan vi konstruerer et gyldigt program, der gør *et eller andet*.

EKSEMPEL

DET FIKTIVE PROGRAMMERINGSSPROG Øvesprog

Lad os opfinde programmeringssproget Øvesprog, der har følgende syntaks:

- ◆ der må kun benyttes symbolerne A, B og E
- ◆ der må ikke være flere B'er i træk

Følgende er så eksempler på syntaktisk korrekt kildekode: "B", "ABE" og "EEAA". Følgende tekster er ikke syntaktisk korrekte: "BBA" (to B'er i træk) og "ABC" (et ikke-tilladt symbol). Vi har kun fastlagt syntaksen for øvesprog, men ikke beskæftiget os med, *hvad* programmerne "B", "ABE" og "EEAA" *gør*.

Et syntaktisk korrekt program kan køres på en computer – med forskelligt resultat alt efter hvad der står i programmet. På den baggrund siger man at et program *betyder* det, som programmet gør, når det køres.

ORDFORKLARING

SEMANTIK

En semantik for et programmeringssprog udtrykker utvetydigt betydningen af ethvert syntaktisk korrekt program.

En semantik bruges af en computer til at udføre et syntaktisk korrekt program, men den bruges også af mennesker til at forstå og ræsonnere om programmer.



Fig. 6.84. Semantik.

Det er en udfordring at udtrykke et bare nogenlunde kompliceret sprogs semantik, sådan at alle – inklusive de computere, der skal afvikle programmer skrevet i det pågældende sprog – er enige om betydningen af *hvert eneste syntaktisk korrekte program*. At udtrykke semantikker kræver en større viden indenfor matematisk logik, og vi kommer ikke ind på det.

En semantik forudsætter en syntaks. En syntaks kan ligge til grund for flere semantikker.

EKSEMPEL

SEMANTIKKER FOR ØVESPROG

Programmeringssproget Øvesprog kan udstyres med forskellige (tåbelige) semantikker, fx

- ◆ Alle syntaktisk korrekte programmer betyder: "Gør ingenting".
- ◆ A svarer til 1, B til 2 og E til 5. Et Øvesprog-program skal opfattes som en plus-stykke, og hvis programmet køres, skal resultatet udskrives på skærmen.

Med første semantik gør alle tre programmer "B", "ABE" og "EEAA" det samme – ingen ting. Med anden semantik forstår programmerne som hhv. "2", "1+2+5" og "5+5+1+1", og ved kørsel af programmerne vil skærmen vise hhv. "2", "8" og "12".

EKSEMPEL

SYNTAKS OG SEMANTIK PÅ DANSK

Symbolfølgen "Ha XΔMs67?" er syntaktisk forkert, hvis man bruger syntaksen for sproget dansk – symbolfølgen bruger ulovlige tegn (symbolet delta (Δ) fra det græske alfabet (ubehøvlet!)).

Symbolfølgen "Marla Ehems driops" bruger kun lovlige tegn, men danner ikke velkendte ord. Symbolfølgen er derfor syntaktisk forkert.

Symbolfølgen "Under han at hoppe rosens foran." bruger kun lovlige ord, men de er ikke sat sammen efter grammatikkens regler. Symbolfølgen er derfor syntaktisk forkert.

Symbolfølger, der ikke er syntaktisk korrekte, gør vi ikke noget forsøg på at tillægge nogen betydning.

Symbolfølgen "Hun stejer bogen, indtil stolen klipper." er syntaktisk korrekt (korrekte tegn og ord med korrekt grammatik og tegnsætning), men semantisk temmelig særlig, hvis vi bruger sædvanlig semantik for sproget dansk: Hvad betyder symbolfølgen – hvorfor stege en bog? Hvad betyder det, at en stol klipper?

Symbolfølgen "Datalogi er sjovt!" er syntaktisk og semantisk korrekt.

TIP

SYNTAKSFEJL OG SEMANTISKE FEJL

Når du programmerer, skal du ikke være bange for at lave syntaksfejl – de bliver opdaget af en såkaldt syntakstjekker, inden programmet nogensinde kommer til at køre. Syntaks-tjekkeren fortæller dig typisk, præcis hvor syntaksfejlen er, så du let kan rette den.

"Semantiske fejl" er, hvad man også kan kalde logiske fejl. Det er op til en programmør selv at finde sådanne fejl, og de kan undertiden være uhyre vanskelige at spotte.

6.3.2. TYPESYSTEM

Udover syntaks og semantik fastlægger specifikationen af et programmeringssprog et *typesystem*. Typesystemet fastlægger, hvilke indbyggede *datatyper* sproget arbejder med, og hvordan datatyperne kan bruges. Indbyggede datatyper kan bruges som byggeklodser, når mere komplekse datastrukturer skal opbygges.

DATATYPE	TYPISK NAVN PÅ ENGELSK	EKSEMPLER PÅ VÆRDIER
Sandhedsværdi	<i>Boolean</i> (efter George Boole)	TRUE FALSE
Heltal	<i>Integer</i>	0 2 200 -45
Flydende tal (komma-tal)	<i>Floating point</i>	1.0 0.5 23.11 -45.4
Tegn	<i>Character</i>	'a' :: ' ' (blanktegn) '\$'
Tekststreng	<i>String</i>	"Jeg er en falk" "200" "2+2=4" "Ost"

Fig. 6.85. Typiske datatyper.

I programmeringssprog arbejder man med *værdier*. Værdier har datatyper. Fx er 1.0 en værdi med datatype komma-tal, og "ost" en værdi med datatype tekststreng.

Programmer, der bruger datatyper forkert, er ugyldige – og typesystemet fanger disse fejl. Typesystemet hjælper altså programmøren med at kode rigtigt.

EKSEMPEL

TYPESYSTEM I AKTION

Lad os forestille os et programmeringssprog med de tre datatyper "tal", "liste af tal" og "tekststreng". Lad os endvidere sige, at symbolet `^` betyder "sæt tekststrenge sammen" i dette sprog. Typesystemet for sproget vil da godkende udtryk som

5+6

"Lille" ^ "bror"

fordi de indgående typer i udtrykkene *passer sammen*. Udtrykkene kan beregnes (og resultatet vil være værdierne 11 og "Lillebror"). Typesystemet vil derimod forhindre udtryk som

2 - [2, 1, 5]

"Lille" / "bror"

da man ikke kan trække en liste fra et tal eller dividere tekst med tekst.

Et gyldigt program er altså både syntaktisk korrekt og overholder typesystemet. Visse typerfejl kan opdages af syntakstjekkeren. De resterende opdages af typesystemet.

Nogle datatyper har altid en *fast størrelse* målt i byte. Hver byte består af 8 bit og kan dermed repræsentere $2^8 = 256$ forskellige værdier. Byte-størrelsen er fastlagt i det enkelte programmeringssprog og fortæller, hvor mange lagerpladser i hukommelsen, der skal bruges til at lagre én værdi af den pågældende datatype. Tegn har en fast størrelse på typisk 1 byte. En tekststreng kan pga af sin variable længde have forskellige størrelser; den opfattes tit som en liste af tegn.

EKSEMPEL

TEGN OG STRENGE

Strenge "Jeg er en falk" kræver mere lagerplads end "ost", da den er længere. Hvis strenge opfattes som lister af tegn, er

"Jeg er en falk" =
['J', 'e', 'g', ' ', 'e', 'r', ' ', 'e', 'n', ' ', 'f', 'a', 'l', 'k']

mens

"ost" = ['o', 's', 't']

Hvis hvert tegn fylder 1 byte, fylder "Jeg er en falk" 14 byte og "ost" fylder 3 byte.

Bortset fra tekststrenge har typerne i Figur 6.85 faste størrelser: Et typesystem fastsætter også bytestørrelser for tal – det er nødvendigt for at kunne regne med tallene. Med et fast antal byte kan man ikke repræsentere alle tal. Tabellen nedenfor viser, hvad 4 og 8 byte giver af repræsentationsområder for hele tal:

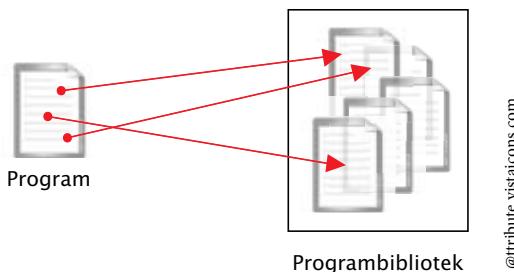
HELTAL			
STØRRELSE	TYPISK NAVN	REPRÆSENTATIONSMÅRÅDE	
		MED FORTEGN	UDEN FORTEGN
4 byte (16 bit)	<i>Int</i>	-2.147.483.648 til +2.147.483.647	0 til 4.294.967.295
8 byte (32 bit)	<i>Long</i>	-9.223.372.036.854.775.808 til +9.223.372.036.854.775.807	0 til 18.446.744.073.709.551.615

Fig. 6.86. Størrelser og repræsentationsområder for heltal.

Simple typer er de datatyper, der kræver et fast antal byte til repræsentationen, fx tal, tegn eller sandhedsværdier. *Sammensatte* typer er datatyper, der bygges ud fra simple typer gennem fx lister. En tekststreng er af typen ”*sammensat*”, da den er en liste af tegn. En liste af tal er også *sammensat*, mens tallene i listen er simple.

6.2.3. PROGRAMBIBLIOTEKER

Et *programbibliotek* hørende til et programmeringssprog er en samling hjælpeprogrammer og hjælpedata, der kan bruges til at skrive nye programmer med. Når et nyt program skrives, kan det referere til funktioner i programbiblioteker.



@ttibute vistaicons.com.

Fig. 6.87. Program refererer funktionalitet i et programbibliotek.

Programbiblioteker

- ◆ hjælper programmører til at skrive programmer hurtigere,
- ◆ hjælper programmører til at skrive bedre programmer – programbiblioteker er som regel gennemtestede og velfungerende,
- ◆ sparar gentagelse af kode.

Et programmeringssprog specificerer som regel et standardbibliotek (engelsk: *standard library* eller *core library*) , der tilbyder grundlæggende funktioner, som de fleste programmører kan tænkes at have brug for. Det er fx funktioner til

- ◆ at kommunikere med operativsystemet; fx ønsker programmet tit at skrive i filer eller reagere på bruger-input, ting, der administreres af operativsystemet,
- ◆ matematiske beregninger,
- ◆ praktisk håndtering af data.

6.4. OVERSÆTTELSE OG FORTOLKNING

Et programmeringssprog er en specifikation, der fortæller en programmør, hvordan han skal *skrive* programmer. I dette afsnit kigger vi på, hvordan man rent faktisk kan komme til at *køre* de programmer, man har skrevet.

6.4.1. HØJNIVEAU- OG LAVNIVEAUSPROG

Som vi lærte i kapitel 2, er en computer er rent hardwaremæssigt opbygget sådan, at den kun kan udføre en begrænset mængde af meget simple instruktioner – til gengæld går det lynende stærkt at afvikle hver enkelt instruktion. Oprindeligt var alle programmerings-sprog maskinafhængige: De blev udviklet til at programmere bestemte maskiner med deres instruktionssæt. Sådanne programmeringssprog kaldes *lavniveau-programmeringssprog*.

EKSEMPEL

ASSEMBLERKODE

I en computer afvikler processoren konstant instruktioner fra maskinens instruktions-sæt. Instruktionerne har læsbare tekstrepræsentationer som assemblerkode (såsom – med FIMA-assembler – `ADD R0 R1 R2` eller `LOAD 44 R2`). Assemblerkode er lavniveau-kode, fordi den afhænger af den maskine, der skal udføre assemblerkoden.

De fleste moderne programmeringssprog er imidlertid *højniveausprog* – sprog, der ikke afhænger af en fastlagt underliggende maskinstruktur med bestemte instruktioner. Sådanne sprog er *processoruafhængige*: Programmer i højniveaukode kan transporteres fra en computer til en anden uden at skulle skrives om. Det kaldes *portabilitet*. Et program i lavniveaukode skal laves om fra bunden, hvis det skal køres på en anden processortype.

Der er kun et enkelt problem med højniveaukode: *processoren forstår den ikke!* Der er to løsninger:

- ◆ *Oversættelse*: Et særligt program, *en oversætter*, oversætter højniveau-kildekoden til maskinkode. Maskinkoden kan herefter udføres. Maskinkoden gør det samme som højniveau-programmet.
- ◆ *Fortolkning*: Et særligt program, *en fortolker*, der kan forstå højniveau-kildekoden, og som selv kan køre på maskinen, udfører programmet.

6.4.2. OVERSÆTTELSE

Vil man køre et højniveauprogram, kan man *oversætte* det til maskinkode. Det gøres med et særligt program, *en oversætter* (engelsk: *compiler*), hvis opgave er at oversætte programkode i et programmeringssprog til programkode i et andet programkode.

ORDFORKLARING

OVERSÆTTELSE

At oversætte kode fra et programmeringssprog til et andet. Den oversatte kode gør præcis det samme som den oprindelige kode.

Når man taler om oversættelse, mener man som regel oversættelse af højniveau-programkode til assembler- eller maskinkode.

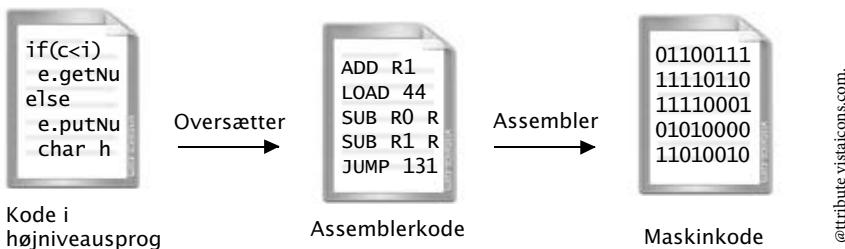


Fig. 6.88. Oversættelse og assemblering. Her ender højniveaukoden – via to oversættelser – som maskinkode-instruktioner, CPU'en forstår.

En oversættelse er en kompleks procedure med adskillige trin, der varierer fra oversætter til oversætter. Typisk forsøger oversætteren, inden oversættelsen, at finde og beskrive eventuelle fejl så præcist som muligt. Vi skitserer her nogle typiske trin for oversættelse af højniveaukode til maskinkode.

OVERSÆTTELSE, TRIN 1: SYNTAKSTJEK

Oversætteren gennemlæser kildekoden og afgør, om syntaksen er i orden eller ej. Det sker med en syntakstjekker, som enten er et selvstændigt program eller en del af oversætteren. Er syntaksen ugyldig, afbrydes oversættelsesprocessen.

OVERSÆTTELSE, TRIN 2: TYPETJEK

Oversætteren undersøger, at kildekodens brug af typer er gyldig. Hvis ikke, afbrydes oversættelsesprocessen.

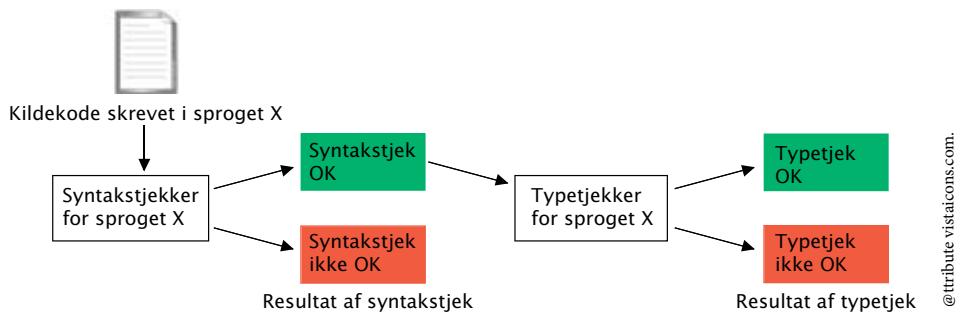


Fig. 6.89. Syntaks- og typetjek.

Attributed vistaicons.com.

OVERSÆTTELSE, TRIN 3: GENERERING AF MASKINKODE

Oversættelsen omsætter kildekoden til maskinkode.

OVERSÆTTELSE, TRIN 4: LINKING

Maskinkoden har muligvis gjort brug af funktioner fra programbiblioteker eller andre programmer. Alle sådanne referencer "ud af selve programmet" kortlægges og kobles sammen med maskinkoden – herved fås en fil med maskinkode, der kan afvikles. En sådan fil kaldes en *executable* ("kan afvikles") og kan fx have fil-endelsen .exe.

Ofte er det dog ikke oversætteren, men et program hørende til operativsystemet, der står for linking.

6.4.3. FORTOLKNING

Oversættelse er ikke den eneste måde at udføre højniveaukode. En *fortolker* for et højniveausprog er et program, der kan forstå kildekode skrevet i højniveusproget og udføre det direkte – uden nogen indledende oversættelse. Vil man fortolke højniveaukode, startes fortolkeren, som herefter udfører højniveau-programkoden *linie for linie*.

ANALOGI

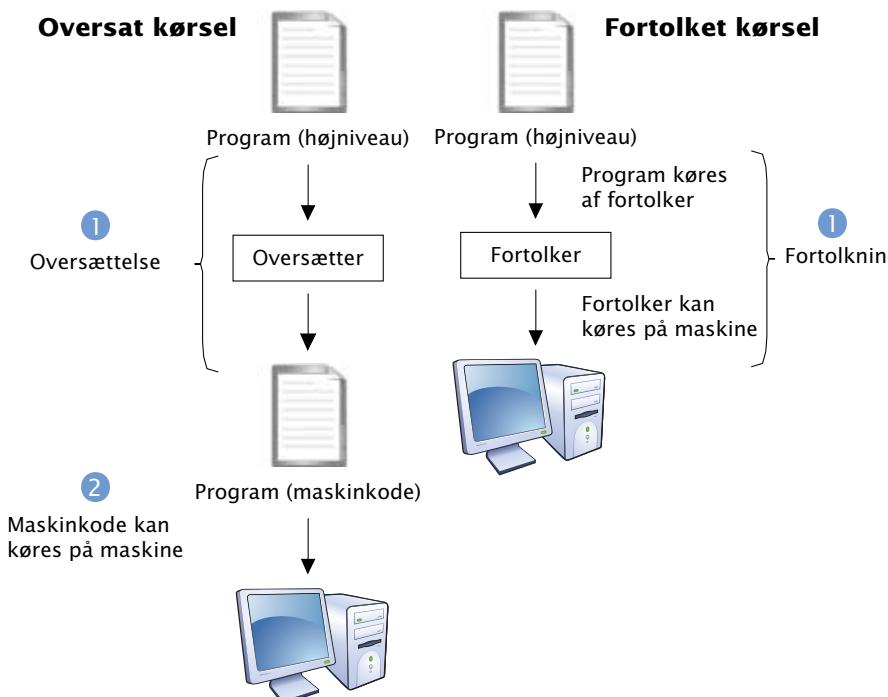
SKRIFTLIG OVERSÆTTELSE ELLER SIMULTANTOLKNING

Vi belyser forskellen mellem oversættelse og fortolkning med en analogi.

Præsident A besøger præsident B og ønsker at overbringe en diplomatisk hilsen. Imidlertid taler og skriver præsident A kun sproget A, mens præsidenten B kun forstår og læser sprog B. Der er to måder at overbringe præsident A's hilsen:

- **Oversættelse:** Præsident A skriver sin hilsen ned på sprog A. En kyndig person oversætter teksten til sprog B og læser den dernæst højt for præsident B.
- **Fortolkning:** Præsident A udtaler sin hilsen, en sætning ad gangen, på sprog A. En simultantolk er til stede, og oversætter, lidt ad gangen, hilsenens ordlyd til sprog B.

Oversat programkørsel består af 2 trin: først en total oversættelse og dernæst kørsel af oversat kode. *Fortolket programkørsel* sker i 1 trin: løbende fortolkning af højniveau-kodelinier.



@tribute vistaicons.com.

Fig. 6.90. Oversat kørsel sker i to trin. Fortolket kørsel sker i et trin.

TIL SÆRLIGT INTERESSEREDE

MELLEMKODE OG VIRTUELLE MASKINER

Vi uddyber nu de 4 trin i oversættelsesprocessen fra afsnit 6.4.2 på side 136. Mange oversættere inkuderer nemlig følgende trin mellem trin 2 og 3:

Oversættelse, trin 2 B: Mellemkode og optimering

Oversætteren omsætter kildekoden til såkaldt mellemkode, et særligt kodeformat, der ligger ”mellem” kildekodens sprog og maskinkode. Oversætteren kigger herefter mellemkoden igennem og kan i nogle tilfælde optimere den.

Den ene store fordel ved at oversætte til mellemkode i stedet for direkte til maskinkode er, at oversætteren kan oversætte til flere forskellige typer maskinkode uden at skulle gennemføre trin 1 – 2 hver gang:

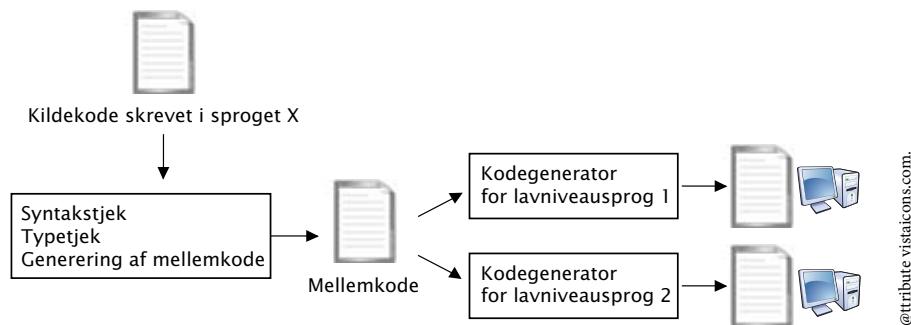


Fig. 6.91. Med mellemkode kan man nøjes med et syntakstjek og et typetjek, selvom programmet skal oversættes til flere forskellige maskiner.

Den anden store fordel er, at mellemkode muliggør *virtuelle maskiner*. En virtuel maskine gør det muligt at køre et program, skrevet i et højniveausprog, på mange forskellige maskiner – uden at skulle lave en oversætter for hver maskinarkitektur. En virtuel maskine til et højniveau-programmeringssprog er en konstruktion med tre dele:

1. en type mellemkode, der minder om maskininstruktioner,
2. en oversætter, der oversætter højniveausproget til denne mellemkode,
3. en fortolker (den virtuelle maskine), der kan afvikle mellemkoden på alle almindelige maskiner (mellemkoden ”simulerer” en maskine, deraf navnet ”virtuel maskine”).

fortsættes...

TIL SÆRLIGT INTERESSEREDE

MELLEMKODE OG VIRTUELLE MASKINER (FORTSAT)

Et program, skrevet i højniveausproget, oversættes nu én gang for alle til mellemkode. Mellemkoden kan så køres på alle maskiner, hvor fortolkeren virker, ved at fortolke mellemkoden.

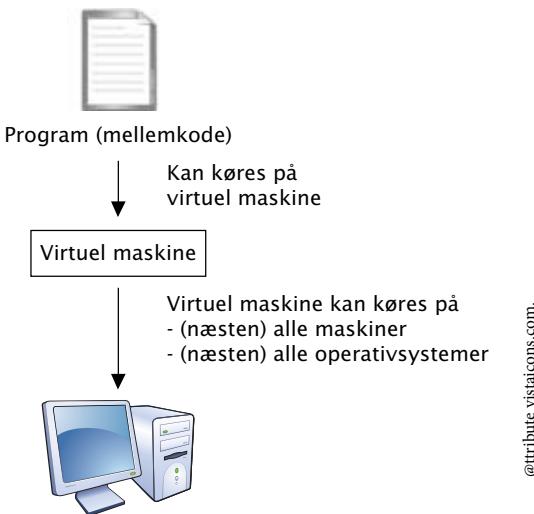


Fig. 6.92. Virtuel maskine.

TIL SÆRLIGT INTERESSEREDE

JVM

På **WWW** hører vi mere om Javas virtuelle maskine, JVM, og den særlige mellemkode, *Java bytecode*, JVM gør brug af. JVM bruges på følgende måde:

1. Et program skrives i højniveausproget Java.
2. Programmet oversættes til Java-bytecode.
3. Java-bytecode-udgaven kan udføres af JVM, som kan køre på næsten alle maskiner.

6.4.4. OPSUMMERING: IMPLEMENTATION AF PROGRAMMERINGSSPROG

En computer udfører et program skrevet i et højniveausprog på en af to udførelsesmåder (engelsk: *execution modes*): oversættelse eller fortolkning. På den måde *implementerer* en oversætter eller en fortolker et programmeringssprog.

ORDFORKLARING

IMPLEMENTATION AF ET PROGRAMMERINGSSPROG

En *implementation* af et programmeringssprog gør det muligt rent faktisk at afvikle kildekode skrevet i programmeringssproget på en bestemt (type af) maskine. En implementation er et program; enten en *oversætter* eller en *fortolker*.

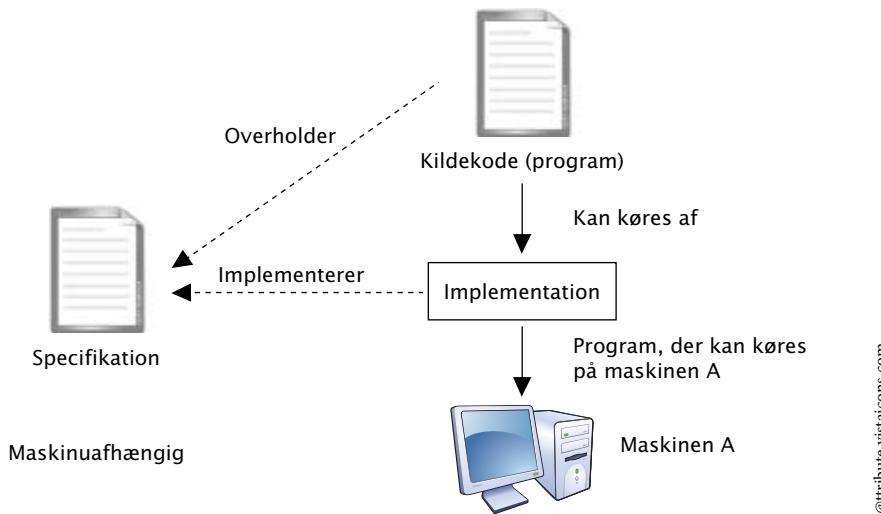


Fig. 6.93. Implementation af programmeringssprog.

6.5. PARADIGMER INDENFOR PROGRAMMERINGSSPROG

Programmeringssprog falder indenfor nogle overordnede såkaldte *paradigmer*, hovedkategorier med hver deres grundlæggende opfattelse af begrebet "programmering". Selvom sprogene falder indenfor forskellige paradigmer, har de ofte samme *udtrykskraft* – dvs., de kan beregne de samme ting.

Hvilket sprog fra hvilket paradigme man vælger til et løse et bestemt programmeringsproblem afhænger altså mestendels af programmørens præferencer. Hvilket konkret sprog han ender med at vælge giver ikke de store begrænsninger eller fordele i forhold til udtrykskraft. Der kan derimod være fordele relateret til effektivitet (både programmeringstid, programhastighed, mv.), portabilitet, dokumentation mv., der indikerer, at et sprog burde vælges fremfor et andet. En dygtig programmør ved altså også, hvornår hvilke sprog er at foretrække.

PARADIGMER	
Imperativt eller tilstandorienteret eller proceduralt	<p>Et imperativt program er en liste af statements (<i>imperativer</i>), som computeren udfører en for en, sekventielt.</p> <p>Hvert statement påvirker computerens tilstand, fx værdien af variable og indhold af lageret.</p> <p>At udføre et program imperativt svarer til, at et menneske (computeren) udfører en lang udregning (programmet) vha en blok til mellemregninger (lageret).</p> <p>Imperative programmer opbygges ofte som et eller flere moduler, der består af <i>procedurer</i> (mere om procedurer i afsnit 6.6.3).</p> <p>Den måde, en processor udfører maskinekode på, kan godt kaldes imperativ.</p>
Objekt-orienteret	Understøtter objekter defineret fx via klasser (mere herom i afsnit 6.7).
Deklarativt	<p>Beskriver en situation via en særlig type oplysninger (<i>deklarationer</i>) og finder løsninger på problemer via forespørgsler (<i>queries</i>).</p> <p>Modsat imperativ programmering, hvor programmet fortæller computeren, præcis hvad der skal gøres, er det med deklarativ programmering op til sprogets implementation at finde svar på forespørgsler.</p>
Funktionelt eller funktionsorienteret	Et program opfattes som et stort regneudtryk, der simplificeres indtil en resultatværdi er fundet.

Fig. 6.94. Vigtige paradigmer indenfor programmering.

I dette kapitel vil vi kun beskæftige os med imperativ programmering og objekt-orienteret programmering. Sprog til databaseprogrammering (jf kapitel 7) er tit deklarative. Her defineres databasens indhold vha. deklarationer, og bagefter udtrækkes database-indhold med forespørgsler. Det fører for vidt at komme ind på funktionelle sprog her.

6.6. IMPERATIV PROGRAMMERING

Imperative programmeringssprog er sprog, hvor kildekoden består af en liste af *statements*, der kan påvirke computerens tilstand: Værdier af variable, elementer i lister, indhold af filer, indhold af lageret mv. Ved kørsel af et imperativt program udføres listen af statements et for et, sekventielt.

Der er en række konstruktioner, man genfinder i de fleste imperative sprog. I dette afsnit gennemgår vi de elementer for et imperativt programmeringssprog, **Imperativtøvesprog**, vi opfinner til lejligheden. **Imperativtøvesprog** omfatter

- ◆ datatyperne `integer` (heltal) og `string` (tekststreng),
- ◆ kommandoen `print x`, der udskriver `x` på skærmen.

Semikolon (;) afslutter en kodelinie (et statement). Tekst efter to skråstreges (\\\) er kommentarer (udføres ikke, når programmet køres).

WWW

ØVELSER I ALLE KONSTRUKTIONERNE

Gennemgangen af de imperative konstruktioner i dette afsnit er kort, men på **WWW** er der rigeligt med øvelser (med ægte programmeringssprog).

6.6.1. VARIABLE OG UDTRYK

Imperativ programmering gør stort set altid brug af variable.

ORDFORKLARING

VARIABEL

Et navn, (*variabelnavnet*), der kan tilknyttes en værdi. En variabel oprettes ved en erklæring (engelsk: *declaration*). Tilknytningen defineres ved en tildeling (engelsk: *assignment*). Den tilknyttede værdi opnås ved at *derefere* variablen. Variablen har som regel en datatype, svarende til datatypen af den *derefererede* værdi.

ANALOGI

VARIABEL ≈ PAPIRLAP

En variabel er som en papirlap, hvor der til hver en tid kan stå en og kun en værdi. Erklæring af en variabel svarer til at tage en papirlap frem for at tage den i brug. Tildeling svarer til at skrive en værdi på lappen (og først viske den forrige værdi ud, hvis der var en). Dereferering sker, når vi aflæser den nuværende værdi på papirlappen.

PROGRAM # 6.1 – GRUNDLÆGGENDE BRUG AF VARIABEL

```
integer x;      \\ ERKLÆRING af variablen x som en variabel  
                \\ med datatype heltal. Variablen x er endnu  
                \\ ikke tilknyttet en værdi.  
  
x = 2;         \\ TILDELING af værdien 2 til variablen x.  
  
print x;       \\ DEREFERERING af variablen x.  
                \\ Skærmen udskriver "2".
```

En variabel kan tildeles værdien af en anden variabel. Skifter den anden variabel senere værdi, påvirker det ikke den første variabel! Vi laver et nyt program til at illustrere:

PROGRAM # 6.2 – TO VARIABLE

```
integer x;      \\ Erklæring af x  
  
integer y = 3;  \\ Erklæring af y og tildeling af værdien 3,  
                \\ samlet i en kodelinie.  
  
x = y;         \\ y derefereres. Tildeling: x har nu værdien 3.  
  
y = 4;         \\ Tildeling: y har nu værdien 4  
  
print x;       \\ x derefereres. Skærmen udskriver "3".
```

En variabel, der er erklæret, men endnu ikke tildelt en værdi, tildeles ofte den specielle værdi `null` (engelsk for *intet*). Sådanne variable kan normalt ikke indgå i beregninger, før de er tildelt en værdi. Bemærk at den særlige værdi `null` ikke er det samme som talværdien 0.

Vi kigger nu på et fejlbehæftet(!) program, der ikke undgår, at `null` indgår i regneudtryk.

PROGRAM # 6.3 – NULL

```
integer ups;      \\ ups erklæres.  
                  \\ ups har datatype integer og værdi null.  
  
integer y = 3;    \\ y erklæres.  
                  \\ y har datatype integer og værdi 3.  
  
integer superups = ups+y; \\ Erklæring af superups, DER VIL  
                           \\ RESULTERE I EN FEJL,  
                           \\ da udtrykket "ups+y", altså "null + 3",  
                           \\ ikke giver mening.
```

TIP**UNDGÅ FEJL MED null**

Fejl i forbindelse med manglende værdi-tildelinger (som i **Program # 6.3**) er almindelige. Når du koder, kan det betale sig fra start af at være opmærksom på, at ingen variable risikerer at være lig med `null` når de skal bruges.

TIP**GIV VARIABLE SIGENDE NAVNE**

Ved at give variable sigende navne letter man programmeringsarbejdet for sig selv. Hvis en talvariabel repræsenterer en persons alder, er variabelnavnet `age` klart at foretrække frem for `x`, `y` eller `shoe-size` (sidstnævnte er ret dumt!).

Syntaksen for et programmeringssprog fastlægger ofte et sæt “reserverede ord”: ord, der har en speciel betydning inden for sproget (i Imperativtøvesprog er fx `integer` et reserveret ord). Sådanne ord kan ikke bruges som variabelnavne.

6.6.2. LISTER

Indenfor programmering er det ofte praktisk – og meget almindeligt – at flere beslægtede elementer af samme datatype samles i en liste. Det kunne fx være en oversigt over personhøjder i centimeter angivet som heltal. En liste med elementer af datatype `<D>` har datatypen `<D>-list`.

Vi kan fx kode listen `[180;169;190;185]` af højderne på de 4 brødre Adam, Bent, Carl og Didrik som følger:

```
integer-list heightList;           \\ Erklærer en heltals-liste,  
                                  \\ der hedder "heightList"
```

```
heightList = [180, 169, 190, 185];
```

Listeindhold kan findes frem via indholdets placering i listen (indholdets index). Fx finder man den 180 cm høje Adams højde på index 0 (den første plads) i listen `heightList` og den 190 cm høje Didriks højde på index 3 (den fjerde plads).

I Imperativtøvesprog gøres det ved at dereferere listen med udtrykket

```
<navn>[<tal>]
```

der giver det element i listen `<navn>` der har index `<tal>`. Udtrykket

```
heightList[0]
```

derefører listen `heightList` og giver elementet på index 0, dvs. tallet 180. En liste af den type, vi har indikeret her, kaldes på engelsk et *array*, og på dansk også en *tabel*, fordi man kan opfatte listen som en tabel, hvor man slår værdier op ved hjælp af index:

INDEX	0	1	2	3
VÆRDI	180	169	190	185

En liste er en *datastruktur*.

ORDFORKLARING

DATASTRUKTUR

Måde at organisere og opbevare en mængde relateret data på.

Lister er langt fra den eneste datastruktur. Der er fx også to-dimensionale lister – det vi normalt ville kalde skemaer. Og køer, stakke, træstrukturer. Endda grafer. Det fører for vidt at komme ind på disse strukturer her, men vi giver en figur, der indikerer disse datastrukturers “udseende”.



Fig. 6.95. Datastrukturer: Skema, kø, stak, træ og graf.

WWW

SPAS MED DATASTRUKTURER

På **WWW** arbejder vi lidt med de forskellige datastrukturer.

6.6.3. METODER OG UDTRYK

Det er meget almindeligt at indkapsle funktionalitet, der er interessant i sig selv eller kan genbruges, i såkaldte *metoder*. Det kunne fx være algoritmer med bestemte delformål. Metoder kaldes også, med stort set samme betydning, *funktioner*, *procedurer* eller *rutiner*.

ORDFORKLARING

METODE, METODEKALD, PARAMETRE, RETURVÆRDI, RETURTYPE

En metode er en blok af statements, der tilsammen udfører en logisk sammenhængende opgave. Metoden aktiveres med et *metodekald*. En metode kan *tage parametre* (input-værdier) og kan give en *returværdi* (output). For en given metode er typen af returværdierne altid den samme og kaldes metodens *returtype*.

Vi illustrerer begreberne med eksempler.

Første eksempel er metoden `print`, der udskriver sin parameters værdi på skærmen. Metoden har én parameter og returnerer ikke nogen værdi. I stedet skrives blot til skærmen.

PROGRAM # 6.4 – METODE UDEN RETUR-VÆRDI

PRINT

```
print(34);          \\ Udskriver "34" på skærmen
```

Udtrykket `print(34)` er et metodekald til metoden `print` med parameteren 34.

Definition af metoder omfatter definition af metodens navn, eventuel returtype, eventuelle parametre, samt metodens ”krop” : De linjer kode, der udtrykker selve det, ”metoden gør”. Syntaksen for definition af funktioner i Imperativtøvesprog er

```
<returtype> <metodenavn>(<parametre>) { <metodens krop> }
```

PROGRAM # 6.5 – METODE UDEN PARAMETRE, MED RETURVÆRDI

RETURNERER ALTID ”ABC”

```
string metode1() {           \\ Oprettelse af metoden metode1
    return "ABC";            \\ med returtype string
}

string a = metode1();        \\ Variablen a med datatype string
                           \\ oprettes og tildeles værdien "ABC"
```

Udtrykket `metode1()` er et metodekald til metoden `metode1`, der ikke tager parametre. Vi kan give metoderne lige de navne, vi vil. Ovenfor har vi ikke givet metoden et sigende navn. Det er ellers praktisk, så det gør vi fra nu af:

**PROGRAM # 6.6 – METODE MED TO PARAMETRE, UDEN RETURVÆRDI
PRINTER SUMMEN AF PARAMETRENE**

```
printSum(x : integer, y : integer){  
    print(x+y);  
}  
  
printSum(2,7);                                \\ udskriver "9" på skærmen
```

**PROGRAM # 6.7 – METODE MED TO PARAMETRE OG RETURVÆRDI
PRINTER PARAMETRENES SUM, RETURNERER DERES SUM**

```
integer returnSum(x: integer, y : integer){  
    print("De to tals sum er:");  
    print(x+y);  
    return x+y;  
}  
  
integer w = return(13,13);                      \\ udskriver følgende på skærmen:  
                                                \\ "De to tals sum er:  
                                                \\ 26"  
                                                \\ Variablen w med datatype integer  
                                                \\ oprettes og tildeles værdien 26.
```

Kombinerer vi metoderne `printSum` og `returnSum` kan vi fx lave

PROGRAM # 6.8 – METODEKALD

```
integer t =  
returnSum(2,4);                                \\ Variablen t oprettes og tildeles  
                                                \\ værdien 6.  
                                                \\ "6" udskrives på skærmen  
  
printSum(t,t);                                \\ "12" udskrives på skærmen
```

ADVARSEL

METODER I PROGRAMMERING ER IKKE “MATEMATISKE FUNKTIONER”

I matematik har en funktion formen $y = f(x)$: Variablen x indsættes, og så kan y beregnes. En matematiker er kun interesseret i at få beregnet y . En metode er noget helt andet end en matematisk funktion:

- metoder behøver ikke tage et input- x (metoder uden parametre),
- metoder behøver ikke give et output- y (metoder uden returværdi),
- selve beregningen af metoden kan have sideeffekter (fx at skrive resultater ud på skærmen).

Med metodebegrebet på plads kan vi også definere et *udtryk* præcist.

ORDFORKLARING

UDTRYK, DATATYPE OG EVALUERING

Et *udtryk* er en kombination af værdier, variable, metoder og operationer, der kan beregnes og til sidst give en værdi. Udtrykket siges at have den datatype, som dets værdi har. Udtryk siges at *evaluere* til deres værdi.

EKSEMPEL

UDTRYK

- “`Bi`” er et udtryk af typen tekststreg, og det evaluerer til “`Bj`”.
- `(5+7)/2` er et udtryk af typen heltal, der evaluerer til `6`.
- `returnSum(2, 3)+11` er et udtryk af typen heltal, der evaluerer til `16`. Under evalueringen af udtrykket vil skærmen udskrive tallet `5`.
- `TRUE+2` er ikke et udtryk, fordi det ikke engang har en type – der er typefejl: Sandhedsværdier kan ikke lægges sammen med tal.
- Vi husker, at tegnet `^` betyder “sammensætning af tekststrenge”. `“DEF”^“metode1()”` er et udtryk af typen tekststreg, der evaluerer til `“DEFABC”`.

6.6.4. KONTROL-FLOW

Når et program er under afvikling, afvikles dets statements som udgangspunkt i rækkefølge. Det er imidlertid uhyre praktisk at kunne styre den rækkefølge, som koden udføres i. På den måde kan koden fx handle på baggrund af betingelser. Den måde, programmer løber gennem koden, når programmet afvikles, kaldes *kontrol-flowet*, og det er det, vi er interesserede i at styre.

Til at styre kontrol-flowet bruges *kontrolstrukturer*.

KONTROLSTRUKTURER		
STATEMENT	ENGELSK NAVN	EFFEKT
Ubetinget hop	<i>Jump</i>	Fortsætter fra et nyt, angivet sted i koden
Forgrening (eller: betinget hop)	<i>Branch</i>	Afhængig af angivne betingelser fortsættes enten uden videre, eller der hoppes til et nyt sted i koden
Løkke	<i>Loop</i>	En angiven mængde statements udføres 0 eller flere gange, afhængigt af angivne betingelser
Metodekald	<i>Method call</i>	Et sæt statements et andet sted i koden udføres, hvorefter der fortsættes
Stop programmet	<i>Halt</i>	Programmet stopper

Fig. 6.96. Oversigt over typiske kontrolstrukturer.

Vi vil gennemgå vi forgreninger og løkker i nogen detalje, mens vi lader Figur 6.97 være den fulde forklaring af “ubetinget hop” og “stop”. Metodekald kiggede vi på i afsnit 6.6.3.



@attribute vistaicons.com.

Fig. 6.97. Hop, forgrening, løkke, metodekald og programstop.

6.6.5. KONTROLSTRUKTUR: FORGRENING

En forgrening har typisk form af en “hvis”-betingelse. I vores fiktive programmerings-sprog har en forgrening formen

```
IF <Bedingelse> THEN <Statements1> ELSE <Statements2> ENDIF
```

Denne overordnede form er meget almindelig. Her er **<Bedingelse>** et udtryk, der evaluerer til en sandhedsværdi og **<Statements1>** og **<Statements2>** serier af statements. **<Statements1>** udføres, hvis **<Bedingelse>** evaluerer til TRUE, og **<Statements2>** hvis **<Bedingelse>** evaluerer til FALSE. Værdien af **<Bedingelse>** kan først beregnes på kørselstidspunktet på programmet.

EKSEMPEL

IF-THEN-ELSE-BETINGELSE

Kodestumpen her, der antager at **x** er en tal-variabel, benytter en forgrening:

```
IF  
  (x < 0)                                \\ <Bedingelse>  
  THEN  
    print("Det var et negativt tal");      \\ <Statements1>  
  ELSE  
    print("Det var måske 0...");           \\ <Statements2>  
    print("...Ellers var det et positivt tal"); \\ <Statements2>  
  ENDIF
```

Har man ikke tænkt sig at have nogen statements i **ELSE**-grenen af en forgrening, kan man benytte den kortere form

```
IF <Bedingelse> THEN <Statements> ENDIF
```

EKSEMPEL

IF-THEN-BETINGELSE

Kodestumpen her, der antager at `x` er en tal-variabel, benytter en forgrening, hvor den ene gren er "tom":

```
IF  
  (x == 0)                                \\ <Betingelse>  
  THEN  
    print("Det var 0! Hurra!");             \\ <statements>  
  ENDIF
```

Her har vi benyttet nogle naturlige måder at formulere betingelser på (nemlig med `<` og `==`, der betyder "mindre end?" og "lig med?"). Generelt er betingelser *logiske udtryk*, dvs. udtryk, der evaluerer til en sandhedsværdi (`TRUE` eller `FALSE`).

TYPISKE LOGISKE UDTRYK		
FORM	BETYDNING	EKSEMPLER
<code><Udtryk1> == <Udtryk2></code>	"Er <code><Udtryk1></code> lig med <code><Udtryk2>?</code> " Evaluerer til <code>TRUE</code> , hvis de to udtryk evaluerer til samme værdi, og <code>FALSE</code> ellers.	<code>5 == (2+3)</code> evaluerer til <code>TRUE</code> <code>null == 2</code> evaluerer til <code>FALSE</code> <code>null == 0</code> evaluerer til <code>FALSE</code>
<code><Udtryk1> != <Udtryk2></code>	"Er <code><Udtryk1></code> forskelligt fra <code><Udtryk2>?</code> " Evaluerer til <code>FALSE</code> , hvis de to udtryk evaluerer til samme værdi, og <code>TRUE</code> ellers	<code>(1+1+1) != 4</code> evaluerer til <code>FALSE</code> <code>"To" != "Tre"</code> evaluerer til <code>TRUE</code>
<code>NOT <Logiskudtryk></code>	"Er <code><Logiskudtryk></code> falskt?" Evaluerer til <code>TRUE</code> , hvis <code><Logiskudtryk></code> evaluerer til <code>FALSE</code> , og <code>FALSE</code> ellers.	<code>NOT TRUE</code> evaluerer til <code>FALSE</code> <code>NOT FALSE</code> evaluerer til <code>TRUE</code>

$<\text{Logiskudtryk1}> \text{ AND } <\text{Logiskudtryk2}>$	<p>"Er begge logiske udtryk sande?"</p> <p>Evaluérer til TRUE, hvis $<\text{Logiskudtryk1}>$ og $<\text{Logiskudtryk2}>$ begge evaluerer til TRUE, og FALSE ellers</p>	FALSE AND TRUE evaluerer til FALSE $(2==2) \text{ AND } (2!=3)$ evaluerer til TRUE
$<\text{Logiskudtryk1}> \text{ OR } <\text{Logiskudtryk2}>$	<p>"Er mindst et af de to logiske udtryk sandt?"</p> <p>Evaluérer til TRUE, hvis en af udtrykkene $<\text{Logiskudtryk1}>$ eller $<\text{Logiskudtryk2}>$ evaluerer til TRUE, og FALSE ellers</p>	FALSE OR TRUE evaluerer til TRUE $(2==2) \text{ OR } (2!=3)$ evaluerer til TRUE FALSE OR $(\text{TRUE } != \text{TRUE})$ evaluerer til FALSE

Fig. 6.98. Grundlæggende logiske udtryk.

Yderligere 4 typer af logiske udtryk er ofte nyttige, nemlig de, der bygges med **>** (større end), **>=** (større end eller lig med), **<** (mindre end) og **<=** (mindre end eller lig med). Fx evaluerer udtrykket $(2+50+50) < 101$ til **FALSE**.

Som regel sammenlignes kun udtryk med samme datatype, når man bruger de logiske operatorer **==** og **!=**. Udtrykket $44 !=$ "agurkesalat" giver altså en typefejl, fordi tal og tekst ikke kan sammenlignes.

6.6.6. KONTROLSTRUKTUR: LØKKER

Ofte har man i et program brug for at udføre et handlingsmønster med et element af gentagelse. Det gøres let med *løkker*.

HANDLINGSMØNSTRE:	EKSEMPEL FRA DAGLIGDAGEN:
Optælle en samling objekter.	Hvor mange bananer er der i skålen?
Udføre nogle bestemte handlinger et antal gange.	Tage ti armbøjninger. (Ti gange skal de samme kropslige bevægelser udføres i samme rækkefølge).
Udvælge eet objekt blandt mange i en samling ved at "bladre" samlingen igennem.	Finde spar konge i et spil kort.
Behandle hvert objekt i en samling.	Sætte frimærke på hvert eneste brev i en ordentlig stabel.

Fig. 6.99. Handlingsmønstre med et element af gentagelse.

En løkke i **imperativtøvesprog** har den meget almindelige form

```
WHILE <Betingelse> DO <Statements> ENDWHILE
```

While betyder “så længe”. Løkkens indhold udføres *så længe betingelsen i løkken er sand*. Mere konkret: Når løkken startes, evalueres det logiske udtryk **<Betingelse>**. Er betingelsen sand, udføres **<Statements>**, hvorefter løkken “genstartes” – det logiske udtryk **<betingelse>** evalueres igen, osv. Evalueres betingelsen på et tidspunkt til FALSE, forlades løkken.

Vi giver et enkelt eksempel på brug af en løkke her. Ægte programmeringssprog har ofte flere forskellige varianter af løkker.

EKSEMPEL

WHILE-LØKKE

Kodestumpen her, der antager at **x** er en tal-variabel, benytter en løkke:

```
WHILE  
(x < 0)                                \\ <Betingelse>  
DO  
    print("x er stadig negativ! Øv!");      \\ <Statements>  
    x = x+1;                                \\ <Statements>  
ENDWHILE  
print("Nu blev x endelig 0 eller positiv! Hurra!");
```

Løkken lægger 1 til **x** så mange gange, at **x** ender med at blive positiv. Hvis **x** fra start er lig med -5, besøges løkkens indre 5 gange. Hvis **x** fra start er lig med +10, besøges løkkens indre slet ikke.

6.7. OBJEKT-ORIENTERET PROGRAMMERING

Ideen i *objekt-orienteret programmering* (forkortet: *OOP*) er at bruge såkaldte “objekter” og deres samspil til at definere programmer med.

Objekter er som “ting”, hver med sine evner og ansvarsområder. Objekter kan kommunikere gennem velspecificerede kanaler. Hvor imperativ programmering går ud på at lade computeren udføre en masse statements efter hinanden, evt. i form af metodekald, virker objekt-orienteret programmering ved, at en masse objekter kommunikerer med hinanden.

6.7.1. HVAD ER ET OBJEKT?

I objekt-orienteret programmering er et objekt en “ting” med følgende egenskaber:

- ◆ man kan snakke med objektet ved at sende beskeder,
- ◆ objektet kan reagere på beskeden,
- ◆ objektets reaktion afhænger af dets tilstand,
- ◆ objektet kan også foretage sig noget, selvom det ikke modtager beskeder.

Endvidere har et objekt en *identitet*. To objekter kan altså godt være forskellige, selvom de opfører sig på samme måde, når de modtager beskeder, og har samme tilstand.

ORDFORKLARING

OBJEKT

Et objekt er en system-repræsentation af en ting eller et begreb. Et objekt har en tilstand, en opførelse og en identitet.

EKSEMPEL

OBJEKTER: FODBOLDE

Fodbold er objekter. To fodbold ligger på en græsplæne, den ene er flad, den anden nypumpet. Du ”snakker” nu med den flade fodbold ved at give den et solidt inderside-spark. Fodbolden reagerer på din ”besked” ved slapt at trille et par meter. Reaktionen afhænger tydeligvis af boldens tilstand. Giver du bagefter den nypumped fodbold samme ”besked”, ”svarer” den ved at trille meget længere, fordi dens tilstand er ”nypumpet”.

EKSEMPEL

OBJEKT-IDENTITET: TERNINGER

Efter et spil Yatzy ligger seks terninger på bordet. De viser alle seks. Terningerne er helt ens og har samme tilstand, nemlig ”Viser seks”. Alligevel er terningerne seks *forskellige* objekter.

EKSEMPEL

OBJEKTER KAN VÆRE ANDET END ”TING”

Begreber, relationer eller følelser kan også være objekter: Sigrids humør er et objekt. Sofie niver nu Sigrid hårdt, uden nogen grund! Altså sendes beskeden ”Sofie niver Sigrid hårdt” til objektet ”Sigrids humør”. Objektet skifter straks tilstanden til ”Gnaven”. Niver Sofie igen Sigrid, skifter objektet tilstand til ”Rasende”. Et objekts reaktion på to helt ens beskeder behøver altså ikke være den samme.

EKSEMPEL

OBJEKT, DER IKKE MODTAGER BESKEDER: KLOKKEN

Objektet "Klokken", der mäter klokken i "hele minutter", har tilstanden "Klokken er TT:MM", hvor TT er timetallet og MM minuttallet. Objektet skifter tilstand hvert minut – uden først at modtage bestemte beskeder.

I en computer er et objekts tilstand repræsenteret som data. Som regel har et objekt et antal *attributter*, variable, der hver har en værdi. Nogle attributter kan ændre værdi gennem objektets levetid, andre kan ikke.

EKSEMPEL

PERSON-OBJEKT

Vi vælger nu at repræsentere folk som **Person**-objekter med attributterne **Navn** og **CPR-nummer**. Hvis Herman Clausen har CPR-nummer 0901941175, kan vi repræsentere ham som et objekt med følgende tilstand:

ATTRIBUT	VÆRDI	DATATYPE
Navn	Herman Clausen	String
CPR-nummer	0901941175	Integer

Da folk kan tage navneforandring, er værdien af attributten **Navn** omskiftelig. Værdien af attributten **CPR-nummer** kan ikke skifte.

Et objekt kan have andre objekter som attributter.

EKSEMPEL

DANMARK

Vi repræsenterer nu Danmark som et **Land**-objekt med bl.a. attributten **Indbyggere**. Værdien af attributten **Indbyggere** er en liste af **Person**-objekter (lister af objekter er selv objekter).

Et objekt kan forstå netop de beskeder, det selv har valgt at ville forstå. Objektet har en grænseflade, der oplyser hvilke beskeder, der kan tages imod – i form af såkaldte *metoder*. Beskeder gives ved at bruge objektets metoder. Metoder kan ændre på objektets tilstand (værdien af dets attributter).

EKSEMPEL

METODER

Vi kigger igen på Herman Clausen, et `Person`-objekt. Objektet definerer metoden `oplysNavn()`

i sin grænseflade. Hvis vi spørger objektet "Hvad er dit navn?" ved at bruge metoden `oplysNavn()`, svarer objektet med returværdien "`Herman Clausen`" (en tekststreng). Har objektet også metoden

`skiftNavnTil(string)`

der foretager en navneforandring, kan vi bruge `Herman Clausen`'s metode `skiftNavnTil("Ib Clausen")`. Effekten er så, at objektets attribut Navn skifter værdi fra "`Herman Clausen`" til "`Ib Clausen`". Her ændrer metoden altså på objektes tilstand.

Med den nye terminologi kan vi sige, at objektets tilstand er værdierne af dets attributter, mens dets opførsel er givet ved dets metoder (og deres returværdier).

Det centrale koncept i OOP er, at objekter kommunikerer med hinanden, hvilket sker ved, at objekterne løbende kalder hinandens metoder.

6.7.2. KASSER

Når vi tænker på objekter fra den virkelige verden, er de naturligt organiseret i *klasser*. Alle fodbold er fx med i klassen `Fodbold`, alle mennesker er med i klassen `Person`, og Odin, Tor og Loke er med i klassen `FigurFraDenNordiskeMytologi`.

Denne tænkemåde er praktisk, når man skal programmere objekt-orienteret. Det ville være upraktisk, ja, ulideligt, hvis man skulle kode attributter, metoder og grænseflade for hvert enkelt objekt, man skulle bruge. Et objekt-orienteret bank-program har fx typisk et `Kunde`-objekt for hver kunde – formentlig tusindvis! I stedet ønsker vi at kode disse tre ting én gang for alle – og det muliggøres af klasser. Formålet med klasser er altså helt klart:

Skriv kun kode én gang!

Heldigvis understøtter de fleste objekt-orienterede sprog klasser.

ORDFORKLARING

KLASSE

En klasse repræsenterer en kategori af ting. Klassen er et stykke kode, som fastlægger grænseflade, attributter og metoder. Klassen bruges som en opskrift til at lave objekter, der repræsenterer eksemplarer af tingen. Sådanne objekter kaldes *instanser* af klassen.

En klasse kan tegnes som en boks:

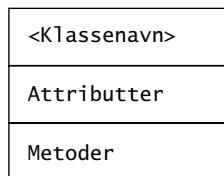
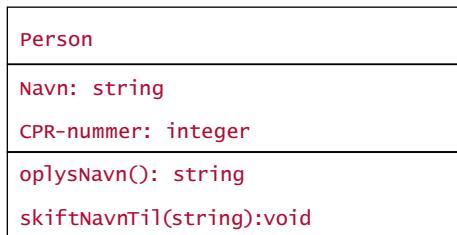


Fig. 6.100. Klasse: Klassenavn, attributter og metoder.

EKSEMPEL

KLASSE OG INSTANS

Herman Clausen, det velkendte `Person`-objekt, er en instans af klassen `Person`. Hans ven, `Person`-objektet Lars Friis, er også en instans af klassen `Person`. Vi kan tegne klassen `Person` som følgende boks:



Bemærk, at vi for attributterne `Navn` og `CPR-nummer` har angivet datatype, mens vi for metoderne `oplysNavn()` og `skiftNavnTil(string)` har angivet parametre og returntype. `oplysNavn()` har ingen parametre og returnerer en `string`. `skiftNavnTil(string)` har en `string`-parameter, men returnerer ingenting – metoder der ikke returnerer noget, siges ofte at have returntypen `void` (engelsk for "tom").

6.7.3. NEDARVNING

En klasse A kan være en *delklasse* af en anden klasse B. I så fald siges klassen B at være klassen A's *superklasse*.

ORDFORKLARING

DELKLASSE

En delklasse er en udvidet, specialiseret udgave af sin superklasse. Den indeholder superklassens metoder og attributter, og muligvis flere. Delklassen siges at nedarve fra sin superklasse.

Delklasser tegnes ofte som i diagrammet i Figur 6.100.

EKSEMPEL

DELKLASSE

Klassen **Mand** er en delklasse af klassen **Menneske**. Klassen **Kvinde** er også en delklasse af **Menneske**.

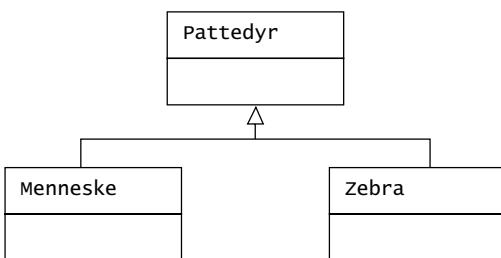


Fig. 6.101. Delklasser.

Klassen **Menneske** er en delklasse af klassen **Pattedyr**, ligesom klassen **zebra** er. **Pattedyr** er en superklasse for **Menneske**. Klassen **Fodbold** er en delklasse af klassen **Bold**, som er en delklasse af **Sportsudstyr**.

Ideen med en delklasse er, at den er let at definere, hvis superklassen allerede er defineret. Man angiver blot i koden for delklassen, hvilken klasse der er superklassen, og så skriver man kode for den ekstra special-funktionalitet, delklassen tilbyder.

EKSEMPEL

BOLD OG SPORTSUDSTYR

Klassen **Bold** er en delklasse af klassen **Sportsudstyr**. Instanser af **Sportsudstyr** kan ikke pumpes op, fordi ikke alt sportsudstyr kan泵es op – det kan fx hverken ketchere, svedbånd eller badmintonbolde. En instans af **Bold** har metoden

pumpOp()

Metoden er specialiseret funktionalitet, som instanser af **Sportsudstyr** ikke har. Den kode, der definerer klassen **Bold**, skal blot indeholde

- ◆ erklæring af klassen **Bold**
- ◆ erklæring af, at superklassen er **Sportsudstyr**
- ◆ erklæring af metoden **pumpOp()**

Klassen **Bold** behøver ikke indeholde de ting, der allerede er defineret af klassen **Sportsudstyr**. Det er fx

- ◆ metoden **pris()**
- ◆ metoden **forhandlere()**

der giver prisen på sportsudstyret, hhv. en liste af forhandlere.

6.8. NØGLEORD

- ◆ Programmering
- ◆ Algoritme
- ◆ Data vs. information
- ◆ Programmeringssprog
- ◆ Syntaks
- ◆ Datatype
- ◆ Højniveausprog
- ◆ Lavniveausprog
- ◆ Oversættelse
- ◆ Fortolkning
- ◆ Variabel
- ◆ Værdi
- ◆ Erklæring
- ◆ Tildeling
- ◆ Dereferering
- ◆ Evaluering
- ◆ Udtryk
- ◆ Metode
- ◆ Forgrening
- ◆ Løkke
- ◆ Objekt
- ◆ Identitet, tilstand og opførsel

KAPITEL 7

DATABASER

Databasesystemer bruges til at opbevare data, så det er *let* og *hurtigt* at finde, gemme og ændre data i systemet. Der er databaser overalt, og de fleste mennesker er i (indirekte) berøring med databaser hver dag.

EKSEMPEL

DATABASER

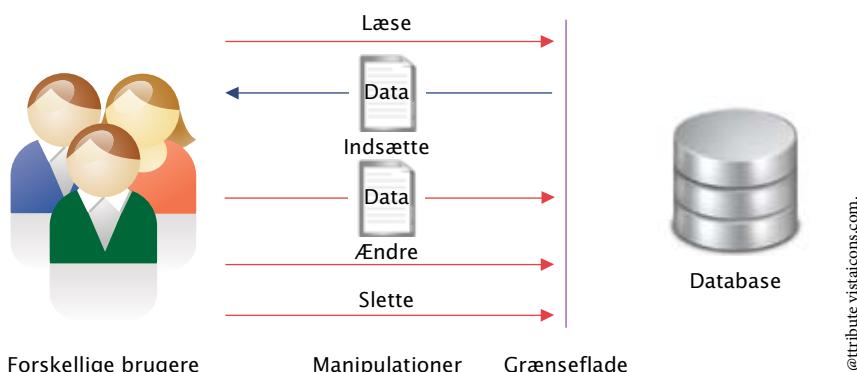
Der hæves penge, bestilles flybilletter og købes varer over internettet – det er transaktioner, der involverer databaser. Virksomheder lagrer informationer om medarbejdere, kunder og økonomi i databaser. Efterretningstjenester lagrer forbryderprofiler i databaser.

ORDFORKLARING

DATABASESYSTEM

Et databasesystem består af en *database*, en struktureret datamængde, samt en *grænseflade* hvorigennem databasebrugere kan *manipulere* databasen, dvs. kan:

- ◆ læse data i databasen
- ◆ indsætte data i databasen
- ◆ ændre i data i databasen
- ◆ slette data i databasen



Attributed via icons.com.

Fig. 7.102. Database med manipulationer via en grænseflade.

Konkret er en database en kompleks samling af sammespillende programmer, kode og filer. Grænsefladen giver brugere af databasen et abstrakt syn på databasen, så de ikke skal bekymre sig om, *hvordan* databasen fungerer. Grænseflader udgøres ofte af internetbaserede programmer. Læsning af en del af databasen kaldes et *dataudtræk*.

EKSEMPEL

GRÆNSEFLADER, UDTRÆK OG MANIPULATION

Når du hæver penge i en automat, er automaten din grænseflade til den database, der indeholder bankens kundekonti. Du kan manipulere med "din" del af databasens indhold ved at hæve penge eller udskrive en kontoversigt. Din hjemlige netbank er en internet-grænseflade, der også kan lave udtræk og manipulere. Bankens ansatte bruger særlige bank-programmer, der kan manipulere enhver kundekonto – og fx hæve gebyrer fra den, ændre dens rentesats eller lukke den (gys!).

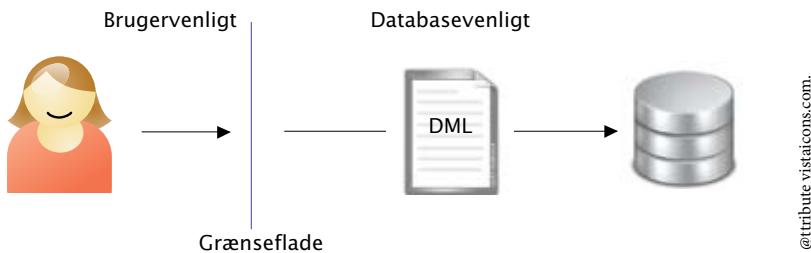
Et godt databasesystem har en række systemkvaliteter, vi værdsætter ved lagring af store mængder data:

- ◆ *pladseffektiv lagring*: databasen er billig at lægge diskplads til,
- ◆ *fleksibelt design*: databasen er let at bruge, vedligeholde og udvide,
- ◆ *data er lagret sikkert*: ondsindede personer kan ikke uretmæssigt ændre databasens indhold,
- ◆ *data er korrekt*: databasens indhold afspejler den virkelighed, databasen modellerer.

I resten af dette kapitel forklares systemkvaliteterne nærmere, og vi kommer ind på, hvordan man kan sikre dem (for en databaseset type, der kaldes en *relationel database*).

7.1. DATAMANIPULATION

Data i en database udtrækkes og manipuleres med kommandoer der kaldes *forespørgsler*. Forespørgsler formuleres i et *DML* (data manipulation language) . Kommunikerer man med databasen gennem en grænseflade, oversætter den ofte brugervenlige handlinger (såsom afkrydsninger i en liste af varenavne eller klik på symboler) til DML-forespørgsler.



@tttribute via icons.com.

Fig. 7.103. Manipulation af database med DML gennem grænseflade.

WWW

FORESPØRGSLER

På **WWW** gennemgår vi alle de almindeligste typer forespørgsler og de konstruktioner, man kan lave med dem for DML-sproget SQL.

7.2. DATAMODEL OG SKEMA

Før en database kan oprettes, skal det besluttes, *hvordan* data skal lagres rent logisk. Vi skal med andre ord fastlægge databasens underliggende *datamodel*. En datamodel beskriver i ”menneskesprog”, hvordan databasens elementer ser ud og forholder sig til hinanden.

ORDFORKLARING

DATAMODEL

En model, der beskriver data, datarelationer og databegrænsninger.

Datamodeller laves ofte som såkaldte *E/R-modeller*. Vi uddyber i afsnit 7.3, hvordan data og datarelationer modelleres med E/R. Udfra en E/R-model kan bygges en relationel database – det kigger vi på i afsnit 7.4. Vi kigger på databegrænsninger for den relationelle database i afsnittene 7.6, 7.8.1, 7.8.2 og 7.8.3.

En af kunsterne i godt databasedesign er at vælge en god datamodel.

For at bygge en faktisk database skal datamodellen først omsættes til et *skema* for databasen. Skemaet formidler datamodellen i et særligt kodesprog, der kaldes et *DDL* (engelsk: *data definition language*) . Et databaseprogram læser DDL-koden og opretter databasen:

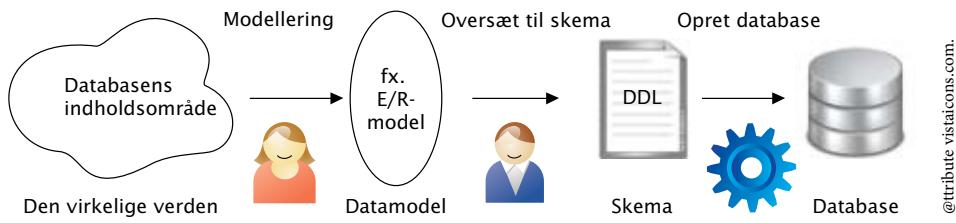


Fig. 7.104. Fra datamodel til skema og fra skema til (tom) database.

Databasen *er tom til at starte med*, men kan efter oprettelse fyldes op gennem DML-forespørgsler.

Skemaet beskriver databasens overordnede design. Skemaet ændrer sig ikke, selvom databasens *indhold* gør.

7.3. DATAMODEL: E/R-MODELEN

E/R står for “Entity/Relationship”. E/R-modellen anskuer den virkelige verden som bestående af basale “enheder”, kaldet *entiteter*, og *relationer* mellem disse entiteter.

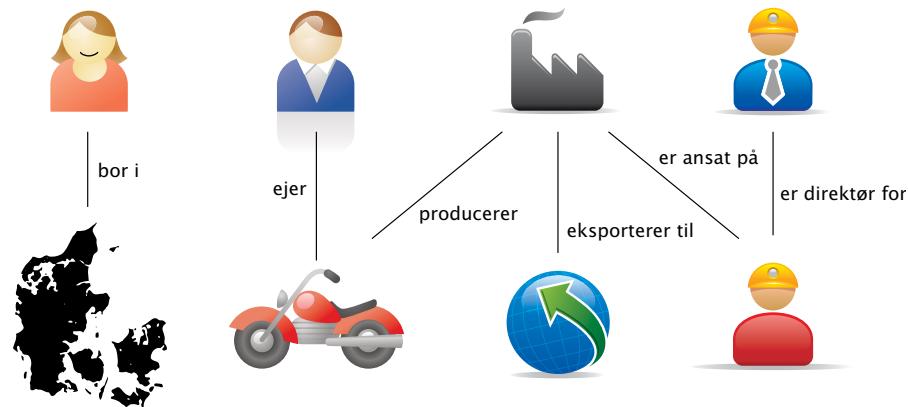


Fig. 7.105. Entiteter og relationer. På figuren ses entiteter, der modellerer personer, lande, fabrikker, stillinger, produkter og eksportdestinationer. Relationerne er angivet med linier.

Entiteters egenskaber kaldes *attributter*. Attributter er også en del af E/R-modellering.



Fig. 7.106. Attributter. På figuren ses en Sparegris-entitet. Sparegris-entiteten modellerer en virkelig sparegris, der indeholder kontanter i 4 forskellige valutaer. Sparegrisen indeholder 100 dollar, 50 euro, 0 yen og 10 pund. dollar-attributten har derfor værdien 100, euro-attributten værdien 50, yen-attributten værdien 0 og pund-attributten værdien 10.

Entiteter, relationer og attributter har vist sig at være godt værktøj til at designe en database med. Vi går derfor i dybden med E/R-modellering – det gør det lettere at forstå relationelle databaser bagefter.

7.3.1. ENTITETER OG ENTITETSKLASSER

Entiteter modellerer ofte fysiske enheder som fx personer, cykler, fly, varer eller kunder. Entiteter samles i *entitetsklasser*. Fx kan Rikke og Søren begge modelleres som person-entiteter i entitetsklassen Person.

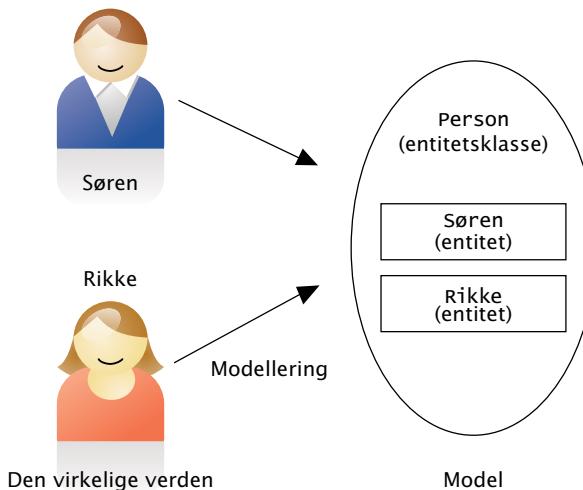


Fig. 7.107. Entiteter og entitetsklasser.

Entitetsklasser kan godt overlappes med hinanden. Entitetsklasserne **Ansæt** og **Kunde** for firmaet Modellering A/S overlapper med de entiteter, der svarer til personer, der er både kunder og ansatte i Modellering A/S.

7.3.2. ATTRIBUTTER

Entiteter forsynes med attributter. Attributterne *beskriver* entiteterne – fx har entitetsklassen **Person** attributterne navn, køn, alder og CPR-nummer, mens entitetsklassen **Cykel** har attributterne stelnummer og antal gear.

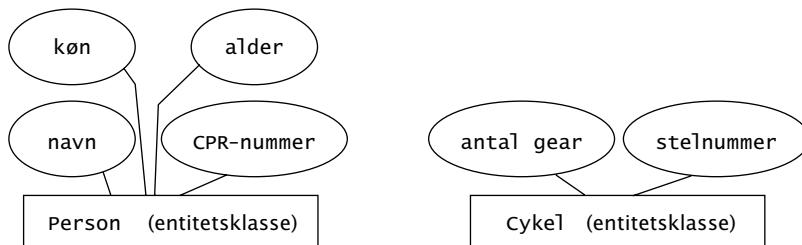


Fig. 7.108. Attributter for entitetsklasserne Person og Cykel.

Hver entitets attributter har *værdier*. Når en database er opbygget på baggrund af en E/R-model, er det især attribut-værdierne, der er det interessante ved databasen. Attributværdierne *afhænger af entitet og kan skifte over tid*.

Attributter antager værdier indenfor en bestemt *værdimængde*. Fx antager attributten **køn** værdier i værdimængden {Mand, Kvinde}, mens attributten **CPR-nummer** antager værdier i værdimængden “de ikke-negative heltaal” og **navn** i værdimængden “personnavne”.

Vi kan altså repræsentere Jacob Maltesen, der har CPR-nummer 090187-1125, som **Person**-entiteten (Navn: Jacob Maltesen, Køn: Mand, CPR-nummer: 090187-1125, Alder: 21).

7.3.2.1. SIMPLE OG SAMMENSATTE ATTRIBUTTER

Attributter er enten *simple* eller *sammensatte*. Simple attributter antager “atomiske værdier”, hvilket løst sagt betyder, at værdierne ikke kan inddeltes i meningsfulde underdele. Sammensatte attributter antager derimod værdier, der meningfuldt kan opsplittes i del-attributter. Fx er attributten **navn** sammensat, fordi den består af de tre del-attributter **fornavn**, **mellemlægnavn** og **efternavn**. Attributten **CPR-nummer** er simpel (tal er ikke sammensat af mindre dele), og det er delattributten **fornavn** også.

7.3.2.2. EN-VÆRDI-ATTRIBUTTER OG FLERVÆRDI-ATTRIBUTTER

Attributter er enten *en-værdi-attributter* eller *flerværdi-attributter*. En-værdi-attributter har kun én værdi tilknyttet (fx er CPR-nummer en en-værdi-attribut), mens flerværdi-attributter har 0, 1 eller flere værdier. En **Tolk-entitet** har fx attributten **sprog**, der oplister hvilke sprog (0, 1 eller flere), tolken kan oversætte imellem.

7.3.2.3. ATTRIBUT-VÆRDIENTEN NULL

Der kan være grunde til, at en entitet “mangler” en værdi for en given attribut:

- ◆ Attributten giver ikke mening for den pågældende entitet. Fx behøver en Person-entitet ikke have mellemnavn(e), og attributten **mellemnavne** har i sådanne tilfælde ingen meningsfuld værdi.
- ◆ Attributtens værdi kendes ikke. Fx kan vi mangle viden om, hvor mange gear cyklen med stelnummeret “WBA 09001291024 S” har.

I sådanne tilfælde angives værdien af attributten til at være **NULL** (engelsk for “tom”).

7.3.3. KANDIDATNØGLER OG PRIMÆRNØGLER

For at kunne udpege en bestemt entitet i en entitetsklasse skal vi kunne identificere den pågældende entitet. Det gøres med en *primærnøgle*.

ORDFORKLARING

PRIMÆRNØGLE

Minimal samling af en eller flere attributter, hvis værdier entydigt bestemmer en entitet i entitetsklassen.

Primærnøgler bruges til at *slå op* med: Kender vi værdien af attributterne i primærnøglen, kan vi slå den entitet op, der matcher værdierne. – Og der vil kun være én entitet, der matcher primærnøglen.

EKSEMPEL

PRIMÆRNØGLE

Entitetsklassen **Person** har attributten **CPR-nummer** som primærnøgle, fordi forskellige personer har forskellige CPR-numre. Vi kan altså “slå personer op” med CPR-numre. Attributsamlingen {**CPR-nummer**, **navn**} er ikke en primærnøgle, fordi den ikke er minimal: Allerede CPR-nummeret bestemmer entiteten.

Attributten **navn** alene er heller ikke en primærnøgle, fordi to forskellige personer kan have det samme navn.

I et E/R-design kan nogle entitetsklasser have flere mulige primærnøgler. Disse kaldes *kandidatnøgler* – da de er kandidater til at være primærnøgler. Det er E/R-designeren, der vælger, hvilken kandidatnøgle der skal være primærnøgle.

EKSEMPEL

KANDIDATNØGLER OG PRIMÆRNØGLER

I et banksystem lagres bankens kunder som *Kunde*-entiteter, der bl.a. har attributterne *kundenummer* (forskellige kunder har forskellige kundenumre), *CPR-nummer*, *kontonumre* (en kunde har en eller flere kontonumre) og *PBS-aftaler* (hver kunde har et antal betalingsservice-aftaler). I denne situation er både *kundenummer* og *CPR-nummer* kandidatnøgler. Da forskellige kunder kan deles om konti (fx ægtefæller) og dermed have de samme kontonumre, er *kontonumre* ikke en kandidatnøgle. Er samlingen {*CPR-nummer*, *kontonumre*} en kandidatnøgle? (Nej – fordi samlingen er ikke minimal!)

7.3.4. RELATIONER OG KARDINALITETER

En relation *sammenkobler to entitetsklasser*. Da en person kan eje en cykel, er *Ejer* en relation mellem entitetsklasserne *Person* og *Cykel*:

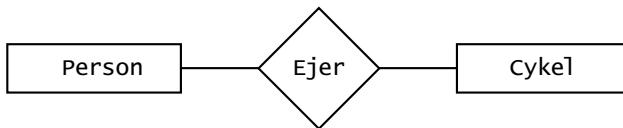


Fig. 7.109. Relation mellem entitetsklasser.

Lad os sige, at Rikke ejer cyklen med stelnummer “WBA 09001291024 S”. Så er relationen “Person-entiteten **Rikke** ejer **Cykel**-entiteten, hvis **stelnummer**-attribut har værdien WBA 09001291024 S” et eksempel på en konkret *Ejer*-relation i E/R-modellen.

Hvis der er en relation mellem to entitetsklasser, siges entitetsklasserne at *deltage* i relationen. Fx deltager både *Person* og *Cykel* i *Ejer*.

En relation har en *kardinalitet*, der udtrykker, “hvordan mange entiteter der kan være på hver side af relationen”. Fx kan en person eje ingen, en, to, tre, eller endnu flere cykler. Hver cykel ejes dog af netop en person – en cykel kan hverken have nul, to, tre eller flere ejere¹. Relationen *Ejer* fra *Person* til *Cykel* er altså en relation, hvor hver *Person*-entitet svarer 0, 1 eller flere *Cykel*-entiteter.

1 Strengt taget kan flere personer dele ejerskabet af en cykel – det ser vi bort fra her.

Lad os sige, at A og B er entitetsklasser med en relation imellem sig. For hver A-entitet A i entitetsklassen A måler kardinaliteten, hvor mange B-entiteter fra entitetsklassen B, A har relationer til. Der er 4 mulige kardinaliteter:

- ◆ **1-til-1:** En entitet i A er relateret til 0 eller 1 entitet i B. En entitet i A er relateret til 0 eller 1 entitet i B.
- ◆ **1-til-mange:** En entitet i A er relateret til 0, 1 eller flere entiteter i B. En entitet i B er relateret til 0, 1 eller flere entiteter i A.
- ◆ **mange-til-1:** En entitet i A er relateret til 0 eller 1 entitet i B. En entitet i B er relateret til 0, 1 eller flere entiteter i A.
- ◆ **mange-til-mange:** En entitet i A er relateret til 0, 1 eller flere entiteter i B. En entitet i B er relateret til 0, 1 eller flere entiteter i A.

For ikke at blande situationerne “1-til-mange” og “mange-til-1” sammen, er det vigtigt at tale om fx en 1-til-mange-relation *fra A til B*, og ikke blot en relation “mellem A og B”.

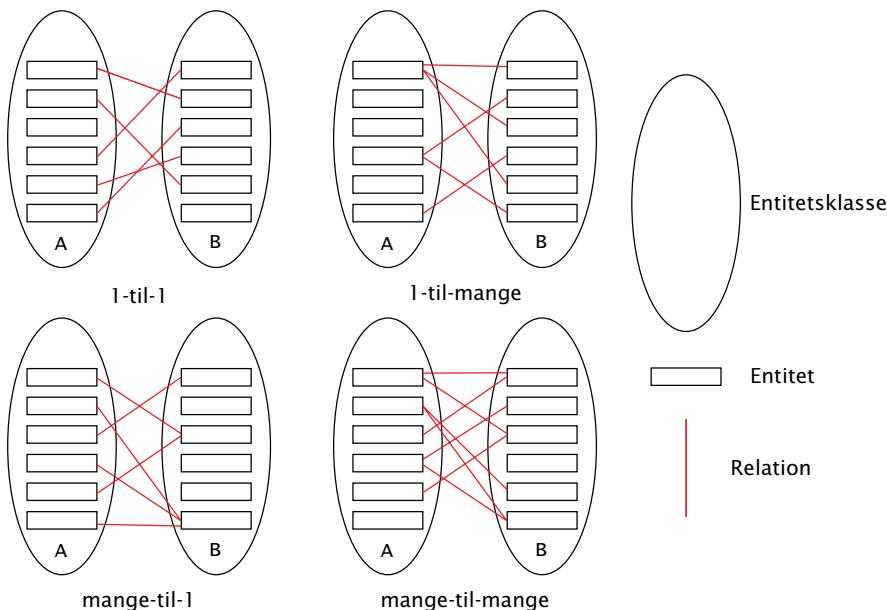


Fig. 7.110. Kardinaliteter for en relation.

EKSEMPEL

1-TIL-MANGE RELATION

Relationen **Ejer** fra Person til Cykel er en 1-til-mange-relation, fordi en person kan eje mange cykler (0, 1 eller flere cykler), og en cykel kan ejes af 1 person.

EKSEMPEL

1-TIL-1 RELATION

Relationen **Er rektor** på fra entitetsklassen **Rektor** til entitetsklassen **Gymnasium** er en 1-til-1-relation, fordi hver rektor er rektor på netop 1 gymnasium, og ethvert gymnasium har netop 1 rektor.

EKSEMPEL

MANGE-TIL-MANGE RELATION

Relationen **Har givet koncert på** fra entitetsklassen **Band** til entitetsklassen **Spillested** er en mange-til-mange-relation, fordi hvert band kan have givet koncert på 0, 1 eller flere spillesteder, og hvert spillested har lagt scene til 0, 1 eller flere koncerter med (varierende) bands.

En entitetsklasse *deltager fuldstændigt* i en relation, hvis alle entiteter i klassen deltager i en relation.

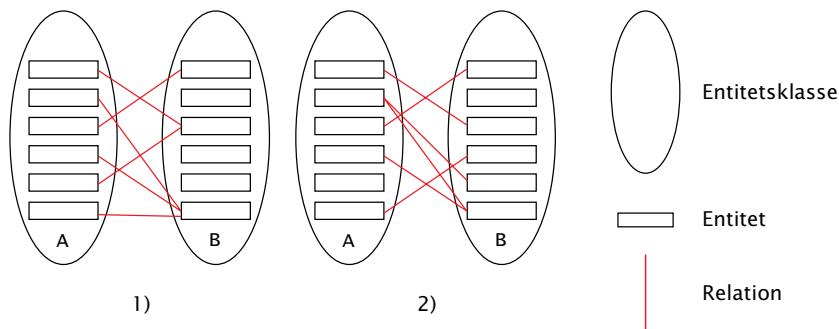


Fig. 7.111. 1) Entitetsklassen A deltager fuldstændigt i relationen. B deltager ikke fuldstændigt.
2) Hverken A eller B deltager fuldstændigt i relationen.

Hvis en entitetklasses deltagelse i en relation er fuldstændig, er det altså “obligatorisk” for entiteterne i den entitetsklasse at være *i en mindst en relation*.

EKSEMPEL**FULDSTÆNDIG OG IKKE-FULDSTÆNDIG DELTAGELSE I RELATION**

Entitetsklassen **Direktør** deltager fuldstændigt i relationen **Er direktør i** – der er ingen direktører, der ikke er direktør i en virksomhed, så enhver **Direktør**-entitet må have en **Er direktør i**-relation til en **Virksomhed**-entitet. **Band** deltager ikke nødvendigvis fuldstændigt i relationen **Har givet koncert på**: man kunne sagtens forestille sig et band, der ikke har givet nogen koncerter på noget spillested endnu.

EKSEMPEL**MULIG OG OBLIGATORISK DELTAGELSE I RELATION**

Vi husker 1-til-mange-relationen **Ejer** fra **Person** til **Cykel**.

En person ejer ikke nødvendigvis en cykel (hver person ejer 0, 1 eller flere cykler). Entitetsklassen **Person**'s deltagelse i relationen er derfor "frivillig" (ikke fuldstændig): Hver **Person**-entitet er relateret til 0, 1 eller flere **Cykel**-entiteter gennem **Ejer**-relationen.

En cykel ejes altid en person, så entitetsklassen **Cykel**'s deltagelse i relationen **Ejer** er "obligatorisk" (fuldstændig): Der kan ikke være en **Cykel**-entitet, som ikke er relateret til præcis en **Person**-entitet gennem **Ejer**-relationen.

"1" betyder altså "præcis 1" (og ikke "0 eller 1") i denne relation.

7.3.5. E/R-DIAGRAMMER

En af E/R-modellens store styrker er, at den kan udtrykkes grafisk som et E/R-diagram. E/R-diagrammet er præcist og hjælper med at visualisere E/R-modellen.

SYMBOL	EKSEMPEL	BETYDNING
Rektangel	Person	Entitetsklasse
Ellipse	navn	Attribut
Ellipse med understreget indhold	CPR-nummer	Primærnøgle eller del af primærnøgle for entitetsklasse
Dobbelt ellipse	sprog	Flerværdi-attribut (0, 1 eller flere værdier)
Diamant	Ejer	Relation
Linie	—————	Forbinde attributter med entitetsklasser og entitetsklasser med relationer

Linier fra relationer kan forsynes med symbolerne 1 og *, der signalerer relationens kardinalitet: 1 betyder "0 eller 1", og * betyder mange (og mange betyder jo 0, 1 eller flere).

KARDINALITET	EKSEMPEL	I SYMBOLER
1-til-1	Et gymnasium har 1 rektor. En rektor er rektor for 1 gymnasium.	<pre> graph LR Rektor[Rektor] -- "1" --> ErRektorFor{er rektor for} ErRektorFor -- "1" --> Gymnasium[Gymnasium] </pre>
1-til-mange	En person ejer 0, 1 eller flere cykler. En cykel ejes af netop 1 person.	<pre> graph LR Person[Person] -- "1" --> Ejjer{Ejer} Ejjer -- "*" --> Cykel[cykel] </pre>
Mange-til-mange	Et band har givet koncert på 0, 1 eller flere spillesteder. Et spillested har haft koncerter af 0, 1 eller flere bands.	<pre> graph LR Band[Band] -- "*" --> HarGivetKoncertPaa{har givet koncert på} HarGivetKoncertPaa -- "*" --> Spillested[Spillested] </pre>

ADVARSEL

PLACERING AF KARDINALITETSSYMBOLER

Der er ikke total enighed i datalogiens verden omkring placeringen af symbolerne 1 og * i en 1-til-mange-relation. Nogle noterer relationen som vi gør, mens andre placerer 1 og * "omvendt" – med sådan en notation ville fx relationen **Ejer** have 1 ud for **Cykel** og * ud for **Person**.

Vi kan præcisere kardinaliteterne på en relation til præcise antal. Vi kan også angive for de deltagende entitetsklasser, om deltagelsen i relationen er fuldstændig (obligatorisk at deltage i) eller ikke-fuldstændig (frivillig at deltage i). For eksempel er relationen **Er barn af** en mange-til-2-relation fra **Barn** til **Voksen**, hvori **Barn** deltager fuldstændigt: hvert barn har præcis 2 forældre, men **Voksen** deltager ikke-fuldstændigt: en voksen person kan have 0, 1 eller flere børn.

KARDINALITET	EKSEMPEL	I SYMBOLER
Mange-til-2	Et barn er barn af præcis to voksne. En voksen kan have 0, 1 eller flere børn.	<pre> graph LR Barn[Barn] -- "*" --- ErBarnAf{Er barn af} voksen[voksen] -- "2" --- ErBarnAf </pre>

Ordet **skal** under en relation betyder fuldstændig deltagelse, mens ordet **kan** betyder ikke-fuldstændig deltagelse. Man kan ”læse relationen op”; fra venstre mod højre oplæses fx ”Et barn skal være barn af to voksne”. Et revolutionerende udsagn, der altså nu også findes som E/R-diagram!

WWW

UML

På [WWW](#) vil du finde henvisninger til læsning om UML (*unified modelling language*). UML er en grafisk udtryksform, der kan bruges til at udtrykke både E/R-modellering og objekt-orienteret modellering. UML er en bredt accepteret standard til formidling af bl.a. databasedesign. De gennemgåede symboler svarer til UML-symboler (på nær brugen af ordene **kan** og **skal**).

7.3.6. E/R-MODELLERING

Skal man lave en E/R-model som baggrund for en database, kan følgende ”opskrift” benyttes:

1. **Brainstorm:** Skriv alle de ord op, der har at gøre med det, modellen handler om.
2. **Sorter (ud fra brugerkrav):** Find ud af, hvem databasens brugere er, og hvad de skal bruge databasen til (hvilke data, de er interesserede i at tilgå eller manipulere). De ord fra ordlisten, der på denne baggrund tydeligvis er irrelevante, kan fjernes fra listen.
3. **Kategoriser:** Kig i den reviderede ordliste. Find ud af, hvilke ord der definerer henholdsvis entitetsklasser, relationer og attributter. Attributværdier bruges ikke til at fastlægge selve E/R-modellen og skal udelades.
4. **Saml og minimaliser:** Tegn E/R-diagrammets grundstamme: Saml de fundne entiteter ved hjælp af relationer og påfør deres attributter. Hvis der er overflødige entitetsklasser, relationer eller attributter, skal de fjernes.
5. **Kardinaliteter:** Alle relationer påføres kardinaliteter.
6. **Primærnøgler:** Alle entitetsklasser skal have en primærnøgle. Hvis der er flere kandidatnøgler, så vælg den mindst indviklede eller den mest meningsfulde som primærnøgle.

Trin 3 er sværest. Det er ikke altid let at beslutte, om et givent koncept fra den virkelige verden skal modelleres som en entitetsklasse, en relation eller en attribut. Følgende tomfingrerregler kan bruges (men ikke med hovedet under armen!):

1. En entitetsklasse "svarer ofte til navneord".
2. En relation "svarer ofte til udsagnsord".
3. En attribut knytter sig til entitetsklasse uden selv at være en entitetsklasse (er tit en evne ved, egenskab ved eller oplysning om en entitet).

7.3.7. EKSEMPEL: E/R-MODELLERING

Vi vil lave en database til Datahøj Gymnasium, som kan fortælle, hvilke lærere der underviser hvilke elever på hvilke hold.

Herudover vil vi lagre information om de enkelte elevers navn og årgang, lærernes fag og initialer (tre bogstaver, fx JEP, ANK eller GIS) samt de enkelte holds fag og holdenes holdkode (H-xxx, hvor xxx er trecifret tal, fx H-190, H-334 eller H-331). Forskellige hold har forskellige holdkoder. Hvert hold undervises kun af en lærer.

Vi starter med at lave en E/R-model med opskriften fra afsnit 7.3.6.

1) BRAINSTORM:

Datahøj Gymnasium, lærer, lærers fag, lærers initialer, JEP, ANK, GIS, lærer underviser i fag, lærer underviser på hold, lærer underviser elev, hold, fag på hold, elev, navn, årgang, går på hold, tidspunkter for undervisning, holdkode, H-190, H-334, H-331, lokale, tysk, engelsk, fransk, datalogi.

2) SORTERING PÅ BAGGRUND AF BRUGERKRAV:

Det indgår ikke i brugerkravene, at man skal kunne finde lokaler eller tidspunkter for holdundervisning, så vi udelader "tidspunkter for undervisning" og "lokale". Da databasen hører til Datahøj Gymnasium, behøver selve gymnasiet ikke være en del af databasen, så vi udelader "Datahøj Gymnasium":

Datahøj Gymnasium, lærer, lærers fag, lærers initialer, JEP, ANK, GIS, lærer underviser i fag, lærer underviser på hold, lærer underviser elev, hold, fag på hold, elev, navn, årgang, går på hold, ~~tidspunkter for undervisning~~, holdkode, H-190, H-334, H-331, ~~lokale~~, tysk, engelsk, fransk, datalogi.

3) KATEGORISERING

Ved at bruge tommelfingerreglerne beslutter vi følgende:

- ◆ entitetsklasser: Lærer, Hold, Elev
- ◆ relationer: Underviser på hold, Underviser elev, Går på hold
- ◆ attributter: fag og initialer (Lærer-attributter), fag (Hold-attribut), navn og årgang (Elev-attributter), holdkode (Hold-attribut)

Ved at lade fag være en Lærer-attribut, kan vi droppe "Underviser i fag" – så den stryger vi fra listen. Vi bemærker, fag er en flerværdi-attribut, fordi en lærer kan undervise i flere forskellige fag. Alle andre attributter er en-værdi-attributter. navn er den eneste sammensatte attribut (sammensat af fornavn, mellemnavne og efternavn).

Vi kan se, at de resterende ord i listen, MPL, H-441, tysk, engelsk, fransk og datalogi, er attributværdier (hørende til attributterne initialer, holdkode og fag (på et hold)). Dem udelader vi af modellen, så den endelige ordliste ser ud som følger:

Østjysk Gymnasium, lærer, lærers fag, lærers initialer, JEP, ANK, GIS, lærer underviser i fag, lærer underviser på hold, lærer underviser elev, hold, fag på hold, elev, navn, årgang, går på hold, tidspunkter for undervisning, holdkode, H-190, H-334, H-331, lokale, tysk, engelsk, fransk, datalogi.

4) E/R-DIAGRAM OG MINIMALISERING:

Vi tegner E/R-diagrammet:

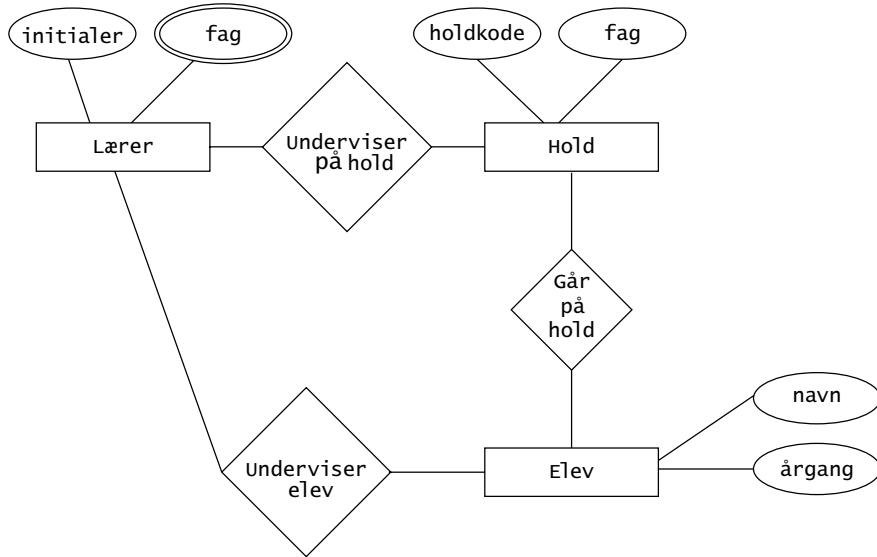


Fig. 7.112. (Foreløbigt) E/R-diagram for undervisningen på Datahøj Gymnasium.

Vi opdager ud fra diagrammet, at der er en relation, der kan undværes – nemlig relationen **Underviser elev**. Vil vi vide, om en bestemt lærer underviser en bestemt elev, kan vi jo finde ud af, hvilke hold læreren underviser, med relationen **Underviser på hold**, og bag-efter med relationen **Går på hold** finde ud af, hvilke elever der går på de pågældende hold. Relationen er derfor overflødig – vi fjerner den.

Figur 7.113 viser et tilrettet diagram.

5) KARDINALITETER

Der er to relationer tilbage, og dem skal vi bestemme kardinaliteter for.

- **Underviser på hold** er en 1-mange-relation fra **Lærer** til **Hold**, fordi en lærer kan undervise flere hold, men et hold kun undervises af 1 lærer.
- **Går på hold** er en mange-mange-relation fra **Elev** til **Hold**, fordi hver elev kan gå på flere hold, og hvert hold kan have flere tilmeldte elever.

På Figur 7.113 er kardinaliteterne indtegnet.

6) PRIMÆRNØGLER

Lærer har primærnøglen **initialer**.

Entitetsklassen **Hold** har primærnøglen **holdkode**. Samlingen {**holdkode**, **fag**} kan også bruges til at slå **Hold**-entiteter op med – men da **holdkode** er simple, er {**holdkode**, **fag**} ikke minimal, så den er ikke en kandidatnøgle. Attributten **fag** alene er minimal, men ikke en kandidatnøgle, da to forskellige hold godt kan have samme fag.

Elev har ikke nogen primærnøgle, fordi der godt kunne være to elever med samme navn på samme årgang. Vi løser problemet ved at tilføje en attribut **elev-ID** til **Elev**, der kan fungere som primærnøgle:

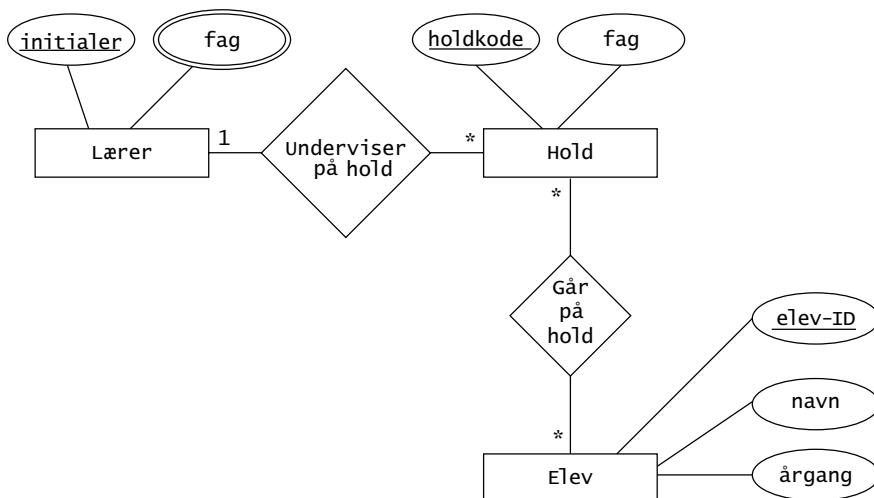


Fig. 7.113. (Færdigt) E/R-diagram for undervisning på Datahøj Gymnasium.

Hvilke entitetsklasser deltager fuldstændigt i hvilke relationer?

7.4. RELATIONELLE DATABASER

Udfra en E/R-model kan man bygge en *relationel database*.

ORDFORKLARING

RELATIONEL DATABASE

Database, der er opbygget som en samling tabeller. Tabellerne er forbundet via eksplikitive relationer. Data i tabelindgangene kan tilgås og manipuleres ved hjælp af forespørgsler.

Hver entitetskasse i E/R-modellen giver anledning til en tabel, hvis søjler er attributter for entitetskassen. Hver række i tabellen svarer til en entitet.

Generelt opskriver vi strukturen af tabeller på formen

<Tabelnavn>(<liste af primærnøgle-attribut(er)>, <liste af attributter>)

Hver relation giver også anledning til en tabel.

7.5. FRA E/R-MODEL TIL TABELLER I RELATIONEL DATABASE

Vi illustrerer nu overgangen fra E/R-model til relationel database via eksemplet med Dataløj Gymnasium fra afsnit 7.3.7.

Vi husker `Elev`-entitetsklassen med attributter `elev-ID` (primærnøgle, derfor understregningen), navn og årgang. Den giver anledning til tabellen med struktur

`Elev(elev-ID, navn, årgang)`

Et udsnit af indholdet i `Elev`-tabellen kunne fx se ud som følger:

Elev		
elev-ID	navn	årgang
008	Herman Gnam Henriksen	3g
071	Sophie Hansen	2g
155	Morten Møller Jensen	2g
189	Christel de La Cour	1g
241	Mette Hannah Louise Juul	1g

Fig. 7.114. Tabel for entitetsklassen Elev.

Her udgør rækken

189	Christel de La Cour	1g
-----	---------------------	----

en enkelt `Elev`-entitet. Vi skriver også enkelte entiteter på såkaldt *tupel-form*:

(189, Christel de La Cour, 1g)

Tabelstrukturen for `Hold`-entitetsklassen bliver

`Hold(holdkode, fag)`

Et udsnit af Hold-tabellen:

Hold	
<u>holdkode</u>	fag
H-331	Fransk
H-334	Latin
H-190	Datalogi
H-641	Fransk
H-082	Historie

Rækken

H-334	Latin
-------	-------

svarer til den Hold-entitet, der har holdkode H-334, og hvor der undervises i latin – på tupelform: (H-334, Latin).

Et udsnit af Lærer-tabellen med struktur Lærer(initialer, fag):

Lærer	
<u>initialer</u>	fag
ANK	Fransk Engelsk Latin
GIS	Oldtidskundskab Billedkunst Idræt
JEP	Datalogi Matematik
KNI	Tysk Fysik
MAR	Musik Dansk
OLP	Historie Samfundsfag Matematik
STT	Fransk Spansk

Fig. 7.115. Tabel for entitetsklassen Lærer.

Det var tabellerne for entitetsklasserne, men også relationerne skal med i databasen – hver bliver til en tabel. Søjlerne i tabellen for en relation er primærnøglerne for de entitetsklasser, der deltager i relationen. Rækkerne er de enkelte relationer mellem entiteter. Fx får relationen **Går på hold** strukturen

Går på hold (elev-ID, holdkode)

Et udsnit af **Går på hold**-tabellen:

Går på hold	
<u>elev-ID</u>	<u>holdkode</u>
008	H-331
008	H-082
071	H-190
155	H-190
189	H-641
241	H-641

Rækken

189	H-641
-----	-------

svarer til relationen **Går på hold** mellem **Elev**-entiteten med **elev-ID** 189 og **Hold**-entiteten med **holdkode** H-641. På tupelform: (189, H-641). Relationens tilstedeværelse i tabellen tolkes i den virkelige verden som “Eleven med elev-ID 189 går på holdet med holdkode H-641”.

Vi danner også tabellen

Underviser på hold (initialer, holdkode)

for relationen **Underviser på hold** mellem entitetsklasserne **Lærer** og **Hold**:

Underviser på hold	
<u>holdkode</u>	<u>initialer</u>
H-331	ANK
H-334	ANK
H-190	JEP
H-641	STT
H-082	OLP

Rækken

H-334	ANK
-------	-----

svarer til relationen `Underviser på hold` mellem `Lærer`-entiteten med initialer ANK og `Hold`-entiteten med holdkode H-334 (og på tupel-form bliver det ... ?).

Vi opdager nu, at tabellen for relationen `Underviser på hold` faktisk kan smeltes sammen med tabellen for entitetsklassen `Hold`: Ud for hvert hold kan vi angive den lærer, der underviser på holdet. På den måde sparer vi tabellen `underviser på hold`. Den udvidede `Hold`-tabel ser ud som følger:

Hold		
holdkode	fag	initialer
H-331	Fransk	ANK
H-334	Latin	ANK
H-190	Datalogi	JEP
H-641	Fransk	STT
H-082	Historie	OLP

Vi opsummerer, hvordan der oversættes fra E/R-model til en samling af tabeller:

- ◆ Hver entitetskasse bliver til tabel af samme navn.
- ◆ Attributterne bliver til søjler i tabellen.
- ◆ De enkelte entiteter i en entitetskasse fremgår som en række i den tilsvarende tabel.
- ◆ Relationer bliver til tabeller.

7.6. INTEGRITET AF DATABASE

En databases indhold skal *altid* repræsentere data, der ikke er i modstrid med sig selv eller databasens design. Det kaldes *integritet*. Hvis databasens integritet ødelægges, er databasens indhold ikke i overensstemmelse med den virkelighed, den forsøger at modellere. Og så kan databasen ikke bruges til noget.

En databases integritet kan skades, hvis

- ◆ forkert data indsættes,
- ◆ data ændres forkert,
- ◆ forkert data slettes.

Som databasedesignere skal vi sikre databasens integritet. Det er derfor nødvendigt at kende en række områder, hvor integritet kan blive et problem:

INTEGRITETSTYPE HAR AT GØRE MED	FORKLARING	EKSEMPEL
Konsistens	Hvis en oplysning er lagret to steder i databasen, skal de stemme overens.	Er oplysningen "Lærer ANK har fagene fransk, engelsk og latin" lagret et sted, må der ikke et andet sted være lagret "Lærer ANK har fagene fransk og fysik".
Kandidatnøgler	Hvis en samling attributter udgør en kandidatnøgle ifølge databasedesignet, skal de altid udgøre en kandidatnøgle.	Attributten initialer er primærnøgle for Lærer . Der må derfor ikke indsættes en Lærer -entitet i tabellen med samme "initialer" som en eksisterende.
Fremmednøgler	Forbindelsen mellem tabeller, der refererer til hinanden, skal altid være korrekt.	Se afsnit 7.8.1. (Overvej hvad der mon sker med de hold, en lærer underviser, hvis læreren slettes fra databasen?)
Kardinaliteter	En relations kardinalitet skal altid være overholdt.	Der må ikke være angivet to lærere som underviser på det samme hold.
Attributter	Det kan bestemmes, at en attribut kun kan antage bestemte værdier.	Elev -attributten årgang må kun antage værdier fra mængden {1g, 2g, 3g}, ikke fx "6g" eller "1h".
Entiteter	Det kan bestemmes at en entitet skal overholde et bestemt format.	Man kunne fx vedtage (men det ville være mærkeligt) at Elev -entiteter ikke må hedde "Hans" eller "Lars".
Andet	En regel om databasens indhold, der ikke er af ovenstående typer.	Hvis en lærer underviser på et hold, ønsker vi at læreren faktisk er i stand til at undervise i det fag, holdet har. ANK, der underviser i fransk, engelsk og latin, må med andre ord ikke undervise hold H-190, der har datalogi.

Fig. 7.116. Integritetstyper.

Der findes heldigvis standardmetoder til at sikre integritet. Første skridt på vejen er *normalformer*.

7.7. NORMALFORMER

Relationelle databaser har en række såkaldte *normalformer*, hvoraf vi kommer ind på første, anden og tredje. En database i tredje normalform (den “flotteste” af de tre første) vil i de fleste tilfælde undgå at data gentages – så der ikke kan opstå *inkonsistens*.

Vi genkalder os gymnasie-databasen med følgende tabeller:

```
Elev(elev-ID, navn, årgang)
Hold(holdkode, fag, initialer)
Lærer(initialer, fag)
Underviser på hold (initialer, holdkode)
Går på hold (elev-ID, holdkode)
```

Vi vil bringe databasen i tredje normalform. Første skridt er at bringe databasen i *første normalform*.

7.7.1. FØRSTE NORMALFORM (1NF)

Man kan ikke slå op i tabeller uden primærnøgler. Derfor kræver fornuftige database-designere primærnøgler på alle tabeller. Sammensatte attributter giver risiko for ekstra arbejde, når databasen skal bruges:

- ◆ det er sværere at få fat i en del af en attribut end hele attributten,
- ◆ det er sværere at ændre en del af en attribut end hele attributten.

Svært arbejde kan resultere i fejl – så lad os undgå problemet i stedet:

ORDFORKLARING

FØRSTE NORMALFORM (1NF)

En tabel er i *første normalform* (forkortet: 1NF), hvis

- ◆ attributter i tabellen er simple en-værdi-attributter
- ◆ tabellen har en primærnøgle

En relationel database er i *første normalform (1NF)*, hvis alle tabeller er i 1NF.

Vi vil nu bringe gymnasiedatabasen på 1NF. Da alle tabeller har primærnøgler, er det kun det første krav af de to 1NF-krav, der kan være et problem.

Entitetsklassen `Elev` overholder ikke kravet til 1NF, fordi `Elev`-attributten `navn` er sammensat. En løsning er at opsplitte den sammensatte attribut `navn` op i de simple attributter `fornavn`, `melleminavn` og `efternavn`:

Elev				
elev-ID	fornavn	mellemnavne	efternavn	årgang
008	Herman	Gnam	Henriksen	3g
071	Sophie	NULL	Hansen	2g
155	Morten	Møller	Jensen	2g
189	Christel	NULL	de La Cour	1g
241	Mette	Hannah Louise	Juul	1g

Bemærk, at vi udfylder med **NULL**-værdier for de elever, der enten ikke har mellemnavne eller ikke har oplyst dem.

Selvom vi nu har bragt **Elev**-tabellen i 1NF, er hele databasen endnu ikke i 1NF, fordi **Lærer**-tabellen indeholder flerværdi-attributten **fag**.

Flerværdi-attributter giver risiko for:

- En database, det er svært at hente, indsætte, slette eller ændre i (hvordan finder vi de lærere, der underviser i fransk?)
- Forkert eller indviklet design.

Derfor kræver 1NF, at alle attributter er en-værdi-attributter. Vi vil gerne have databasen på 1NF, og da en lærer altid har 2 eller 3 fag, laver vi følgende opsplitning:

Lærer			
initialer	fag1	fag2	fag3
ANK	Fransk	Engelsk	Latin
GIS	Oldtiduskundskab	Billedkunst	Idræt
JEP	Datalogi	Matematik	NULL
KNI	Tysk	Fysik	NULL
MAR	Musik	Dansk	NULL
OLP	Historie	Samfundsfag	Matematik
STT	Fransk	Spansk	NULL

Nu er **Lærer**-tabellen på 1NF.

Da ingen andre tabeller har sammensatte attributter eller flerværdi-attributter, er databasen på 1NF.

ADVARSEL

1NF OG “STÆRK 1NF”

Blandt dataloger er der ikke universel enighed om, hvad 1NF indebærer. Nogle mener, at det hører med til 1NF, at ingen attributter må have værdien **NULL**. For at kende forskel, kalder vi det for ”Stærk 1NF”. Hverken den nye Elev-tabel eller den nye Lærertabel på ”Stærk 1NF”. På **WWW** kan du læse mere om 1NF og Stærk 1NF.

7.7.2. ANDEN NORMALFORM (2NF)

ORDFORKLARING

ANDEN NORMALFORM (2NF)

En tabel er i *anden normalform* (forkortet: 2NF), hvis den er i 1NF, og der desuden gælder a, b eller c:

- tabellen indeholder ikke en sammensat kandidatnøgle (en nøgle bestående af 2 eller flere attributter),
- tabellen indeholder 1 sammensat kandidatnøgle, og alle andre attributter i tabellen er afhængige af hele den sammensatte kandidatnøgle,
- tabellen indeholder flere sammensatte kandidatnøgler, der hver opfylder b.

En relationel database er i 2NF, hvis alle dens tabeller er i 2NF.

Vores database er i øjeblikket i 2NF, fordi ingen tabeller har en sammensat kandidatnøgle.

For at forstå betingelsen b, og for at illustrere overgangen fra 1NF til 2NF, udvider vi nu vores E/R-design for Datahøj Gymnasium til også at indeholde information omkring møderne for gymnasiets 4 elevudvalg: Festudvalget, Skolebladet, Elevrådet og Skolekommedieudvalget. Festudvalget, Skolebladet og Elevrådet mødes altid i lokale L432 ved siden af kantinen, mens Skolekommedieudvalget altid mødes i gymnastiksalen.

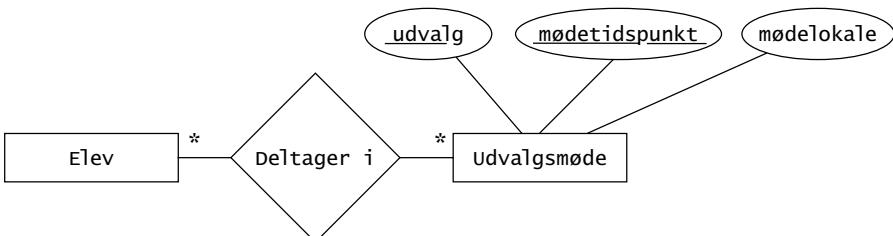


Fig. 7.117. Tilføjelse til E/R-modellen for Datahøj Gymnasium: Entitetsklassen Udvalgsmøde og relationen Deltager i.

Da to forskellige udvalg kan holde møde samtidigt, er mødetidspunkt ikke en kandidatnøgle. Det samme udvalg kan holde møde på forskellige tidspunkter, så udvalg er heller ikke en kandidatnøgle. Da 3 af udvalgene holder møde i samme lokale, er mødelokale heller ikke en kandidatnøgle. Men tilsammen fastlægger mødetidspunkt og udvalg entydigt et udvalgsmøde, så {mødetidspunkt, udvalg} er en (sammensat!) kandidatnøgle for Udvalgsmøde (- og faktisk den eneste mulige primærnøgle).

Databasen udvides med følgende to tabeller:

Udvalgsmøde (mødetidspunkt, udvalg, mødelokale)

Deltager i (elev-ID, mødetidspunkt, udvalg)

Nu er databasen imidlertid ikke længere i 2NF: Attributten mødelokale for et udvalgsmøde afhænger kun af udvalg, og ikke af hele primærnøglen {mødetidspunkt, udvalg}.

Det giver risiko for

- *inkonsistens i databasen*: hvis et nyt udvalgsmøde indsættes i databasen skal mødelokale indskrives igen, og skriver vi forkert, foreslår databasen to forskellige mødelokaler til samme udvalg!
- *manglende information i databasen*: hvis Festudvalget ikke har nogle planlagte møder, så giver databasen ikke mulighed for at lagre informationen om, at Festudvalget mødes i lokale L432.
- *pladsspild*: hvis Elevrådet har fire planlagte møder, er oplysningen om at elevrådet mødes i lokale L432 lagret fire gange!

Inden vi redder databasen ud af denne suppedas, forklarer vi begrebet *afhængighed* (der indgår i 2NF, punkt b):

ORDFORKLARING

AFHÆNGIGHED

En attribut a i en tabel *afhænger* af attributterne {b₁, ..., b_n}, hvis en bestemt værdi af attributterne {b₁, ..., b_n} altid giver en bestemt værdi af attributten a.

Afhængighed er et begreb, der generaliserer begrebet kandidatnøgle. Enhver attribut er afhængig af en kandidatnøgle, men en attribut kan godt være afhængig af en samling attributter, der ikke er en kandidatnøgle.

EKSEMPEL

AFHÆNGIGHED

Attributten **mødelokale** for et udvalgsmøde afhænger af **udvalg**, fordi et bestemt udvalg altid har samme mødelokale.

Attributten **mødetidspunkt** er afhængig af **mødetidspunkt**.

Lærer-attributten **initialer** er ikke afhængig af **fag**.

Er **Hold**-attributten **fag** afhængig af holdkode?

Vi vender nu tilbage til den udvidede gymnasie-database, der i øjeblikket ikke er i 2NF. Vi bringer den i 2NF ved at opsplitte tabellen udvalgsmøde i to tabeller:

Udvalgsmødetidspunkt(**mødetidspunkt**, **udvalg**)

Udvalgsmødelokale(**udvalg**, **mødelokale**)

Databasen er nu i 2NF! (Hvorfor?)

7.7.3. TREDJE NORMALFORM (3NF)

ORDFORKLARING

TREDJE NORMALFORM (3NF)

En tabel er i *tredje normalform* (forkortet: 3NF), hvis

- tabellen er i 2NF,
- ingen af tabellens attributter afhænger indirekte af en kandidatnøgle.

En relationel database er i tredje normalform, hvis alle dens tabeller er i 3NF.

Vores database for Datahøj Gymnasium er allerede i 3NF (hvorfor?).

Vi kan illustrere 3NF-problemet med følgende tabel, **vinder**, over nogle vinderklubber gennem årene af de to europæiske fodboldturneringer Champions League og UEFA Cup:

Vinder			
<u>turnering</u>	<u>sæson</u>	<u>k1ub</u>	<u>hjemland</u>
UEFA Cup	2005/2006	Sevilla	Spanien
UEFA Cup	2000/2001	Liverpool	England
Champions League	1994/1995	AFC Ajax	Holland
UEFA Cup	1999/2000	Galatasaray	Tyrkiet
Champions League	2001/2002	Real Madrid C.F.	Spanien
Champions League	2000/2001	FC Bayern München	Tyskland
UEFA Cup	2006/2007	Sevilla	Spanien

Tabellen er i 2NF (hvorfor?). Men tabellen er ikke i 3NF, fordi attributten hjemland, der til-syneladende kun afhænger af attributten klub, indirekte afhænger af primærnøglen {turnering, sæson} – værdien af klub afhænger nemlig af {turnering, sæson}. Risikoen:

- ◆ *inkonsistens*: en letsindig databasebruger kan indsætte rækken

(Champions League, 1999/2000, Real Madrid C.F., Sverige)

i tabellen. Klubben Real Madrid C.F. er nu parret med to forskellige hjemlande i tabellen, og hvilket land er så det rigtige?!

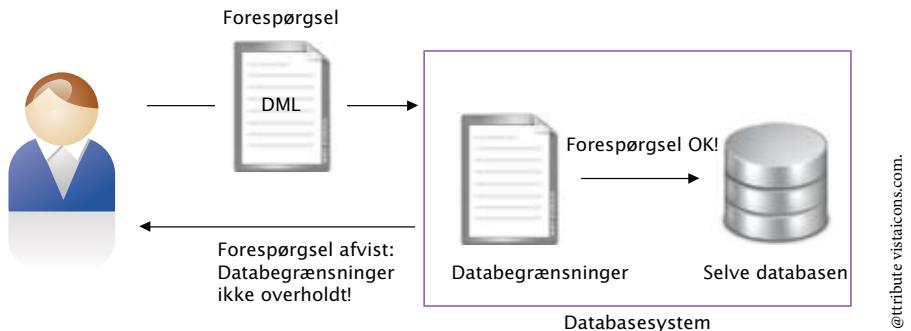
Løsningen er at opsplitte i følgende to tabeller, der begge er på 3NF (hvorfor?):

Vinder			Klubhjemland	
Turnering	sæson	klub	klub	hjemland
UEFA Cup	2005/2006	Sevilla	Sevilla	Spanien
UEFA Cup	2000/2001	Liverpool	Liverpool	England
Champions League	1994/1995	AFC Ajax	AFC Ajax	Holland
UEFA Cup	1999/2000	Galatasaray	Galatasaray	Tyrkiet
Champions League	2001/2002	Real Madrid C.F.	Real Madrid C.F.	Spanien
Champions League	2000/2001	FC Bayern München	FC Bayern München	Tyskland
UEFA Cup	2006/2007	Sevilla		

Fig. 7.118. Den oprindelige Vinder-tabel opsplittes i Vinder og Klubhjemland for at opnå 3NF.

7.8. DATABEGRÆNSNINGER

Udover normalformer kan databasens integritet sikres gennem såkaldte *databegrænsninger*. Databegrænsninger er et regelsæt, der fortæller hvilke dataværdier, vi vil acceptere i databasen. Databasesystemet oprettes med alle begrænsningerne angivet fra start. Hver gang en bruger vil indsætte, slette eller ændre data i databasen, konsulteres begrænsningerne – overtrædes begrænsningerne, afvises forsøget på at opdatere databasen:



Attribute vistaicons.com.

Fig. 7.119. Databegrænsninger medvirker til at sikre database-integritet.

Det letteste eksempel på en databegrænsning er begrebet *primærnøgle*. I tabellen `Elev(elev-ID, navn, årgang)`, der indeholder `Elev`-entiteten (189, Christel de La Cour, 1g), er det ikke tilladt at indsætte en `Elev`-entitet med samme `elev-ID` som Christel, fordi `elev-ID` er primærnøgle. Fx må vi ikke indsætte (189, Søren Justesen, 1g).

WWW

ANGIVELSE AF BEGRÆNSNINGER MED SQL

Syntaksen for begrænsninger varierer alt efter databasesprog. På **WWW** kan du se hvordan man gør med sproget SQL.

7.8.1. FREMMEDNØGLER

Vi har allerede set, at primærnøgler er et eksempel på en databegrænsning. Man kan også have *fremmednøgler*.

ORDFORKLARING

FREMMEDNØGLE

En attribut, eller en samling af attributter, der er primærnøgle (eller kandidatnøgle) for en anden tabel.

EKSEMPEL

FREMMEDNØGLE

Husk følgende tabeller fra gymnasie-databasen:

`Lærer(initialer, fag1, fag2, fag3)`

`Hold(holdkode, fag, initialer)`

Attributten `initialer` i tabellen `Hold` er en fremmednøgle til tabellen `Lærer`. Det markerer vi i tabelstrukturen på følgende måde:

`Hold(holdkode, fag, initialer [:Lærer])`

EKSEMPEL

FREMMEDNØGLE

Husk tabellerne `Vinder` og `Klubhjemland` fra Figur 7.118 på side 186. `klub`-attributten i tabellen `Vinder` er fremmednøgle for tabellen `Klubhjemland`:

`Vinder(turnering, sæson, klub [:Klubhjemland])`

Databasesystemet opretholder følgende begrænsning relateret til primærnøgler:

Forbyd indsættelse af en række, der kopierer en eksisterende primærnøgle.

Det vidste vi i forvejen. Databasesystemet håndhæver nogle lignende begrænsninger, der har at gøre med fremmednøgler. Det er databasedesigneren, der bestemmer, præcis hvordan disse begrænsninger er udformet – vi ser her på et par eksempler.

EKSEMPEL

UKENDT INDSÆTTELSE

Forsøger man at indsætte

(Champions League, 1998/1999, Forenede Databaseprogrammørers Boldklub)

i tabellen `Vinder(turnering, sæson, klub[:Klubhjemland])`, støder databasesystemet på et problem: der ikke er et fremmednøgle-match på "Forenede Databaseprogrammørers Boldklub" i tabellen `Klubhjemland(klub, hjemland)`. Vi får altså en klub ind i systemet, hvis hjemland vi ikke kender. Er det OK? Det er op til databasedesigneren at beslutte. Hvis ja, er der ingen problemer. Hvis nej, må indsættelsen afvises.

EKSEMPEL

SLETNING AF RÆKKE, DER BLIVER REFERERET AF FREMMEDNØGLE

Læreren med initialer STT bliver pensioneret. Vi ønsker at slette **Lærer**-entiteten
(STT, Fransk, Spansk, **NULL**)

fra tabellen **Lærer**. Tabellen **Hold** har en fremmednøgle (attributten **initialer**) til ta-
bellen **Lærer** og der er match på attribut-værdien STT: Tabellen **Hold** indeholder **Hold**-
entiteten

(H-641, Fransk, STT).

Databasedesignet skal afgøre, hvad der skal ske med franskholdet nu. Kan et hold
stå uden lærer, sådan at STT kan slettes – vi kunne jo ændre **Hold**-entiteten til (H-641,
Fransk, **NULL**)? Eller skal sletningen af STT fra databasen afvises, indtil en ny lærer er
sat på holdet?

EKSEMPEL

OPDATERING AF RÆKKE, DER BLIVER REFERERET AF FREMMEDNØGLE

Hvis læreren STT ikke går på pension, men bare skifter initialer til STK (som følge af et
giftermål, men det er anden historie), kan vi lave en opdatering, der “breder sig som
ringe i vandet” gennem databasen:

1. Opdater **Lærer**-entiteten til (STK, Fransk, Spansk, **NULL**)
2. Opdater alle fremmednøgle-forekomster af STT i databasen til STK, i dette til-
fælde opdateres **Hold**-entiteten til (H-641, Fransk, STK).

WWW

FREMMEDNØGLER

På **WWW** kan du læse om

- ◆ *hvilke fremmednøgle-begrænsninger, man kan angive som databasede-
signer,*
- ◆ *hvordan man angiver begrænsninger med fremmednøgler i database-
sproget SQL.*

7.8.2. DOMÆNE-BEGRÆNSNINGER OG TUPEL-BEGRÆNSNINGER

Domænebegrænsninger bruges til at præcisere en attributs værdimængde. Man kan fx ønske sig, at en attribut

- ◆ ikke må antage værdien **NULL**,
- ◆ skal antage talværdier i et bestemt interval, fx mellem 0 og 100,
- ◆ skal antage værdier fra en bestemt liste af værdier,
- ◆ kun må antage værdier, der afhænger af indhold fra andre tabeller,
- ◆ ...

Sådanne attributbegrænsninger angives, når tabellen oprettes.

EKSEMPEL

NULL IKKE TILLADT

Attributterne fornavn og efternavn for en **Elev**-entitet må ikke være **NULL**, da skolen skal kende elevens fornavn og efternavn.

EKSEMPEL

TALVÆRDIER I INTERVAL

En attribut **point** for entitetsklassen **Konkurrencedeltager** kunne udtrykke et pointtal mellem 0 og 100, som en entitet har opnået ved at skyde med flitsbue i en konkurrence.

EKSEMPEL

BESTEMTE VÆRDIER

Attributten **fag** for både **Lærer**- og for **Hold**-entitetsklassen antager værdier fra følgende liste af værdier:

{Spansk, Billedkunst, Fysik, Arabisk, Biologi, ... , Astronomi}

EKSEMPEL

TILLADTE VÆRDIER AFHÆNGER AF ANDEN TABEL

Tabellen **Vinder** fra Figur 7.118 er en liste af *europæiske* fodboldmestre. Vi må altså kun indsætte **Vinder**-entiteter med en **Klub**-attribut, der svarer til en klub med et europæisk **Hjemland**. Imidlertid fremgår oplysninger om klubbers hjemland ikke af tabellen **Vinder**, men af **Klubhjemland**. Så for at opretholde betingelsen laver vi en *tabelafhængig domænebegrænsning* for attributterne **Klub**. Nu forsøger vi at indsætte

(Champions League, 1998/1999, C.A. Boca Juniors)

i tabellen **Vinder**. Domænebegrænsningen aktiveres og slår op i **Klubhjemland**-tabellen med nøglen "C.A. Boca Juniors". **Klubhjemland** lister hjemlandene for alle europæiske klubber, men indeholder ikke klubber fra resten af verden. Da C.A. Boca Juniors er et argentisk hold, finder domænebegrænsningen ikke noget match i tabellen **Klubhjemland**. Derfor vil domænebegrænsningen forhindre vores forsøg på at indsætte C.A. Boca Juniors.

Udover domænebegrænsninger kan man lave begrænsninger på formatet af en tupel. Disse begrænsninger er som regel en sammenkobling af flere attributter og angives også, når tabellen oprettes.

EKSEMPEL

TEMPERATUR I JANUAR 1985

En interessant(?) tabel over temperaturer i verdens lande i januar 1985 modellerer verdens lande som **Land**-entiteter:

Land (landenavn, laveste temperatur januar 1985, højeste temperatur januar 1985)

For at temperatur-tabellen skal give mening, ønsker vi betingelsen

Laveste temperatur i januar 1985 ≤ højeste temperatur i januar 1985

opfyldt for alle lande. Tabellen skal altså afvise indsættelse af fx rækken

(Afghanistan, 11, -4)

7.8.3. EKSPPLICITTE BEGRÆNSNINGER

Vi har set på nøgle-, attribut- og tupelbegrænsninger. Faktisk man kan lave lige de regler for datas udseende i en database, man har lyst til. Enhver begrænsning, der ikke kan angives som nøgle-, attribut- eller tupel-begrænsning kan angives som en *eksplicit begrænsning*. Sådanne eksplisitte begrænsninger konsulteres hver gang databasen forsøges ændret med en ændring, der vedrører tabeller med den eksplisitte begrænsning. Hvis den eksplisitte begrænsning overtrædes, afvises database-ændringen.

EKSEMPEL

EKSPPLICIT BEGRÆNSNING

I Datahøj Gymnasium-modellen kan vi fx indføre følgende (mere eller mindre velovervejede) eksplisitte begrænsninger

- Der må kun gå elever fra samme årgang på et hold.
- Ingen elever må gå på et fysikholt.
- Alle elever skal gå på et datalogihold .
- Læreren med initialer ANK må ikke undervise et hold i engelsk, hvis der er en 1g-elev på holdet, der hedder Markus, Andreas eller Sophie.

Jo mere indviklet begrænsning, desto længere tid tager det at undersøge, om begrænsningen er overholdt. Som databasedesigner bør man derfor sigte mod at designe en database med så få eksplisitte begrænsninger som muligt.

7.9. OPSUMMERING: SIKRING AF DATABASESENS INTEGRITET

Vi opsummerer her, hvordan de forskellige integritetstyper i en database sikres:

INTEGRITETSTYPE	SIKRING AF INTEGRITETSTYPE VIA BEGRÆNSNINGER
Konsistens	<i>Kombination af:</i> Normalformer Nøgleangivelse Eksplisit begrænsning
Kandidatnøgler	
Fremmednøgler	
Kardinaliteter	
Attributter	Domænebegrænsning
Entiteter	Tupel-begrænsning
Andet	Eksplisit begrænsning

Fig. 7.120. Sikring af databaseintegritet.

7.10. BACKUP OG TILGÆNGELIGHED

En database er ikke meget værd, hvis dens data bliver slettet eller ændret til forkert eller meningsløst data. Derfor skal en database have fornuftige *backup-mekanismer* til situationer, hvor uheldet er ude. Typisk er en database bygget som en samling af flere forskellige computere, der tilsammen huser al data i flere eksemplarer (backup-kopier). En særskilt backup-system sørger for, at opdateringer i *master-data* også foretages i backup-data.

En database med gode backupmekanismer, der desuden er velbeskyttet mod spionage, tyveri, naturkatastrofer, ruminvasion mv., er *stadic værdiløs*, hvis dens brugere ikke kan få adgang til den. Derfor sigter man som database-ansvarlig mod at højne databasens *tilgængelighed*, dvs.

- ◆ databasens *oppe-tid*: databasen skal kunne kontaktes så tit som muligt, helst altid,
- ◆ databasens *svar-tid*, også under spidsbelastninger: selvom der er mange samtidige brugere, skal det ikke gå langsomt at kommunikere med databasen.

Teknikken *load balancing* fordeler mange samtidige brugeres databaseforespørgsler fornuftigt på de computere, der huser databasesystemet. På den måde oplever alle brugere i gennemsnit mindst mulig belastning.

7.11. NØGLEORD

- ◆ Database
- ◆ Manipulation af database: Læse, indsætte, opdatere, slette
- ◆ E/R-model
- ◆ Entitet
- ◆ Entitetsklasse
- ◆ Relation
- ◆ Attribut
- ◆ Relationel database
- ◆ Tabel
- ◆ Primærnøgle
- ◆ Database-integritet
- ◆ Første normalform
- ◆ Anden normalform
- ◆ Tredje normalform

KAPITEL 8

DIGITALE MULTIMEDIER

Med computerteknologi kan tekst, billeder, lyd og film kombineres meningsfuldt i *multimedier*. Multimedier bruges til meget forskelligt:

- ◆ Oplysning
- ◆ Underholdning
- ◆ Kunst
- ◆ Reklamer
- ◆ Videnskab
- ◆ Forretning
- ◆ Politik
- ◆ ...

Men hvad er et *medie*?

ORDFORKLARING

MEDIER OG MULTIMEDIER

Et *medie* lagrer information og/eller formidler information af bestemte indholdstyper i bestemte formater. Et medie, der lagrer og/eller formidler flere forskellige indholds-typer samtidigt kaldes et *multimedie* ("multi" betyder "mange"). Multimedier er ofte interaktive.

Termen "digitalt medie" bruges typisk meget løst om både hardware- og softwareenheder.

EKSEMPEL

MEDIER, MULTIMEDIER, INDHOLDSTYPER OG FORMATER

Harddisker, USB-nøgler, CD'er og DVD'er er alle medier. Filer, databaseindhold og programkode kan også (med lidt god vilje) kaldes medier.

Tekst, billede, lyd, animation og video er alle eksempler på indholdstyper.

Tekst kan lagres i mange formater, fx som ren tekst, formatteret tekst eller webtekst med links. Lyd kan lagres i mange forskellige formater, fx er der specielle formater til CD-lyd, mens musik, der afspilles over internettet, bruger andre formater.

En website med billeder, links og tekst er et multimedie. En internet-blog er interaktiv, indeholder tekst og billeder og er dermed også et multimedie. Det er en film også – den kombinerer lyd og animation. Typiske brugerprogrammer er også multimedier – der er både billeder, tekst og interaktion. Det samme gælder sociale online-netværk, emails og nyhedsfeeds.

Som vi aner, er der mange "multimedier". Generelt kan man dog forvente, at folk bruger ordet "multimedier" om digitale medier, der på en *interaktiv måde kombinerer tekst, billeder, lyd og/eller video*.

For at forstå digitale multimedier skal vi forstå *de byggesten*, multimedier kan være opbygget af:

- ◆ digital grafik (afsnit 8.1),
- ◆ digital lyd (afsnit 8.2),
- ◆ digital video (afsnit 8.3).

Til sidst kigger vi på, hvordan multimedier forbruges over et netværk (afsnit 8.5).

WWW

MERE OM MULTIMEDIER

På **WWW** kan du læse nærmere om to populære eksempler på multimedier: *interaktive websider* og *computerspil*.

8.1. GRAFIK

Computergrafik handler om digitale billeder:

- ◆ Fremstilling af billeder (fotos med kamera eller skabes direkte på computeren)
- ◆ Ændring af billeder (med billedbehandlingsprogrammer)
- ◆ Publicering af billeder (typisk på nettet)

- ◆ Data-repræsentation (hvordan lagres et billede)
- ◆ Skærmfremvisning (hvordan omsættes en datarepræsentation til et visuelt udtryk, mennesker kan sanse)
- ◆ ...

I dette kapitel kigger vi på data-repræsentation og skærmfremvisning.

WWW

EKSPERIMENTER MED GRAFIK

På **WWW** finder du ideer og værktøjer til, hvordan du selv at fremstiller og redigerer digital grafik. Du finder også en liste over almindelige filformater for digitale billeder.

8.1.1. PIXELS OG SKÆRMFREMVISNING

Displayet på en computerskærm er inddelt i et skema af felter, lidt ligesom et skakbræt. Hvert felt kan lyse i en individuel farve. Felterne kaldes *pixels* (pixel er en forkortelse af *picture element*) .

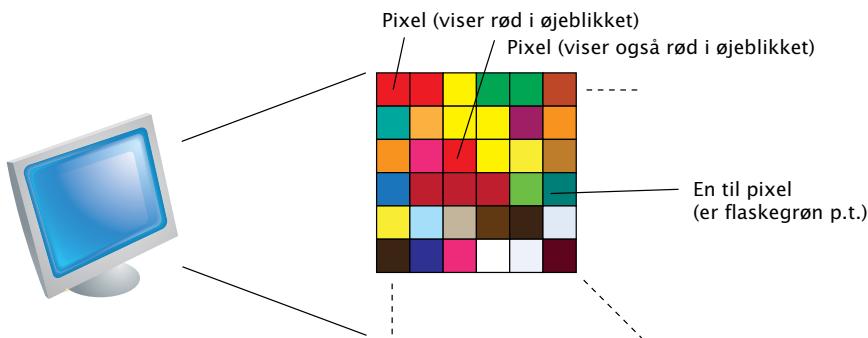


Fig. 8.121. Computerskærmens display er inddelt i pixels. Hver pixel kan vise en farve ad gangen.

En "gammeldags" film er i virkeligheden en stribe billeder, der vises med fx 1/24 sekunds mellemrum. Tilsvarende opdateres farven af hver enkelte pixel på en computerskærm typisk med 1/100 sekunds mellemrum – begge dele hurtigere end det menneskelige øje kan nå at opfatte. På den måde opleves ikke irritation eller flimren. Computerskærme kan på den måde, som film, give illusionen af bevægelse på skærmen.

8.1.2. FARVEMODELLER

Mennesker er vant til at se farver ved at genstande *reflekterer lys* forskelligt. Rødt blæk, fx, opfattes rødt, fordi det absorberer visse bølgelængder af indkommende lys og kun reflekterer lys tilbage, som det menneskelige øje opfatter som rødt. Blander vi gul og blå blæk, fås en grøn farve, fordi kombinationen af blæk nu “absorberer al farve, der ikke er grøn”.

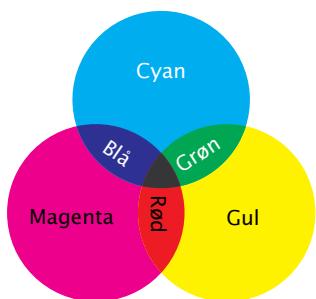


Fig. 8.122. Subtraktiv farvemodel.

Opfattelse af farver på denne “sædvanlige” måde kaldes *subtraktiv*, fordi visse bølgelænger ”fratrækkes” det hvide lys. En *subtraktiv farvemodel* er en model med grundfarver, der ved at blive kombineret subtraktivt kan skabe et helt farvespektrum.

Figur 8.122 viser den meget udbredte subtraktive farvemodel CMY, hvis navn er en forkortelse af modellens 3 grundfarver *cyan* (turkisblå), *magenta* (rødlilla) og *yellow* (gul). Trækkes alle farverne fra det hvide lys, fås sort.

Computerskærme fungerer ikke subtraktivt, fordi de ikke reflekterer lys, men *udsender lys*. Her trækker de enkelte grundfarver ikke bølgelængder ud af hvidt lys, men tilfører bølgelængder til mørke.

For at give farveoplevelser skal en computerskærm derfor ikke bruge en subtraktiv model, men en *additiv farvemodel*. Hver enkelt pixel på skærmen belyses med forskellige grundfarver i forskellige styrker (grundfarverne ”lægges sammen”). Herved opnås forskellige farve-synsindtryk for den, der kigger på skærmen. Den mest anvendte additive farvemodel er RGB-modellen med de tre grundfarver *rød*, *grøn* og *blå* (- gæt hvad RGB står for). De tre grundfarver i denne model kan lægges sammen og danne farver fra et helt farvespektrum. Lægges alle farverne sammen, fås hvid:

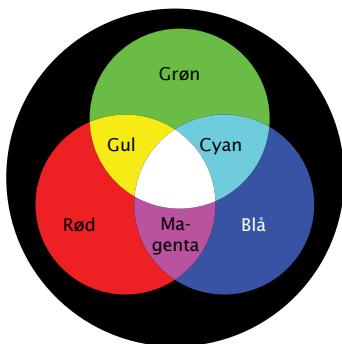


Fig. 8.123. Additiv farvemodel.

I RGB-modellen er rød + grøn = gul. Den er altså tydeligt anderledes end vores ”sædvanlige opfattelse af farver”, den subtraktive forståelse, hvor grøn + rød = brun.

CMY-MODELLEN	RGB-MODELLEN
Subtraktiv farvemodel	Additiv farvemodel
Farver er et resultat af reflekteret lys	Farver er et resultat af udsendt lys
Cyan + magenta + gul = sort	Rød + grøn + blå = hvid
Til tryk. Bruger blæk til farve. (I afsnit 8.1.6 hører vi mere om farveprint.)	Til computerskærme. Bruger lys til farve.

Fig. 8.124. Sammenligning af CMY- og RGB-farvemodellerne.

8.1.3. BILLEDKVALITET

Operativsystemet har et særligt stykke hukommelse afsat til at kontrollere farverne på skærmen. Denne grafik-hukommelse kaldes *video RAM* (forkortet: VRAM) og er typisk ikke en del af selve computerens RAM. Hver pixel på skærmen svarer til et lille stykke VRAM, hvor operativsystemet kan sætte den pixels aktuelle *farvekode*. De enkelte pixels farvekoder skifter løbende, og skærmen fremviser løbende de farvekoder, der er lagret i VRAM'en.

Det antal bits, VRAM'en afsættes pr pixel til farvekoder, kaldes for *farvedybden* for displayet. En computerskærm kan typisk indstilles til en af flere forskellige farvedybder.

Hvis en computerskærm har en farvedybde på to, og altså kun kan vise to farver – fx sort og hvid, eller, som mange tidlige computerskærme, grøn og sort – behøves kun 1 bit til at lagre farvekoden for en pixel. Et 0 betyder hvid (eller grøn) og et 1 betyder sort.

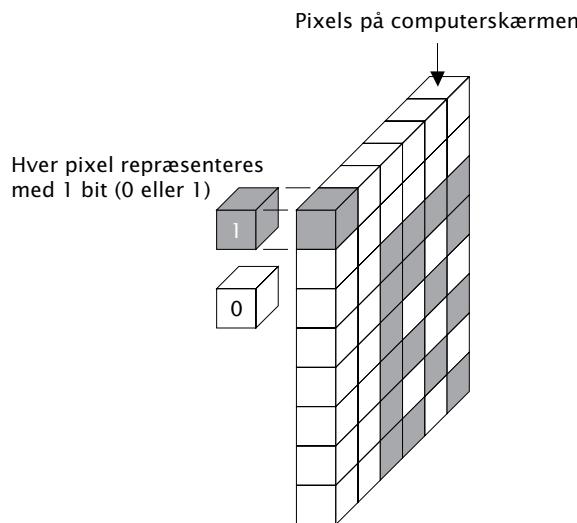


Fig. 8.125. 2-bit farvedybde.

Hvis der bruges flere bits pr pixel, opnås flere mulige farver. Fx fås i alt $256 = 2^8$ forskellige farver med 8-bit farvekoder. Sådanne displays kaldes 8-bit eller 256-farver. Det kan løbende skifte, præcis hvilke 256 farver, skærmen skal vælge imellem, men der kan maksimalt være 256 forskellige farver på skærmen ad gangen. Når skærmen laver en opdatering af sine pixelfremvisninger, kigger den på den enkelte pixels 8-bit-farvekode og slår derefter op i en tabel – også lagret i VRAM'en – der oversætter farvekoder til faktiske farver. Tabellen kaldes CLUT (*colour lookup table*). Med en 8-bit farvedybde har CLUT'en 256 indgange. Hvis operativsystemet vil ændre udvalget af farver, ændres blot i CLUT'en.

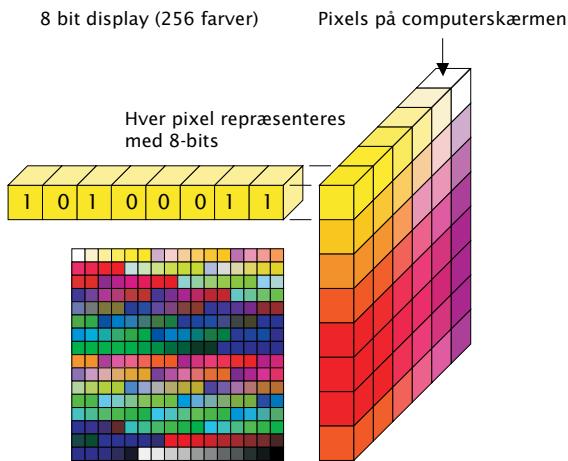


Fig. 8.126. 8-bit-farver med CLUT.

Giver man hver pixel 24 bit (8 bit til rød, 8 til grøn, 8 til blå), kan stort set *fotorealistisk effekt* på skærmen opnås. 24-bit-displays kaldes *true-color*, fordi man opnår farveeffekter, der er lige så realistiske som på et fotografi – “de sande farver” vises. Der er mulighed for lidt over 16 millioner samtidige farver (helt præcist er der $2^{24} = 2 \times 2 = 16.777.216$ forskellige farver).

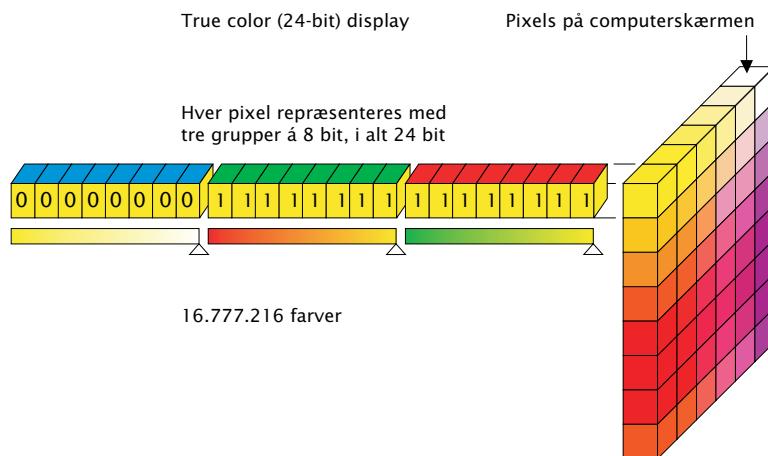


Fig. 8.127. True color.

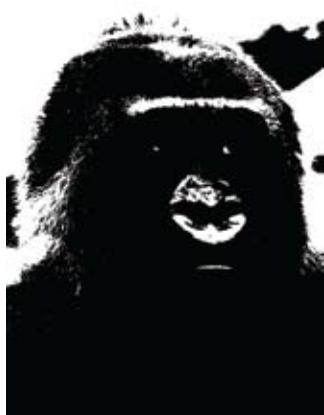
Med true color bruges ikke noget CLUT, fordi udvalget af farver ikke skifter.

8.1.4. BITMAP-BILLEDER

Et bitmap-billede repræsenterer et billede som et rektangulært gitter af pixels. Hver indgang i gitteret rummer en farvekode. Ved billedets *opløsning* forstår man antallet af pixels, der indgår i billedet. Det angives typisk som sidelængderne på det rektangulære pixelgitter. Fx kan en oplosning være 80 x 200 pixels (et billede, der er 80 pixels højt og 200 pixels bredt).

En billedfil består af

- ♦ en “palet” – medmindre det er et true color-billede,
- ♦ information om, hvilke pixels, der har hvilke farver (enten fra paletten eller fra de 16 millioner true color-farver).



Hvid-sort-palet med farvedybde på 1 bit



4-bit farvepalet



Gråtonepalet med farvedybde på 8 bit



True color (24 bit)

Fig. 8.128. 3 eksempler på paletter samt true color.

Paletten oversætter farvekoder af en fast *farvedybde* til faktiske farver. På den måde spiller en palet i en bitmap-billedefil samme rolle som CLUT'en for en farveskærm.

24-bit-billeder lagres typisk som tre “ens-farvede lag” – et rødt lag, et grønt lag og et blåt lag. Hvert “ensfarvet” lag fylder som et selvstændigt 8-bit billede.

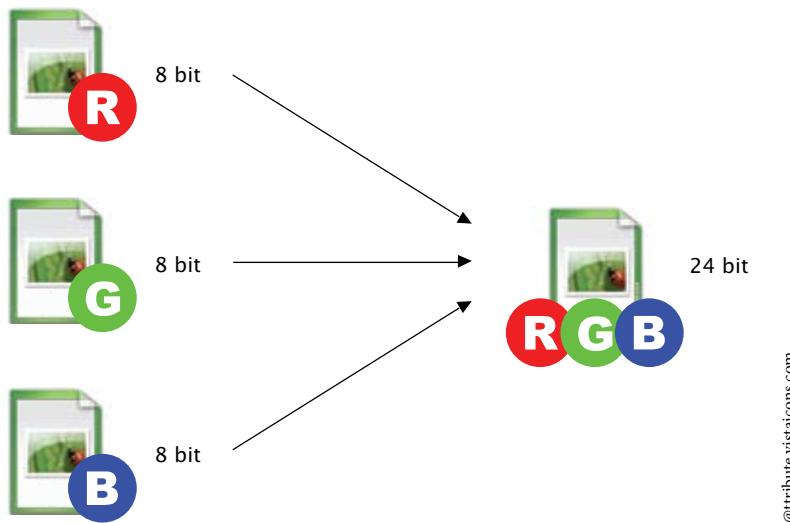


Fig. 8.129. Et 24-bit-billede lagres som 3 billeder i 8-bit.

Kvaliteten af et bitmapbillede er bestemt ved

- antal pixels (opløsning),
- informationsmængde i hver pixel (*bitdybden* – alstår palettens farvedybde).

Vi kan se på Figur 8.128, at jo større farvedybde, desto “glattere” og mere virkelighedstroser billedet ud. På Figur 8.130 kan vi se, at et lav oplosning ser grov og firkantet ud, mens en høj oplosning ser mere virkelighedstro ud.

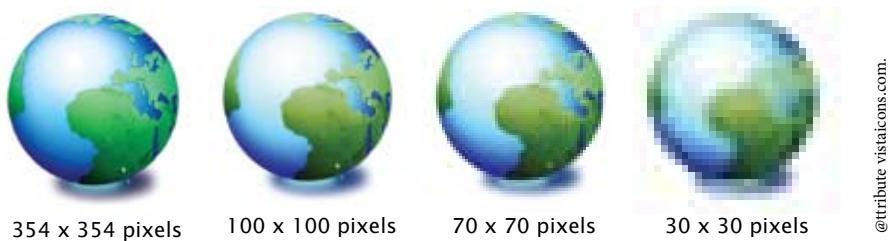


Fig. 8.130. Fra høj oplosning til lav oplosning.

Bitmapgrafik er oplosningsafhængigt. Hvis bitmaps forstørres og fremvises i høje oplosninger ”mistes” kvalitet: Billeder ”pixeleres” (bliver firkantede) eller utydelige (compute- ren forsøger at udglatte).

8.1.5. VEKTORGRAFIK

Bitmapbilleder lagrer billeder som et skema af pixels med hver deres farve. *Vektorgrafik*, derimod, lagrer billeder som samlinger af matematisk definerede kurver og linier. Fx kan vi gemme to farvede cirkler som

- ◆ cirkernes ligninger (matematiske udtryk, der udtrykker cirklerne),
- ◆ de to farkekoder.

Når et billede lagret som vektorgrafik skal fremvises på en skærm, oversætter et program matematikken til et aktuelt skærm-bitmap, der så fremvises.



Fig. 8.131. Vektorgrafik fremvises som skærm-bitmap.

Fordi alle former og kurver på billedet er matematiske objekter, kan de forstørres så meget det skal være uden tab af kvalitet. Fx kan et logo for en fægteklub, gemt som vektorgrafik, ligeså vel printes på et visitkort som på en plakat, der dækker en hel husmur. Situationen er altså helt anderledes end i Figur 8.130.



Fig. 8.132. Skalering af vektorgrafik: Vektorgrafik-original til venstre, til højre en kraftig opskalering – helt uden kvalitetstab.

For at redigere i vektorgrafik skal vektorerne ændres, i stedet for de enkelt pixels. Vektorer ændres ved matematiske transformationer som fx at strække, bøje, spejle eller farve.

Vektorgrafik kan

- ◆ fjerne unødige detaljer – fx til *informationsgrafik* (piktogrammer, logoer, skilte, videnskabelige figurer, mv.),
- ◆ opskaleres uendeligt uden tab af detaljer.

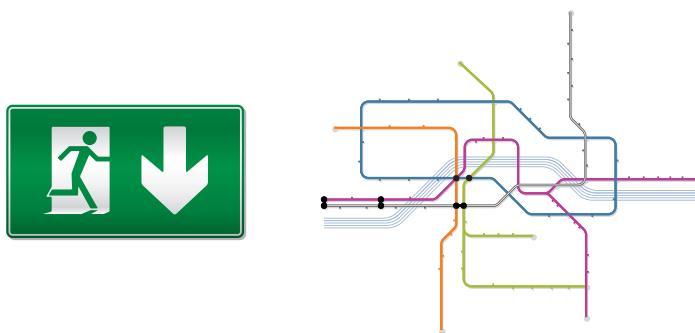


Fig. 8.133. Eksempler på informationsgrafik: piktogram og metrokort.

Vektorgrafik er dog ikke en erstatning af bitmapgrafik. Bitmaps

- ◆ kan forstås og redigeres uden matematisk indsigt,
- ◆ håndterer fotos bedre end vektorgrafik,
- ◆ er bedre til detaljer end vektorgrafik.

8.1.6. UPRENT AF GRAFIK

En printer virker ved at inddale papiret i et rektangulært gitter af prikker ("dots") og fyldte farvet blæk i hver prik – lidt ligesom et billede inddeltes i pixels.

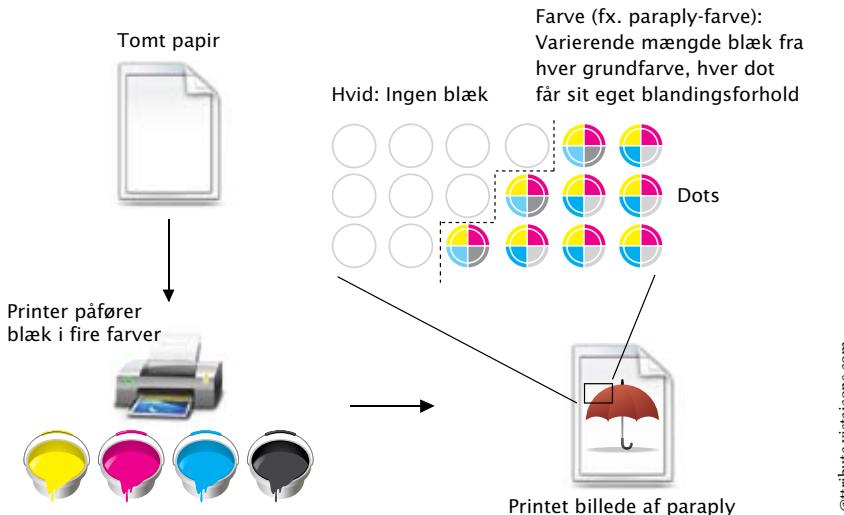


Fig. 8.134. Printeren fylder blæk i et gitter af dots.

Grafik kan printes i forskellige kvaliteter, typisk målt i, hvor finkornet printeren er. Det opgiver man typisk i *DPI* (dots per inch). DPI måler, hvor mange farvede prikker printeren kan lave på en tomme. Jo flere, desto bedre kvalitet.

CMYK er en udbredt farvemodel til farveprint. En CMYK-printer har 4 farver blæk, som har givet modellen dens navn: *cyan* (turkisblå), *magenta* (rødlilla), *yellow* (gul) og *key*-farven (egentlig: "nøglefarve", ofte sort). Ved print påføres blækket i 4 lag, typisk i samme rækkefølge som forkortelsen antyder. CMYK-modellen er *subtraktiv*: Den virker ved at lade blækket "maskere" visse farver fra den hvide baggrund (papiret). Blækket "fratrækker" visse bølgelængder fra de hvide lys. Jo flere farver der anvendes, des mørkere (jf. Figur 8.122). Ved at variere blandingsforholdet mellem de 4 farver blæk opnår printeren et farvespektrum.

8.1.7. TABSFRI KOMPRIMERING AF GRAFIK

Billedfiler rummer tit "overflødig" information. Ved at lagre billedet smartere, kan man spare plads. Det kaldes *tabsfri komprimering*.

EKSEMPEL

LAGRING AF SIMPELT BILLEDE

Et billede, der er sort næsten over det hele, behøver ikke have farvekoden for "sort" gemt i hver eneste pixel. Et mere besparende format for lagring af dette billede ville være "sort i alle pixels, undtagen følgende: ... ". I dette format bruges ikke plads til at huske, hvilke pixels der er sorte – kun hvilke, der *ikke* er sorte.

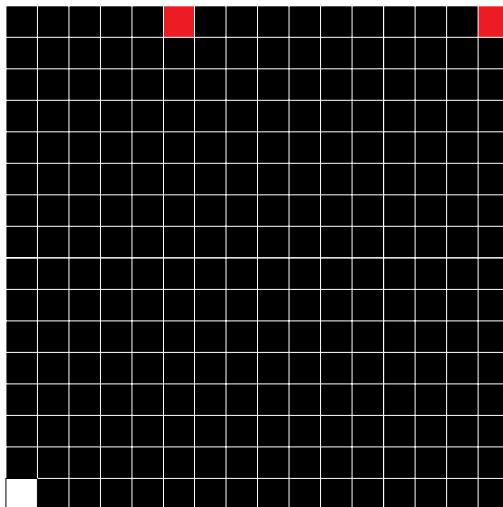


Fig. 8.135. Simpelt billede på 256 pixels.

Det smukke 4-bit-bitmap-billede, man ser i Figur 8.135, har en oplosning på 16x16 pixels = 256 pixels. Kun 3 af disse pixels er ikke sorte. Lagret som 256 farvekoder på 4 bit fylder billedet

$$256 \text{ pixels} \times 4 \text{ bits/pixels} = 1024 \text{ bits.}$$

Vi lagrer nu i stedet billedet som reglen "Standardfarve: sort" og listen

[(1,5) : Rød , (1,16) : Rød , (16,1) : Hvid]

af "undtagelses-pixels" – hver undtagelsespixel har en position (der fylder 8 bits) og en farvekode (på 4 bits). Under antagelse af, at farvekoden for "standardfarven" kan gemmes på 4 bits plads, fylder billedet i det nye format blot

$$4 \text{ bits} + 3 \times (8 \text{ bits} + 4 \text{ bits}) = 40 \text{ bits.}$$

Besparelsen på 1024 bits – 40 bits = 984 bits opnår vi, fordi vi lagrer al informationen om de sorte pixels på kun 4 bit.

Tankegangen fra eksemplet kan videreføres til ethvert billede, uanset oplosning og farvedybde. Ideen er, at

de mest forekommende farver skal fylde mindst at lagre.

Ideen adskiller sig altså fra bitmap-tankegangen, hvor alle pixels fylder lige meget. Til gen-gæld kræver teknikken, at man kender de enkelte farvers hyppighed i et billede.

Med *Huffman-kodning* kan man give hver farve i et bitmap-billede en ny farvekode. Hvis farven er meget hyppigt forekommende, bliver den nye farvekode meget kort. Hvis farven er meget sjælden, bliver den nye farvekode lang.

EKSEMPEL

HUFFMAN-KODNING

Vi forestiller os et 4-bit-billede på 1024 pixels x 1024 pixels = 1.048.576 pixels. Billedet bruger kun 5 farver, blå, rød, gul, grøn og lilla. Følgende tabel viser deres hyppigheder:

Farve	Blå	Rød	Gul	Grøn	Lilla
4-bit farvekode	1100	1001	1100	1010	1110
Hyppighed	1.000.000	40.000	8.000	500	76

Billedet er altså overvejende blåt. Billedet fylder med 4-bit-farvekoder

$1.048.576 \text{ pixels} \times 4 \text{ bits/pixel} = 4.194.304 \text{ bits}$ (ca 0,5 MB).

Huffman-kodning giver hver pixel nye farvekoder af *varierende længde*, baseret på hyppigheden.

Farve	Blå	Rød	Gul	Grøn	Lilla
Huffman-farvekode	0	101	100	111	1101
Længde af Huffman-farvekode	1	3	3	3	4

Efter Huffman-kodning skal billedet rumme

- ◆ en oversættelse af Huffman-koder til de gamle farvekoder,
- ◆ Huffman-koderne for hver pixel.

fortsættes...

EKSEMPEL

HUFFMAN-KODNING (FORTSAT)

Hvis vi ser bort fra de få bits, der skal bruges til at lagre oversættelsen, fylder det Huffman-kodede billede

1.000.000 blå pixels x 1 bit / blå pixel +
40.000 røde pixels x 3 bits / rød pixel +
8.000 gule pixels x 3 bits / gul pixel +
500 grønne pixels x 3 bits / grøn pixel +
76 lilla pixels x 4 bits / lilla pixel =
1.000.000 bits + 120.000 bits + 24.000 bits + 1500 bits + 296 bits =
1.145.796 bits (ca 0,15 MB).

Vi har altså høstet en plads-besparelse på omkring 0,35 MB.

Der er en god grund til, at Huffman-farvekoderne kom til at se ud som vist – og at der fx ikke er nogen Huffman-koder med et bit-længde på 2. Det fører imidlertid for vidt at komme ind på, hvordan Huffman-koder konstrueres og bruges. Det vigtige at notere sig her, er hyppighedsprincippet.

8.1.8. GRAFIKKOMPRIMERING MED TAB

Er man villig til at acceptere mindre forringelser i kvaliteten af et billede, kan man spare plads ved at komprimere grafikfiler ”med tab” – altså med tab af kvalitet. To oplagte måder at komprimere et billede på er

- at skære ned på antal farver (farvedybden),
- at skære ned på antal pixels (opløsningen).

Der findes imidlertid smartere komprimeringsmåder, som bevarer farvedybde og oplosning. Sådanne komprimeringer er avancerede. De smider kun grafisk information væk, man har svært ved at se som menneskelig beskuer, fx meget fine farvforskelle. Pladsbesparelserne, man kan opnå med sådanne komprimeringer, er større end med tabsfri komprimeringer – og resultatet ofte lige så godt (for det menneskelige øje).

8.2. LYD

For at forstå digitale repræsentationer af lyd, må vi først forstå hvad lyd er. En *lyd* er fysisk set en trykbølge i luften. Bølgen kaldes en lydbølge. Når lydbølgen rammer vores ører, *hører* vi lyden – luftens vibrationer omsættes i ørets indre organer til en lydoplevelse.

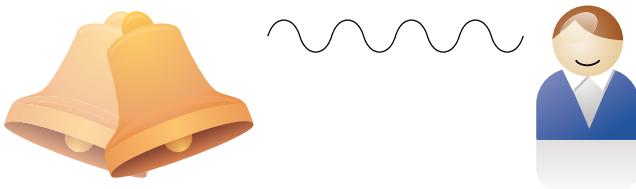


Fig. 8.136. Lydbølge (Ahhh! Kirkeklokker tidligt søndag morgen!).

8.2.1. OPTAGELSE AF LYD I DIGITALT FORMAT

Oprindeligt optog man lyd *analogt*: Man repræsenterede den kurvede lydbølge med en "kurvet tilnærmelse". På LP'er bliver lydbølgene fx repræsenteret som kurvede niveauforskelle i en lang rille i en lakplade.

Nu om dage bruger man især *digital representation*, hvor lydbølgene repræsenteres som en "firkantet tilnærmelse":

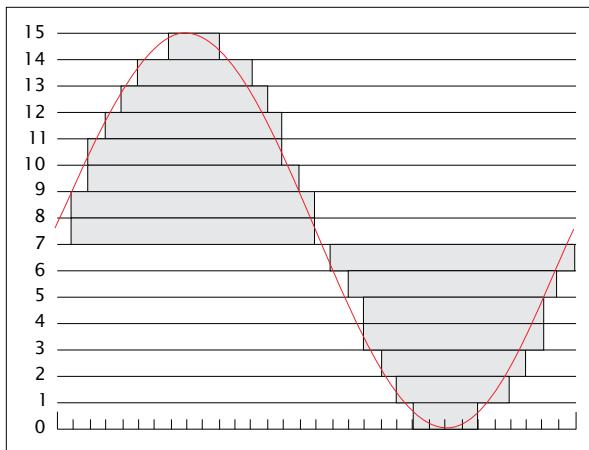


Fig. 8.137. Firkantet tilnærmelse (grå) af lydsignal (rød).

Den firkantede tilnærmelse har den fordel, at den kan repræsenteres præcist med et endeligt antal talværdier – svarende til hjørnepunkterne i de indgående firkanter. Med andre ord: repræsentationen er digital.

For at opnå den firkantede tilnærmelse foretages en række *sampling*er. Hver sampling måler groft sagt lyden lige på det tidspunkt, samplingen foretages, og omsætter målingen til en talværdi. Med tilstrækkelig mange samplingser af høj nok kvalitet fås en god firkantet tilnærmelse til den kurvede lydbølge.

Konkret opfanger en mikrofon lydsignalet og omsætter det til et elektrisk signal, der videresendes til et apparat, vi her omtaler som en sampler. Sampleren udtager samplingser et bestemt antal gange hvert sekund – hver gang måles den aktuelle styrke af det elektriske signal, og det omsættes til en talværdi.

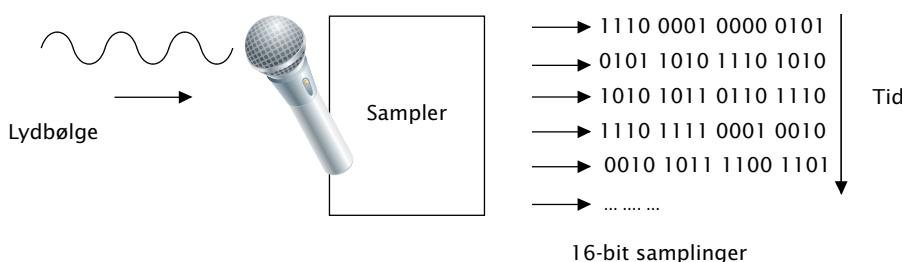


Fig. 8.138. Fra lydbølge til digital lyd (optagelse).

Hver talværdi repræsenteres som en samling af bits. Hver bitsamling har samme størrelse, og denne størrelse – præcisionen i samplingsen – kaldes samplerens *bitdybde*:

$$\text{bitdybde} = \text{antal bits} / \text{sampling}$$

Antallet af samplingser, sampleren udtager pr sekund, kaldes *samplingsfrekvensen*:

$$\text{samplingsfrekvens} = \text{antal samplingser} / \text{sekund}$$

Man mäter tit samplingsfrekvensen med enheden *hertz* (1 hertz = 1/sekund).

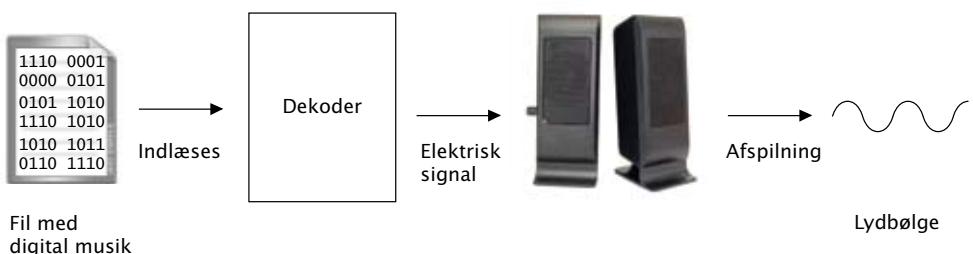
EKSEMPEL

CD-OPTAGELSE

Musik på CD'er er lagret digitalt. Optagelser på CD har typisk en bitdybde på 16 (dvs. hver sampling fylder 16 bit) og en samplingsfrekvens på 44,1 kilohertz = 44.100 hertz (dvs. der er 44.100 samplingser pr sekund).

8.2.2. AFSPILNING AF DIGITAL LYD

Man afspiller digital lyd ved sende det igennem en *dekoder*. Dekoderen omsætter de digitale koder til et sammenhængende elektrisk signal, der sendes til en højttaler. Højttaleren omsætter løbende det indkommende elektriske signal til svingninger af en membran – det resulterer i en trykbølge, der af menneskers ører opfattes som lyd.



Attributed via icons.com.

Fig. 8.139. Fra digital lyd til lydbølge (afspilning).

Generelt gælder, at jo større samplingsfrekvens og jo større bitdybde, den digitale lyd har, des bedre lydkvalitet opleves, når den digitale lyd omsættes til virkelig lyd.

For at have et enkelt samlet mål for lydkvalitet af en digital lydoptagelse taler man ofte om optagelsens *bitrate*. Bitraten måler optagelsens kvalitet i bits/sekund – groft sagt: jo flere bits pr sekund, des bedre kvalitet.

Ud fra bitdybde og samplingsfrekvens kan bitraten bestemmes som

$$\text{bitrate} = \text{bitdybde} \times \text{samplingsfrekvens}$$

EKSEMPEL

BEREGNING AF BITRATE

Hvis en lydoptagelse har en bitdybde på 12 bits og en samplingsfrekvens på 20.000 samplinger pr sekund, bliver bitraten

$$\begin{aligned}\text{bitrate} &= 12 \text{ bits / sampling} \times 20.000 \text{ samplinger / sekund} \\ &= 240.000 \text{ bits / sekund} = 0,24 \text{ Mbit / sekund}\end{aligned}$$

8.2.3. KOMPRIMERING AF LYD

Det menneskelige øre er fintfølende. Derfor bruger man digitale lydoptagelser i en kvalitet, der tilfredsstiller det menneskelige øre 100 %, til fx CD'er, lydspor til film i biografer og i professionelle musikstudier. Sådanne optagelser fylder imidlertid meget. Man ønsker ofte at minimere pladsforbruget til lydfiler, fx fordi

- ◆ digital lyd på nettet skal hentes eller høres over en forbindelse med begrænset kapacitet,
- ◆ pladsen er begrænset på steder som musikservere, transportable musikafspillere, mv.

Derfor *komprimerer* man ofte lydfiler, efter at de er blevet optaget i høj kvalitet.

En komprimeret lydfil fylder væsentligt mindre end en ukomprimeret. Komprimering udnytter, at alt lyd i en ukomprimeret fil ”fylder lige meget”. Ukomprimerede lydfiler er altså *ikke pladseffektive*.

EKSEMPEL

KOMPRIMERING

To højkvalitets-lydoptagelser, A og B, varer begge 1 minut. A er en optagelse af absolut stilhed, mens B er en optagelse af noget af Beethovens Skæbnesymfonii. Lydoptagelserne er lagret i samme ukomprimerede format og fylder derfor begge to *det samme!*

En komprimering kan bl.a. udnytte, at visse lydmønstre enten er ”lette” at lagre eller mindre væsentlige for den menneskelige lytteoplevelse. Komprimering virker derfor ved at få dele i lydfilen, ”der ikke bør fyde så meget”, til at fyde mindre – det kan fx være

- ◆ stilhed – kan lagres pladseffektivt,
- ◆ gentagne lyde – kan lagres pladseffektivt,
- ◆ lyde, der næsten er uden for det menneskeliges øres opfattelsesevne – kan slettes,
- ◆ lyde, der næsten ikke kan skelnes fra resten af lydbilledet af det menneskelige øre – kan slettes.

Der findes mange forskellige komprimeringsalgoritmer, og det er forskelligt fra algoritme til algoritme, hvad der komprimeres hvordan.

EKSEMPEL

KOMPRIMERING (FORTSAT)

Lad os nu sige, at begge optagelserne A og B fra det forrige eksempel fylder 10 MB. Den komprimerede udgave af A fylder nærmest ingenting – måske 0,1 MB – fordi det er ”let” at repræsentere stilhed. Den komprimerede udgave af B, derimod, fylder stadig en del efter komprimeringen – måske 6 MB – fordi Skæbnesymfonien er et komplekst stykke lyd.

Der er to typer komprimering:

- ◆ tabsfri komprimering (engelsk: *lossless compression*),
- ◆ komprimering med tab (engelsk: *lossy compression*).

Tabsfri komprimering får data til at fyde mindre, *uden at information går tabt*. Tabsfri komprimering sletter altså ingen lyde. Med tabsfri komprimering kan man ofte få lyddata til at fyde halvt så meget. Tabsfri komprimering af lyd foregår på bit-niveau, ligesom tabsfri komprimering af billeder (se afsnit 8.1.7).

Komprimering med tab smider information væk – til gengæld kan lyddata ofte komme til at fylde op mod 10 gange så lidt som før komprimeringen. Når man komprimerer med tab, smider komprimeringsalgoritmen “det mindst vigtige” lyddata ud. Den præcise fastlæggelse af, hvilke dele af lyd, der er “vigtige” for den menneskelige lytteoplevelse, fastlægges med *psykoakustiske heuristikker* (på dansk: overordnede retningslinier baseret på en masse undersøgelser med mennesker, der lytter til musik i forskellige komprimeringer).

Bitraten sænkes med komprimering. Jo bedre komprimeringsalgoritme, desto mere kan bitraten sænkes relativt uden at skade lydkvaliteten.

Lydfiler, der er tabsfrit komprimeret, kan dekomprimeres – og så er originalen genskabt. Lydfiler, der komprimeret med tab, kan også dekomprimeres – men resultatet er ikke af lige så høj kvalitet som originalen.

WWW

LYDFILFORMATER

På [WWW](#) finder du en liste af kendte lydfils-formater, katalogiseret med typiske komprimeringstyper og bitrater.

8.3. VIDEO

Ligesom en traditionel film er en sekvens af enkeltbilleder, er en digital video en sekvens af digitale billeder. Billederne fremvises med en konstant hastighed, fx 24 eller 30 billeder i sekundet. Man omtaler de enkelte billede som *frames*.

Som med digitale billeder kan en digital video være bitmapbaseret eller vektorgrafisk baseret. En video med bitmap-frames fremviser blot en række bitmap-billeder efter hinanden. Billederne behøver i princippet ikke have nogen tilknytning til hinanden. En videosekvens baseret på vektorgrafik fremviser derimod en række matematiske transformationer, der anvendes i rækkefølge, på en start-opstilling af de matematiske figurer, der indgår i billederne.

Hvor lydsporet til en traditionel film er optaget separat fra selve strimlen med filmens frames, omfatter moderne filformater til digital video også lydsporet.

Fuldstændig parallelt til situationen med digital lyd har digitale videofiler

- ◆ en *bitrate* (målt i bits / sekund), der måler hvor mange bit 1 sekund af videoen fylder,
- ◆ en *bitdybde* (målt i bits / frame), der måler størrelsen af hvert enkelt billede i bits,
- ◆ en *frame-frekvens* (målt i frames / sekund), der måler, hvor mange frames videoen fremviser i sekundet.

Man ønsker at spare plads på lagring af og båndbredde til transport af digital video – fuldstændig som med digital lyd. Derfor er næsten alle digitale videoformater *komprimerede*. En video kan komprimeres på flere måder (samtidigt!):

- ◆ lydsporet kan komprimeres,
- ◆ de enkelte frames kan komprimeres (intra-frame-komprimering),
- ◆ en samling af frames kan komprimeres (inter-frame-komprimering).

Vi kender allerede til komprimering af billeder (afsnit 8.1.7) og komprimering af lyd (afsnit 8.2.3), så kun det sidste punkt behøver en uddybning.

Lad os forestille os en film, hvor 10 frames i træk viser en panda ligge og dase i en bambuslysnings. Der sker ikke noget fra billede til billede: Bambusen bevæger sig ikke, og pandanen er pænt mæt og ligger helt stille. De 10 frames er altså *ens!* Der er ingen grund til at lagre alle 10 frames i deres fulde størrelse. I stedet lagres kun den første frame fuldstændigt sammen med en angivelse af, at de næste 9 frames er identiske. Den første frame fungerer dermed som såkaldt *key frame* for de efterfølgende 9 frames. Hvis de 9 efterfølgende frames havde mindre afvigelser fra key framen, ville man – frame for frame – angive hvilke forandringer den aktuelle frame rummer i forhold til den foregående.

Jo flere bevægelser og sceneskift der er i en video, des mindre kan inter-frame-komprimering benyttes med udbytte.

WWW

VIDEOFILFORMATER

På **WWW** finder du en liste af kendte videofil-formater, katalogiseret med typiske komprimeringstyper og bitrates.

8.4. 3D-GRAFIK

En computerskærm har 2 dimensioner: Højde og bredde.

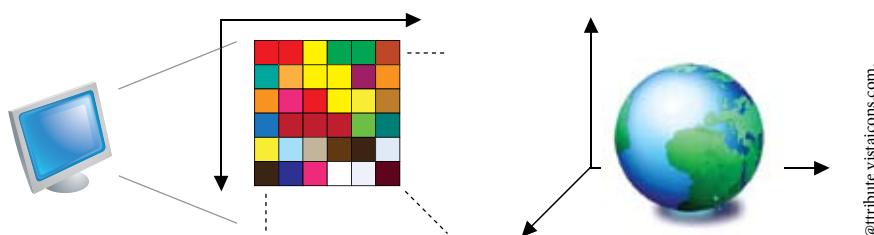


Fig. 8.140. Computerskærm: to dimensioner. Virkeligheden: tre dimensioner.

Virkelighedens rum er som bekendt *3-dimensionalt*. Gengivelse af rumlige objekter som grafik på computerskærme er imidlertid nødvendigvis 2-dimensional. Hvad menig giver det så tale om 3D-grafik?

ORDFORKLARING

3D-GRAFIK

Computergrafik, der er repræsenteret som 3-dimensionalt geometrisk data (en *3D-model*). Når grafikken skal fremvises, beregnes et 2D-billede, der viser et "syn" på det 3-dimensionale data.

Med 3D-grafik kan opnås 2D-billeder, der i højere grad end 2D-grafik giver beskueren en fornemmelse af rumlighed. 3D-modeller bruges også til andet end fremvisning – fx simulationer og beregninger.

WWW

3D-GRAFIK

På **WWW** finder du

- ◆ opgaver, hvor du selv kan eksperimentere med 3D-grafik,
- ◆ links til mere viden om 3D-grafik.

8.5. MULTIMEDIER OVER NETVÆRK: STREAMING

Multimedier, der forbruges over netværk, er allestedsnærværende: Folk snakker sammen på tværs af kontinenter med IP-telefoni eller i virtuelle verdener. Forretningsmænd mødes i telekonferencer. Elever med langt til skole modtager video-fjernundervisning. Der spilles computerspil over lokalnetværk, folk ser streamede film og lyttere hører internett-radio.

Lad os kalde multimedieapplikationer, der afvikles over netværk, for *netværksmultimedieapplikationer*. Netværksmultimedieapplikationer har helt andre krav end dem, protokoller til emails eller filtransport kræver (jf. kapitel 5):

- ◆ en netværksmultimedieapplikation er *folsom over for forsinkelser*,
- ◆ en netværksmultimedieapplikation kan *acceptere tab af data*.

Faktisk er det lige omvendt for fx transmission af emails: Forsinkelse er irriterende, men ikke skadeligt, mens tab af data er uacceptabelt.

Vi kigger i det følgende på tre typer afvikling af multimedie over netværk:

- ◆ streaming af lagret data,
- ◆ live streaming,
- ◆ real-time interaktiv lyd eller video.

ORDFORKLARING

STREAMING

At lyd- eller film-data løbende leveres, lidt ad gangen, og løbende afspilles.

Med streaming tænker man sig, at browseren modtager lyd- eller film-filen i en “strøm af data”, deraf navnet “streaming”. Browseren siges at *stremme* en film, når den påbegynder afspilningen, selvom hele filen med filmen endnu ikke er hentet fra dens placering på netværket.



Fig. 8.141. Streaming.

Konkret består “strømmen” af datapakker, som afsender har opdelt lyd- eller videofilen i.

Modtager opsamler løbende de indkommende pakker i en *buffer*, hvorfra de videresendes til afspilning. Det er typisk ikke browseren, men et medieafspilningsprogram, der står for både *buffering* og afspilning.

Streaming står på den måde i modsætning til et normalt download-forløb, hvor en hel fil hentes, inden den afspilles. Vi regner da heller ikke download-først-og-afspil-derefter-programmer (fx fildelingsprogrammer) med som netværksmultimedieapplikationer. Med sådanne programmer afspilles multimedieindholdet (lyd eller film) ikke over netværket, men lokalt på computeren. Sådanne programmer er egentlig bare programmer til filtransport.

8.5.1. STREAMING AF LAGRET DATA

Når en bruger streamer en lagret fil over nettet fremfor at downloade den, er det især fordi brugeren ønsker at komme hurtigt i gang med at afspille filen.

Men samtidig ønsker brugeren også

- ◆ at kunne spole frem og tilbage i filen,
- ◆ at afspilningen ikke afbrydes, fordi manglende fildele endnu ikke er downloadet (man ønsker *uafbrudt afspilning*).

Disse to ønsker harmonerer ikke altid med, at hele filen ikke er til rådighed fra start. Derfor venter en streaming-afspillere ofte med at påbegynde afspilning, indtil tilstrækkeligt meget data er opsamlet i bufferen til at uafbrudt afspilning kan lykkes.

ANALOGI

TRAGT OG KALAHAKUGLER

Torben skal have 400 kalahakugler gennem en stor tragt. Hvis tragten er fuld af kugler, plumper der kalahakugler ud af dens munding med en konstant hastighed på 10 kugler per sekund. Torbens kone Gertrud kan desværre kun hælde 5 kalahakugler ned i tragten per sekund. For ikke at opleve at tragten ”går istå” (“mangler kugler”), holder Torben for tragtens munding, indtil tragten er lidt over halvt fyldt op. Så slipper han og lader kuglerne trille igennem. Fordi tragten på det tidspunkt allerede er halvfylt, vil tragten først tømmes helt efter at Gertrud er færdig med at hælde kugler i.

Situationen minder om hvad en streaming-applikation gør for at sikre uafbrudt afspilning:

Streaming-applikation	=	Torben
Server, der streames fra	=	Gertrud
Pakke med streaming-data	=	Kalahakugle
Buffer	=	Tragt
Uafbrudt afspilning	=	Kugler triller igennem tragt med 10 kugler per sekund. Tragten løber først tør, når der ikke er flere kugler.

EKSEMPEL

WEBSIDER MED VIDEOER

De fleste websider med videoer tillader brugerne at streame videoerne. Videoerne er optaget på forhånd og ligger lagret som videofiler på webside-serveren.

Streaming af lagrede filer på det tidspunkt brugeren ønsker det, kaldes også *on demand-streaming*.

Streaming foregår typisk fra en særlig streaming-server.

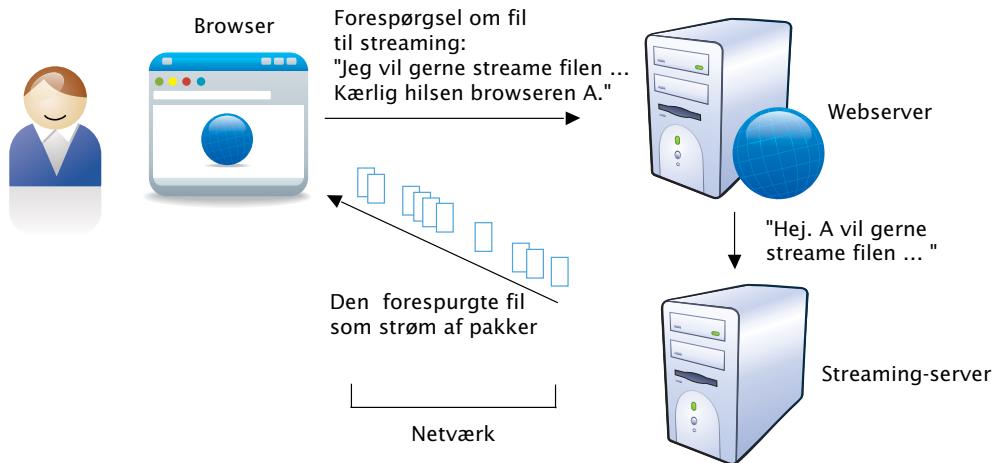


Fig. 8.142. Streaming fra streaming-server.

8.5.2. LIVE STREAMING

Live streaming minder meget om traditionelt *broadcast* af TV eller radio. Forbrugere af den datastrøm, der streames, "tuner ind" (finder den rette internetadresse og påbegynder streaming fra den) og lytter eller kigger med.

EKSEMPEL

INTERNETRADIO OG DIREKTE TV OVER NETTET

Internetradio er et typisk eksempel på lyd, der streames live. Ser man direkte TV-udsendelser overnettet, er der tale om video, der streames live.

Det er klart, at man ikke kan spole frem i en live stream – men man forlanger som bruger uafbrudt afspilning. Vil man spole tilbage i en live stream eller sætte på pause og fortsætte "fra samme sted" senere, er en typisk metode at *optage* live stream'en – lidt som man optager traditionelt TV på video eller radio på kassettebånd. Tilbagespoling og pause er på den måde ikke en del af den egentlige streaming-funktion.

8.5.3. REAL TIME-INTERAKTION

Der er tale om real time-interaktion, hvis brugere skal udveksle lyd eller video i *real time*, altså i princippet “uden forsinkelser”.

EKSEMPEL

IP-TELEFONI OG VIDEOKONFERENCER

Både med IP-telefoni og videokonferencer ønsker brugere at udveksle data i real time.

Under real time-interaktion kan der naturligvis hverken spoles frem, tilbage eller sættes pause.

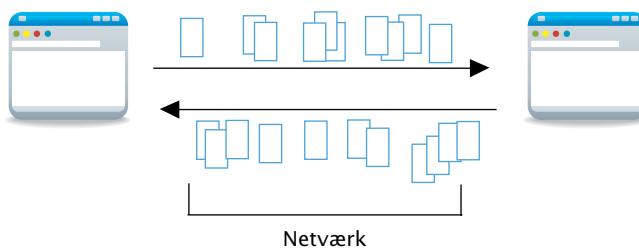


Fig. 8.143. Real-time interaktion: Pakker streames i to retninger samtidigt.

8.5.4. UDFORDRINGEN: BEST EFFORT

For en live streaming- eller real time-interaktiv netværksmultimedieapplikation er det utåleligt med for store forsinkelser. Da netværksmultimedieapplikationer er live eller interaktive, kan man heller ikke uden videre lave en stor start-forsinkelse og så regne med uafbrudt afspilning derefter. Sådanne netværksmultimedieapplikationer bruger derfor typisk *upålidelig transport*, fordi det er *hurtigt* (pålidelig transport er for langsomt).

Som nævnt i kapitel 5 er netværkets vilkår for levering af datapakker en såkaldt “best effort”-service – “så godt som muligt”. Upålidelig datatransmission er også “best effort” – så der er følgende risici ved upålidelig transport af data på netværk:

- ◆ pakker kan forsinkes,
- ◆ pakker kan ankomme i forkert rækkefølge hos modtager,
- ◆ pakker kan tabes.

Ankommer pakker så forsinket, at modtageren irriteres, anses pakken for tabt. Når det kommer til film og lyd, irriteres mennesker typisk ved forsinkelser på mere end 100 – 200 millisekunder.

Hvis pakker ankommer i forkert rækkefølge, er det kun et problem med de pakker, der ankommer for sent i forhold til deres afspilningstidspunkt. Disse pakker kasseres, og kan dermed anses for at være gået tabt.

Forsinkede pakker og pakker, der ankommer ude af rækkefølge, kan altså opfattes som tabte pakker. For en programmør, der er ved at skrive en netværksmultimedieapplikation, gælder det altså om at overvinde problemet *pakketab*. Der er 4 grundideer til at overvinde pakketab i en netværksmultimedieapplikation:

- ◆ repetition
- ◆ interpolation
- ◆ ekstra information i pakker
- ◆ fletning

Med *repetition* erstattes en tabt pakke af den foregående pakke. Da både lyd og film ofte ”lokalt” ikke varierer meget, er strategien ikke helt dum: En gentagen pakke vil med god sandsynlighed ikke se radikalt anderledes ud end den tabte.

Med *interpolation* erstattes en tabt pakke af et passende ”gennemsnit” af de foregående og efterfølgende pakker. Der er mange måder at interpolere lyd og billeder på, og det fører for vidt at komme ind på metoderne her. Igendem: Da lyd og film lokalt ikke varierer meget, er en interpolation typisk et godt bud på indholdet af den tabte pakke. Pakke-interpolation giver typisk bedre brugeroplevelser end pakke-repetetion.

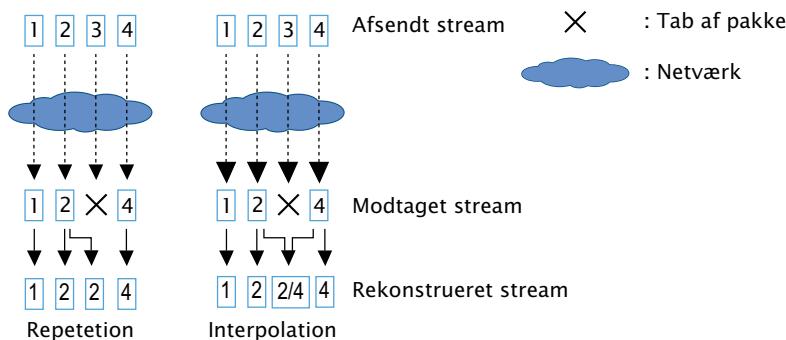


Fig. 8.144. Repetition og interpolation.

Ved at tilføje ekstra information til pakker kan man fx gøre det lettere at skabe en interpolation, hvis en pakke skulle gå tabt. Man kan gå så vidt som til at tilføje hver pakke hele næste pakke i en ”lav kvalitet”. På den måde opleves den tabte pakke kun som en meget kortvarig kvalitetsforringelse.

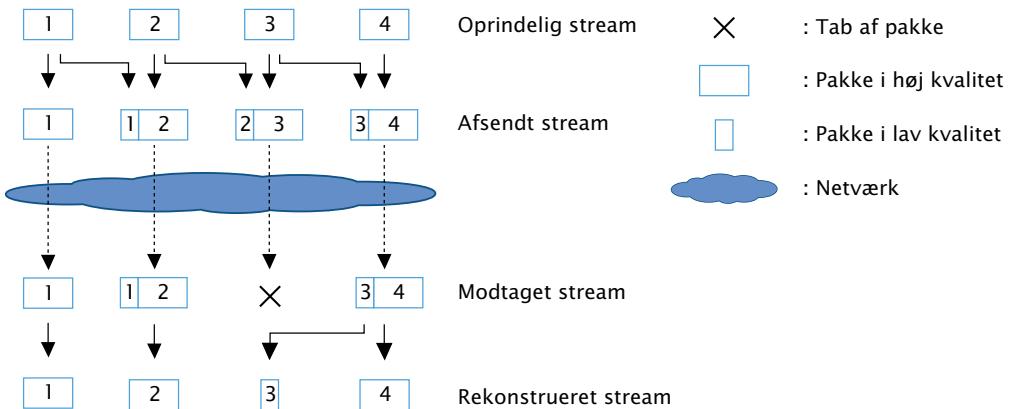


Fig. 8.145. Ekstra information i pakker.

Endelig, med *fletning*, opdeler afsender pakkerne i mindre dele, der flettes til nye pakker inden afsendelse. Modtager ”af-fletter” de indkommende pakker inden afspilning. Tabes en pakke, vil det opleves som flere små tab fremfor et stort tab. Det giver typisk en bedre brugeroplevelse – bedre end både interpolation og repetition.

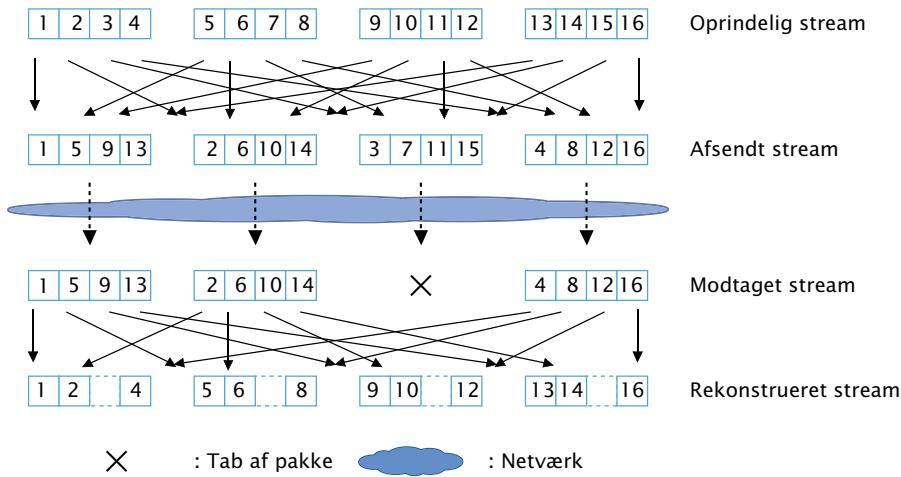


Fig. 8.146. Fletning.

STREAMING AF MIDDAGSMÅLTID

4 forskellige kunder, A, B, C og D, i restaurant Den Gyldne Stream vil hver især have "streamet" den samme menu: et middagsmåltid bestående af forret (suppe), hovedret (300 gram steg) og dessert (tre kugler is).

Første kunde, A, ønsker repetition i tilfælde af, at de upålidelige tjenere skulle tabe en ret på vej fra køkkenet til hans bord. Kunde B ønsker interpolation i tilfælde af en tabt ret. C ønsker ekstra retter i lav kvalitet medsendt hver enkelt ret – hvis nu en ret skulle gå tabt. D kræver fletning af måltiderne.

Det går nu hverken værre eller bedre end at alle fire kunder modtager første ret fejlfrit, anden ret går tabt, og tredje modtages ret fejlfrit. Lad os se på, hvad de enkelte kunder får at spise:

A får suppe, dernæst suppe igen ("repetition"), dernæst tre kugler is.

B får suppe, dernæst en halv portion suppe med halvanden kugle is i ("interpolation"), dernæst tre kugler is

C får først serveret suppe og en ussel kødluns på 40 gram ("næste pakke medsendes i lav kvalitet"), der svagt minder om de 300 grams hovedrets-portioner, kokken laverude i køkkenet. Da C ikke modtager nogen hovedret, ser C sig nødsaget til at spise den usle luns, der blev leveret sammen med suppen, til hovedret. Til sidst får C is.

D får først en trediedel portion suppe, 100 gram hovedrets-steg og en kugle is (retterne er "flettet"). Dernæst taber tjeneren en identisk portion. Til sidst overbringer tjeneren D yderligere en trediedel portion suppe, 100 gram hovedrets-steg og en trediedel portion is. D spiser de to trediedele suppe, dernæst de 200 gram kød og til sidst de to kugler is.

Alle 4 kunder er tilfredse (men nogen måske mere end andre): De fik jo alle en forret, en hovedret og en dessert.

Situationen minder om, hvad en netværksmultimedieapplikation kan gøre for at overkomme pakketab:

4 streaming-typer	≈ Kunderne A, B, C og D
Server, der streames fra	≈ Køkken
Pakke med streaming-data	≈ Ret
Netværk, der streames over	≈ Upålidelige tjenere
Pakketab	≈ Tjener taber ret

8.6. NØGLEORD

- ◆ Multimedie
- ◆ Subtraktiv farvemodel (fx CMYK)
- ◆ Additiv farvemodel (fx RGB)
- ◆ Pixel
- ◆ Farvedybde
- ◆ Print af grafik
- ◆ Bitmap vs. vektorgrafik
- ◆ Digital lyd
- ◆ Bitrate
- ◆ Samplingsfrekvens
- ◆ Tabsfri komprimering
- ◆ Komprimering med tab
- ◆ Streaming

KAPITEL 9

IT-SIKKERHED

Sikkerhed er helt afgørende for mange IT-systemer.

EKSEMPEL

SYSTEMER, HVOR SIKKERHED ER KRITISK

Et usikkert kontrolsystem på et atomkraftværk, i et fly eller i en rumfæргé kan skade mennesker, dyr eller miljø. En usikker hæveautomat kan tillade ondsindede personer at hæve andre folks penge. Et usikkert hospitalssystem kan udstille personfølsomme oplysninger som fx patientjournaler. En usikker tyverialarm på et museum kan tillade tyveri af uerstattelige kunstværker. En usikker hjemmecomputer risikerer at blive inficeret med ondsindede programmer, aflyttet eller overtaget af hackere.

Som eksemplet viser, er IT-sikkerhed en bredt favnende disciplin. Dette kapitel beskæftiger sig kun med de IT-tekniske aspekter af IT-sikkerhed, selvom man med rette kan sige at IT-sikkerhed også handler om fx psykologi, jura og økonomi.

ORDFORKLARING

IT-SIKKERHED

IT-sikkerhed handler om at bygge IT-systemer, der giver mindst mulig risiko for

- ◆ interne fejl,
- ◆ uhedl,
- ◆ ondsindede angreb.

Skulle systemet alligevel udsættes for fejl, uhedl eller angreb, skal IT-sikkerhed sørge for, systemet fungerer pålideligt.

9.1. IT-SYSTEMER OG AKTØRER

Et *IT-system* kan i sikkerhedsmæssig sammenhæng være mange ting – vi vil her skelne mellem tre niveauer:

1. Et hardware- eller softwareprodukt.
2. En enkelt computer.
3. En samling computere i et lokalnetværk.

Hvert IT-system har *en eller flere brugere*.

En bruger af et IT-system har en række rettigheder. Groft sagt sikrer *udvikling af IT-systemer*, at brugere kan udøve deres rettigheder (fx Anna kan læse og skrive i denne fil), mens *sikring af IT-systemer* sørger for, at hverken brugere eller andre kan gøre ting, de ikke har rettigheder til (fx Terese kan ikke slette Annas fortrolige data).

Lad for nemheds skyld en *aktør* betyde enten en person (fx en privatperson eller en ansat i et firma) eller en software-enhed (fx en computervirus eller et antivirusprogram). Begrebet bliver praktisk, fordi vi ønsker at omtale ondsindede personer og ondsindede programmer under et.

9.2. MÅL MED IT-SIKKERHED

Sikkerhed er groft sagt en velfungerende *beskyttelse af værdier*, fx menneskeliv, natur, penge eller kunst. Indenfor IT-sikkerhed er de værdier, der skal beskyttes, *information*, fx kreditkortnumre, kodeord, personfølsom data eller oplysninger om et atomkraftværks IT-sikkerhedssystem. Uretmæssig omgang med eller ødelæggelse af information, der skal beskyttes, er et sikkerhedsbrud. Får en spion fat i statshemmeligheder, kan han videresælge dem til terrorister. Får en tyv fat i et kreditkortnummer, kan han hæve penge, der ikke er hans. Ødelagte konto-data i en bank kan resultere i en personlig fallit – eller at banken skal af med millioner af kroner.

Visse ideer fra traditionel fysisk beskyttelse af genstande kan oversættes til IT-sikkerhed (fx ideen om adgangskontrol). Der er imidlertid en lang række forskelle på traditionel sikkerhed og IT-sikkerhed – vi opnoster et par stykker:

- ◆ Hvis et maleri bliver stjålet, er det væk. Hvis en information aflyttes, fx et kreditkortnummer, har tyven blot en kopi. Man kan altså ikke umiddelbart vide, om ens kreditkortnummer er blevet stjålet, bare ved at tjekke, om man stadig har det.
- ◆ Modsat traditionel sikkerhed kan det være svært at forstå, *hvad* IT-sikkerhed skal beskytte – informationer er mere uhåndgribelige end malerier, guld eller fladskærme.
- ◆ De fleste låser deres bil af, inden de forlader den. Ingen efterlader en 1000-krone-seddel på gaden og regner med at finde den dagen efter. Folk er opmærksomme på traditionelle sikkerhedstrusler (fx tyve). Men mange ved slet ikke, at de bør tænke på IT-sikkerhed.
- ◆ IT-kriminalitet udøves typisk ved at sende “onde bits i en ond rækkefølge” gennem et lille internet-stik i væggen. Så selv om man er opmærksom på at være IT-sikker, kræver det en større teknisk indsigt at blokere for “onde bits” og samtidig lade “gode bits” slippe igennem.

9.2.1. FORTROLIGHED, INTEGRITET OG TILGÆNGELIGHED

Normalt skelner man mellem 3 overordnede typer af informationsbeskyttelse:

- ◆ *Fortrolighed*: Forhindring af, at information uretmæssigt afsløres.
- ◆ *Integritet*: Forhindring af, at information uretmæssigt ændres eller slettes.
- ◆ *Tilgængelighed*: Sikring af, at information altid er tilgængelig og brugbar, når dens retmæssige brugere ønsker.

EKSEMPEL

FORTROLIGHED, INTEGRITET OG TILGÆNGELIGHED: BANKKONTO

Et banksystem beskytter Ingrids konto, herunder hendes kreditkortnummer og saldo. *Fortrolighed* er, at kun Ingrid og hendes bankrådgiver kender til kontooplysningerne – og ikke afslører det til tilfældige personer på gaden. *Integritet* er, at Ingrids saldo er ”som den bør være”: saldoen bliver kun ændret, hvis Ingrid sætter penge ind eller hæver penge på kontoen, og saldoen bliver i så fald opdateret korrekt. *Tilgængelighed* er, at Ingrid kan hæve og indsætte penge – og på andre måder tilgå sin konto – når hun har ret til og brug for det.

EKSEMPEL

FORTROLIGHED, INTEGRITET OG TILGÆNGELIGHED: FORRETNINGSHEMMELIGHED

Det landsdækkende firma DK-Hotdog sælger hotdogs. Chefen udtaenker nu *cold dog'en*. Den kolde hotdog skal lanceres i firmaets pølsevogne inden for et år. Indtil da er opfindelsen en forretningshemmelighed. Firmates chef lægger en tekst-fil med en beskrivelse af cold dog'en på DK-Hotdogs website, så de ansatte kan lære, hvordan cold dog'en tilberedes.

Fortrolighed er, at kun firmaets ansatte kan læse beskrivelsen. Det skal altså sikres at konkurrenten Dansk Skodmad ikke opsnapper beskrivelsen. DK-Hotdogs website skal altså have *adgangkontrol*.

Integritet er, at kun firmaets chef kan ændre i beskrivelsen. Da det er chefen, der har udtaenkt cold dog'en, skal ansatte forhindres i (med vilje, eller ved en fejl) at ændre beskrivelsen til noget forkert.

Tilgængelighed er, at alle firmaets ansatte faktisk kan komme til at læse beskrivelsen når som helst. DK-Hotdogs website skal altså ikke gå ned, og den skal have et system, der sikrer alle medarbejdere adgang.

TIP

CIA-MODELLEN

På engelsk forkortes confidentiality (fortrolighed), integrity (integritet) og availability (tilgængelighed) ofte til ”CIA”, og man taler om *CIA-modellen* for IT-sikkerhed.

9.2.2. TRUSLER, SÅRBARHEDER OG MODMIDLER

Indenfor IT-sikkerhed er det nyttigt at skelne mellem *trusler* mod et system og systemets *sårbarheder*.

ORDFORKLARING

SÅRBARHED

Systemsvaghet.

En sårbarhed kan ofte give en ondsindet aktør uautoriseret adgang til en information.

ORDFORKLARING

TRUSSEL OG TRUSSELSAKTØR

En *trussel* er en potentiel fare for et system. En *trusselsaktør* kan gøre truslen til virkelighed. En trusselsaktør kan være en ondsindet aktør, der kan finde en sårbarhed og udnytte den mod systemet, eller en velmenende aktør, der ved en fejl kan skade et sårbart system.

Overordnet kan man sige, at

$$\text{trussel} = \text{sårbarhed} + \text{trusselsaktør}.$$

TRUSSELSAKTØR	SÅRBARHED(ER)	TRUSLER
Computervirus	Manglende antivirussoftware, sikkerhedshuller i software	Virusinfektion
Hacker	Manglende adgangskontrol	Uautoriseret adgang til information Systemet bliver overtaget
Systembruger	Bruger systemet forkert	Systemnedbrud Tab af data
Ildebrand	Brændbart materiale, ingen ildslukker	Tab af computere og information, måske endda menneskeliv

Fig. 9.147. Eksempler på trusselsaktører, sårbarheder og trusler – der vil blive uddybet i resten af dette kapitel.

Vil man nedsætte risikoen for, at en trussel bliver til virkelighed, må man bygge værn, trussel for trussel. Man kalder sådanne værn for *modmidler*. Man som regel ikke kan fjerne trusselsaktører – de kan være alt fra hackere i fjernøsten eller medarbejdere med rent mel i posen til ondsindende programmer i omløb på internettet udenfor menneskelig kontrol. Modmidler virker derfor som regel ved at reducere et systems sårbarhed.

ORDFORKLARING

MODMIDDEL

Et *modmiddel* mindske en trussel, som regel ved at mindske en sårbarhed.

Et modmiddel kan konkret være mange ting, fx en *firewall*, en softwarekonfiguration eller en bestemt procedure, som brugere af et IT-system skal følge.

SÅRBARHED	MINDSKES AF MODMIDDEL	TYPE AF MODMIDDEL
Manglende antivirussoftware	Antivirussoftware	Software
Manglende adgangskontrol	Firewall	Hardware
Systembruger bruger systemet forkert	Indstil systemet, så brugere har mindst mulig risiko for at begå fejl	Software-konfiguration
Systembruger bruger systemet forkert	Undervis brugere i, hvordan systemet bruges korrekt	Procedurer, brugere skal følge

Fig. 9.148. Eksempler på typer af modmidler.

Selvom sårbarheden "manglende adgangskontrol" mindskes med modmidlet, er den ikke udryddet. En sårbarhed kræver ofte flere modmidler for at blive udryddet. Men selv med uhyrlige bunker af modmidler:

Et IT-system er aldrig 100 % sikkert!

9.2.3. PRIVACY

Man kan opdele IT-sikkerhedsmålet *fortrolighed* i

- ◆ fortrolighed på et *personligt plan*: Evnen til og retten til at beskytte personlige hemmeligheder.
- ◆ fortrolighed på *gruppeplan*: Evnen til og retten til at beskytte hemmeligheder internt i en organisation, branche, virksomhed, institution, kommission, forskergruppe, ...

Fortrolighed på det personlige plan kaldes *privacy* (engelsk for “privat-hed”). Der er mange ting fra privatsfæren, man som enkeltperson (eller familie) ikke ønsker udbasuneret til alle og enhver – fra politisk ståsted, religion, hårfarve, økonomisk status, sygdomme eller medicinforbrug til information om, hvor man befinner sig på hvilke tidspunkter, hvornår man står op og ligger ned, hvad man spiser til morgenmad eller hvad man skriver sms'er til sine kammerater om.

IT-systemernes stadige udbredelse i hverdagsslivet har gjort debatten om privacy aktuel: Hvilen information må websider lagre om deres besøgende? Må en virksomhed overvåge de ansattes brug af internettet? Må politiet aflytte borgeres internetbrug?

Selvom enkeltlænde lovgiver omkring fx lagring af persondata, er der et stykke vej fra at håndhæve disse love til fx at kunne sige, at vi færdes anonymt på internettet:

- ◆ Al internettrafik går gennem en internetudbyder (ISP), der ofte lagrer information om,
 - ◆ hvilke computere, der har kontaktet hvilke andre computere via internettet
 - ◆ hvilke data, de kommunikerende computere har udvekslet
- ◆ Søgemaskiner husker søgninger fra de enkelte computere
- ◆ Internetforretninger husker købsinformation om kunder
- ◆ Sociale online-fora har personadresser mv. lagret
- ◆ ...

Men selvom vi stoler på vores internetudbyder og kun afslører information, vi ønsker at afsløre, til internetforretninger, søgemaskiner, mv. – er der stadigt langt til anonymitet: Blot det *at sende en enkelt pakke* på et netværk afslører information: En pakke fra computer A til computer B afslører fx, at A og B snakker sammen. Hverken A eller B er altså som udgangspunkt *anonyme* – og selve det faktum, at der er trafik imellem A og B, kan observeres. Det kunne sagtens tænkes, at A og B gerne ville snakke sammen, uden at nogen vidste, at der overhovedet fandt en samtale sted. Dette ønske kaldes *ikke-observabilitet*.

Vi har altså identificeret to delaspekter af privacy:

- ◆ anonymitet,
- ◆ ikke-observabilitet.

Som privatperson kunne man ønske sig maksimal privacy. I realiteten er det dog meget svært at undgå at afsætte elektroniske spor, når man fx har med internettet at gøre.

9.2.4. UAFVISELIGHED

Ideen med at underskrive et dokument er, at underskriveren ikke senere kan “løbe fra sin underskrift”. Har man skrevet under på en aftale om at sælge villaen, kan man ikke bagefter afvise at sælge. Det kaldes *uafviselighed*. Uden uafviselighed ville underskriften være værdiløs.

Et tilsvarende problem gør sig gældende i IT-verdenen. Man ønsker at kunne underskrive visse elektroniske informationer på en uafviselig måde. På den måde kan modtager være sikker på, afsender ikke ”løber fra” løfter, der er givet i den elektroniske information. Det kaldes *uafviselighed* og er et aspekt af informationens integritet.

ORDFORKLARING

UAFVISELIGHED

En handling er uafviselig, hvis den, der udfører handlingen, ikke bagefter kan påstå, at vedkommende ikke udførte handlingen.

EKSEMPEL

DIGITAL SIGNATUR

Et underskrift foretaget med en digital signatur er uafviselig.

Vi kommer ind på digital signatur i afsnit 9.5.7.

9.3. SÅRBARHEDER OG TRUSLER

Der er 3 typer trusler mod et IT-systems sikkerhed:

- ◆ *naturskabte*: fx jordskælv, brand, oversvømmelse eller strømafbrydelse
- ◆ *tekniske*: computerne i IT-systemet kan gå i stykker
- ◆ *menneskelige*: mennesker kan
 - ◆ begå fejl,
 - ◆ spionere, stjæle og sabotere.

Vi vil kun beskæftige os med menneskeskabte trusler.

9.3.1. VELMENENDE PERSONER (BRUGERE)

Et computersystem er ikke mere sikkert end de mennesker, der betjener sig af det – brugerne. De færreste brugere har kriminelle hensigter, men kan alligevel

- ◆ begå fejl,
- ◆ være skødesløse med kodeord eller andre følsomme oplysninger,
- ◆ lade sig narre.

Menneskelige fejl tegner sig for størsteparten af alle brud på sikkerheden i ethvert IT-system. En stor del af sådanne brud på sikkerheden sker, hvor naive eller fortravlede brugere afgiver følsomme oplysninger. Følsomme oplysninger afgives typisk på to måder:

- ◆ *Intetanende*: Det er svært at undgå at afgive oplysninger, hvis man færdes på internettet: Fx kan websteder sagtens lokke en masse information om en internetsurfers computer ud af hans browser, og offentliggør man et dokument på internettet, kan det indeholde information om forfatterens computer og det netværk, den er tilkoblet.
- ◆ *Frivilligt*: Brugere narres til at udføre falske instruktioner – som at åbne en bestemt website, åbne en vedhæftet fil i en email, angive et kodeord, el.l. Websiden eller den vedhæftede fil indeholder ondsindet kode, der udnytter de følsomme oplysninger, som brugeren har afgivet frivilligt. Brugerne lader sig narre, fordi instruktionerne tilsyneladende kommer fra fx IT-afdelingen i deres firma, en ven fra adresselisten, el.l. En ondsindet aktør udgiver sig altså for at være en bekendt – det kaldes *social engineering*.

9.3.2. ONDSINDEDE PERSONER

Der er ikke én bestemt type person, der skriver ondsindede programmer: Der er unge og gamle, professionelle og amatører, europæere og asiater, rige og fattige. Men *hvorfor* overhovedet skrive ondsindet kode? Her er nogle af de væsentligste svar på spørgsmålet:

- ◆ *Penge*: Computerbrugere verden over har i stigende grad brug for at logge ind på forskellige portalér og betale med kreditkort i netbutikker. Ved at stjæle kodeord eller kreditkortnumre kan ondsindede personer stjæle penge. De kan også videresælge information eller lave mere organiseret kriminalitet.
- ◆ *Politiske årsager*: Er man utilfreds med det politiske indhold på en webside, kan man jo bare gøre websiden utilgængelig eller ændre indholdet på den – så er den klaret! Det er ulovligt, men ikke desto mindre en almindelig type angreb.
- ◆ *Berømmelse*: Både unge og erfarne programmører kan ønske berømmelse eller anseelse – og forsøger at opnå det ved at skrive en effektiv computervirus (uheldigt!).
- ◆ *Forskning og udvikling af antivirus-software*: Professionelle programmører udfordrer konstant sikkerheden i eksisterende operativsystemer, browsere og andre programmer. Finder de et sikkerhedshul, sørger de for at lave sikkerheds-software, der kan lukke hullet. På den måde kan man være på forkant med de IT-kriminelle.

En *hacker* er en person, der tiltvinger sig uretmæssig adgang til et IT-system.

9.3.3. KODEORD

Kodeord giver adgang til information, som kun kenderen af kodeordet bør have adgang til. Alligevel regnes kodeord ikke for en særlig sikker adgangskontrol – mange brugere

- ◆ *bruger gætbare kodeord* som fx konens fødselsdag eller hundens navn.
- ◆ *bruger svage kodeord*, der kan knækkes på kort tid med en kodeordsknækker. Et kodeord regnes for svagt, hvis det er kort, eller gætbart, eller danner eksisterende ord. Fx er “?}_1”, “Bastian” og “Jegerenfalk” svage kodeord.
- ◆ *skriver kodeord ned* for at huske dem. En ondsindet aktør kan finde det nedskrevne kodeord og misbruge det.
- ◆ *afslører kodeordet* til ondsindede personer, der udgiver sig for at være en anden end de er (social engineering).
- ◆ *bruger samme kodeord flere steder*: De fleste skal huske en stor bunke kodeord til mange forskellige systemer – PIN-kode, kode til email, kode til arbejdscomputeren, mv. For at kunne huske kodeordene, bruges så vidt muligt det samme kodeord alle steder. Det gør det lettere for en angriber at knække kodeordet.

De bedste kodeord er genereret tilfældigt – på den måde kan de ikke sammenkædes med brugeren. Fx er kodeordene

“_t[71WLNiNX”, “x?[3qD€ct(GnM” og “+(LaNqZ-/67-8”

genereret tilfældigt og meget lidt gætbare. Til gengæld er de svære at huske. En endnu bedre løsning er *smart cards*, der genererer et nyt, tilfældigt kodeord hver gang, kodeordet skal bruges. På den måde undgås genbrug af kodeord fuldstændigt.

9.3.4. SOFTWARE

Mange sårbarheder opstår, fordi eksisterende programmer ikke er sikkert kodet. Der kan være *bugs* (deciderede fejl i koden), manglende sikkerhedstjek eller bagdøre, som (den ellers velmenende) programmør har glemt at lukke. Ondsindede aktører kan undersøge software-usikkerhederne og udnytte dem til målrettede angreb.

EKSEMPEL

INTERNETBROWSERE

I 2004 rådede CERT (*US Computer Emergency Readiness Team*, en IT-sikkerhedsorganisation) internetbrugere til at bruge en hvilken som helst anden browser end Internet Explorer, fordi der var så mange sikkerhedshuller i programmet. Selvom andre browsere i principippet var lige så usikre, havde Internet Explorers daværende allestedsnærværelse gjort programmet til et oplagt hackermål. Hvert år finder man hundredevis af sikkerhedshuller i aktuelle internetbrowsere, som skyldes forhastet eller uigenremtænkt programmering. Sådanne huller lukkes ved at installere sikkerhedsopdateringer.

9.3.5. MALWARE

Malware er en samlebetegnelse for ondsindet kode. Ordet er en sammentrækning af *malicious* (engelsk for ondsindet) og *software*. Der findes adskillige typer malware, hvoraf nogle af de mere kendte omfatter

- ◆ virus,
- ◆ orme,
- ◆ spyware,
- ◆ trojanske heste.

At kode er *ondsindet*, betyder at koden griber ind i et systems stabilitet og sikkerhed på en uønsket måde, fuldstændig som en sygdom griber ind i en krops sundhed og stabilitet. Påvirkningerne af malware kan variere:

- ◆ *mindre påvirkninger*: små forstyrrelser og systemlangsomhed,
- ◆ *alvorlige påvirkninger*: sletning af programmer eller data, ændring af indstillinger i operativsystem,
- ◆ *katastrofale påvirkninger*: ændring af data, omfattende forsinkelser og forstyrrelser, udspionering eller videresalg af følsomme data.

En *virus* er et lille stykke programkode, der blot ønsker at reproducere sig selv. For at gøre det inficerer virusen et andet program ved at vedhæfte en kopi af sin programkode i programmet. Udover at sprede sig selv og dermed tvinge folk til at spilde tid på fjerne den, kan virusen have direkte skadelige virkninger, som fx at slette data.

EKSEMPEL

CREEPER

Den første (veldokumenterede) virus dukkede op lang tid før internettet blev folkej: I starten af 1970'erne opdagede amerikanske militærfolk virusen Creeper på deres interne datanetzværk, ARPA-nettet. Virusen gjorde ingen egentlig skade, men udskrev følgende besked på skærmen: "I'm the creeper! Catch me if you can!" – hvorefter den kopierede sig selv videre til yderligere computere på netværket.



Fig. 9.149. Virus, orm og trojansk hest.

I modsætning til en virus er *en orm* et selvstændigt program. Uafhængigt af andre programmer spredt ormen sig selv til andre computere – via fx adresselister i emailprogrammer.

EKSEMPEL

DEN KÆRLIGHED, DEN KÆRLIGHED

En berømt computer-orm hed "I love you" og oversvømmede internettet i år 2000. Ormen ankom til intetanende ofre som en email med overskriften "I love you", tilsyneladende fra en bekendt i adresselisten. Nysgerrige emailbrugere, der åbnede den vedhæftede fil i emailet, blev inficeret. Ormen overskrev musik, film, tekstdokumenter mv. med teksten "I love you!" og sendte sig selv videre til alle i emailprogrammets adresseliste. Mere end 45 millioner computerbrugere verden over blev ramt. I Danmark blev computersystemer hos Tele Danmark, TV2 og Folketinget sat ud af drift.

Spyware er spion-software (deraf navnet). Spyware installerer sig på en computer uden brugerens viden. Hvor en virus- eller ormeinfektion typisk vil afsløre sig selv hurtigt, skjuler spywaren sig, folger med i brugerens færden på nettet og kan i principippet registrere al information, brugeren afgiver – fx kodeord og personfølsom data. Den indsamlede information bruges til at lave en personlig profil af brugeren. Spywaren sender som regel den oprettede profil til et tvivlsomt firma (som har lavet spywareprogrammet).

- ◆ I bedste fald bruger firmaet profilen til at målrette fx reclamer eller email og sender det til brugeren. Spywaren er altså skyld i spildtid og spild af computerkrafter.
- ◆ I værste fald sender offentliggør eller videresælger firmaet de personlige oplysninger, stjæler penge eller afpresser brugeren.



Fig. 9.150. Spyware.

Trojanske heste er opkaldt efter den berømte træhest, grækerne indtog byen Troja med: Grækerne byggede en stor, hul træhest. En håndfuld græske soldater gemte sig i den, mens resten af den græske hær gemte sig udenfor bymuren, ude af synet for trojanske udkigsposter. Trojanerne troede nu, at grækerne havde opgivet deres langvarige belejring af byen, var rejst hjem og havde efterladt hesten som afskedsgave. Byporten blev åbnet, hesten trukket ind og så blev sejren ellers fejret med betragtelige mængder vin. Samme nat krvlæde grækerne lydløst ud fra hestens bug og åbnede byporten. Den udhvidelede græske hær kunne da frit strømme ind i byen og besejre de døddrukne modstandere.

En trojansk hest er et ondsindet program (≈ grækere), som programmøren har gemt i et uskyldigt udseende program (≈ træhesten). Hvis man intetanende kører det “uskyldige” program, installeres det skjulte program, som derefter kan udføre uønskede handlinger. Det kunne fx

- ◆ opføre sig som spyware,
- ◆ oprette en “bagdør”, som giver afsenderen adgang til computeren,
- ◆ sætte computeren ud af drift,
- ◆ slå antivirusprogrammer og firewalls fra, inden et større angreb sættes ind.

En trojansk hest spredes ved at godtroende brugere installerer “hesten” uden at opdage “grækerne” – på den måde er trojanske heste anderledes end virus og orme, der er installerer sig selv (virus og orme er *selv-replikerende*).

EKSEMPEL

FALSK LOTTOSPIL

En trojansk hest kunne fx ankomme i form af en email, angiveligt fra en kammerat, med en opfordring til at hente og installere et sjovt lille lottospil fra internettet. Spillet hentes og installeres, og alle er glade – et stykke tid. Imidlertid var emailens afsender ikke den kammerat, han udgav sig for at være – brugeren, der installerede spillet, var utsat for social engineering. Spillet er en trojansk hest og installerede en bagdør, da det blev installeret. Bagdøren giver den trojanske hests afsender fuld kontrol over den inficerede computer: Alle tastetryk (fx kodeord) kan overvåges, angriberen kan ændre systemfiler, lytte med på mikrofonen eller kigge med på den inficerede computers webcam.

9.3.6. SPAM

Spam er uønsket information, fx emails, reklamer eller pop-up-vinduer. Spam er ofte ganske ufarligt, men det irriterer og tager tid. Åbnes en spam-email, risikerer man ikke blot at spilde tid på fx løgnagtige tilbud i emailet, men også at der bliver sendt information tilbage til afsenderen om, at adressen er aktiv. Resultatet: mere spam.

9.3.7. KOMMUNIKATION OVER NETVÆRK

Netværkssikkerhed handler om at *kommunikere sikkert* over et netværk. Kommunikationen kan fx være udveksling af kærestebreve mellem hemmelige elskende eller HTTP-forespørgsler til en webserver. Alt efter sammenhængen kræver sikker kommunikation mellem aktørerne A og B, at forskellige (eller alle) af følgende sikkerhedsål er opfyldt:

- ◆ *fortrolighed*: Uvedkommende kan ikke opsnappe kommunikationen.
- ◆ *integritet*:
 - ◆ *data-integritet*: Indholdet af en besked fra A til B kan ikke ændres under transporten over netværket.
 - ◆ *autenticitet*: En uvedkommende aktør C kan ikke udgive sig for at være A eller B.
 - ◆ *uafviselighed*: Hvis A sender B en besked, kan A ikke bagefter nægte det.
- ◆ *tilgængelighed*: Uvedkommende kan ikke afbryde A's og B's mulighed for at kommunikere.

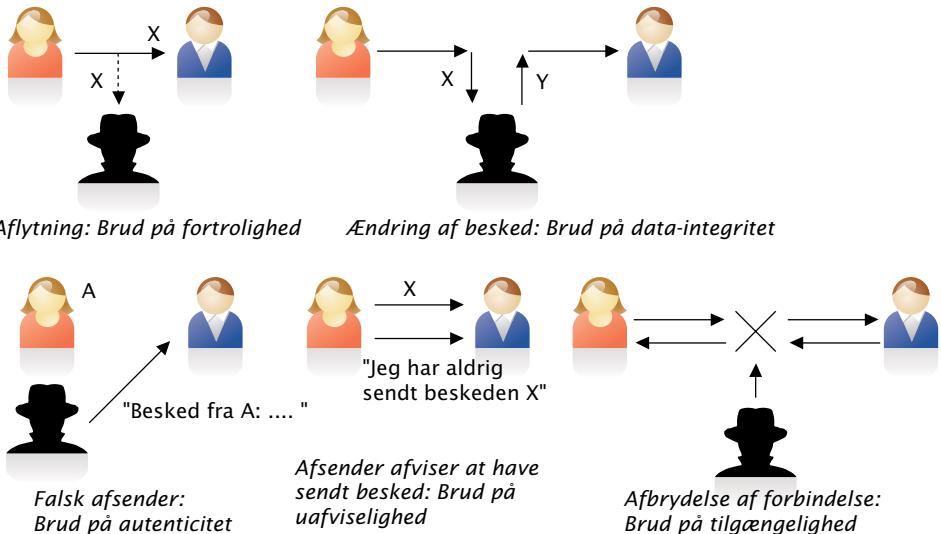


Fig. 9.151. Trusler: Brud på fortrolighed, integritet og tilgængelighed.

Inden du læser resten af dette afsnit, kan det være en god ide at repetere kapitel 5 (især begreberne protokol, port og netværkslag).

Mange netværk har sårbarheder, der muliggør trusler mod sikker kommunikation, fx:

- netværket kan bruge usikre protokoller,
- netværket kan tillade aflytning ude- eller indefra,
- netværket kan være "for åbent" – fx acceptere potentielt skadelig kommunikation på porte, der ikke bruges til standardformål.

Hvilke konkrete trusler findes der så?

Packet sniffing: Pakker i netværkslaget kan blive aflyttet eller kopieret, uden at det bliver opdaget. Ondsindede aktører kan dermed

- opsnappe fortrolig kommunikation,
- indsamle information om et IT-system og bruge det til at udføre et detaljert angreb.

Spoofing og phishing: En ondsindet aktør udgiver sig for at være en anden, end han er. Man bruger som regel ordet spoofing om "aktivt" bedrag af denne art, fx email-henvendelser med falsk afsender, mens phishing bruges om mere "passivt" bedrag, fx at oprettholde en falsk netbanks-webseite, der lokker folk til at indtaste kreditkortinformationer.

Replays: Et replay-angreb forklares bedst ved et eksempel: Lad os sige, at A er en netbank-kunde og B en netbank. A henvender sig nu til netbanken og oplyser kodeord mv. A betaler sit kontingent til tennisklubben. Derefter logger A af netbanken og tager til tennis. Hvad A og B ikke har opdaget, er at hackeren C har "optaget" A's logon-information til netbanken B – ved packet sniffing. Lidt senere, mens A træner sin serv, "afspiller" C den del af "samtalens" mellem A og B, der bestod i, at A loggede på netbanken. Heraf navnet "replay" – C genafspiller dele af en kommunikation. Netbanken B opfatter C's henvendelse som om A henvender sig igen, og den giver adgang til A's konto. Resultatet er, at C har adgang til A's netbank!

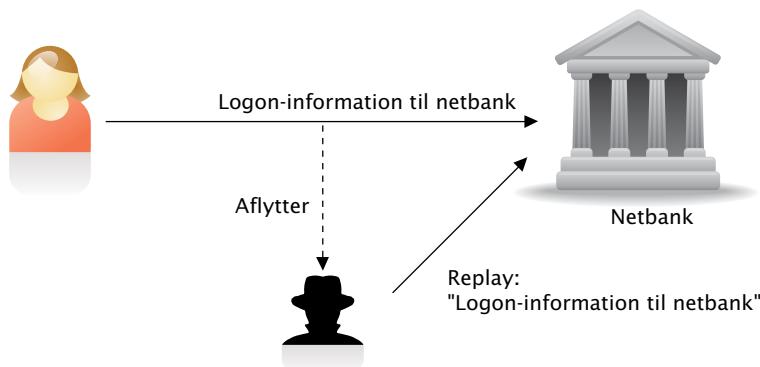


Fig. 9.152. Replay-angreb.

Man in the middle-angreb (forkortet: MITM-angreb): Hvis en ondsindet aktør C kan opsnappe al information, der udveksles mellem A og B, kan han lave et MITM-angreb – C er "manden i midten". Fx kan C opsnappe en besked fra A, ændre den, og sende den videre til B, uden at B opdager, at beskeden ikke længere er fra A.

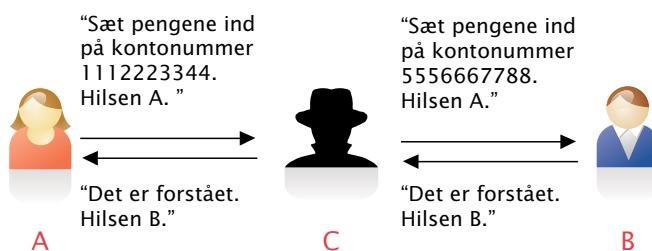


Fig. 9.153. Man in the middle-angreb.

Hvis en man-in-the-middle kaprer en forbindelse fuldstændigt, og fx smider B af, kaldes det en *session hijacking* (at “hijacke” betyder at kapre).

DoS: Ondsindede aktører kan umuliggøre kommunikation ved fx at “lægge en webserver ned”. Det gøres ved at genere webserveren med så mange henvendelser, at den ikke kan passe sit job overfor andre klienter. Sådanne afbrydelser kaldes *Denial of Service* (DoS), fordi parter, der ønsker at kommunikere, nægtes adgang til det.

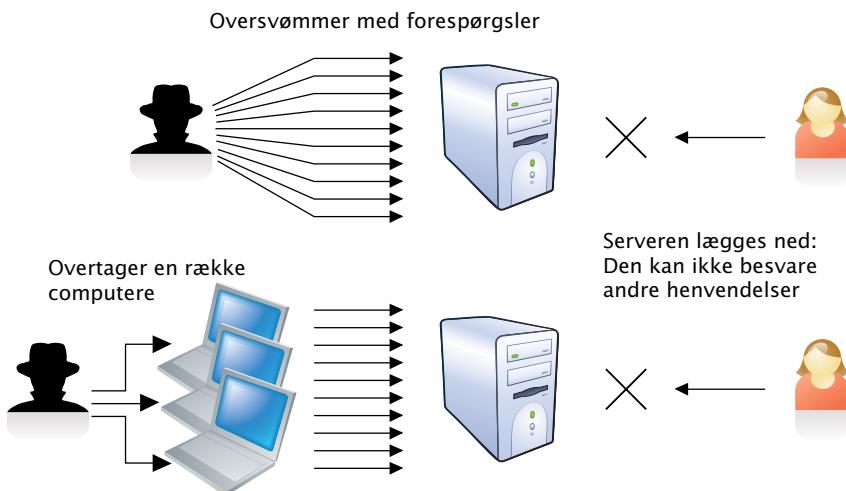


Fig. 9.154. DOS og DDoS.

En særlig ondsindet type DoS-angreb er såkaldte *distribuerede DoS-angreb* (DDoS), hvor en hacker kaprer en række computere til at hjælpe sig med et målrettet angreb.

9.3.8. TRÅDLØSE NETVÆRK

Trådløse netværk har ikke bare alle de samme sårbarheder som kabel-netværk, men også en lang række ekstra. Vi gennemgår nogle af de væsentligste sårbarheder og trusler:

Traditionelt har trådløse netværk *tilladt enhver at koble sig på* – uden nogen form for tjek. Denne sårbarhed giver angribere adgang til netværket.

Trådløse netværk er *lette at opdage, aflytte og koble sig på* – modsat et kabelnetværk, som man kun koble sig på med et kabel, der er i fysisk i kontakt med en netværksenhed. En bygning med et trådløst netværk har – med en metafor – “netværkskabler hængende ud af vinduerne i store klaser”, og forbipasserende kan bare slutte ondsindede apparater til disse “netkabler”.

Trådløse netværk er *lette ofre for DoS-angreb*: Et trådløst netværk er følsomt over for støj på de frekvensområder, der bruges til data-transmission.

Ondsindede trådløse netværk kan opstilles, fx nær eksisterende trådløse netværk. Hvis brugere ved en fejl benytter sig af det ondsindede netværk, fx i stedet for det netværk, de troede de brugte, kan de afsløre fortrolig information. Indenfor trådløse netværk omtaler man problemet – altså det, at man som aktør på et trådløst netværk ikke kan vide, om anden aktør er en ”ven” eller en ”fjende” – med det farverige navn ”De Byzantinske Generalers Problem”.

WWW

TRÅDLØSE NETVÆRK

På **WWW** finder du links til mere viden om trådløse netværk – og også en forklaring af, hvor de byzantinske generaler kommer ind i billedet.

9.4. MODMIDLER

For at mindske et IT-systems sårbarheder kan en lang række modmidler tages i brug. Modmidler kan groft sagt opdeles i

- *fysiske*, fx overvågningskameraer, pigtrådsindhegning og sprinkleranlæg,
- *tekniske*, fx antivirussoftware, firewalls og kryptering,
- *menneskelige*, fx agtpågivenhed og sikkerhedsprocedurer.

Yderligere kan modmidler inddeltes i

- *protektive*: beskytter mod brud på sikkerheden,
- *detektive*: kan opdage brud på sikkerheden,
- *reaktive*: kan genoprette sikkerheden, hvis der er sket et brud på sikkerheden.

EKSEMPEL

TYPER AF MODMIDLER

En pigtrådsindhegning er fysisk-protektiv. Typisk antivirus-software er både teknisk-protektiv, teknisk-detektiv og teknisk-reakтив. Sikkerhedsprocedurer, der følges til punkt og prikke af alle brugere, er et menneskeligt-protektivt modmiddel. Et sprinkleranlæg er fysisk-detektivt og fysisk-reaktivt.

9.4.1. AUTORISATION

Fortrolighed, integritet og tilgængelighed kan alle formuleres som egenskaber ved *autoriseret adgang* til information:

- ◆ *fortrolighed*: Kun personer og systemer, der er autoriseret til at læse informationen, har adgang til informationen.
- ◆ *integritet*: Kun personer og systemer, der er autoriseret til at ændre eller slette informationen, har adgang til informationen.
- ◆ *tilgængelighed*: Alle personer og systemer, der er autoriseret til at læse og/eller ændre informationen, kan komme til det.

En såkaldt *sikkerhedspolitik* for IT-systemet definerer, hvem der er autoriseret til at gøre hvad. Som udgangspunkt siger man, at en IT-politik bør følge princippet om *least privileges* (engelsk for “færrest privilegier”): Enhver aktør er kun autoriseret til netop det, han/hun/den/det har brug for at være autoriseret til.

EKSEMPEL

LEAST PRIVILEGES

Lad os kigge på IT-systemet “et operativsystem”. Her fastlægger operativsystemet, hvilke autorisationer de enkelte brugerprogrammer har. En webbrowser (der er et brugerprogram) bør ifølge principippet om least privileges ikke være autoriseret til at ændre i systemfiler – det er ikke en nødvendig autorisation for at webbrowseren kan fremvise webindhold. Et operativsystem, der giver en webbrowser for kraftfulde autorisationer – fx adgang til at ændre systemfiler – udsætter sig selv for fare: Narres en bruger til at åbne en skadelig webside, kan websiden narre browseren til at ændre fatalt i systemfilerne. Det kan ikke ske med least privileges.

9.4.2. ADGANGSKONTROL

Vi har lige konstateret, at uautoriseret adgang er en meget generel trussel mod IT-sikkerhed. Det er derfor ingen overraskelse, at *adgangskontrol* er et centralet modmiddel. En adgangskontrol er protektiv og kan være af fysisk, teknisk eller menneskelig art.

Når en aktør vil tilgå et system med adgangskontrol, afgør adgangskontrollen, om aktøren er autoriseret:

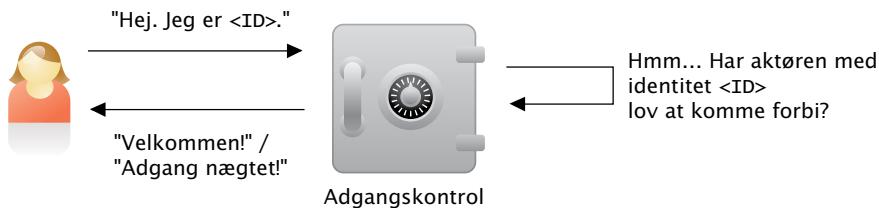


Fig. 9.155. Autentificering 1.0 (protokol). 1) Aktøren oplyser sin identitet <ID>. 2) Adgangskontrol-len tjekker, at aktøren med identitet <ID> har adgang til systemet.

Adgangkontrollens afgørelse kaldes en *autentificering* – en autentificering kan enten resultere i et “velkommen” eller et “adgang nægtet”.

De to skridt i en autentificering, som Figur 9.155 viser, er imidlertid ikke tilstrækkelige i sig selv: En ondsindet aktør A kan blot oplyse <B-ID> for en anden aktør B, der har de adgangsrettigheder, A ønsker at gøre brug af til onde formål. Derfor suppleres en identifikation altid med ekstrainformation, *autenticitetsinformation*. Adgangskontrolen kan nu sammenholde identifikation med autenticitetsinformation og derved afgøre, om aktøren virkelig er den, vedkommende giver sig ud for at være – altså, at det oplyste <ID> er autentisk.

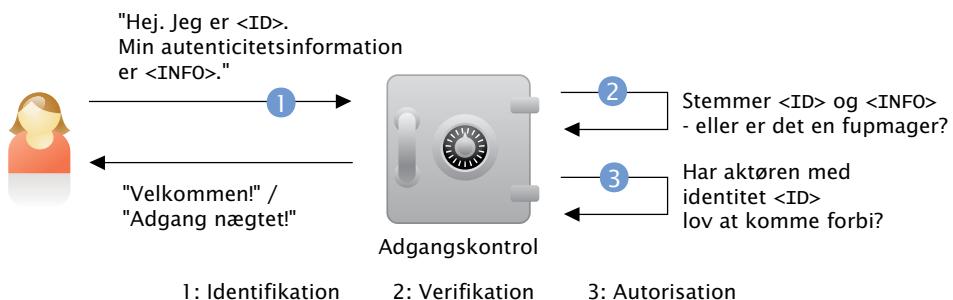


Fig. 9.156. Autentificering 2.0 (protokol).

Den reviderede autentificeringsprotokol har altså følgende tre skridt:

1. *Identifikation*: Aktøren oplyser identitet <ID> og autenticitetsinformation <INFO>.
2. *Verifikation*: Adgangskontrollen tjekker, at den oplyste autenticitetsinformation <INFO> stemmer med den oplyste identitet <ID>, sådan at aktørens identitet virkelig er <ID>.
3. *Autorisation*: Adgangskontrollen tjekker, at aktøren med identitet <ID> har adgang til systemet.

Vi opsummerer:

$$\text{autentificering} = \text{identifikation} + \text{verifikation} + \text{autorisation}.$$

En særlig type autenticitetsinformation er den digitale signatur (se afsnit 9.5.7).

EKSEMPEL

BRUGERNAVN OG KODEORD

Mange brugersystemer tilgås gennem en adgangskontrol, hvor brugernavn og kodeord skal indtastes. Her består autentificeringen i

- ◆ identifikation = opgivelse af brugernavn <ID>,
- ◆ autenticitetsoplysning = opgivelse af kodeord <INFO>,
- ◆ verifikation = tjek af, at <ID> er en gyldig bruger, og at <ID> har kodeord <INFO>.

Hvis et ugyldigt brugernavn oplyses, vil adgangskontrollen nægte adgang med begründelsen "bruger ukendt". Oplyses et korrekt brugernavn, men en forkert adgangskode, vil adgangskontrollen opfatte den oplyste identifikation som ikke-autentisk – fordi der er uoverensstemmelse mellem brugernavn og kodeord. I denne situation nægter adgangskontrollen adgang med begründelsen "kodeord forkert".

EKSEMPEL

BIOMETRISK AUTENTIFICERING

Et IT-system kan bruge biometrisk autentificering af mennesker. Det betyder, at unikke biologiske egenskaber måles som autenticitetsoplysniger, når mennesket præsenterer sin identitet. De biologiske egenskaber, der måles, kan fx være øjets irismønster eller fingeraftryk.

Den opmærksomme læser har allerede nu opdaget, at autentificeringsprotokollen Autentificering 2.0 i Figur 9.156 har et sikkerhedshul! Den er nemlig sårbar overfor *replay*-angreb, som vi hørte om i afsnit 9.3.7 (hvorfor?). For at undgå *replay*-angreb må man tage skrappe midler i brug – bl.a. kryptografi – vi kommer nærmere ind på hvordan i afsnit 9.5.6.

9.4.3. KRYPTOGRAFI

Du har sikkert hørt om kryptografi – læren om at kryptere. Kryptering er helt afgørende for IT-sikkerhed. Med rigtigt anvendt kryptering kan man opnå:

- ◆ fortrolighed,
- ◆ data-integritet,
- ◆ autenticitet af afsender- og modtager-identitet,
- ◆ uafviselighed.

EKSEMPEL

LØNFORHØJELSE

Du er ansat i Krypto Labs DK. En dag får du en email fra chefen, der fortæller dig, at din løn stiger til 100.000 kroner om måneden. Fordi der er brugt kryptering (på den rigtige måde), kan du være sikker på, at emailen faktisk kom fra din chef (autenticitet), at emailens indhold ikke er blevet ændret, siden din chef afsendte den (data-integritet), at ingen andre har læst emaileten, mens den rejste fra din chefs computer over netværket til din computer (fortrolighed), og at din chef ikke kan nægte at have sendt emaileten, når han senere ombestemmer sig (uafviselighed).

Med “rigtigt anvendt” menes, at kryptering skal bruges i en sikkerhedsprotokol uden sikkerhedshuller – ellers er krypteringen værdiløs. Vi kommer ind på kryptering i afsnit 9.5.

9.4.4. ANTIVIRUS-SOFTWARE

Et udbredt modmiddel mod truslen “malware” er antivirus-programmer, som kan opdage malware, inden det når at inficere en computer i et IT-system – eller kurere systemet, hvis det er blevet inficeret. Malwaren uskadeliggøres eller, bedre, fjernes.

Sådanne programmer har to primære metoder til at opdage malware:

- ◆ *Scan efter malware*: Systemet scannes jævnligt efter mønstre, der matcher virusdefinitioner i en opdateret oversigt over kendt malware.
- ◆ *Overvåg systemet*: Hvis mistænkelige begivenheder indtræffer – hvis fx filstørrelser ændres uden grund, eller programmer holder op med at virke.

Hvis en mistænkelig fil eller mistænkelig program-opførsel opdages, sørger antivirus-programmet for at uskadeliggøre filen eller programmet – eller alarmere brugeren.

9.4.5. FIREWALLS

Som vi ved, er der “gode” og “onde” aktører på internettet. Med et IT-system, der er koblet på internettet, er det afgørende at forhindre ondsindende aktører udefra i at tiltvinge sig adgang. Ligesom man i middelalderen beskyttede borge med en voldgrav og en vindebro, beskytter man nu om dage IT-systemer med *firewalls*. En firewall er protektiv.

ORDFORKLARING

FIREWALL

En kombination af hardware og software, der afgrænsler et internet-opkoblet IT-system fra resten af internettet. Firewallen tillader noget trafik at passere og blokerer for anden trafik.

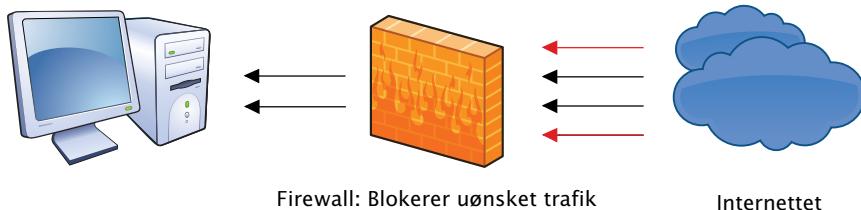


Fig. 9.157. Firewall.

Ideen med en firewall er, at den kun lukker ”velkommen” trafik ind i IT-systemet – det kaldes *filtrering*, fordi trafikken sluses gennem et filter. En firewall frasorterer en stor del af den uønskede trafik, men kan aldrig give fuldstændig beskyttelse.

TIL SÆRLIGT INTERESSEREDE

FILTERING I FLERE LAG AF PROTOKOLSTAKKEN

Firewalls filtrerer ofte på flere niveauer i protokolstakken:

- ◆ *pakkefiltrering i netværkslaget*: Firewall'en inspicerer de enkelte ankommande pakker. Hvis en pakke kommer fra en tvivlsom adresse, har et ukendt format, genkendes som skadelig, eller vil kontakte en port, der ikke ellers bruges, kasseres pakken.
- ◆ *beskedfiltrering i applikationslaget*: Det kan være nødvendigt at filtrere trafik på basis af fx enkeltbrugeres identiteter. Sådan filtrering kræver information fra applikationslaget.

Udover at filtrere fører de fleste firewalls en sikkerhedslog (se afsnit 9.4.10).

9.4.6. INTRUSION DETECTION-SYSTEMER (IDS)

Kan man ikke forhindre indtrængen, kan man forsøge at opdage forsøg på indtrængen. Et *intrusion detection system* (IDS) kan opdage brud på sikkerheden gennem overvågning – og er altså et detektivt modmiddel. Der findes mange typer IDS, bl.a.:

- ◆ *Fysisk IDS*: Opdager fysisk indtrængen. Fx er tyverialarmer og overvåningskameraer fysiske IDS'er.
- ◆ *Netværks-IDS*: Opdager uautoriseret indtrængen på et netværk ved at overvåge netværkstrafik.
- ◆ *Computerbaseret IDS*: Opdager uautoriseret indtrængen på en computer. Et antivirus-program er typisk et computerbaseret IDS.

9.4.7. SPAMFILTRE

Spamfiltre scanner indkommende emails med en opslagsteknik, der minder om virke-måden for antivirussoftware og IDS-systemer. Hvis en scanning resulterer i et match i en opdateret oversigt over kendt spam, kasseres emailen.

9.4.8. AWARENESS

De fleste brud på IT-sikkerhed skyldes menneskelige fejl. Menneskelige fejl kan i et vist omfang undgås ved at uddanne og oplyse brugere af et IT-system om, hvilke trusler der lurer. En bruger, der forstår farer og risici, får skaerpet sin agtpågivenhed – og opfører sig dermed mere sikkert og bliver sværere at narre til fx at udføre falske ordrer. Denne agtpå-givenhed kaldes med et engelsk ord for *awareness*.

Folk, der fremstiller IT-systemer, højner IT-systemets sikkerhed ved fra start at udvise awareness. Fx laver programmører, der er opmærksomme på at programmere sikkert, mere sikre programmer: De anvender sunde principper for softwarearkitektur, undgår kendte sikkerhedsproblemer og tester programmer for at undgå bugs, der sidenhen kan udnyttes af ondsindede aktører.

9.4.9. FYSISK SIKKERHED

Fysiske sikkerhedsforanstaltninger beskytter et IT-system mod ulykker som indbrud, gasudslip eller ildebrand på det sted, IT-systemet fysisk er. Fysiske sårbarheder mindskes med åbenlyse modmidler som fx fysiske adgangskontroller – det kunne være svært bevæbnede dørvagter. Også detektive og reaktive modmidler kan tages i brug, fx overvågningskameraer, nødudgange, backup-servere, ildslukkere og nødstrømanlæg.

9.4.10. SPORBARHED

Som vi har set, kan en adgangskontrol medvirke til at forhindre uønskede begivenheder. Virkeligheden viser desværre, at vi ikke med 100 % sikkerhed kan forhindre alle uønskede begivenheder:

- ◆ selvom kun autoriserede handlinger udføres i et system, kan det alligevel føre til brud på sikkerheden
- ◆ enhver adgangskontrol, uanset hvor smart og gennemtænkt den er, kan snydes af en udspukuleret misdæder

Et fornuftigt ekstra krav at stille indenfor IT-sikkerhed er derfor, at enhver aktør stilles til regnskab for sine egne handlinger. Det kræver viden om, *hvem der har gjort hvad hvornår*.

ORDFORKLARING

SPORBARHED

Hver gang en begivenhed indtræffer, der er relevant i forhold til et systems IT-sikkerhed, noteres begivenhed, tidspunkt og aktør i en særlig systemfil, *sikkerhedsloggen*.

Sikkerhedsloggen er en slags “sikkerheds-dagbog” for systemet.

EKSEMPEL

SPORING

Følgende er eksempler på ting, det kunne være relevant at logge i en sikkerhedslog for et lokalnetværk af computere på fx en skole:

- ◆ Tidspunkt, dato og logon-ID for alle forsøg på at logge på en computer.
- ◆ Brug af administrator-konti.
- ◆ Brug af hardware – hvem har printet hvornår, hvem har haft en CD i en computer hvornår, mv.
- ◆ Hvilke systemfiler der åbnes og lukkes hvornår.

Listen kunne fortsættes. Det er op til det enkelte systems sikkerhedsansvarlige at fastlægge, præcis hvad der skal logges. En meget omfattende sikkerhedslog kunne fx endda medtage ”hvilke taster der trykkes på hvornår”.

Med sporbarhed kan man efter et eventuelt brud på sikkerheden *spore*, hvem der gjorde hvad hvornår. Det kræver blot et opslag i sikkerhedsloggen. Er der foregået ulovligheder, fx, kan sikkerhedsloggen bruges til bevisførelse.

For at implementere sporbarhed skal systemet kunne *identificere* og *autentificere* aktører.

9.5. KRYPTOGRAFI

Kryptografi er læren om at *kryptere data*. Krypteret data er tilsløret: Kun afsender og de modtagere, der ved, hvordan data kan *dekrypteres*, kan forstå indholdet. Bliver krypteret data opsnappet af en, der aflytter forbindelsen, får vedkommende ikke noget ud af det.

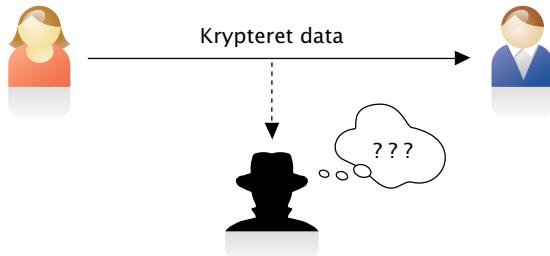


Fig. 9.158. Kryptering.

9.5.1. EKSEMPEL: KRYPTERING Å LA CÆSAR

Allerede den romerske kejser Julius Cæsar brugte kryptering, når han sendte taktiske ordrer med kurører mellem sine udsendte generaler. På den måde kunne hverken en spion, der udgav sig for at være kurør, eller overfaldsmænd, der bemægtigede sig den krypterede besked, afløre beskeden. Cæsars teknik var simpel – han erstattede simpelt hen hvert bogstav i sin besked med bogstavet tre gange længere henre i alfabetet:

Før kryptering: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Efter kryptering: DEFGHIJKLMNOPQRSTUVWXYZABC

Hvis vi koder teksten "Top secret!" med Cæsars metode, får vi den krypterede tekst "Wrs vhfuhw!". Uforståeligt – medmindre man kender krypteringsmåden.

Det er klart, at man også kunne kryptere ved at forskubbe et andet antal pladser end lige netop 3.

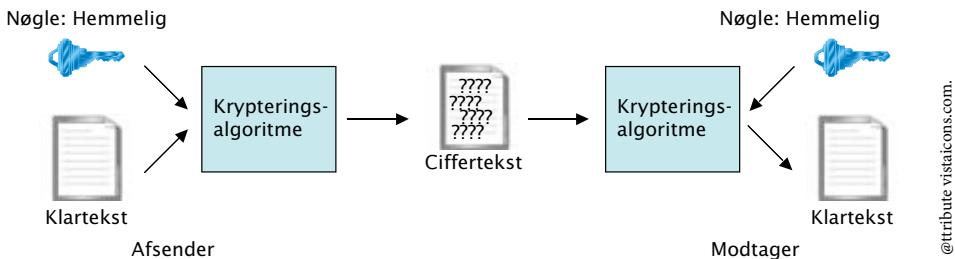
9.5.2. KLARTEKST, CIFFERTEKST OG NØGLE

Ikke-krypterede beskeder er som postkort: Uvedkommende kan læse indholdet under transporten. Krypterede beskeder er som breve i lukkede kuverter: Uvedkommende kan ikke læse indholdet (men kan som regel se afsender og modtager).

Til kryptering bruges en *krypteringsalgoritme*. Man bruger primært kryptering der ikke afhænger af, at krypteringsalgoritmen er hemmelig. Princippet om, at krypteringsalgoritmen bør være offentligt kendt, kaldes for *Kerckhoffs princip*.

Lad os sige, at Alice vil sende en besked til Bob, fx "Jeg elsker dig, Bob." (Alice og Bob er tilbagevendende figurer i litteratur om kryptografi). Beskeden, inden den er blevet krypteret, kaldes for *klarteksten*. Alice krypterer nu sin besked med en eller anden krypteringsalgoritme, sådan at den resulterende *ciffertekst* er uforståelig for udenforstående, fx aflyttersken Eva. Idet krypteringsalgoritmen er offentligt kendt, bliver Alice nødt til at bruge en eller anden hemmelig information, hvis hun vil undgå, at Eva dekrypterer beske-

den. Denne hemmelige information kaldes *nøglen*. Nøglen bruges til at "låse" cifferteksten med – Eva kender ikke nøglen, der fx kan være et tal, et stykke tekst eller et kodeord. For at dekryptere cifferteksten til klartekst, skal Bob bruge den rigtige nøgle til at "låse cifferteksten op" med.



@tribute.vistaicons.com.

Fig. 9.159. Kryptering: Inden afsendelse krypteres klartekst til ciffertekst med nøglen. Ved modtagelse dekrypteres cifferteksten med nøglen.

En krypteringsalgoritme tager altså to input: Klartekst og nøgle, og producerer et output: cifferteksten. Vil man dekryptere, bruges også to input: Ciffertekst og nøgle. Outputtet er da klarteksten.

EKSEMPEL

CÆSARKRYPTERING

I forrige afsnit krypterede vi beskeden "Top secret!" med Cæsars metode og fik den uforståelige tekst "Wrs vhfuhw!" ud af det. Klarteksten er i dette tilfælde "Top secret!", nøglen er tallet 3, mens cifferteksten er "Wrs vhfuhw!". For at dekryptere cifferteksten, skal vi blot bruge nøglen (tallet 3) – så kan vi "gå den anden vej".

9.5.3. SYMMETRISK KRYPTERING

Cæsars kryptering er *symmetrisk*.

ORDFORKLARING

SYMMETRISK KRYPTERING

Symmetrisk kryptering er kryptering, hvor afsender og modtager skal bruge den samme nøgle: Afsender krypterer med nøglen, modtager dekrypterer med nøglen.

Sikkerheden ved en symmetrisk krypteringsalgoritme afhænger af, at nøglen svær at gætte. Jo mere kompleks nøgle, desto sværere er det at knække koden. Til gengæld tager kryptering og dekryptering længere tid. Cæsars algoritme har en simpel nøgle og er dermed let at gætte. Der er kun 29 forskellige nøgler (hvorfor?), og de er hurtigt gennemprøvet.

EKSEMPEL

DES OG AES: FORTROLIGHED

DES (Data Encryption Standard) er en symmetrisk krypteringsalgoritme, der blev lancet i 1977. DES bruger et 56-bit lang *binært tal* som nøgle. Der er altså 2^{56} , eller lidt over 70 millioner millarder, forskellige DES-nøgler. Man skulle tro, at DES dermed er ubrydelig, men et effektivt kryptoanalysehold dekrypterede en DES-besked på mindre end 4 måneder i 1997. Siden voksede computerkraften, og i 1999 behøvedes kun 22 timer for at dekryptere en DES-besked. For at imødegå ondsindede personer med masser af computerkraft til at knække DES-krypterede beskeder, lancerede NIST (en amerikansk organisation for tekniske standarder) i 2001 efterfølgeren til DES: AES (Advanced Encryption Standard). AES bruger nøgler med bitlængde 128, 192 eller 256. En maskine, der kan knække en DES-krypteret besked på 1 sekund, skal bruge omrent 150 billioner år på at knække en besked, der er AES-krypteret med en 128-bits nøgle.

WWW

KERBEROS: AUTENTIFICERING

På **WWW** kigger vi på en udbredt protokol til autentificering, der bygger på symmetrisk kryptering, Kerberos (protokollen er opkaldt efter den trehovedede hunde-skildvagt, der i den græske mytologi bevogter indgangen til underverdenen).

9.5.4. ASYMMETRISK KRYPTERING

Symmetrisk kryptering forudsætter, at afsender og modtager i al hemmelighed har kunnet blive enige om, hvilken nøgle der skal bruges. Men det kræver jo (sikker) kommunikation! Hvor Cæsar og hans generaler kunne mødes over en gladiatorkamp og aftale nøgler, mens tilskuernes opmærksomhed var rettet mod arenaen, er det meget sjældent tilfældet, at folk der skal sende krypteret data til hinanden over fx internettet, kan mødes i det virkelige liv og aftale hemmelige nøgler først. Hvem ved, måske bor Alice i Grønland og Bob i Tunesien – og de skal udveksle krypteret data med det samme – hvad gør de?

Svaret er *asymmetrisk kryptering* (en genial teknik opfundet i 1970’erne), hvor afsender og modtager ikke behøver at have udvekslet en hemmelig nøgle for at kunne kommunikere sikkert. Asymmetrisk kryptering kan ikke bare bruges til fortrolig kommunikation, men også til autentificering og digitale signaturer.

ORDFORKLARING

ASYMMETRISK KRYPTERING

Kryptering, hvor afsender og modtager hver har sit eget sæt nøgler, en *privat nøgle* og en *offentlig nøgle*. Den private nøgle er hemmelig og kun kendt af ejermanden, mens alle har adgang til at se den offentlige nøgle. Data krypteret med den offentlige nøgle kan kun dekrypteres med den private nøgle. Data krypteret med den private nøgle kan kun dekrypteres med den offentlige nøgle.

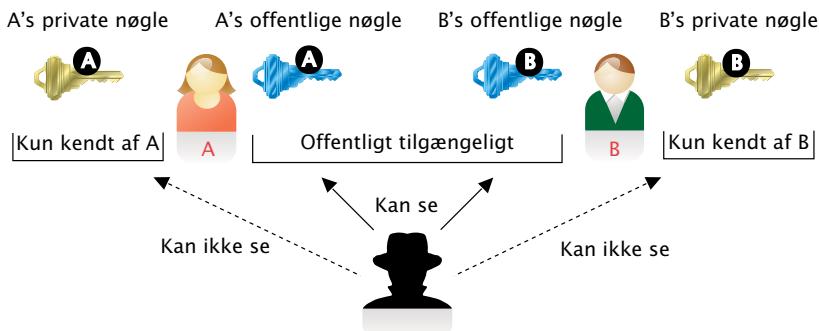


Fig. 9.160. Offentlig og privat nøgle.

TIP

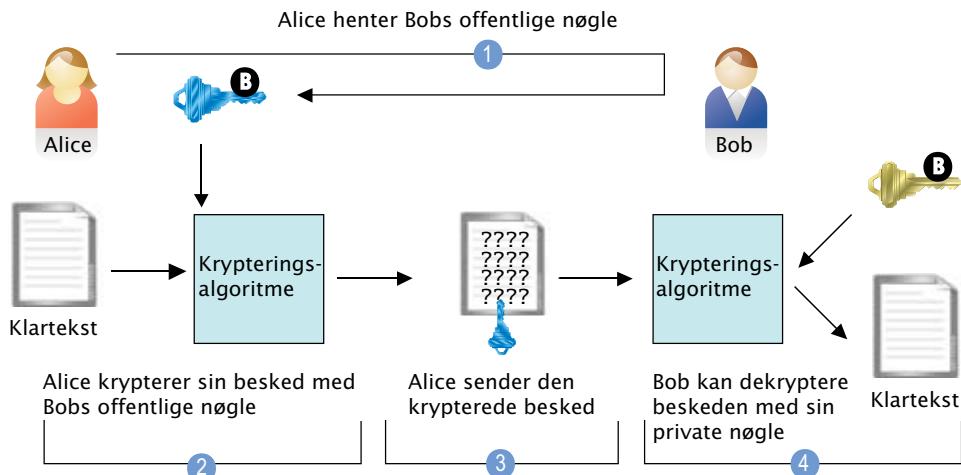
PUBLIC KEY-KRYPTERING

I kraft af, at asymmetrisk kryptering hviler på brugen af offentlige nøgler, modsat symmetrisk kryptering, hvor der kun er hemmelige nøgler, kaldes asymmetrisk kryptering også for *public key-kryptering* (public betyder offentlig på engelsk).

Hvis Alice og Bob kommunikerer med asymmetrisk kryptering, er der altså i alt 4 nøgler i spil (modsat asymmetrisk kryptering, hvor der var 1 nøgle): Alices private nøgle, Alices offentlige nøgle, Bobs private nøgle og Bobs offentlige nøgle.

Vi tager et eksempel på asymmetrisk kryptering:

1. Alice vil sende en besked, L, til Bob. Hun henter derfor Bobs offentlige nøgle, $Bob_{\text{offentlig}}$. Den kan alle hente, idet den er offentlig.
2. Alice krypterer sin besked, L, ved hjælp af Bobs nøgle, og får cifferteksten $Bob_{\text{offentlig}}(L)$. Hun bruger (i overensstemmelse med Kerckhoffs princip) en velkendt krypteringsalgoritme.
3. Alice sender den krypterede besked til Bob.
4. Bob modtager $Bob_{\text{offentlig}}(L)$. Han kender, som den eneste, sin private nøgle, Bob_{privat} . Bob kan derfor dekryptere $Bob_{\text{offentlig}}(L)$. Det gør han ved at beregne $Bob_{\text{privat}}(Bob_{\text{offentlig}}(L)) = L$.
5. Hvis Bob beslutter sig for at svare, bruger han den helt tilsvarende procedure: han henter først Alices offentlige nøgle, krypterer sit svar med den, og sender den krypterede besked til Alice. Fordi Alice kender sin private nøgle, kan hun dekryptere Bobs besked.



Attributed via icons.com.

Fig. 9.161. Asymmetrisk kryptering med privat og offentlig nøgle.

ANALOGI

AFLÅSTE KISTER

Dronning Alice af Alicien vil gerne sende et hemmeligt brev til Kong Bob af Bobbistan. Desværre har dronning Alice kun en mistænkelig kurier, Igor, til sin rådighed. Igor kunne sagtens finde på selv at læse brevet undervejs eller udlevere det til fremmede. Hun sender derfor Igor til Bobbistan for at hente Kong Bobs hængelås. Igor adlyder. I Bobbistan udleverer Kong Bob sin hængelås (åben). Hængelåsen er ikke i sig selv værdifuld, så Igor stjæler den ikke på hjemvejen. Hjemme i Alicien putter Dronning Alice brevet ned i en kiste, spænder jernkæder om kisten og låser kæderne fast med Kong Bobs hængelås. Dronning Alice sender Igor afsted med den aflåste kiste. Igor kan ikke åbne kisten, og undlader derfor at stjæle den – det ville han alligevel ikke få noget ud af. Møder han mistænkelige fremmede på rejsen, vil de heller ikke kunne åbne kisten og se brevet derinde. Vel fremme i Bobbistan modtager Kong Bob kisten. Efter at have sendt Igor hjemad, fremdrager Kong Bob sin nøgle, sætter den i hængelåsen – den passer! – og låser op. Brevet er nået sikker frem.

Fortællingen kan bruges til at forstå asymmetrisk kryptering med:

Hemmeligt brev	= besked, Alice vil sende til Bob
Kong Bobs hængelås	= Bobs offentlige nøgle
Kiste låst med Kong Bobs hængelås	= besked krypteret med Bobs offentlige nøgle
Kong Bobs nøgle	= Bobs private nøgle
Igor	= transport over et netværk, der måske bliver aflyttet

EKSEMPEL

RSA

RSA er en asymmetrisk krypteringsalgoritme (opkaldt efter opfinderne Ron Rivest, Adi Shamir og Leonard Adleman). RSA er en globalt accepteret standard for kryptering.

9.5.5. AFTALE AF SYMMETRISK NØGLE: DIFFIE-HELLMAN

Asymmetrisk kryptering kan bruges til at aftale en hemmelig symmetrisk nøgle med. Til formålet bruges *Diffie-Hellman-protokollen*, opfundet af – ja … – Diffie og Hellman.

Lad os sige, at Alice og Bob ønsker at aftale en symmetrisk nøgle til at udveksle data fortrigtlig med. De gennemfører derfor skridtene i Diffie-Hellman-protokollen:

1. Alice og Bob udveksler offentlige nøgler – selvom netværket er usikkert, er der ingen problemer, fordi nøglerne er offentlige.
2. Alices Diffie-Hellman-program tager hendes private nøgle og Bobs offentlige nøgle og beregner et tal K1 ved hjælp af Diffie-Hellman-algoritmen.
3. Bobs Diffie-Hellman-program tager hans private nøgle og Alices offentlige nøgle og beregner et tal K2 ved hjælp af Diffie-Hellman-algoritmen.
4. Fordi Diffie-Hellman-algoritmen er baseret på smuk og smart matematik, gælder
 - a. $K_1 = K_2$.
 - b. En ondsindet aktør, der har aflyttet udvekslingen af de offentlige nøgler, kan ikke beregne tallet K1 – for at gøre det, skal han bruge enten Bobs eller Alices private nøgle – og den har han ikke.

Fordi der gælder lige præcis både a og b, kan Alice og Bob bruge den beregnede værdi som symmetrisk nøgle.

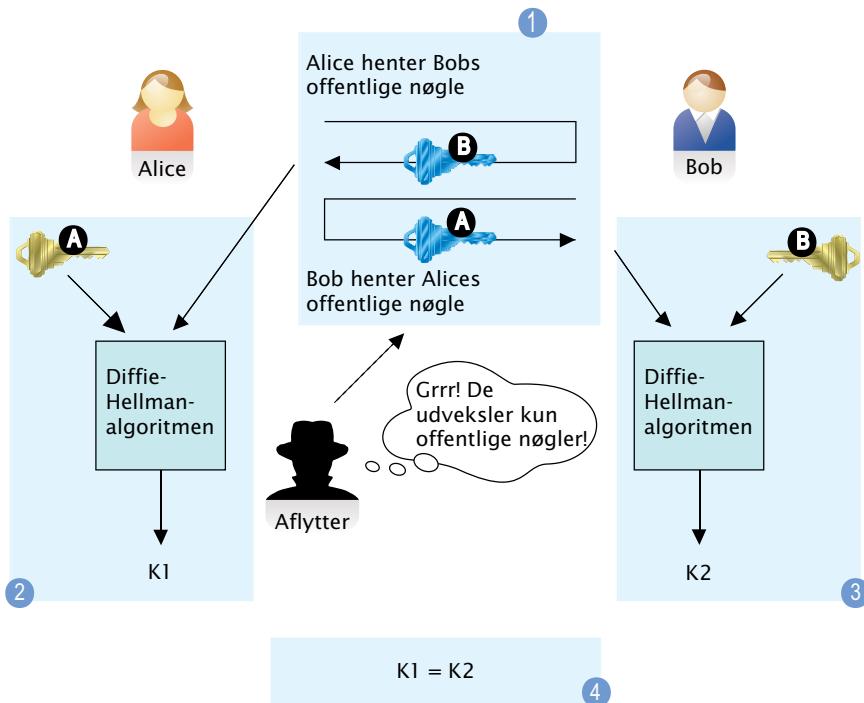


Fig. 9.162. Diffie-Hellman.

9.5.6. AUTENTIFICERING MED RANDOM CHALLENGE

Vi husker autentificeringsprotokollen i Figur 9.156. For at sikre protokollen mod replayangreb må den ændres. Vi forsyner derfor vores autentificeringsprotokol med en mekanisme til *random challenge* og *response*.



Fig. 9.163. Alice autentificerer sig overfor Bob med challenge-response. Ondskindede aktører, der giver sig ud for at være Alice, har ikke en chance.

Til hver *challenge* svarer et rigtigt *response*. Lad os sige, at Alice vil autentificere sig overfor Bob. Alice henvender sig hos Bob og siger "Jeg er Alice. Jeg vil gerne autentificeres." Bob svarer med en random challenge – en tilfældig udfordring – som kun Alice kender det rigtige svar (*response*) til. Alice sender sit svar til Bob. Fordi svaret er sandt, ved Bob, at det må være Alice han taler med.

Ideen med random challenge og response er, at hver gang Alice vil autentificere sig, får hun en ny udfordring – og derfor er det nu et nyt svar, der er det eneste acceptable svar. Da udfordringen udvælges tilfældigt af Bob, kan hverken Alice eller andre udregne på forhånd, hvilken udfordring der skal svares på. En aflytter kan altså ikke bevidstløst lave et replay-angreb med Alices response, fordi et response ikke kan genbruges.

EKSEMPEL

CHALLENGE-RESPONSE

Alice vil autentificere sig over for Bob. Bob udfordrer derfor med den tilfældigt valgte udfordring "Hvad er navnet på den bamse, du havde som 8-årig?" (*challenge*). Alice svarer "Bamse-Lillian!" (*response*). Bob konstaterer, at svaret er sandt. Samtidig ved han, at kun Alice kender til Bamse-Lillian, så det må altså være Alice, han taler med. Næste gang Alice vil autentificere sig, giver Bob hende spørgsmålet "Hvad var det første ord, du lærte på fransk?". Alice svarer denne gang med "crayon". Bob konstaterer at svaret er sandt. Samtidig ved han, at kun Alice kender svaret, så det må være Alice han snakker med. Svaret "Bamse-Lillian" er helt klart et forkert svar på challenge nummer to. En replay-angriber ville derfor ikke med succes kunne udgive sig for at være Alice.

I praksis laves random challenge-response med en kompliceret kombination af asymmetrisk og symmetrisk kryptering. Det vigtige her er blot at forstå principippet.

9.5.7. DATA-INTEGRITET MED HASHING

På internettet bruges en checksum til at opdage, om en pakke er blevet ændret under sin rejse fra afsender til modtager. En tilsvarende teknik, *hashing* (af det engelske ord *hash*, der betyder noget i retning af hakkelse), bruges til at sikre data-integritet af beskeder.

ORDFORKLARING

HASH-FUNKTION

Til hashing bruges en *hash-funktion*. Hash-funktionen er en algoritme, der tager en besked (kort eller lang) som input og leverer en hash-værdi af en bestemt størrelse som output. En hash-funktion beregner altid samme output-hash-værdi for den samme input-besked. En hash-funktion er *en-vejs*: Udfra en hash-værdi kan den oprindelige besked ikke genskabes.

EKSEMPEL

HASHING

Alexander vil sende beskeden “**Du er sød! Alexander**” til Malene over et usikkert netværk. For at sikre sig, at Malene ikke modtager en forkert besked, hasher Alexander beskeden med en hash-funktion og beregner hash-værdien “**507113e9**”. Alexander sender nu både besked og hash-værdi til Malene. Beskeden når frem til Malene, og hun beregner hash-værdien af den modtagne besked. Hvis hun får hash-værdien “**507113e9**”, kan hun sammenligne med Alexanders medsendte hash-værdi og konstatere, at beskeden er nået uændret frem. Hvis hun derimod får en anden hash-værdi, må beskeden være blevet ændret under sin rejse på netværket. I dette tilfælde kasserer Malene beskeden.

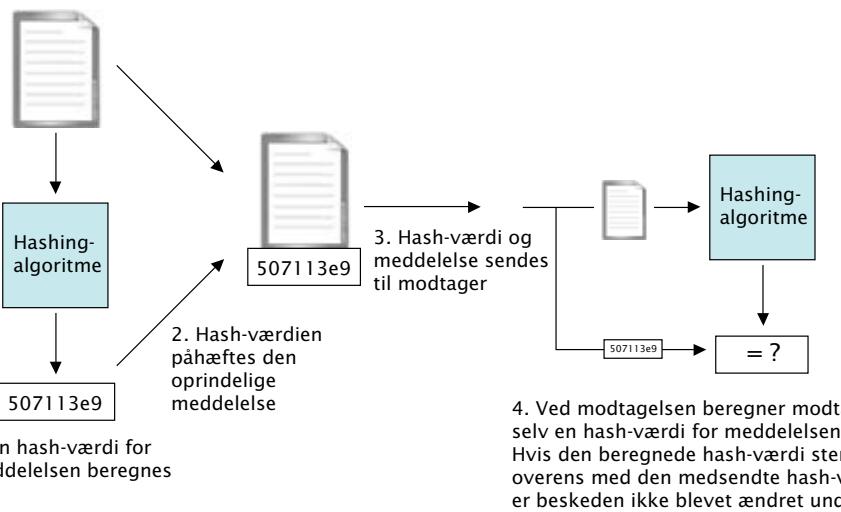


Fig. 9.164. Hashing.

Hashing bruger ingen nøgler. Hashing-algoritmen er heller ikke hemmelig – i overensstemmelse med Kerckhoffs princip. Fidusen med hashing er, at det er en en-vejs-proces. Opsnapper en ondsindet aktør en hash-værdi, fx “aac16ea6”, *kan han ikke genskabe den besked*, der gav hash-værdien.

Der er altså en afgørende forskel på, hvordan kryptering med nøgler fungerer, og så hashing. Med den rette nøgle kan man dekryptere en krypteret besked. Ingen kan nogensinde “dekryptere” en hash-værdi og få den originale besked tilbage.

TIL SÆRLIGT INTERESSEREDE

KOLLISIONSFRI HASH-FUNKTIONER

Der findes mange anvendte hash-funktioner. En god hash-funktion er *kollisionsfri*, dvs, forelagt en besked og dens hash-værdi er det *umuligt* at finde en anden besked med samme hashværdi.

9.5.8. DIGITAL SIGNATUR

En *digital signatur* er en elektronisk underskrift. Den identifierer og autentificerer underskriveren, og den er uafviselig. Sagt på dansk: Man skal kunne afgøre, om en digital signatur tilhører en bestemt person, og signaturen må ikke kunne kopieres.

EKSEMPEL

BRUG AF DIGITAL SIGNATUR

Digitale signaturer bruges overalt, hvor der indgås kontrakter eller økonomiske aftaler over netværk, fx mellem personer og banker, mellem banker og banker, eller mellem virksomheder.

Inden vi ser på, *hvordan* man laver en digital signatur, kigger vi på et eksempel, der illustrerer, *hvorfor* digitale signaturer er nødvendige. Hashing-teknikker alene er nemlig ikke nok til at sikre besked-autenticitet:

EKSEMPEL

HASHING UDEN AUTENTIFICERING

Alexander sender (besked,hash-værdi)-parret

(“Du er sød! Alexander”, “507113e9”)

til Malene over et usikkert netværk. Erika, der er jaloux, opsnapper både besked og hash-værdi. Hun ændrer beskeden til “**Jeg hader dig, Malene. Alexander**” og beregner en ny hash-værdi, “**029a159e**”, svarende til den nye besked. Erika sender nu (besked,hash-værdi)-parret

(“**Jeg hader dig, Malene. Alexander**”, “**029a159e**”)

til Malene. Malene modtager beskeden, beregner hash-værdien “**029a159e**” og tror nu – fordi hash-værdierne stemmer overens – at beskeden ikke er blevet ændret undervejs.

Vil man sikre sig, at beskedens afsender er den rigtige, må *digitale signaturer* tages i brug.

En digital signatur implementeres med en kombination af asymmetrisk kryptografi og hashing:

ORDFORKLARING

DIGITAL SIGNATUR

En digital signatur på en besked er en hash-værdi af beskeden, krypteret med afsenderens private nøgle.

Ideen er, at *hvis* kan man dekryptere hash-værdien med en eller andens offentlige nøgle, så *må* den “en eller anden” lige præcis være afsenderen. Signeringen sikrer på den måde autentificering og uafviselighed. Hashingen sikrer data-integritet.

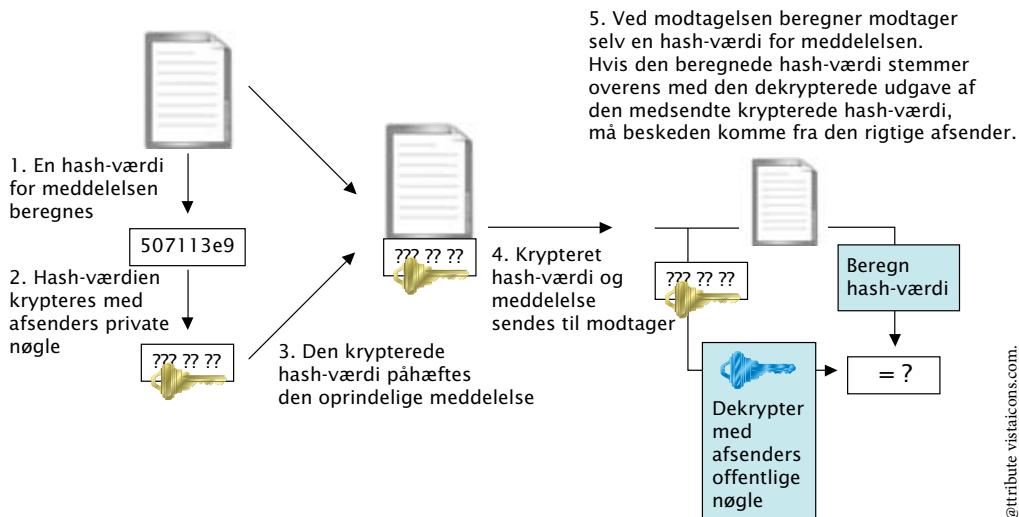


Fig. 9.165. Digital signatur.

Vi illustrerer ideen med et eksempel.

EKSEMPEL

DIGITAL SIGNATUR

Alexander vil sende beskeden “**Du er sød! Alexander**” til Malene. For at sikre data-integritet og for at Malene ved, at beskeden er fra ham, gør Alexander følgende:

1. Alexander beregner hash-værdien “**507113e9**” af beskeden “**Du er sød! Alexander**”.
2. Han krypterer hash-værdien “**507113e9**” med sin private nøgle og får den krypterede hash-værdi “**bdb74befc887188**”. Kun Alexanders offentlige nøgle kan dekryptere den krypterede hash-værdi.
3. Alexander sender (besked, krypteret hash-værdi)-parret

("Du er sød! Alexander", "bdb74befc887188")

til Malene.
4. Malene beregner først hash-værdien af den modtagne besked – hun får hash-værdien “**507113e9**”, fordi en hash-funktion altid giver den samme hash-værdi for den samme input-besked.
5. Malene dekrypterer herefter den modtagne krypterede hash-værdi “**bdb74befc887188**” med Alexanders offentlige nøgle. Det giver hende værdien “**507113e9**”.
6. Fordi den modtagne og den beregnede hash-værdi er ens, er beskeden ikke blevet ændret undervejs. Fordi Alexanders offentlige nøgle kunne bruges til at dekryptere med, *må* beskeden være fra ham.

9.5.9. PKI

Man skulle tro, at alle vores trængsler vedrørende sikker kommunikation var overstået, nu hvor vi både har kryptering, hashing og digitale signaturer. Der er imidlertid én ting, vi har haft som stiltiende antagelse gennem hele krypteringsafsnittet: Nemlig at

afsender og modtager stoler på hinanden.

Hvad nu hvis Josefine vil kommunikere med Stefan, men ikke er sikker på, om Stefan er ondsindet – er han til rødvin og madvarer eller røveri og malware?

Løsningen på problemet er at lave en *public key-infrastruktur (PKI)*. En PKI er et system, der tager højde for ”hvordan aktører stoler på hinanden”. De vigtigste elementer i PKI er

- ◆ certifikater,
- ◆ certifikat-autoriteter (CA'er).

Hver aktør (fx Josefine) i et PKI registreres hos en certifikat-autoritet, der mod konkrete oplysninger om fx navn, adresse, mv., udsteder et personligt *certifikat*. Vil Josefine snakke

med Stefan, spørger hun sin certifikat-autoritet, om Stefan er registreret der. Hvis han er, får Josefine Stefans certifikat tilsendt – det indeholder Stefans offentlige nøgle – og Josefine kan derefter snakke med Stefan. Josefine vælger altså at stole på de aktører, der er registreret hos hendes certifikat-autoritet.

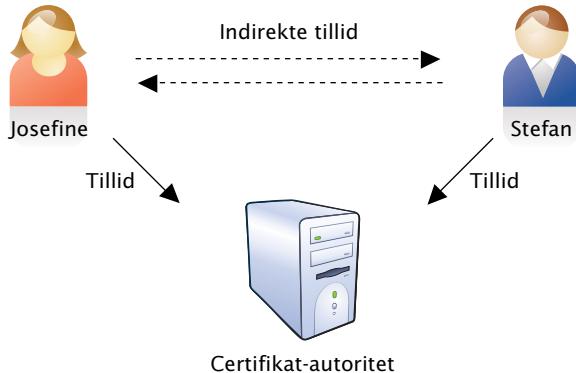


Fig. 9.166. Josefine stoler på sin CA. Stefan stoler på sin CA. Derfor stoler Josefine og Stefan indirekte på hinanden.

9.5.10. KRYPTOGRAFISK OPSUMMERING

Vi opsummerer alle de mange kryptografiske begreber.

... sikrer ...	fortrolighed	data-integritet	autenticitet	uafviselighed
Kryptering ...	X	-	-	-
Hashing ...	-	X	-	-
Digital signatur ...	-	X	X	X
Kryptering og digital signatur ...	X	X	X	X

Fig. 9.167. Ikke alle kryptografiske teknikker sikrer alle IT-sikkerhedsmål. Med de rette kombinationer af kryptografiske teknikker kan man opnå de sikkerhedsmål, en given beskedudveksling kræver.

9.6. REALISERING AF IT-SIKKERHED

Som allerede nævnt kan et IT-system aldrig blive 100 % sikkert. Det er derfor op til sikkerhedspolitikken for et givent IT-system at fastlægge hvor meget anstrengelse der skal bruges på IT-sikkerhed. Jo mere sikkerhed, jo større omkostninger.

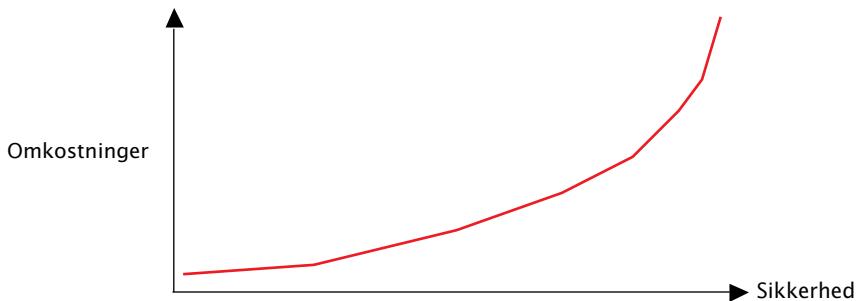


Fig. 9.168. Omkostninger og sikkerhed.

9.6.1. OMKOSTNINGER

Ethvert IT-system har brugere. Brugerne, der ikke nødvendigvis ved noget om IT-sikkerhed, har alligevel bestemte IT-sikkerhedsmæssige behov. Samtidig er det klart, at jo flere sikkerhedsprocedurer, brugere af system skal lære at bruge, desto sværere er det at bruge systemet. Et godt system har derfor en fornuftig balance mellem at være *sikkert* og at være *let at bruge*.

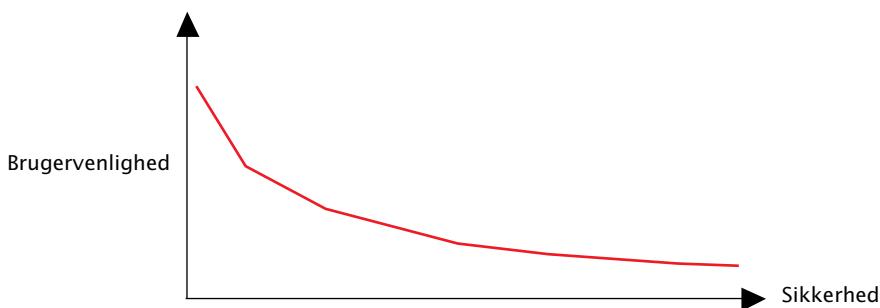


Fig. 9.169. Et system uden nogen sikkerhedsforanstaltninger er let at bruge, men meget sårbart. Et system med mange sikkerhedsforanstaltninger er svært at bruge, men ikke så sårbart.

Hvis et system har sikkerhedsprocedurer eller -foranstaltninger, der på klodset vis afbryder eller irriterer brugere, ender det som oftest med, at systemets brugere blæser på sikkerheden – enten undlader de at følge sikkerhedsprocedurer, eller også foretager de sig uhensigtsmæssige ting for at spare tid.

EKSEMPEL

SKIFTENDE KODEORD

Mange systemer vil sikre sig mod svage bruger-kodeord og kræver derfor, at alle brugere skifter kodeord en gang imellem – fx en gang om måneden. Fordi brugere ikke gider huske på nye kodeord hele tiden, vælger brugere tit at skifte mellem 2 kodeord, eller bruge et gennemskueligt system til at vælge næste kodeord med. Det gør det ikke væsentligt sværere for en angriber, og sikkerhedsforanstaltningen med de roterende kodeord ender med ikke at højne sikkerheden, men blot gøre brugerne frustrerede.

Sikkerhedsforanstaltninger kræver også tid og ressourcer af andre end brugerne:

- ◆ Sikkerhed kræver computerressourcer
- ◆ Sikkerhedsprodukter (fx firewalls eller antivirusprogrammer) skal udvælges, installeres og vedligeholdes
- ◆ Sikkerhed skal betales

IT-sikkerhed er derfor en omkostning, der skal retfærddiggøres. Det kan være svært at argumentere for denne omkostning, da resultaterne af velfungerende sikkerhed er ikke så synlige (der skete *ikke* noget). Det er selvfølgelig klart, at omkostningerne ved manglende sikkerhed kan være fatale.

9.7. NØGLEORD

- ◆ IT-sikkerhed
- ◆ Fortrolighed
- ◆ Integritet
- ◆ Tilgængelighed
- ◆ Data-integritet
- ◆ Autenticitet
- ◆ Uafviselighed
- ◆ Privacy
- ◆ Autorisation
- ◆ Adgangskontrol
- ◆ Trussel
- ◆ Sårbarhed
- ◆ Modmiddel
- ◆ Malware
- ◆ Firewall
- ◆ Sporbarhed
- ◆ Kryptografi
- ◆ Krypteringsalgoritme
- ◆ Klartekst
- ◆ Cifertekst
- ◆ Nøgle
- ◆ Symmetrisk kryptering
- ◆ Hashing
- ◆ Digital signatur

KAPITEL 10

PROJEKTARBEJDE I DATALOGI

Et *datalogisk projekt* består i at løse en opgave ved hjælp af datalogi – så mon ikke der skal programmeres?! Men programmering er ikke bare programmering. For at få et datalogiprojekt til at blive en succes er det afgørende ikke bare at kode løs med hovedet under armen. Et program er ubrugeligt, hvis

- ◆ det løser en forkert opgave,
- ◆ det ikke kan laves,
- ◆ der er ikke tid (eller penge) til at lave det færdigt,
- ◆ det skal kasseres efter to måneder, fordi det er forældet,
- ◆ programmets målgruppe ikke kan finde ud af at bruge det.

Dette kapitel giver korte ideer til, hvordan du

- ◆ strukturerer en arbejdsproces, så projektet bliver realistisk at fuldføre og løser de rigtige opgaver,
- ◆ fremtidssikrer det produkt, du udvikler,
- ◆ gør slutproduktet brugervenligt.

WWW

EKSEMPEL PÅ PROJEKT OG ARBEJDSPROCES

På **WWW** finder du et eksempel på et datalogiprojekt med en beskrivelse af den tilhørende arbejdsproces. Eksemplet sætter kød på alle de begreber, metoder og advarsler, der nævnes i flæng i dette kapitel.

10.1. ARBEJDSPROCES

Der er ikke én rigtig måde at strukturere en datalogisk arbejdsproces på. Faktisk findes der utallige modeller for arbejdsprocesser – med hver deres fordele og ulemper. En god datalogisk proces kommer dog omkring det meste af følgende:

- ◆ brainstorm
- ◆ analyse
- ◆ kravspecifikation

- ◆ modellering og design
- ◆ implementation
- ◆ fejlfinding (*test*)
- ◆ fejlretning (*debugging*)
- ◆ dokumentation
- ◆ brugervejledning

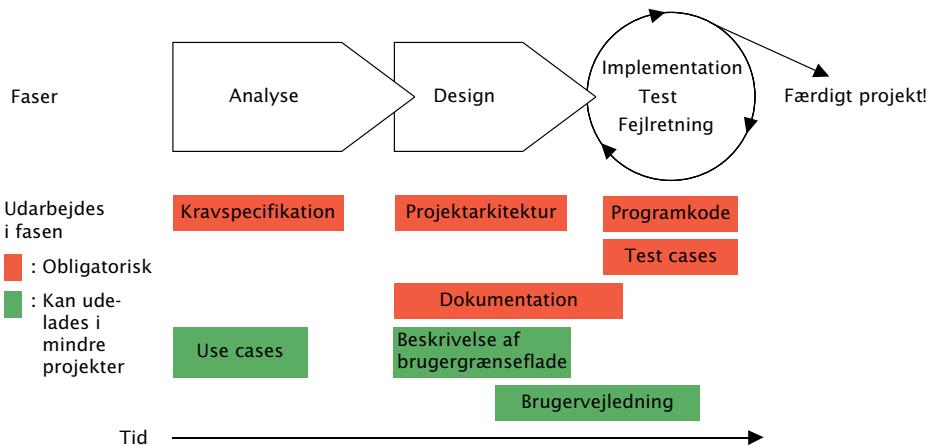


Fig. 10.170. Eksempel på model for arbejdsproces.

Som listen viser, er selve kedefasen – implementationsfasen – blot en af riktig mange faser, der er mindst lige så vigtige. En god arbejdsproces kan kendes på

- ◆ godt forarbejde,
- ◆ kort implementationsfase,
- ◆ få fejl.

En dårlig arbejdsproces kan kendes på

- ◆ manglende forarbejde,
- ◆ lang og forvirrende implementationsfase,
- ◆ mange fejl, sidste-øjebliks-beslutninger og inkonsistenser, der fører til en lang og utilstrækkelig fejlfindings- og fejlretningsfase.

Det kan være en ide at arbejde i flere faser samtidigt. Fx er det en god ide at udarbejde dokumentation løbende. Det kan også være en god ide at gennemløbe et antal faser flere gange – en sådan gentagende arbejdsproces kaldes *iterativ*. På den måde kan et projekt opbygges ”lidt ad gangen”.

Det kan også være en ide at udtaenke forskellige arbejdsroller, som de enkelte medlemmer i en projektarbejdsgruppe kan varetage. En kan fx være test-ansvarlig, mens en anden er dokumentationsansvarlig.

10.1.1. BRAINSTORM

I den helt indledende fase dannes en projektarbejdsgruppe og mødes for at blive inspireret til at løse projektopgaven. Alle ideer, input, meninger og drømme om slutproduktets udseende og funktion er velkomne. Der skrives ned, tegnes, snakkes, synges, ...

Brainstorm-fasen resulterer i:

- ♦ en usorteret liste af ideer. Listen er typisk kreativ og uden tekniske detaljer.

ADVARSEL

HVAD KAN GÅ GALT I BRAINSTORM-FASEN?

En ukontrolleret brainstorm-fase kan blive ustruktureret eller langvarig. Det er derfor fornuftigt at tidsbegrænse brainstorm-fasen.

10.1.2. ANALYSE OG KRAVSPECIFIKATION

I analysefasen analyseres, hvilke behov det kommende produkt skal dække. Det gøres ved at kortlægge, *hvad de kommende brugere forventer af produktet* – fx via interviews eller spørgeskemaer. Undersøgelsen kan gøre brug af brainstormen – fx kan man spørge brugere om deres favorit blandt flere forskellige løsningsforslag. Brugeres forventninger svarer ofte til bestemte handlinger og kan i første omgang modelleres som såkaldte *use cases* – korte tekstbeskrivelser af forskellige specialelafde af produktets brug.

Ud over brugerkrav til produktet kan der være krav til fx driftsomkostninger.

Behovene udmønter sig en detaljeret række af krav. Kravlisten kaldes en *kravspecifikation*. Det er udfra kravspecifikationen, at produktet kan designes. Hvert krav i en kravspecifikation skal være

- ♦ *præcist formuleret*, så der ikke er uklarhed om, hvad kravet går ud på,
- ♦ *test-bart*, så det kan undersøges, om et slutprodukt faktisk opfylder kravet,
- ♦ *give mening* i forhold til produktet.

Hvis et krav ikke giver mening, bør det kasseres eller genovervejes. Tivilsommeh krav kan udsættes for en “hvorfor?”-prøve: Er der ingen god grund til at have kravet, kasseres det.

Kravspecifikation fastlægger altså ikke alene, hvad projektet skal kunne, men *afgrænser* også projektet.

Analysefasen resulterer i:

- ◆ Evt. use cases.
- ◆ En kravspecifikation.

ADVARSEL

HVAD KAN GÅ GALT I ANALYSE-FASEN?

- ◆ Det er svært at fastlægge krav, fordi brugere af et fremtidigt produkt sjældent kan formulere præcise krav selv. Det er altså analytikerens opgave at omforme brugerudsagn til præcise krav.
- ◆ Krav kan blive formuleret vagt eller flertydigt, så brugere og designere viser sig at have forskellige tolkninger af kravet.
- ◆ Der er risiko for, at kravene *ikke* tilsammen dækker de faktiske krav, fremtidige brugere af produktet vil stille.
- ◆ Den vigtigste regel er dog: *Opfind ikke unødige krav!*

10.1.3. MODELLERING OG DESIGN

I modellerings- og designfasen fastlægges den logiske struktur af det produkt, man vil lave.

Er der tale om et objekt-orienteret design, fastlægges klasserne, deres klassehierarkier, attributter og indbyrdes afhængigheder. Er der tale om en relationel database, fastlægges tabeller (gerne ud fra E/R-overvejelser) og attributter, og tabellerne normaliseres. Er der tale om en webside, bestemmes udseende og funktionalitet i overordnede træk.

Design går altså ud på at fastlægge

- ◆ komponenter,
- ◆ afhængigheder og forbindelser mellem komponenter,
- ◆ hvordan komponenter har brug for at kommunikere med "omverdenen".

Ligesom en arkitekt beskriver en planlagt bygning med en samling tekniske og grafiske dokumenter, beskriver en projektdesigner *projekt-arkitekturen*. Projekt-arkitekturen kan beskrives på flere niveauer og fra flere synspunkter og både som tekst og diagrammer. Det er en kunst at lave en god projektarkitektur (mere herom i afsnit 10.3).

Designfasen resulterer i:

- ◆ Evt. overordnet beskrivelse af nødvendig kildekode
- ◆ Beskrivelse af systemets vigtigste dele
- ◆ Beskrivelse af de processer, systemet omfatter
- ◆ Evt. udvidede use cases: Beskrivelser af, hvordan brugere er i kontakt med systemet: Hvilke henvendelser kan en bruger gøre, og hvordan, og hvilke svar kan systemet leve?
- ◆ Beskrivelse af *brugergrænseflade* (mere herom i afsnit 10.2).

ADVARSEL

HVAD KAN GALT I DESIGN-FASEN?

- ◆ Den største risiko forbundet med designfasen er, at mange fristes til at springe for let hen over den. Et ufærdigt design hæmmer resten af projektet.
- ◆ Et design, der ikke tydeligt dækker alle krav i kravspecifikationen, er ikke færdigt.
- ◆ I designfasen er det undertiden nødvendigt at træffe beslutninger, der ikke bunder i kravspecifikationen og dermed ikke har at gøre med slutproduktets funktionalitet. Sådanne *designbeslutninger* bør dokumenteres grundigt – ellers bliver de forvirrende senere, når man ikke forstår, hvorfor beslutningerne er taget.
- ◆ Et godt design tilstræber at være *teknologi-neutralt* – det indeholder ingen henvisninger til bestemte programmerings-, web- eller databaseteknologier. På den måde hæmmes implementationsfasen ikke af begrænsninger ved specifikke teknologier.

10.1.4. IMPLEMENTATION

I implementationsfasen implementeres det designede system. Med et godt design er det at implementere (næsten) som at følge en kogebogsopskrift.

I implementationsfasen fastlægges først, hvilke (velegnede) teknologier, der skal bruges til at implementere designet: Der vælges fx programmeringsprog, databaseteknologi samt udviklingsværktøj. Derefter kodes der.

Når du skriver kode, så skriv god kode. God kode

- ◆ *har kommentarer*, så man kan læse, hvad koden gør,
- ◆ er *effektiv*, så der ikke spildes computerkraæfter på at udføre den,
- ◆ er *robust*, dvs. forudsiger og håndterer ugyldigt eller ondsindet input,
- ◆ er *sikker*, dvs. undgår så vidt muligt sikkerhedshuller.

ADVARSEL

HVAD KAN GÅ GALT I IMPLEMENTATIONS-FASEN?

Et uklart design kan føre til, at beslutninger, der burde være taget i designfasen, tages i implementationsfasen. Resultatet bliver et ufleksibelt og uigenremskueligt produkt. Hvis man i løbet af implementationsfasen opdager, at designet er uklart eller urealistisk, bør designet tilpasses, inden implementationsfasen fortsætter.

10.1.5. FEJLRETNING (DEBUGGING)

Et program med mange fejl i er ubrugeligt. Derfor skal et program, allerede mens det er ved at blive til, løbende *debugges*, altså: der skal luges ud i programfejl (*bugs*). Ingen programmør skriver fejlfri kode fra start, så en god programmør er nødvendigvis en god debugger. Debugging udføres efter behov undervejs i implementationsfasen.

Når man skriver kode, kan følgende gode råd følges for at undgå fejl:

- ◆ Kod en enhed ad gangen.
- ◆ Test så vidt muligt hver enkelt enhed straks efter kodningen. (En sådan test kaldes en *unit test*).
- ◆ Tænk dig om, mens du koder. Mener du det, du koder?

Er der alligevel fejl i koden – hvilket opdages ved de løbende unit tests – må man debugge. Der findes værktøjer til debugging, men processen består i det store hele af at teste separate dele af koden. Hvis en del fungerer, må fejlen ligge i anden del. Debugging kræver talent og overblik – men øvelse gør mester.

ADVARSEL

HVAD KAN GÅ GALT UNDER DEBUGGING?

Forhastede fejl-retninger kan føre til nye fejl. Hvis du opdager en fejl, så marker hvor fejlen er – og tænk så over, hvordan den skal rettes, inden du retter den.

10.1.6. TEST

Testfasen påbegyndes, når implementationsfasen er afsluttet. Testfasen har et primært formål: at teste, at implementationen faktisk overholder kravspecifikationen. Det kan gøres ved at tilrettelægge et antal *test cases*, der hver tester et eller flere krav.

En test case specificerer

- ◆ hvilke krav den tester,
- ◆ en række skridt, der skal udføres,
- ◆ det forventede resultat af at udføre skridtene.

Hvis alle test cases kan udføres fejlfrit – dvs, med det resultat, der er angivet som forventet resultat – overholder implementation kravspecifikationen.

Test cases kan udføres enten af mennesker eller programmer. Resultatet af at afvikle en test case skal kunne dokumenteres.

ADVARSEL

HVAD KAN GÅ GALT I TESTFASEN?

- ◆ For at kunne analysere fundne fejl er det vigtigt at kunne genskabe den samme fejl flere gange (indtil den er blevet rettet). Uhedligt tilrettelagte test cases tester på en måde, der ikke kan genskabe resultater af enkelte tests.
- ◆ Det kan være kedeligt at teste. Måske skal man teste samme ting mange gange, og så skal man oven i købet dokumentere alle testresultaterne. Man kan fristes til at springe for let over testfasen.
- ◆ Tests udføres som regel mod slutningen af en projektfase, hvor der er for kort tid til deadline til at udføre en ordentlig test.

10.1.7. DOKUMENTATION

Et produkt, der ikke er dokumenteret, kan hverken forstås, tilrettes eller videreudvikles af andre end dem, der har lavet det. Efter kort tid kan de heller ikke selv huske produktdetaljerne. Produktet bliver dermed umuligt at ændre.

Dokumentationen er et eller flere tekniske dokumenter, der overordnet beskriver både kravspecifikation, design, faktisk implementation og udførte tests og deres resultateter. Der findes en række værktøjer til automatisk dokumentation af fx kildekode.

10.1.8. BRUGERVEJLEDNING

Et produkt, der ikke har en brugervejledning, er svært at bruge. En brugervejledning forklarer i ikke-tekniske vendinger

- ◆ hvad produktet kan bruges til,
- ◆ hvem der kan bruge produktet under hvilke omstændigheder,
- ◆ hvordan produktet bruges.

Hvis et produkt ikke er meget indviklet og har en letforståelig brugergrænseflade, er en brugervejledning mindre vigtig. Det kan fx være tilfældet for webprodukter med klare formål.

10.2. BRUGERGRÆNSEFLADER

En brugergrænseflade modtager *input* og leverer *output*. Brugergrænsefladen afskærmer brugeren fra de bagvedliggende funktionelle detaljer. Brugergrænsefladen kan modtage input på mange måder:

- ◆ som tastetryk og musebevægelser i en grafisk brugergrænseflade (en *GUI – graphical user interface*),
- ◆ som tastetryk og musebevægelser på en website,
- ◆ som fysiske berøringer af touch screens (fx billetbestilling),
- ◆ som tastetryk på et håndholdt apparat,
- ◆ kommandoer i en kommandolinie,
- ◆ gennem fysiske håndtag (fx joysticks til spillekonsoller, pause-knap på DVD-afspiller),
- ◆ ...

Input kan også beregnes af sensorer i grænsefladen, der fx mäter lyde, lufttryk, luftfugthed eller bevægelse. Output er ofte lyd og grafik, men kan også være fx papir-udprint eller joystick-bevægelser.

10.2.1. BRUGERVERNLIGHED

En bil, der ikke kan bakke eller bremse, er ikke en god bil, selvom den har et flot indtræk, er god at sidde i og er nem at betjene. Et dårligt produkt (et, der ikke kan dække en brugers behov) bliver ikke et godt produkt af at få en flot og funktionsdygtig brugergrænseflade: Det er som at give en trold læbestift på og tro, at den så er en prinsesse.

En bil, der fungerer perfekt, men er grim, har spyd i stedet for sæder og et kosteskaft i stedet for et rat er heller ikke en god bil. Et ellers godt produkt er dårligt, hvis brugergrænsefladen er uhensigtmæssig.

	DÅRLIG BRUGERGRÆNSEFLADE	GOD BRUGERGRÆNSEFLADE
DÅRLIGT PRODUKT	⊗ : Brugere skifter produkt	⊗ : Brugere er kortvarigt glade pga den gode grænseflade, men skifter så produkt
GODT PRODUKT	⊗ : Brugere ærgres og ender med at skifte produkt	⊗ : Brugere er tilfredse

Fig. 10.171. Produktdesign og brugergrænseflade: Begge dele er vigtigt for at opnå tilfredse brugere.

En god brugergrænseflade er *brugervenlig*. Jo mere brugervenlig, desto lettere og hurtigere er det er for en bruger

- ◆ at levere input,
- ◆ at forstå output.

Størrelsesordenen af en brugers anstrengelser afhænger af brugeren: hvordan tænker og føler brugeren? Hvor god er brugeren til at tænke logisk? Hvor god er brugeren til at tænke visuelt? – Kort sagt: Hvor god er brugeren til at gætte sig frem til, hvordan produktet bruges? Mantraet indenfor brugergrænseflader er

Don't make me think!

Altså: Få aldrig en bruger til at tænke! Hvis en bruger skal tænke sig om eller gætte for at bruge en grænseflade, er den dårligt designet. Grænsefladen skal derfor udformes med udgangspunkt i brugeren. Det kaldes *brugercentrering*.

GODT: BRUGERCENTRERING GRÆNSEFLADEN ...	DÅRLIGT: INGEN BRUGERCENTRERING GRÆNSEFLADEN ...
... er opbygget med udgangspunkt i brugerens viden	... er opbygget med udgangspunkt i special-viden, man ikke kan forvente af brugeren
... er opbygget med udgangspunkt i genkendelse	... er opbygget med udgangspunkt i genkalde
... giver brugerne en fornemmelse af at vide, hvad de laver	... forvirrer brugerne
... lader brugeren bruge de typer input, brugeren bedst kan lide	... skal bruges på 1 bestemt måde
... er organiseret efter brugernes brugsmønstre: de mest almindelige funktioner er de lettest tilgængelige	... er organiseret efter en dybere, men for brugere skjult logik: de funktioner, en bruger hyppigst ønsker at bruge, er ikke nødvendigtvis nemme at komme til

Fig. 10.172. Grundelementer i brugercentrering.

10.3. SOFTWARE-ARKITEKTUR

I dette afsnit kigger vi på principper, som hjælper med at skrive *god software*.

ORDFORKLARING

SOFTWARE-SYSTEM

Et større program eller en samling af samarbejdende programmer med et bestemt formål. Programmerne er typisk spredt over flere computere, eventuelt i et større netværk. Et software-system har brugere og systemudviklere. Systemudviklerne er programmører, der har lavet systemet og står for vedligeholdelse og opdateringer af systemet.

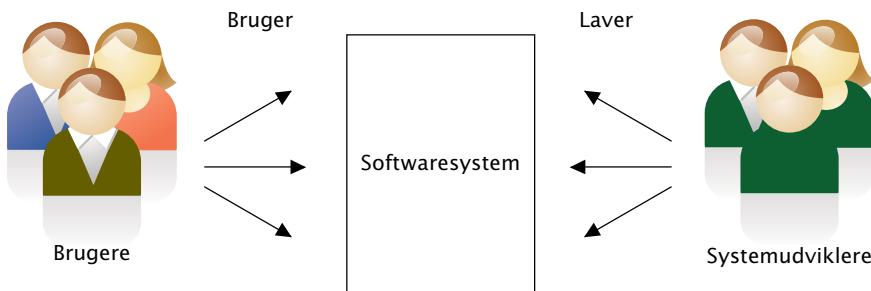


Fig. 10.173. Et softwaresystem med brugere og systemudviklere.

EKSEMPEL

SOFTWARE-SYSTEMER

Software-systemer er fx systemer til online-billetbestilling, styreprogrammer til rumfærger, søgemaskiner, portaler eller budgetsystemer til virksomheder.

Et godt software-system har en række forskellige kvaliteter.

SYSTEM-KVALITET	BESKRIVELSE	EFTERSPØRGES AF
Brugbart og brugervenligt	Systemet tilbyder relevante funktioner. Systemet er let at komme til og let og hurtigt at bruge.	Brugere
Korrekt og robust	Systemet er fejlfrit (korrekthed). Systemet har på forhånd have en plan parat, hvis der alligevel indtræffer fejl eller uregelmæssigheder (robusthed).	Alle
Fleksibelt	Programmet har dele, der kan udskiftes. Programmet kan let ændres eller udvides (så fejl kan for eksempel rettes).	Systemudviklere
Let at genbruge	Systemets enkeltdele kan i høj grad genbruges i fremtiden	Systemudviklere
Let at forstå og teste	Systemet er opdelt i overskuelige dele, der hver har en klarlagt funktion. Systemet er veldokumenteret. Det er let at teste, om systemet virker korrekt.	Systemudviklere

Fig. 10.174. Systemkvaliteter.

EKSEMPEL

SOFTWARE-SYSTEM MED KVALITETSMÄNGLER

Ariane 5 er et europæisk raket-affyringssystem. Den 4. juni 1996 skulle Ariane 5's første (ubemandede) test-raket, Flight 501, affyres. På grund af en fejl i kontolsystemets software forlod raketten sin planlagte bane efter 37 sekunder. Raketten blev nødt til at selv-destruere og brænde op i atmosfæren for ikke at forårsage en endnu større ulykke. Det er en af de mest berømte softwarefejl i historien.

Så hvordan sikres det, at systemets kvalitet bliver høj? Det fundamentale problem er:

Der er grænser for, hvor meget et menneske kan forstå på én gang.

Software-systemer er ofte meget indviklede med kildekode på hundrede-tusinder eller endda millioner af linier kode! Fx er operativsystemet Windows XP på omkring 40 millioner linier kode. For at et menneske skal kunne overskue millioner af linie kode, må koden være organiseret på en logisk måde og opdelt i logiske dele, der kan overskues hver for sig.

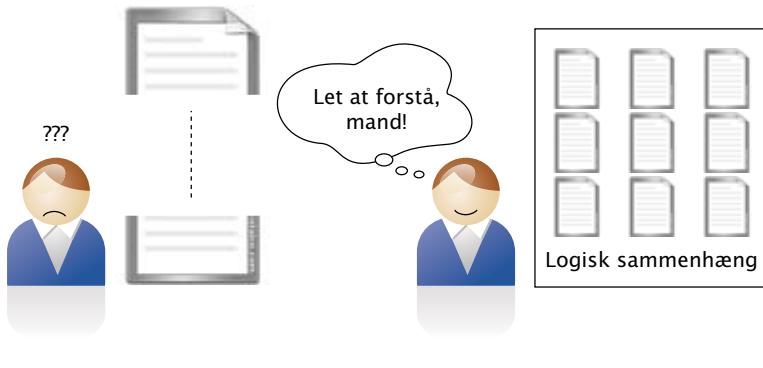


Fig. 10.175. Millioner af linier kode. Ud i et? Nej tak. Opdelt? Ja tak.

ANALOGI

KLÆDESKAB

De fleste mennesker opbevarer deres tøj (~ kodelinjer) sorteret efter type – nederdele, sokker og fjerboaer hver for sig – på forskellige hylder i klædeskabet (~ opdelt i logiske enkelt-dele). På den måde er det hurtigt at finde tøj, man skal bruge, og overskue den samlede mængde tøj. Alternativet – en “tøj-mødding” midt på gulvet – er hverken let at overskue eller finde tøj i.

Opdeling (del og hersk!) er blot et af flere principper indenfor design af en *god software-arkitektur*.

ORDFORKLARING

SOFTWARE-ARKITEKTUR

Strukturen af et software-system.

En rodet software-arkitektur giver fejl og er umulig at vedligeholde. En god software-arkitektur, derimod, er ikke alene overskuelig, men hjælper med at højne mange af systemkvaliteterne i Figur 10.174. Vi giver nu en række bredt accepterede principper for god software-arkitektur.

PRINCIP 1

OPDEL I DELPROBLEMER

Opdel systemet i dele, der løser hver deres delproblem.

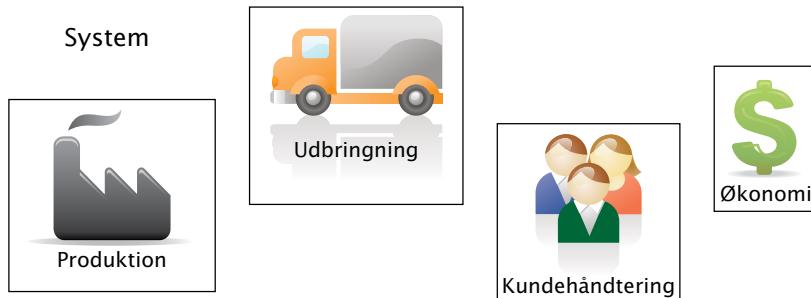


Fig. 10.176. Et system opdeles i delsystemer, der har hver sit ansvarsområde.

Ikke enhver inddeling er en god inddeling. Vi skal derfor undersøge, hvad der gør en inddeling brugbar. En god opdeling kombinerer flere måder at opdele problemområdet på. Figur 10.176 viser et system, der opdelt efter problemområdet. Det er også meget almindeligt opdele langs aksen fra “menneskenær” til “systemnær”:

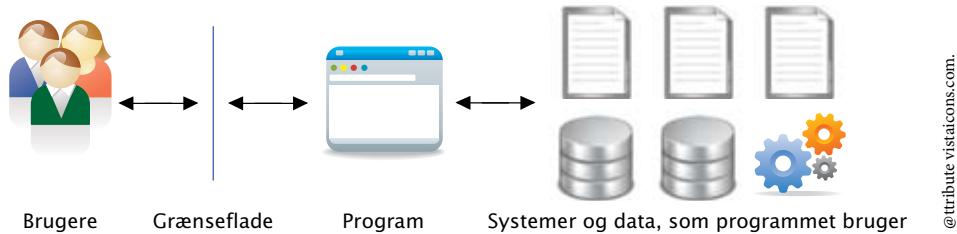
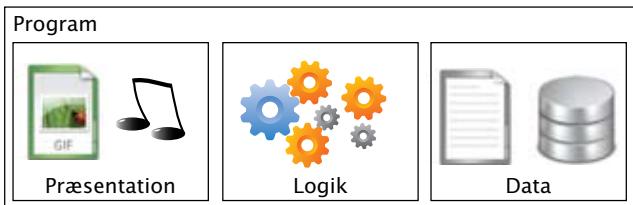


Fig. 10.177. Fra menneskenær til systemnær.

En typisk opdeling langs denne akse er følgende 3-delning:

1. *Præsentations-del*: Præsenterer programmer til brugere, fx som åbent vindue med knapper og grafik. Modtager bruger-input.
2. *Logik-del*: Foretager beregninger og reagerer på bruger-input.
3. *Data-del*: Henter og gemmer data og indstillinger.



@attribute vistaicons.com.

Fig. 10.178. Opdeling af program i præsentation, logik og data.

EKSEMPEL

OPDELING I PRÆSENTATION, LOGIK OG DATA

Et tekstbehandlingsprogram kan fremvise den skrevne tekst og modtage tastatur-indtastninger (præsentation), omsætte indtastningerne til faktiske programhandlinger (logik) og gemme dokumenter (data).

Princip # 1 kaldes også princippet om *separation of concerns* (adskillelse af bekymringer). Det bruges allevegne: Et hotel fungerer fint, selvom hotellets økonomiske direktør ikke kan rede en seng, og stuepigen ikke kender til hotellets finanser – hver varetager sin opgave og stoler trygt på, at den anden varetager sin opgave på bedste vis. På samme måde kan et computerprogram opdeles i dele, der – uden at bekymre sig om hinandens anliggender – løser hver deres opgave. For at realisere **Princip # 1** bruges tit principperne # 2, # 3 og # 4:

PRINCIP 2

GRÆNSEFLADER OG INDKAPSLING

Forskellige dele af et system må kun kommunikere gennem veldefinerede grænseflader. Delsystemers indre skjules for resten af systemet, som kun kender delsystemets grænseflade.

Hvis et system har en grænseflade, behøver systemets brugere (mennesker eller andre systemer) ikke bekymre sig om, hvordan det virker – bare systemets opfører sig, som grænsefladen fortæller. Man siger, at et delsystems grænseflader *abstraherer* delsystemets indre væk: Brugerne behøver kun tænke på, hvad delsystemet gør, ikke hvordan – hvilket bidrager til en sammenhængende, abstrakt forståelse.

EKSEMPEL

BIL: GRÆNSEFLADE

En bil har en grænseflade: Der er et rat, en gearstang, gaspedal, kobling, bremser, mv. Man kører bilen ("kommunikerer med den") ved at dreje rattet, trykke på gaspedalen, mv. ("kommunikation gennem grænsefladen"). En bilist bekymrer sig ikke om, hvorfor bilen virkelig kan dreje, bare fordi rattet drejes, eller hvordan motoren virker – bare bilen opfører sig som forventet.

EKSEMPEL

SODAVANDSAUTOMAT: GRÆNSEFLADE

En sodavandsautomat har en grænseflade: Der er et hul til møntindkast, en knap til at vælge sodavand med og en skuffe, sodavanden leveres i. Automaten betjenes ved at trykke på knappen og indkaste mønster. Tørstige automat-brugere bekymrer sig ikke om, hvordan automatens indre virker, bare den giver den ønskede sodavand.

Grænseflader muliggør, at implementationen af et indkapslet delsystem problemfrit kan udskiftes, uden at resten af systemet skal opdateres. Fx kan motoren i en bil skiftes, eller mekanikken i sodavandsautomaten udskiftes, uden at bilister og tørstige bagefter skal finde nye måder at betjene apparaterne på.

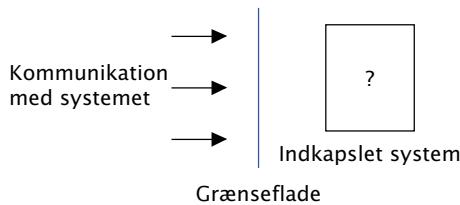


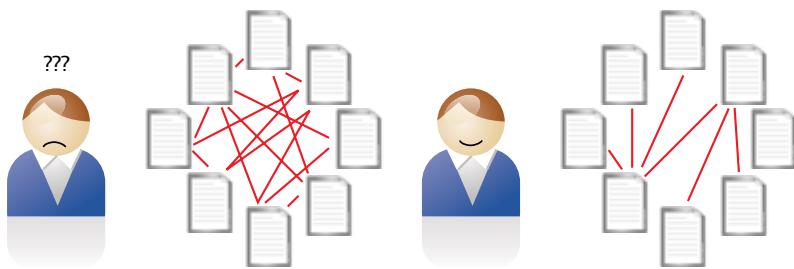
Fig. 10.179. Grænseflade og indkapsling.

Hvis et delsystem A har brug for at kommunikere med et andet delsystem B, siger man, at A er afhængigt af B. For at gøre det lettest muligt at udskifte et delsystem har man følgende princip:

PRINCIP 3

MINIMER AFHÆNGIGHEDER

Lad kun delsystemer snakke sammen, hvis det er absolut nødvendigt.



@tttribute vistaicons.com.

Fig. 10.180. Få afhængigheder er bedre end mange afhængigheder.

PRINCIP 4

SVAG KOBLING

Delsystemer, der snakker sammen, skal udveksle så lidt og så simpel information som muligt. Det betyder, at alle grænseflader skal være så små og simple som muligt.



@tttribute vistaicons.com.

Fig. 10.181. Svag kobling er bedre end stærk kobling.

Disse principper kan opsummeres som

PRINCIP 5

OPDEL I MODULER (= PRINCIPPERNE # 1, # 2, # 3 OG # 4)

Opdel systemet i *moduler*, der kommunikerer mindst og enklest muligt med hinanden.

ORDFORKLARING

MODUL

Et modul er et stykke software, der er indkapslet af en grænseflade, og som løser et bestemt overordnet problem. Et modul har så få afhængigheder som muligt, og de afhængigheder, der er, er tydeligt angivet. Et modul skal gerne kunne genbruges i senere programmer.

EKSEMPEL

MODULER

Moduler kan fx være filer, funktioner, programbiblioteker, delvist selvstændige programmer eller, i objekt-orienterede sprog, klasser.

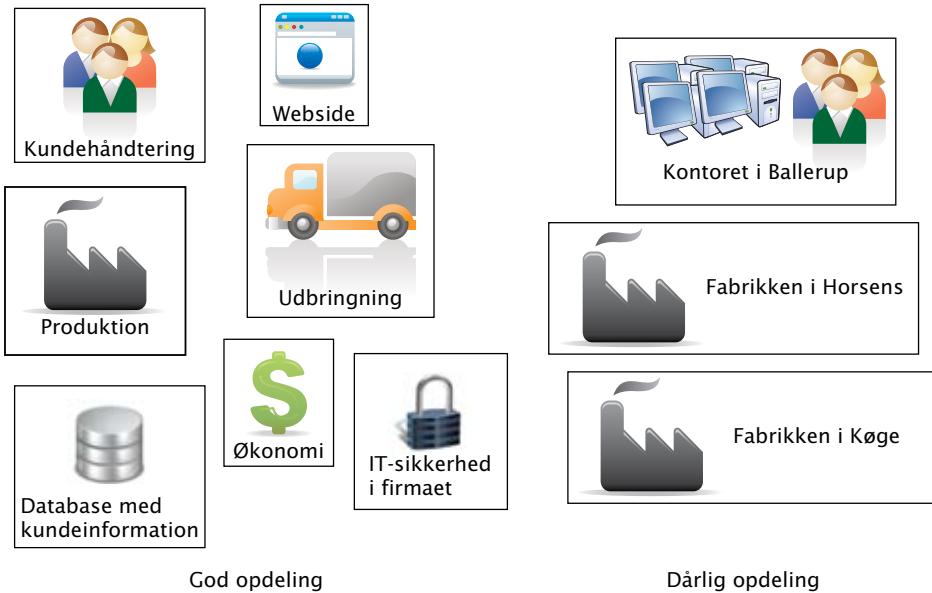
Moduler i en software-arkitektur bør svare til de problemer og delproblemer, der skal løses. Uhensigtsmæssig modellering resulterer i ”forkerte” moduler: De er svære for programmører at kode, og skulle det alligevel lykkes, er resultatet et system, som brugerne har svært ved at bruge, og som er dyrt og svært at vedligeholde.

PRINCIP 6

GOD MODELLERING

Brug en god model af den opgave, systemet løser, som udgangspunkt for selve koden.

Vi skal altså designe de ”rigtige moduler”. Godt design kræver erfaring og kendskab til de problemer, modulerne skal løse.



@tribute vistaicons.com

Fig. 10.182. God og dårlig opdeling i moduler.

Hvis modelleringen er god, behøver modellen ikke blive udskiftet, selvom brugere af et system skulle opfinde nye ønsker til systemet.

PRINCIP 7

SKRIV KUN KODE ÉN GANG!

Hvis det er vigtigt at to kopier af et stykke kode altid er i overensstemmelse med hinanden – så skal der ikke være to kopier.

Skriver man den samme kode flere gange, kunne man ikke bare have brugt sin tid bedre, men man har også startet et vedligeholds-problem: De to steder, koden er skrevet, holder sig ikke synkroniseret af sig selv.

Tilsammen sikrer de principper, vi har skitseret her, at programmet er fleksibelt, let at genbruge dele af og let at forstå og teste – fordi dele kan testes separat. En god modellering medvirker også til at sikre brugbarhed og brugervenlighed, fordi modellen afspejler brugeres behov og let kan justeres til ændringer i brugeres behov. Principperne sikrer ikke direkte systemkvaliteterne korrekthed og robusthed, men – om ikke andet – er det lettere at skrive korrekt og robust kode, hvis koden er organiseret i en god arkitektur.

STIKORD

1-til-1-relation, 168
1-til-mange-relation, 167
3D-grafik, 215

A

ACK, 102, 117
ADD (FIMA-instruktion), 36
Additiv farvemodel, 197
Adgangskontrol, 241
Adresse (i hukommelse), 16
Adresserum, 52
AES, 250
Afbrydelse, 47
Afhængighed, 184
Aktør, 225
Algoritme
 deterministisk, 123
 eksempel på, 121, 122, 123
 fuldstændig, 123
 hashing-, 256
 krypterings-, 248
 ordforklaring, 123
 programmer som algoritmer, 121
 routing-, 108
 stoppende, 123
 til digital signatur, 259
ALU
 aritmetik, 26
 division med 0, 28
 ordforklaring, 22
 overflow, 28
Analogi
 aflåste kister, 252
 brevskrivende børnefamilier, 99
 cookies ≈ post-its, 79

forwarding ≈ vejvisning i rundkørsel, 107
instruktioner ≈ legoklodser, 23
klædeskab, 275
modtage email ≈ samtale, 83
mærkeligt værtshus, 78
operativsystem ≈ regering og politi, 45
personnummer og postadresse, 111, 113
pålidelig og upålidelig transport, 101
routere er som rundkørsler, 90
skriftlig oversættelse vs.
 simulantolkning, 137
streaming af middagsmåltid, 222
tragt og kalahakugler, 217
variabel ≈ papirlap, 142
Analyse (del af datalogisk arbejdsproces), 266
Antivirus-software, 244
Applikationslag, 97
Arbejdsproces. Se Datalogisk arbejdsproces
Ariane 5, 274
Aritmethic-logical unit. Se ALU
ARP, 112
ARP-tabel, 112
Array. Se Liste
Assemblerkode, 35
Asymmetrisk kryptering
 eksempel på, 251, 253, 258
 ordforklaring, 250
Asynkron beskedudveksling, 60, 61
Attributter
 entitet-attributter, 162, 164
 objekt-attributter, 154

- Autentificering
autentificering 1.0-protokollen, 242
autentificering 2.0-protokollen, 242
eksempler på, 243
forklaring af, 243
med random challenge, 254
- Autorisation, 241, 242
- Autoritativ navne-server, 86
- Awareness, 246
- B**
- Backbone, 90
- Best effort, 105, 219
- Binære tal, 16
- BIOS, 64
- Bitmap-grafik, 201
- Bitrate, 211
- Bits, 16
- Bootchip, 62
- Bootstrapping, 62
- Brainstorm (del af datalogisk arbejdsproces), 266
- Broadcast, 110
- Browser, 65
- Brugercentrering, 272
- Brugergrænseflade, 271
- Brugertilstand, 49
- Brugervejledning, 270
- Brugervenlighed, 271
- Bug (fejl i programkode), 269
- Bundkort, 13
- Bus, 14
- Bytes, 16
- Båndbredde, 90
- C**
- CA, 259
- Central processing unit. Se Processor
- CERT, 233
- Certifikat, 259
- Challenge-response, 254
- Chat, 83
- Checksum, 101, 108
- CIA, 226
- Cifertekst, 248
- Clock-cyklus, 25
- Clock-frekvens, 25
- CMY, 197
- Computer, 11
- Cookie, 79, 80
- CPU. Se Processor
- Cæsars kryptering, 248
- D**
- Data, 124
- Database
1NF, 181
2NF, 183
3NF, 185
databegrænsninger, 186
eksempel på, 159
integritet af, 179, 193
manipulation af, 160
normalformer, 181
ordforklaring, 159
primærnøgle, 165
relationel, 175
stærk 1NF, 183
tabel, 175
udtræk, 160
- Datacelle (i hukommelse), 16, 30
- Data-link, 88
- Data-link-lag, 110
- Datalogisk arbejdsproces, 265
analyse, 266
brainstorm, 266
dokumentation, 270
fejlretning (debugging), 269
implementation, 268
kravspecifikation, 266
modellering og design, 267

- test, 269
 unit test, 269
 use cases, 266
 Datamodel, 161
 Datastruktur
 eksempler på, 145
 liste, 144
 ordforklaring, 145
 Datatype
 eksempler på, 130, 148
 ordforklaring, 148
 DDL, 161
 DDoS, 239
 Debugging (del af datalogisk arbejdsproces), 269
 Del og hersk, 275
 Delklasse, 157
 Denial of service. Se DoS
 Dereferere (at dereferere en variabel), 142
 DES, 250
 Design (del af datalogisk arbejdsproces), 267
 Diffie-Hellman-protokollen, 253
 Digital signatur, 258, 259
 Distribueret denial of service. Se DDoS
 DML, 160
 DNS
 ordforklaring, 86
 servicemodel, 98
 Dokumentation (del af datalogisk arbejdsproces), 270
 Domain name system. Se DNS
 Domæne, 85
 Domæne-begrænsning (databegrænsning i database), 190
 Don't make me think, 272
 DoS, 239
 Dots, 205
 DPI, 205
 Driver, 15
 Dynamisk lagerallokering, 52
 Dynamisk webside, 69
- E**
 E/R-diagram, 169, 170, 174, 175
 E/R-modellen
 eksempel med Datahøj Gymnasium, 172
 forklaring af, 162
 modellering, 171
 Eksplicit begrænsning (databegrænsning i database), 192
 Email, 81
 Enhed
 eksempler på, 13
 I/O-enhed, 14
 ordforklaring, 13
 regneenhed. Se ALU
 ydre enhed, 14
 Enkelt-cyklus-arkitektur, 26
 Entitet, 162, 163
 Entitetsklasse, 163
 Erklæring (af variabel), 142
 Ethernet, 113
 Evaluering (af udtryk), 148
 Event, 71
- F**
 Farvedybde, 198
 Farvemodel, 197
 Fejlretning (del af datalogisk arbejdsproces), 269
 Fetch/execute
 eksempel på
 instruktionsafvikling, 32
 ordforklaring, 25
 Fil, 18
 Fildeling, 83
 Filsystem, 54
 Filtrering af netværkstrafik, 245
 FIMA
 eksempel på FIMA-program, 39, 40
 FIMA-simulator, 36
 forklaring af, 35
 Firewall, 244

Fler-cyklus-arkitektur, 26

Forgrening, 149

Fortolkning, 137

Fortrolighed, 226, 237

Forwarding, 106, 116

Frame, 111, 116

Fremmednøgle, 187

Fuldstændig deltagelse (i relation), 168

Fysisk lag, 113

G

GET (HTTP-metode), 76

Grafik, 195

Grænseflade

bruger-, 271

eksempler på, 271, 278

mellom protokoller i

protokolstak, 95

som princip i software-arkitektur, 277

til database, 160

til kommunikation med objekt, 154

til protokol-software, 97

H

Handshake, 102

Harddisk, 17

Hardware, 11

Hashing, 255

Heltal (datatype), 131

Hexadecimale tegn, 111

Hjemmeside. Se Webside

HTML, 69

HTTP

metoden **GET**, 76

metoden **POST**, 76

ordforklaring, 74

request, 76

response, 77

servicemodel, 98

som eksempel på protokol i

applikationslaget, 97

som tilstandsløs protokol, 78

statuskoder og statusbeskeder, 77

Huffman-kodning, 207

Hukommelse

adresse i, 16

controller, 16

datacelle i, 16

ordforklaring, 15

som del af computerens hjerne, 12

Højniveau-programmeringssprog, 41,

134

I

I/O, 14

I/O-enhed, 14

Identifikation, 242

IDS. Se Intrusion detection system

IF THEN ELSE, 149, 150

Ikke-observabilitet, 229

Imperativtøvesprog, 142

Implementation

af programmeringssprog, 140

af protokols servicemodel, 95

del af datalogisk arbejdsproces, 268

Index (i liste), 145

Indkapsling, 277

Information, 124

Instans

af klasse, 156

af program, 56

Instant messaging, 83

Instruktion

afvikling af instruktioner, 22, 32

clock-cyklus, 25

ikke-privilegeret, 49, 51

kan indlæses som data, 30

ordforklaring, 20

privilegeret, 49, 51

Instruktionssæt, 23

integer, 142

Integritet (mål med IT-sikkerhed), 226, 237
Internettet, 65, 88
Interrupt handler, 47
Interrupt request. Se IRQ
Intrusion detection system, 245
IP, 108
IP-adresse, 107, 113, 115
eksempel på, 85
lokale IP-adresser, 109
offentlige IP-adresser, 109
ordforklaring, 85
oversættelse fra webadresse til IP-adresse med DNS, 86
IRQ, 47
IRQ-registeret, 30
ISP, 90
IT-sikkerhed
fysisk, 246
og brugervenlighed, 261
og omkostninger, 261
ordforklaring, 224
IT-system, 225

Klient/server-princippet
chat, 83
DNS, 86
eksempler på, 67
email, 81
ordforklaring, 67
push og pull, 82
Kodeord, 232
Komma-tal (datatype), 131
Komprimering
af grafik, med tab, 208
af grafik, tabsfri, 205
af lyd, 212
af video, 214
Huffman-kodning, 207
Kontrolstruktur, 148
Kravspecifikation, 266
Krop (i definition af metode), 146
Krypteringsalgoritme, 248
Kryptografi, 247
Kvittering, 103

L

Lagdelt kommunikation i netværk
applikationslag, 93, 97
data-link-lag, 93, 110
eksempel med alle fem lag i
aktion, 114
fysisk lag, 93, 113
netværkslag, 93, 105
transportlag, 93, 98
LAN, 88
Lavniveau-programmeringssprog, 41, 134
Least privileges, 241
Liste, 144, 154
LOAD (FIMA-instruktion), 38
Logisk og fysisk adresserum, 53
Logiske udtryk, 151
Lokal navne-server, 86
Lydbølge, 209
Løkke, 151

J

Jeg er en falk. Ja!
JUMP (FIMA-instruktion), 40
JUMPOZERO (FIMA-instruktion), 40
JVM, 139

K

Kandidatnøgle, 166
Kardinalitet, 166, 170
Kerberos, 250
Kerckhoffs princip, 248
Kerne, 46
Kerneltilstand, 49
Kildekode, 127
Klartekst, 248
Klasse, 156

M

- MAC-adresse, 113, 115
 - ordforklaring, 111
- MAC-protokol, 111
- Malware, 233
- Man in the middle-angreb, 238
- Mange-til-mange-relation, 168
- Markup-kode, 68
- Maskinkode, 34
- Mellemkode, 138
- MEMWRITE** (FIMA-instruktion), 38
- Metode, 146, 147, 155
- Metodekald, 146
- MITM-angreb. Se Man in the middle-angreb
- Model
 - E/R-modellen, 162
 - fra virkelighed til repræsentation, 125
 - modellering som del af datalogisk arbejdsproces, 267
 - opdeling i moduler, 280
 - som forenkling, 126
- Modmiddel
 - eksempler på, 228
 - ordforklaring, 228
 - typer af, 240
- Modul, 280
- Mulig deltagelse (i relation), 168
- Multimedie, 194, 195

N

- NAT, 109
- Nedarvning (fra superklasse til delklasse), 157
- Netværkslag, 105
- NIST, 250
- Normalformer (for database), 181
- null, 143, 165
- Nøgle (input til krypteringsalgoritme), 249

O

- Objekt-orienteret programmering, 152
 - delklasse, 157
 - klasse, 156
 - objekt, 153
 - attributter, 154
 - eksempel på, 153, 154
 - metoder, 155
 - ordforklaring, 153
- Obligatorisk deltagelse (i relation), 168
- Offentlig nøgle, 250
- OOP. Se Objekt-orienteret programmering
- Operand (i ALU-operation), 27
- Operativsystem, 43, 44, 45
- Opløsning, 201, 203
- Orm, 234
- OS. Se Operativsystem
- Oversættelse
 - ordforklaring, 135
 - trin for trin, 135

P

- P2P, 83
- Packet sniffing, 237
- Pakke, 105, 115
- Paradigmer (indenfor programmeringssprog), 140
- Parametre, 146
- PC-registeret, 30
- Phishing, 237
- Pipelinet arkitektur, 26
- Pixel, 196
- PKI, 259
- Platform, 43
- Point-to-point, 110
- Port, 99, 115
- POST (HTTP-metode), 76
- Post (kommunikation med tre lag), 91
- POST (power on self-test), 62
- Primærnøgle, 165, 166
- Primært lager, 16

- `print`, 142
 Privacy, 229
 Privat nøgle, 250
 Proces
 adressering over netværk, 99
 ordforklaring, 55
 procestilstande, 58
 Processskift, 57
 Processor
 ordforklaring, 22
 som del af computerens hjerne, 12
 Program, 20
 Programbibliotek, 133
 Programmering, 120
 Programmeringsprog
 implementation af, 140
 ordforklaring, 127
 paradigmer, 140
 semantik, 128
 specifikation af, 127
 standardbibliotek, 133
 syntaks, 128
 typesystem, 130
 Projekt-arkitektur, 267
 Protokol
 ARP, 112
 Diffie-Hellman, 253
 HTTP, 74
 IP, 108
 ordforklaring, 72
 SMTP, 82
 TCP, 104
 til spørgsmål og svar i
 datalogitime, 73
 UDP, 101
 Protokolstak
 internet-protokolstakken, 97
 ordforklaring, 96
 Præsentation, logik og data, 277
 Public key-infrastruktur. Se PKI
 Public key-kryptografi, 250
 Push og pull, 82
 Pålidelig transport, 100, 101, 102
 kvittering, 103
 sekvensnummer, 102, 104
 timeout, 104
- ## R
- RAM. Se hukommelse
 Random access memory. Se hukommelse
 Random challenge, 254
 Read only memory. Se ROM
 Ready (procestilstand), 57
 Register
 IR, 24, 30, 33
 ordforklaring, 29
 PC, 24, 30, 33
 R0, R1, R2 og R3, 36
 Regneenhed. Se ALU
 Relation, 162, 166
 Relationel database, 175, 176
 Replay-angreb, 238
 Repræsentation
 af data, 124
 analog, 126, 209
 digital, 126, 209
 Request og response, 66, 74
 Returtype, 146
 Returværdi, 146
 RGB, 197
 Rod-navne-server, 86
 ROM, 17
 bootchip, 63
 MAC-adresse, 111
 Router, 88, 106, 116
 ordforklaring, 89
 Routing, 106, 108
 RSA, 253
 Running (procestilstand), 56
- ## S
- Sampling, 210
 bitdybde, 210
 samplingsfrekvens, 210

- Samtidighed, 58
 Sandhedsværdi (datatype), 131
 Script
 - client-side, 70
 - ordforklaring, 70
 - server-side, 70
 Segment, 100, 102, 115
 Sekundært lager, 17
 Sekvensnummer, 102
 Semantik
 - eksempler på, 129, 130
 - ordforklaring, 128
 Separation of concerns, 277
 Session, 78
 Sikkerhedspolitik (for IT-system), 241
 Skedulering, 57
 Slutsystem, 88
 SMTP
 - forklaring af, 82
 - servicemodel, 98
 - som eksempel på protokol i applikationslaget, 97
 Software, 12
 Software-arkitektur, 275
 Software-system, 273
 Spam, 236
 Spoofing, 237
 Sporbarhed, 247
 Spyware, 234
 SQL, 187
STOP (FIMA-instruktion), 37
STORE (FIMA-instruktion), 38
 Streaming
 - ekstra information i pakker, 220
 - fletning, 221
 - interpolation, 220
 - live streaming, 218
 - ordforklaring, 216
 - real time-interaktion, 219
 - repetition, 220
 - udfordringen med pakketab, 219**string**, 142
 Styresystem. Se Operativsystem
- SUB (FIMA-instruktion), 36
 Subtraktiv farvemodel, 197
 Superklasse, 157
 Svag kobling, 279
 Symmetrisk kryptering, 249
SYN, 102
 Synkron beskedudveksling, 59, 61
 Synkronisering, 59
 Syntaks
 - eksempler på, 128, 130
 - ordforklaring, 128
 Systemkald, 50
 Systemkvalitet
 - for database, 160
 - for software-system, 274
 Systemressource, 45
 Sårbarhed
 - eksempler på, 227
 - ordforklaring, 227

T

- Tabel (i database), 175
 TCP, 104
 Tegn (datatype), 131
 Tekststreg (datatype), 131
 Terminated (procestilstand), 56
 Test (del af datalogisk arbejdsproces), 269
 Test case, 269
 Tier-1, tier-2, tier-3, 90
 Tildeling (af værdi til variabel), 142
 Tilgængelighed, 226, 237, 239
 Transportlag, 98
 Trojansk hest, 235
 True color, 200
 Trussel
 - eksempler på, 227, 230
 - menneskelige fejl, 231
 - ondsindede personer, 231
 - ordforklaring, 227

Trusselsaktør
 eksempler på, 227
 ordforklaring, 227
Tråde, 61
TTL, 108
Tupel-begrænsning (databegrænsning i database), 190

U

Uafviselighed, 230
UDP, 101
Udprint af grafik, 205
Udtryk, 148
UML, 171
Uniform resource locator. *Se URL*
Upålidelig transport, 100, 101
URL, 68

V

Variabel, 142
Vektorgrafik, 203
Verifikation, 242
Video, 213
Virtuel hukommelse, 53
Virtuel maskine, 138
Virus, 233
Von Neumann-princippet, 31

W

Waiting (procestilstand), 57
WAN, 88
Web browser. *Se Browser*
Webadresse, 68
Webdokument. *Se Webside*
Webserver, 68
Webside, 68
WHILE, 152

ILLUSTRATIONER

KAPITEL 1

- Fig. 1.1: @ttribute vistaicons.com / iStock Photo, © hhv. Leo Blanchette, Timur Arbaev, Andrew Buckin, Georgios Alexandris, Luis Sandoval Mandujano, Natalia Siverina
- Fig. 1.2: @ttribute vistaicons.com / http://en.wikipedia.org/wiki/Image:Firefox-logo.svg / http://upload.wikimedia.org/wikipedia/en/1/10/Internet_Explorer_7_Logo.png / iStock Photo, © Aliaksandr Kakouka
- Fig. 1.3: @ttribute vistaicons.com / iStock Photo, © hhv. Leo Blanchette, Timur Arbaev, Georgios Alexandris
- Fig. 1.4: © Forfatteren
- Fig. 1.5: iStock Photo, © Leo Blanchette
- Fig. 1.6: © Forfatteren
- Fig. 1.7: © Forfatteren

KAPITEL 2

- Fig. 2.8: @ttribute vistaicons.com / iStock Photo, © Julie Felton
- Fig. 2.9: © Forfatteren
- Fig. 2.10: © Forfatteren
- Fig. 2.11: © Forfatteren
- Fig. 2.12: © Forfatteren
- Fig. 2.13: © Forfatteren
- Fig. 2.14: © Forfatteren
- Fig. 2.15: © Forfatteren / @ttribute vistaicons.com / http://en.wikipedia.org / wiki/Image:Firefox-logo.svg / http://upload.wikimedia.org / wikipedia/en/1/10/Internet_Explorer_7_Logo.png / http://en.wikipedia.org/wiki/Image:OfficeWord.png
- Fig. 2.16: © Forfatteren
- Fig. 2.17: @ttribute vistaicons.com / iStock Photo
- Fig. 2.18: @ttribute vistaicons.com / iStock Photo, © Er Ten Hong / © Systime

KAPITEL 3

- Fig. 3.19: @ttribute vistaicons.com / iStock Photo, © Luis Sandoval Mandujano, Natalia Siverina, Nataliya Kostenyukova, Er Ten Hong http://upload.wikimedia.org/wikipedia/en/1/10/Internet_Explorer_7_Logo.png / http://en.wikipedia.org/wiki/Image:OfficeWord.png
- Fig. 3.20: © Forfatteren
- Fig. 3.21: © Forfatteren
- Fig. 3.22: iStock Photo, © Nataliya Kostenyukova / © Forfatteren

- Fig. 3.23: iStock Photo, © Nataliya Kostenyukova / © Forfatteren
Fig. 3.24: © Forfatteren
Fig. 3.25: © Forfatteren
Fig. 3.26: © Forfatteren
Fig. 3.27: © Forfatteren
Fig. 3.28: © Forfatteren / Systime
Fig. 3.29: @ttribute vistaicons.com
Fig. 3.30: © Forfatteren
Fig. 3.31: © Forfatteren
Fig. 3.32: © Forfatteren
Fig. 3.33: © Forfatteren
Fig. 3.34: © Forfatteren / Systime
Fig. 3.35: © Forfatteren / Systime
Fig. 3.36: © Forfatteren

KAPITEL 4

- Fig. 4.37: iStock Photo, © hhv. Jakub Semeniuk, Er Ten Hong, Aliaksandr Kakouka
Fig. 4.38: iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka
Fig. 4.39: @ttribute vistaicons.com / iStock Photo, © Er Ten Hong
Fig. 4.40: @ttribute vistaicons.com / iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka
Fig. 4.41: @ttribute vistaicons.com / iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka, Julie Felton
Fig. 4.42: iStock Photo, © Aliaksandr Kakouka
Fig. 4.43: tv.: © Forfatteren / th.: © Polfoto/Graham Whitby-Boot/Allstar England
Fig. 4.44: @ttribute vistaicons.com / iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka, Pierangelo Rendina, Javier Bernal Martínez
Fig. 4.45: © Forfatteren
Fig. 4.46: iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka, Edward Grajeda
Fig. 4.47: iStock Photo, © hhv. Aliaksandr Kakouka, Edward Grajeda
Fig. 4.48: @ttribute vistaicons.com / iStock Photo, © Aliaksandr Kakouka
Fig. 4.49: iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka

KAPITEL 5

- Fig. 5.50: iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka, Jim Snyder, © Systime
Fig. 5.51: iStock Photo, © Aliaksandr Kakouka, © Forfatteren/Systime
Fig. 5.52: iStock Photo, © Edward Grajeda, Marko Radunovic
Fig. 5.53: iStock Photo, © hhv. Er Ten Hong, Murat Sen, Konstantin Sukhinin
Fig. 5.54: iStock Photo / © Forfatteren
Fig. 5.55: © Forfatteren
Fig. 5.56: © Forfatteren
Fig. 5.57: © Forfatteren
Fig. 5.58: © Forfatteren
Fig. 5.59: iStock Photo, © hhv. Er Ten Hong, Julie Felton, Marko Radunovic
Fig. 5.60: © Forfatteren
Fig. 5.61: iStock Photo, © hhv. Jim Snyder, Marko Radunovic, Aliaksandr Kakouka

Fig. 5.62: © Forfatteren
Fig. 5.63: © Forfatteren
Fig. 5.64: © Forfatteren
Fig. 5.65: © Forfatteren
Fig. 5.66: © Forfatteren
Fig. 5.67: iStock Photo, © Marko Radunovic
Fig. 5.68: © Forfatteren
Fig. 5.69: iStock Photo, © hhv. Jim Snyder, Marko Radunovic, Aliaksandr Kakouka
Fig. 5.70: iStock Photo, © hhv. Jim Snyder, Aliaksandr Kakouka
Fig. 5.71: iStock Photo, © Aliaksandr Kakouka
Fig. 5.72: © Forfatteren
Fig. 5.73: iStock Photo, © hhv. Jim Snyder, Aliaksandr Kakouka/© Systime
Fig. 5.74: iStock Photo, © hhv. Jim Snyder, Aliaksandr Kakouka
Fig. 5.75: © Forfatteren
Fig. s. 114a: iStock Photo, © hhv. Er Ten Hong, Marko Radunovic
Fig. s. 114b: iStock Photo, © hhv. Er Ten Hong, Marko Radunovic, Julie Felton
Fig. s. 116: iStock Photo, © Jim Snyder / © Forfatteren
Fig. s. 118: © Forfatteren

KAPITEL 6

Fig. 6.76: © Forfatteren / Systime
Fig. 6.77: iStock Photo, © hhv. David Foreman, Onur Döngel / © Forfatteren
Fig. 6.78: iStock Photo, © Jacob Semeniuk / © Forfatteren
Fig. 6.79: © Forfatteren
Fig. 6.80: @ttribute vistaicons.com / © Forfatteren
Fig. 6.81: @ttribute vistaicons.com / © Forfatteren
Fig. 6.82: @ttribute vistaicons.com / © Forfatteren
Fig. 6.83: @ttribute vistaicons.com / iStock Photo
Fig. 6.84: @ttribute vistaicons.com / iStock Photo
Fig. 6.85: © Forfatteren
Fig. 6.86: © Forfatteren
Fig. 6.87: @ttribute vistaicons.com
Fig. 6.88: @ttribute vistaicons.com
Fig. 6.89: © Forfatteren
Fig. 6.90: @ttribute vistaicons.com / iStock Photo
Fig. 6.91: @ttribute vistaicons.com / iStock Photo
Fig. 6.92: @ttribute vistaicons.com / iStock Photo
Fig. 6.93: @ttribute vistaicons.com / iStock Photo
Fig. 6.94: © Forfatteren
Fig. 6.95: © Forfatteren
Fig. 6.96: © Forfatteren
Fig. 6.97: @ttribute vistaicons.com
Fig. 6.98: © Forfatteren
Fig. 6.99: © Forfatteren
Fig. 6.100: © Forfatteren
Fig. 6.101: © Forfatteren

KAPITEL 7

- Fig. 7.102:@ttribute vistaicons.com / iStock Photo
Fig. 7.103:@ttribute vistaicons.com / iStock Photo
Fig. 7.104:@ttribute vistaicons.com / iStock Photo, Julie Felton / © Forfatteren
Fig. 7.105:iStock Photo, © Marko Radunovic
Fig. 7.106:iStock Photo, © Er Ten Hong
Fig. 7.107:iStock Photo
Fig. 7.108:© Forfatteren
Fig. 7.109:© Forfatteren
Fig. 7.110:© Forfatteren
Fig. 7.111:© Forfatteren
Fig. 7.112:© Forfatteren
Fig. 7.109:© Forfatteren
Fig. 7.113:© Forfatteren
Fig. 7.114:© Forfatteren
Fig. 7.115:© Forfatteren
Fig. 7.116:© Forfatteren
Fig. 7.117:© Forfatteren
Fig. 7.118:© Forfatteren
Fig. 7.119:@ttribute vistaicons.com / iStock Photo
Fig. 7.120:© Forfatteren

KAPITEL 8

- Fig. 8.121:iStock Photo / © Forfatteren
Fig. 8.122:iStock Photo, © Alejandro Raymond / © Forfatteren
Fig. 8.123:iStock Photo, © Aliaksandr Kakouka / © Forfatteren
Fig. 8.124:© Forfatteren
Fig. 8.125:© Systime
Fig. 8.126:© Systime
Fig. 8.127:© Systime
Fig. 8.128:iStock Photo, © Alicia Higgins
Fig. 8.129:@ttribute vistaicons.com
Fig. 8.130:@ttribute vistaicons.com
Fig. 8.131:@ttribute vistaicons.com / iStock Photo, © Julie Felton
Fig. 8.132:iStock Photo, © Er Ten Hong
Fig. 8.133:iStock Photo, © hhv. Tim Stapleton, Richard Simpkins
Fig. 8.134:@ttribute vistaicons.com / iStock Photo, © hhv. Er Ten Hong, Aliaksandr Kakouka,
Fig. 8.135:© Forfatteren
Fig. 8.136:iStock Photo, © Er Ten Hong
Fig. 8.137:© Systime
Fig. 8.138:iStock Photo, © Aliaksandr Kakouka / © Forfatteren
Fig. 8.139:@ttribute vistaicons.com / iStock Photo, © Georgios Alexandris
Fig. 8.140:@ttribute vistaicons.com / iStock Photo/ © Forfatteren
Fig. 8.141:© Systime A/S
Fig. 8.142:iStock Photo, © hhv. Er Ten Hong, Marko Radunovic
Fig. 8.143:iStock Photo, © Er Ten Hong

Fig. 8.144:© Forfatteren
Fig. 8.145:© Forfatteren
Fig. 8.146:© Forfatteren

KAPITEL 9

Fig. 9.147:© Forfatteren
Fig. 9.148:© Forfatteren
Fig. 9.149:<http://www.geeks.com/techtips/2006/Images/virus-tt94.gif>
Fig. 9.150:iStock Photo, © Mark Stay
Fig. 9.151:iStock Photo
Fig. 9.152:iStock Photo, Er Ten Hong
Fig. 9.153:iStock Photo
Fig. 9.154:iStock Photo, © Aliaksandr Kakouka
Fig. 9.155:iStock Photo, © Er Ten Hong / © Forfatteren
Fig. 9.156:iStock Photo, © Er Ten Hong / © Forfatteren
Fig. 9.157:iStock Photo, © Jim Snyder
Fig. 9.158:iStock Photo
Fig. 9.159:@ttribute vistaicons.com / iStock Photo, © Todd Harrison
Fig. 9.160:@ttribute vistaicons.com / iStock Photo, © Todd Harrison
Fig. 9.161:@ttribute vistaicons.com / iStock Photo, © Todd Harrison
Fig. 9.162:@ttribute vistaicons.com / iStock Photo, © Todd Harrison
Fig. 9.163:@ttribute vistaicons.com / iStock Photo, © Todd Harrison
Fig. 9.164:@ttribute vistaicons.com
Fig. 9.165:@ttribute vistaicons.com / iStock Photo, © Todd Harrison
Fig. 9.166:iStock Photo
Fig. 9.167:© Forfatteren
Fig. 9.168:© Forfatteren / Systime
Fig. 9.169:© Forfatteren / Systime

KAPITEL 10

Fig. 10.170: © Forfatteren / Systime
Fig. 10.171: © Forfatteren
Fig. 10.172: © Forfatteren
Fig. 10.173: iStock Photo
Fig. 10.174: © Forfatteren
Fig. 10.175: @ttribute vistaicons.com / iStock Photo
Fig. 10.176: iStock Photo, © Er Ten Hong, Marko Radunovic
Fig. 10.177: @ttribute vistaicons.com / iStock Photo, © hhv. Er Ten Hong, Julie Felton
Fig. 10.178: @ttribute vistaicons.com / iStock Photo, © Julie Felton
Fig. 10.179: © Forfatteren
Fig. 10.180: @ttribute vistaicons.com / iStock Photo
Fig. 10.181: @ttribute vistaicons.com / iStock Photo
Fig. 10.182: @ttribute vistaicons.com / iStock Photo, hhv. Er Ten Hong, Marko Radunovic