DKuug nr. 181 Juli 2016

DKuug NYT

OSD-2016 blev afholdt på Metropol - professionshøjskole i København

Mødes

Dele

Erfare

Lære

Dansk Forum for Åbne Systemer

DKUUG - Unix Brugere (Linux/BSD) system administratorer

Community for IT-specialister og IT-interesserede.

Erlang: Actor programmeringsmodellen
Billeder fra OSD, kommandolinien side 19
- og lidt mere

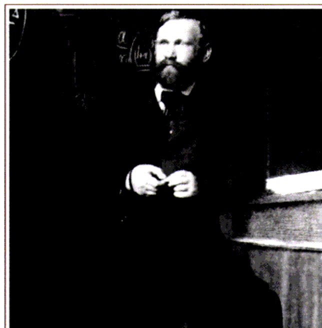
De måske egnede ...

Jagten efter det perfekte programmeringssprog

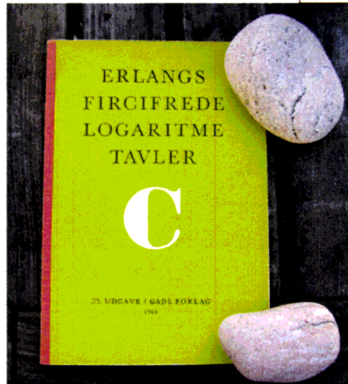
En skolekammerat, der efter at have været skak-mester og kanon-skuds-beregner blev computer-specialist, sagde disse kloge ord: ***Jeg sværger ikke til ét programmeringssprog, men bruger det sprog, som egner sig bedst til den foreliggende opgave.***

Erlang er hverken værre eller bedre end andre programmeringssprog, men har en række måder at skrive parallel processer, som gør sproget interessant og egnet til en bestemt type opgaver.

Erlang programmeringssproget er opkaldt efter danskeren Agner Krarup Erlang.
Samtidigt er det jo ***Er****icssons* ***Lang****uage*!


Agner Krarup Erlang - med kridt i hånd i en undervisningssituation

Vi er nok stadig nogle stykker, der kan huske at gymnasiets logaritmetabeller var lavet af en dansk matematiker - Erlang. Han kom fra et fattigt hjem, men med hjælp fra legater og fjerne slægtninge fik han mulighed for at studere matematik på Københavns Universitet, hvor han tjente til videre studier ved at undervise. Senere blev han ansat i Københavns Telefon A/S (KTAS) og det var her, at han udviklede queueing matematik.



Han ønskede at kunne beregne en passende størrelse af en telefoncentral, og også at kunne anslå et rimeligt tal for hvor mange telefondamer, man burde ansætte, for at kunderne fik god service.

Men A.K.Erlang havde altså ikke noget at gøre med programmeringssproget Erlang.

Hvorfor endnu et sprog?

Tim Bray fra det nu hedengangne SUN Microsystems, udtrykte sin beundring for programmeringssproget Erlang på denne måde, citatet er hentet fra Wikipedia.org artiklen om Erlang (Programming Language):

> *If somebody came to me and wanted to pay me a lot of money to build a large scale message handling system that really had to be up all the time, could never afford to go down for years at a time, I would unhesitatingly choose Erlang to build it in. (Tim Bray, director of Web Technologies at Sun Microsystems, expressed in his keynote at OSCON in July 2008.)*

> *Hvis nogen kom til mig og ville betale mig en masse penge for at skrive et stor-skala message-handling system, som virkelig skulle kunne køre uafbrudt hele tiden, det ville ikke være acceptabel, at det gik ned en eneste gang igennem perioder på flere år, så ville jeg uden tøven vælge Erlang til at bygge systemet. (Tim Bray, direktør ved SUN's web-teknologi division, i et foredrag ved OSCON Juli 2008.)*

Message-handling som han nævner, forekommer i mail-queue, og i packet-switching systemer af forskellig art og er med større netværk så aktuelt som nogensinde.

Tim Brays kærlighedserklæring til Erlang programmeringssproget nævner to krav til det system, som han gerne ville have mange penge for at lave: 1. Det skal være et stort system, og 2. oppetiden skal være nær ved 100%.

1. Store systemer bliver mere komplekse

Vi kan ikke undgå kompleksitet i dag - vor tids systemer bliver stadig større, typisk for at kunne give flere muligheder til brugerne, men også på grund af hurtigere hardware, som kan flere ting på én gang - GPS i mobiltelefonen, trådløse printere med drivere af alle mulige slags. Store websites, som betjener tusinder af samtidige brugere - fx. Amazon.com, som bruger Erlang til SimpleDB.

Men kompleksiteten er også en achilleshæl, fordi overskueligheden bliver mindre. Et ad hoc programmeringssprog - med henblik på en bestemt type opgaver - kan komme til nytte ved at have et lager af komplicerede ting pakket ind i enkle sætningsregler.

2. Oppetid tæt på 100%

Telefonsystemer, som også skal kunne bruges som nødtelefoner, må have forskellige mekanismer til at klare overbelastning af nettet. Ericsson bruger Erlang til programmering af switche.

Hvorfor ikke blive ved med det sprog man kender?

Der findes (naturligvis) funktioner til C sproget, som kan skabe letvægtsprocesser, kommunikere med dem osv. Erlang kører som fortolket kode og kan faktisk derfor lave "billigere" processkift og multitasking; det er muligt at have flere tusinde småprocesser (actors) på et almindeligt system, der kører Erlang.

Men alligevel tænkte jeg at det kunne være morsomt at prøve at programmere en af de opgaver, som vises i artiklen i dette nummer, i C. De simple eksempler kan løses uden threads - faktisk kan de illustreres med funktionspointere, funktionen "vågner" når den bliver kaldt. Forskellen er selvfølgelig, at en Erlang *actor* kan lave noget selv om den ikke blev "vækket".

I C må man kunne skabe nogle flere threads og gemme et kommunikations-kald til dem i en funktionspointer, og det hele lægges i en linked liste ... Men det er lettere sagt end gjort, og ***hermed udloddes en flaske god rødvin*** hvis du, kære læser, kan skrive et C-program, der implementerer en actor - proces, der kan svare på et message, men hvis der ikke kommer et message, skriver den "Hallo" på skærmen efter et par sekunders venten.

Iøvrigt i dette nummer:

OSD-2016 var en succes - og vi bringer billeder. Et tilbageblik i DKUUGs annaler viser at Eric Raymonds argumenter for Open Source blev omtalt for 17-år siden, og det fortjener et par ord.

Det lovede indslag om scripting (programmering) af mail udskydes til næste nummer, i stedet handler Kommandoklummen om hvorfor kommandolinien giver større frihed til at vælge.

■

DKUUG

DKuug-NYT er medlemsblad for DKuug, foreningen for Åbne Systemer og Internet
Nr. 181 - Juli-August 2016

Udgiver:
DKUUG
Fruebjergvej 3
2100 København Ø
Tlf. 39 17 99 44
email: blad@dkuug.dk

Redaktion:
Donald Axel (ansvarshavende)

Forsidecredits:
Redaktionen

Design og layout:
DKUUG/Donald Axel med LibreOffice

Annoncer:
pr@dkuug.dk

Tryk:
Lasertryk i Aarhus

Oplag:
300 eksemplarer

Artikler og inlæg i DKUUG-Nyt er ikke nødvendigvis i overensstemmelse med redaktionens eller DKUUGs bestyrelses synspunkter.

Eftertryk i uddrag med kildeangivelse er tilladt.

Deadline for nr. 182: Fredag d. 2. september 2016.

Medlem af Dansk Fagpresse
DKUUG-Nyt
ISSN-1395-1440

Vores møder og **foredrag** holdes - med mindre andet udtrykkeligt angives - på vores adresse:

DKUUG
SYMBION
Fruebjergvej 3
2100 København Ø

Hvis man kommer lidt før, er der tid til en snak på kontoret. DKUUG bor i en virksomhedsfarm, Symbion, hvor der er åbne døre indtil kl.18 eller 19 (afhængig af mødetidspunkt). Efter den tid har vi på foredragsaftener en vagt ved døren.

INDHOLD:

OSD 2016 *af Donald Axel* 4

Open Source - Den magiske kedel koger endnu *af Donald Axel* 6

Introduktion til Erlang *af David Askirk* 8

Kommando(linie)centralen 19

Kalender:

Ekstraordinær generalforsamling tirsdag 6. september kl.18, sted:

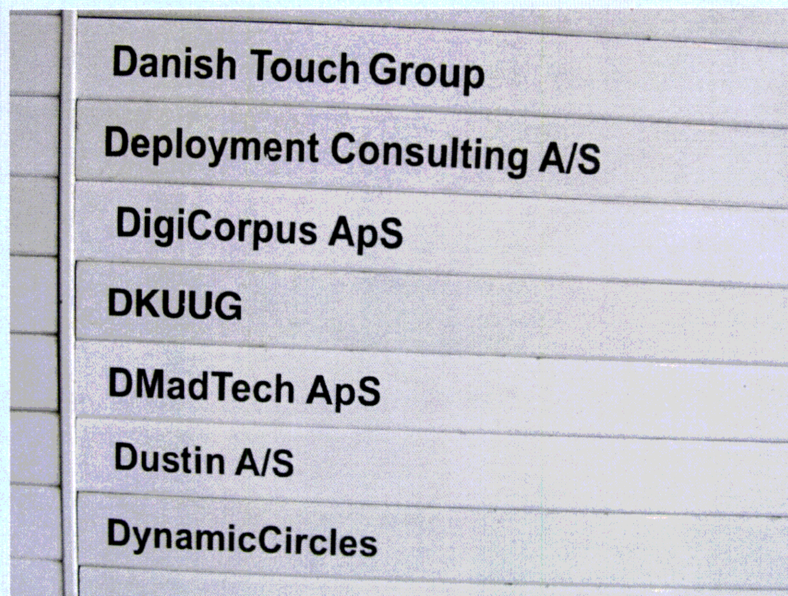
Foreningens kontor,

Symbion, Fruebjergvej 3

DK-2100 København Ø

Onsdage : møde på kontoret fra kl.18 ca. - somme tider 17 eller før.

VI modtager gerne indlæg til bladet.



DKUUG udgiver foreningsbladet DKUUG NYT 3 eller 4 gange om året - og oftere, hvis der er stof nok. Bladet fremstilles indtil videre med LibreOffice. Vi kan godt hjælpe med brug af Libre Office og dele templates mv.

"If the automobile had followed the same development cycle as the computer,a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside."

(Robert X. Cringely)

Indtryk fra OSD 2016

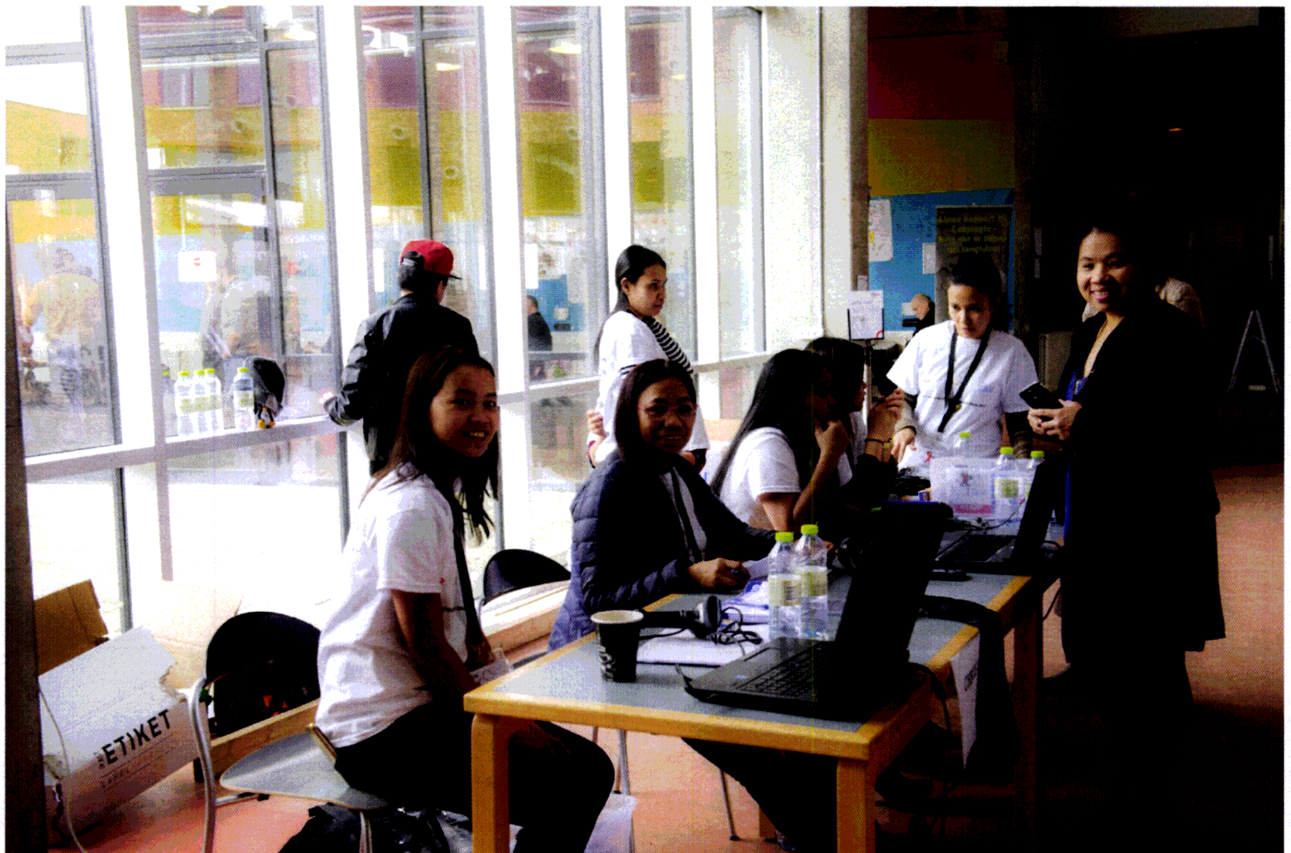
OSD 2016 var godt besøgt og det fungerede godt med husly i Metropol Professionshøjskolen i Sigurdsgade - med både nye og gamle lokaler og et stort auditorium

For enden af Sigurdsgade bygges ny station til den Københavnske Metro - og derfra graver tunnelboremaskinerne sig igennem undergrunden. I fremtiden vil Sigurdsgade være et meget centralt sted i København.

En professionshøjskole svarer nogenlunde til det, som man i USA kalder et college - det er lidt mindre krævende end et universitet, mere profession, mindre forskning. Prøv at tænke på hvordan det passer sammen med et fag som computer-science.

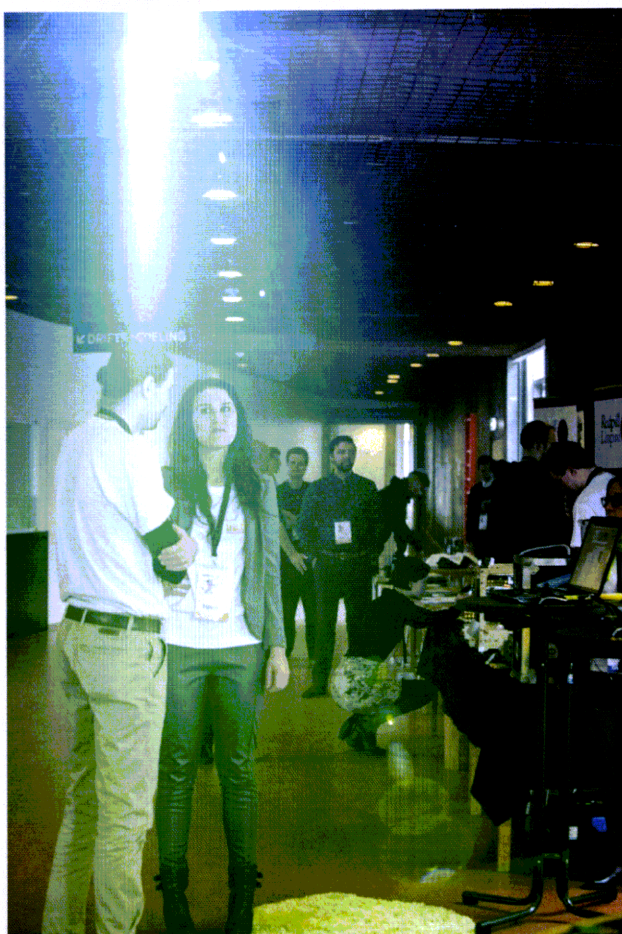
I den anden ende af Sigurdsgade skimtes Zoologisk Museum i Universitetsparken.

De rare damer hilser og kontrollerer at man har adgangskort:



Tak til OSD-fotografen og til Opensourcedays.org for brug af billederne.

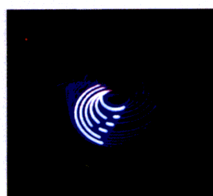
Der var en række udstillere i hallen.



Freja Wedensborg er mest kendt som ankerperson bag en række crypto-parties, som dækker et nyt behov, der er opstået for især journalister: Hvordan mailer man fortroligt med sin kilde.

Ole Tange var også på banen med et foredrag om kryptografi, simpel kryptering af mail og andre lettilgængelige teknikker, som dog kun er et første skridt i retning af sikker kommunikation.

Ole Tange kom også ind på, hvorfor det er nødvendigt. Det er jo ikke for sjovs skyld, at de kendte wikileaks afslørede ormehuller i moralen blandt politikere og andre topfolk. Men en vigtig detalje er, at hvis ingen bruger kryptering eller kun gør det yderst sjældent, så vil ethvert krypteret brev vække opmærksomhed!



Labitat - maker-community - var på pletten igen med sjove og lærereige demonstrationer af hvad man selv kan gøre.

DKUUG støtter Labitat, der har åbent hus hver tirsdag og iøvrigt også gerne tager imod gæster, når man ringer på eller ringer i forvejen.

Labitat IRC er altid åbent for et spørgsmål om teknik.

Hallen er lys og venlig - tiltrækkende sted med få skridt til cafeteria og kaffebar.

Vort demokrati mangler noget teknisk hjælp

Bliv kryptohelt nu

siger skærm-billedet - og som en første vejledning blev forskellige applikationer nævnt, Tutanota, Haiku, og en bog for begyndere var på tilbud:

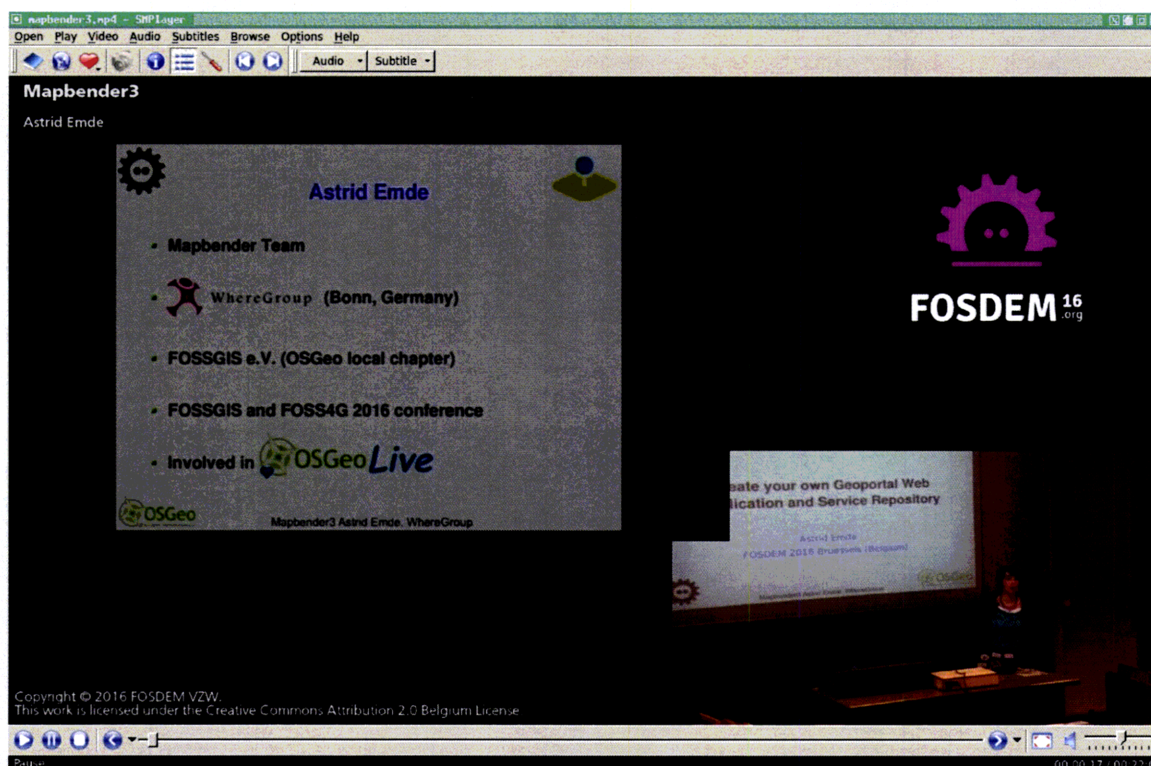
Open source -

Den magiske kedel koger endnu

Af Donald Axel

Den entusiasme, som man oplevede på Linuxforum for 12-15 år siden er stadig i live. Der var en del grå hår på OSD-2016, men der var også unge mennesker og entusiasme. Som en, der har været med længe, er det på sin plads her at glædes over at den store energi til at organisere events her og der:

Udover vores danske ***Open Source Days*** (omtalt på foregående sider) og ***Bornhack.dk***, som DKUUG også støtter, er der ***FOSDEM***, som samler mange mennesker og som er helt åbent - ingen tilmelding, ingen betaling. Det stiller faktisk nogle store krav til logistikken.



Video'er fra FOSDEM-2016 er nu på nettet og kan downloades frit og let og bekvemt, størrelsen (og dermed kvaliteten) er sat til et behageligt niveau, som giver korte downloadtider og trods alt en pæn opløsning.

Emnerne spænder vidt.

Men ingen dvæler ved hvorfor der er tilslutning til Open Source og hvad fordele og økonomien bag det er.

Måske har Eric Raymond ramt noget rigtigt med sin analyse af Open Source fænomenet .

I udgivelsen "The Magic Cauldron" skriver han om de kræfter, der styrer udviklingen og tilslutningen til Open Source.

[...] we now will consider (from entirely within the realm of scarcity economics) the modes of cooperation and exchange that sustain open-source development. While doing so we will answer the pragmatic question ``How do I make money at this?", in detail and with examples. First, though, we will show that much of the tension behind that question derives from prevailing folk models of software-production economics that are false to fact.

(A final note before the exposition: the discussion and advocacy of open-source development in this paper should not be construed as a case that closed-source development is intrinsically wrong, nor as a brief against intellectual-property rights in software, nor as an altruistic appeal to `share'. While these arguments are still beloved of a vocal minority in the open-source development community, experience since [CatB] has made it clear that they are unnecessary. ***An entirely sufficient case for open-source development rests on its engineering and economic outcomes -- better quality, higher reliability, lower costs, and increased choice.****)*

Oversat: ***Et helt tilstrækkeligt argument for Open Source udvikling hviler på resultaterne i ingeniørmæssigt og økonomisk udbytte -- bedre kvalitet, større stabilitet og pålidelighed, lavere omkostninger, og større valgmuligheder.***

Det er store ord, og det var dem, der gav genklang i DKUUG-nyt 1999-4 (se næste side - bladene kan iøvrigt downloades fra vores website.)

Af disse argumenter er det først og fremmest bedre kvalitet og økonomi, som tæller.

Men sådan som jeg ser det, er det, der tænder entusiasmen ikke kun et ønske om stabil software - det være sig et game, en geo-app eller musik-editor. Nej! det er hands-on oplevelsen, dette at kunne gøre tingene selv, prøve store softwaresystemer, sammenligne, programmere tilføjelser.

Og hands-on kræver masser af information, manualer, vejledninger, som skal være billige eller gratis at få fat på. Derfor spiller Open Source en enorm rolle for de fleste studerende.

DKUUG

Vejen til viden om Åbne Systemer og Internet

nyt

3/april 1999

Tjen penge på Open Source

Vi giver opskriften på, hvorledes man tjener på noget, der er gratis

Frihed til desktoppen

WP 8.0 og KDE: Frie desktops

Wassenaar

Hvad er nu det for noget?

Freelance-arbejde i IT-branchen

Godt eller skidt?

Artiklen fra dengang, 1999, konkluderer at man kunne tjene penge på at give den service, det er at få tingene til at køre.

Det er endnu mere sandt i dag, men der er nu ikke kommet mange forretninger, som leverer service for ved at installere, tilpasse og evt. sørge for drift og backup. De steder, der var Open Source højborge, undervisningssteder, generer sig ikke for at installere systemer, som baserer sig på Microsofts produkter, fordi især login for tusinder af studerende og lærere er en svær nød at knække for Open Source systemfolk - det kan lade sig gøre med OpenLdap, men det kræver indsigt på programmør-niveau, og det gør arbejdsgivere utrygge: "Der skal ikke udvikles her! - en udviklingsafdeling er noget dyrt stads." Men på den anden side må enhver IT-specialist konstant uddanne sig for at kunne løse de nye opgaver, man kan ikke købe sig til ny viden.

Unge mennesker, som læser denne artikel, vil komme til at arbejde for firmaer, der ikke eksisterer før om 10-20 år. Hvordan kan du uddanne dig til det?

(Forfatteren Fareed Zakaria, Information 9.Juli).

■

Tjen penge på Open Source

Af Peter Toft
<pto@sslug.dk>,
Hanne Munkholm
<hanne@aub.dk> og
Kenneth Geisshirt
<kneth@sslug.dk>

I denne artikel gives en redegørelse for begrebet Open Source, og der svares bl.a. på det ofte stillede spørgsmål: Hvordan kan man tjene penge på noget, der er gratis?

Formål

Vi vil i denne artikel argumentere for at Open Source Software er bedre end det mere traditionelle lukkede software. Det vil blive beskrevet, hvorfor Linux-verdenens måde at udvikle software på har store fordele - dels med hensyn til stabilitet og sikkerhed, men også med hensyn til den strategiske risiko, som en moderne erhvervsleder altid må tage, når der købes software.

Artiklen er kraftigt inspireret af Open Source bevægelsen og Eric S. Raymonds artikler og foredrag. Eric S. Raymond er en (nu meget kendt) systemprogrammør fra USA, der har studeret og dokumenteret mange af de stærke ting, som kendetegner udviklingen af det meget udbredte Linux styresystem.

Indledning til Open Source

I de første måneder af 1998 opstod et nyt begreb blandt Internet-brugere: Open Source. Et af mønstrene er, at udviklingsprojekterne ofte bruger Internettet som det vigtigste medium, dvs. at e-post og WWW er vigtige elementer hvor udviklerne kommunikerer. Et andet fælles mønster er den udviklingsmodel, som man benytter i projekterne (ofte uden bevidst at tænke over den).

Flere af de store firmaer anerkender nu Open Source og anvender det som udviklingsmodel: SUN, Silicon Graphics og Netscape.

Der findes mange tusinde mere eller mindre kendte Open Source projekter. De allermest kendte er

- Linux - Med en vækst på 212% i 1998 og ca. 20 millioner brugere er Linux det styresystem, der virker mest lovende af alle. Linux er i dag det mest anvendte styresystem til web, news og ftp-servere med en markeds andel på 25%-33%.
- Apache - Webserveren Apache kan downloades gratis fra Internettet og er af en meget høj kvalitet. Netop den høje kvalitet har betydet, at Apache har en markedsandel på omkring 50%.
- Samba - Samba er blandt mange andre ting et stykke software, som kan fungere som filserver for et Microsoft Windows 95 netværk. Programmet betyder at en edb-afdeling kan benytte UNIX-baserede servere, imens slutbrugerne bruger Windows og de mange applikationer, som findes til Windows 95. Mange firmaer anvender i dag Samba

Introduktion til Erlang

Af David Askirk

instruktør ved SuperUsers - Karlebogård, Hillerød

Erlang er et sprog der har arvet en del ting, blandt andet fra Prolog. Erlang hører til i den funktionelle verden.

For at komme i gang med Erlang, kan man, efter man har installeret Erlang, starte Erlang shellen ved at skrive `erl`.

Så kommer følgende frem på skærmen:

```
$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Eshell V7.3  (abort with ^G)
1>
```

Den har en prompt `1>` der melder at Erlang er klar til at modtage kommandoer fra os.

Inden vi går igang med at lave filer, skal vi lige se på noget generel syntax.

Hvis vi gerne vil skrive en kommentar kan vi bruge `%`. Erlang shellen indeholder forskellige matematiske operationer, som eks. plus `+`. Linjer afsluttets for det meste med punktum `.`.

```
$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Eshell V7.3  (abort with ^G)
1> % Jeg er en kommentar
1> 2+3.
5
2>
```

Erlang er lidt specielt når det kommer til variabler og typer. Variabler er altid med stort bogstav som `A` eller `AntalBrikkerIPuslespil`.

Når der er tildelt en værdi til en variabel kan den ikke ændres, dette gælder selvom man blot ændrer værdien til noget af samme type. Dette er noget anderledes fra andre programmeringssprog, men meget brugt i matematikken.

Dette er måske noget man kan undre sig over, men ved at gøre at variabler ikke kan ændres kan man lave kode uden sideeffekter, og hvis man gerne vil lave kode med meget parallelitet, så er sideeffekter noget man gerne vil af med.

Den anden ting, som Erlang har, som nogle andre sprog også har, men ikke mange, er atoms. Værdien af et atom er atom'et selv.

Eksempel:

```
6> hello.
hello
7>
```

Her bliver atom'et `hello` oprettet. Et atom er meget billigt rent hukommelsesmæssigt, og er et stærkt værktøj når vi kommer videre og skal lave kommunikation.

Hvis vi gerne vil pakke værdier ind, så kan der bruges en tuple. En tuple er en samling af værdier. Erlang har en garbage collector, som rydder op i hukommelsen, også tuples bliver ryddet op.

Men lad os lige se det i kode:

```
1> Fornavn = {navn, david}.
{navn,david}
2> Efternavn = {efternavn, fotel}.
{efternavn,fotel}
3> ForOgEfter = {forOgEfternavn, Fornavn, Efternavn}.
{forOgEfternavn,{navn,david},{efternavn,fotel}}
```

I linje 1 og 2 bliver der lavet to tuples, den ene med fornavn og den anden med efternavn. Husk at en variabel altid starter med stort bogstav, hvorimod et atom starter med et lille bogstav. I linje 3 bliver de to tuples kombineret og resultatet kan ses.

For at hente værdier ud af en tuple laver bruger man en Erlang variant af pattern matching. Lad os se på et eksempel:

```
1> Point = {point, 3, 14}.
{point,3,14}
```

Her erklærer vi et punkt. Nu vil vi så gerne have X og Y ud af punktet:

```
2> {point, X, Y} = Point.
{point,3,14}
3> X.
3
4> Y.
14
5>
```

Her bliver der matchet på at det er en tuple der starter med point, herefter kommer der et tal, og herefter et andet. Vi kan se værdierne af X og Y i linje 3 og 4.

Vi kommer mere ind på pattern matching, når vi når til kommunikation.

Den sidste datatype vi skal kigge på er listen.

En liste er en række af elementer, som ikke behøver at have samme type. Lad os se på nogle lister:

```
1> IndkoebsListe = [{maelk, 3}, {rugbroed, 1}, {leverpostej, 1}].
[{maelk,3},{rugbroed,1},{leverpostej,1}]
2> BlandetListe = [1, 2+2, {person, niller}].
[1,4,{person,niller}]
```

I linje 1 bliver der lavet en indkøbsliste og i linje 2 er der en liste med blandende elementer. Lister kan man sige er bygget op af dens hoved (head) samt dens hale (tail). Så hovdet er det første element og halen er resten af listen.

Dette kan vi bruge til at skille listen ad. For at finde ud af hvad vi skal købe ville man først finde det første element i indkøbslisten, herefter det næste, og sådan ville man forsætte indtil man er færdig. Se følgende kode:

```
3> [FoersteTing|Resten1] = IndkoebsListe.
[{maelk,3},{rugbroed,1},{leverpostej,1}]
4> FoersteTing.
{maelk,3}
5> Resten1.
[{rugbroed,1},{leverpostej,1}]
```

Vi mangler at behandle strenge. Men strenge er blot lister af heltal. For at kende forskel på dem, bruger vi " til at indkapsle strenge.

```
6> [1,2,3].
[1,2,3]
7> [65,66,67].
"ABC"
8> "Hello world".
"Hello world"
```

I linje 6 bliver der lavet en almindelig liste, i linje 7 bliver der også lavet en liste, men læg mærke til at den skriver "ABC" ud på skærmen. I linje 8 bliver der skrevet "Hello world" netop som en streng. Men bagved er det en liste af heltal.

Med alt dette på plads kan der laves nogle moduler. Denne del vil fokusere på sekvensel programmering. Senere vil vi nå til asynkron programmering.

Et modul er et erlang program. Lad os lave et modul, hvor vi har nogle regne funktioner til at udregne størrelsen af nogle geometriske figurer.

Et module starter altid med en erklæring om hvilket module det er. Dernæst kommer der en erklæring om hvilke funktioner som bliver eksporteret, altså, hvilke af modulets funktioner, der skal være til rådighed for dem som gerne vil bruge modulet.

Den følgende kode ligger i en fil der hedder geo.erl:

```
-module(geo).
-export([area/1]).
area({rect, Width, Ht})      -> Width * Ht;
area({circle, R})            -> 3.14159 * R * R;
area({triangle, Bottom, Ht}) -> Bottom * 0.5 * Ht.
```

Den første linje fortæller at geo er modulet. Den anden linje fortæller hvilke funktioner der bliver eksporteret og stillet til rådighed for andre. Notationen area/1 fortæller det er funktionen area, som tager et input. Dette er /1 der angiver dette. Dette kaldes *aritet* af en funktion.

area funktionen benytter sig af tuples, hvor der bliver matchet, og det er dette der gør at man kan kalde funktionen det samme. For at fortælle Erlang at det er den samme funktion, bliver hver linje afsluttet med ';' undtagen den sidste som bliver afsluttet med '.'. I filen bliver de forskellige definitioner af area funktionen også kaldt clauses. Generelt skal der ';' mellem clauses.

Når modulet skal bruges skal det først oversættes. Se følgende session:

```
1> c(geo).
{ok,geo}
2> geo:area({circle, 1}).
3.14159
3> geo:area({rect, 2, 3}).
6
```

I linje 1 bliver modulet compilet og loadet ind i erlang miljøet. Hvis man kigger i sin mappe kan man se en geo.beam fil, som er den oversatte erlang fil.

I linje 2 og 3 bliver der lavet nogle kald til funktionen. Læg mærke til at funktions-kaldene præfikses med **geo:**.

Modul med funktioner med forskellig aritet

Lad os tage et eksempel med en funktion der har forskellige aritet, altså forskellige antal elementer. Samtidig viser den også det at gå igennem en liste for at behandle hvert element.

```
-module(mysum).
-export([sum/1]).

sum(L) -> sum(L, 0).

sum([], X)    -> X;
sum([H|T], X) -> sum(T, (X+H)).
```

I dette modul bliver der kun exporteret en funktion, men der er faktisk to funktioner. Den funktion der bliver eksporteret er den med sum som kun tager et argument.

Nemlig en liste med tal som der skal summeres over.

Den anden funktion benytter sig af at der er to tilfælde når man går igennem en liste. enten så er listen tom, eller listen har to elementer, det første element og resten af listen. Resten af listen kan godt være tom. Hvis vi når til den tomme liste, så er vi færdige og kan returnere den sum vi er kommet til. Ellers tager vi det første element og ligger det til vores accumulator variabel, samt gå videre med resten af listen.

Her kommer lige et eksempel der viser, hvordan den kører:

```
Input: [1,2,3,4,5]

sum([1,2,3,4,5]) kalder
sum([1,2,3,4,5], 0) Den midlertidige sum er 0 og vi kalder følgende:
sum([2,3,4,5], 0+1) Den kalder videre til:
sum([3,4,5], 1+2) som kalder videre til:
sum([4,5], 3+3) som kalder videre til:
sum([5], 4+6) som kalder videre til:
sum([], 5+10) som kalder videre til:
sum([], 15) Her er listen tom, så 15 bliver returneret som resultatet
```

Men lad os se på brug i erlang:

```
1> c(mysum).
{ok,mysum}
2> sum([1,2,3,4,5]).
** exception error: undefined shell command sum/1
3> mysum:sum([1,2,3,4,5]).
15
4>
```

Her sker der flere ting. I linje 1 bliver modulet oversat og indlæst. I linje 2 bliver funktionen kaldt, men uden modul henvisning, derfor fejler erlang. I linje 3 bliver der lavet det korrekte kald og resultatet 15 bliver returneret.

Guards er vagter der kan sige under hvilke forudsætninger en bestemt funktion skal køre.

Et eksempel på en guard er dette:

```
max(X, Y) when X > Y -> X;
max(X, Y)            -> Y.
```

Her er selve guard-nøgleordet 'when'. Her går vi ind og siger at funktion max(X, Y) som returnere X hvis X er større end Y. Hvis den guard ikke er bliver sand, bliver funktionen ikke kørt. I det andet tilfælde, sluger den alle andre tilfælde, så derfor bliver Y returneret.

Actor-modellen

Actor modellen er en model der har fokus på parallel programmering. Dette opnår den ved ikke at dele noget hukommelse mellem de forskellige tråde, men derimod kommunikere mellem processer. Denne kommunikation foregår asynkront.

For at definere så lad os lige se på hvad parallel og hvad concurrent betyder.

Jeg spiser en is og kan derfor ikke sende en sms om den dejlige is jeg spiser til min ven.

Hvis jeg udførte tingene parallelt, ville jeg kunne spise is med den ene hånd og sende sms med den anden.

Hvis jeg udførte dem concurrent så ville jeg kunne spise en skefuld is, derefter lægge skeen ned, skrive et ord på telefonen, lægge telefonen og forsætte is og så fremdeles indtil sms er sendt, og isen er spist.

Det næste, som er meget interresant i Erlang sammenhæng er det asynkrone.

Asynkront vil være at sende en sms, lægge telefonen i lommen og spise is, indtil man modtager en besked som man kan mærke med en vibration i lommen.

I Actor modellen er hver actor en selvstændig enhed med egen hukommelse, og en postkasse som andre processer kan putte beskeder i. Dette gør man nemmere kommer udenom typiske multi-thread udfordringer såsom deadlock og at man oplever at tråde spænder ben for hinanden.

Actor-modellen i Erlang

Erlang bruger actor modellen til at lave high performance og high concurrent systemer.

Verden er concurrent! Hvis man kigger på mennesker kan man se at enhver person kan reagere for sig selv, samt har sin egen hukommelse.

Den måde som mennesker sender beskeder til hinanden er via tale. Vi kan så spørge om den anden person har modtaget beskeden og de kan så bekræfte eller afkræfte det.

Erlang fungerere på denne måde. Nogle ville endda sige at denne Actor-model er en klart bedre måde at lave store systemer der snakker sammen, end at bruge fx. object orienteret programmering. Tankegangen med at have mange små actorere der hver i sær står for et ansvarsområde og kommunikere ser man også brugt andre steder, fx. i micro-services, som har lånt lidt fra actor modellen.

Der er tre ting man skal vide når man skal lave concurrent programmering i Erlang.

```
Pid = spawn(Fun)
```

spawn laver en ny process og tildeler et nyt Pid (Processid). Det er gennem dette Pid vi skal kommunikere med den nye actor.

For at sende en besked til Pid skal følgende bruges:

```
Pid ! Message
```

Her bliver Message sendt til processen med id'en Pid.

Den sidste del er at modtage en besked.

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
    ...
end
```

Den sidste del er modtagelsesdelen hvor vi kan modtage forskellige beskeder med forskellige mønstre. Hvert mønster kan evt. have en vagt på, og der kan udføres et eller flere expressions.

Men lad os lave en beregningsserver til vores beregningsfunktioner:

```
-module(geoserver).
-export([loop/0]).

loop() ->
    receive
        {rect, Width, Ht} ->
            io:format("Rektangle areal er: ~p~n", [Width*Ht]),
            loop();
        {circle, R} ->
            io:format("Arealet af en cirkel er: ~p~n", [3.14159 * R * R]),
            loop();
        {triangle, Bottom, Ht} ->
            io:format("Arealet af en trekant: ~p~n", [Bottom * 0.5 * Ht]),
            loop();
        Other ->
            io:format("Arealet af ~p kan ikke beregnes~n", [Other]),
            loop()
    end.
```

Her er der først de to kendte, nemlig module og export. Derefter bliver der erklæret en loop funktion. Denne funktion bliver kaldt igen og igen, og dette giver samme effekt som en uendelig løkke.

I loop funktionen er der et receive statement, som kan håndtere fire forskellige slags beskeder. Rektangle, cirkel, trekant og resten. Hvert mønster kalder loop funktionen igen. Den sidste Other er ikke nødvendig, men ved at have den med kan vi håndtere nogle fejl tilfælde.

Her kommer en skærm session, som viser, hvordan det kan køres:

```
1> c(geoserver).
{ok,geoserver}
2> Pid = spawn(fun geoserver:loop/0).
<0.41.0>
3> Pid ! {rect, 10, 20}.
Rektangle areal er: 200
{rect,10,20}
4> Pid ! {circle, 3}.
Arealet af en cirkel er: 28.274309999999996
{circle,3}
5> Pid ! {triangle, 10, 5}.
Arealet af en trekant: 25.0
{triangle,10,5}
6> Pid ! {banan, 34}.
Arealet af {banan,34} kan ikke beregnes
{banan,34}
```

Lad os gennemgå det linje for linje. Linje 1 oversætter og indlæser modulet. Linje 2 opretter en ny process og tildeler processid'et og gemmer i variablen Pid. Derefter bliver de fire muligheder afprøvet ved at sende beskeder til processen. Læg mærke til den sidste. Det er ***Other*** muligheden der kommer i spil der.

Nu blev det sagt at ved at bruge actor programmering kan man lave noget der minder om den virkelige verden. Lad os se på et eksempel af det:

```
-module(person).
-export([person/2]).

person(Name, Age) ->
    receive
        {hello, OtherName} ->
            io:format("Goddag ~p. Mit navn er ~p~n", [OtherName, Name]),
            person(Name, (Age + 1));
        {tellAboutYou} ->
            io:format("Goddag, mit navn er ~p og jeg er ~p år gammel.~n",[Name, Age]),
            person(Name, (Age + 1))
        after 1000 ->
            io:format("I dag skete der ingenting~n"),
            person(Name, (Age + 1))
    end.
```

Dette er også en løkke, men læg mærke til at hver gang løkken bliver kaldt, bliver alder forøget med en. Name og Age er personens tilstand. Den bliver vedligeholdt løbende. I dette eksempel bliver der brugt en timeout. Det gøres med after. Her er der sat op at hvis der ikke er kommet en besked inden for 1000 ms, så bliver koden udført, hvor den skriver at der ikke skete noget idag, samt derefter kalder personen igen.Lad os se hvordan det bruges:

```
1> c(person).
{ok,person}
2> Pid = spawn(fun() -> person:person("Niels", 0) end).
<0.41.0>
I dag skete der ingenting
I dag skete der ingenting
```

```
3> Pid ! {hello, "Else"}.
Goddag "Else". Mit navn er "Niels"
{hello,"Else"}
I dag skete der ingenting
I dag skete der ingenting
I dag skete der ingenting
```

I linje 1 bliver modulet oversat og indlæst. I linje 2 bliver der oprettet en ny process, med personen. Den starter med navnet Niels og alderen 0 år.

Allerede der er den startet, og læg mærke til at der når at komme to dage, hvor der ikke sker noget førend der bliver sendt en besked. Den bliver håndteret og der bliver sagt goddag til Else, så kommer der tre gange at der ikke skete noget.

Nu kommer der et lidt størrere eksempel der viser hvad man også kan i Erlang.

Beregning af primtal i Erlang

En sjov måde at beregne primtal på er ved at splitte op i divisons processor. En division process kan tage et tal og se om det tal som division processen har, går op i det tal der er modtaget. Hvis det er tilfældet bliver tallet stoppet, ellers bliver det sendt videre i kæden.

Ved at lave en kæde af disse og binde dem sammen kan man sende et tal gennem kæden og kæden filterer ikke primtal fra.

Kæden kan bygges dynamisk og udvides til man ikke kan klare flere processer.

Når et led modtager et tal, som den ikke kan se om er primtal eller ej, sender den tallet videre. Hvis der ikke er et led at sende videre til opretter den et nyt led og binder sig til det som det næste led.

Første del er af definere en primtal divisions process.

En divisions process skal køre et loop, hvor den modtager et tal, undersøger om det er et tal der skal kasseres, - ellers skal det sendes videre. Der skal kun gøres noget når det er et tal der ikke skal kasseres, så lad os nøjes med at håndtere, at der er et tal, der skal sendes videre. Til dette bruges modulus operatoren "`/=`". Se kode eksemplet, module codelisting1.

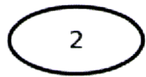
```
-module(codelisting1).
-compile(export_all).

divProc(Myprime) ->
    io:format("~w is a prime~n",[Myprime]),
    receive
        {No} when No rem Myprime /= 0 ->
            Pid = spawn(fun() -> divProc(No) end),
            divProc(Myprime, Pid);
        stop ->
            true
    end.

divProc(Myprime, NextLink) ->
    receive
        {No} when No rem Myprime /= 0 ->
            NextLink ! {No},
            divProc(Myprime, NextLink);
        stop ->
            true
    end.
```

Ingen noder oprettet

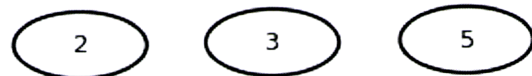
2 bliver sendt



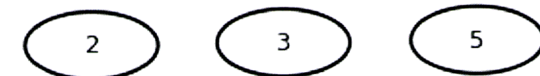
3 bliver sendt

4 bliver sendt

5 bliver sendt



6 bliver sendt



I denne er der først modul erklæringerne, dernæst erklæring om at vi gerne vil eksportere alle funktioner i vores fil.

Den første divProc er den der bruges når den er det nyeste led i kæden. Til at starte med vil kæden ikke bestå af nogle led, men vi kan så starte den op. Se kørselseksempel senere.

Det første den gør er at printe at et tal der er et primtal. Dette er fordi når et tal kommer i gennem hele kæden må det være et primtal, da der så ikke er flere led der kan sortere tallet fra. Derfor må et tal der kræver at der bliver lavet et nyt led, være et primtal.

Dernæst står programmet i et modtagelsesloop og venter på der bliver sendt et tal til aktøren. Hvis det er et tal som som ikke er deleligt med det primtal som processen har i variablen Myprime, så skal det sendes videre. Hvis processen er det nyeste led, så skal der oprettes en ny. Dette gøres ved spawn kommandoen. Dernæst kaldes `divProc/2` med det primtal som processen hele tiden har, samt Pid (Process-ID) på det næste led i kæden. Læg mærke til at der kun bliver oprettet en ny process, på den gamle bliver logikken kun ændret en smule. Nemlig ændret til at den nu sender et tal videre, istedet for at oprette ny process hvis den støder på et tal som den ikke har set før.

Hvis dette gemmes i en fil ved navn codelisting1.erl, så kan det afprøves på følgende måde. Der er tilføjet kommentarer til højre for session-eksemplet i rammen:

Compile Erlang koden og gør den klar til brug i erlang sessionen, det er **`c(codelisting1).`**

1. Opret det første led i kæden med primtallet 2, som er det første primtal. Vi får tilbage på skærmen at 2 er et primtal.
2. Send tallet 3 til det første led i kæden. Bagved bliver der nu oprettet en ny process som har tallet 3.
3. Send tallet 4 til det første led i kæden. Da det første leds tal er 2, og 2 går op i 4, bliver der ikke gjort mere, og tallet kommer derved ikke videre end til det første led.
4. Send tallet 5 til det første led. Dette tal vil så vandre ned i gennem kæden og blive tilføjet kæden, som nu består af 2,3 og 5.
5. Send tallet 6. 2 går op i 6, og derfor bliver tallet stoppet ved første led.
6. Send tallet syv. 7 vil igen vandre ned i kæden og blive koblet på som et nyt led bagerst i kæden
7. Send tallet 8. Igen, 2 går op i 8, og derved bliver 8 stoppet allerede ved første led.
8. Send tallet 49 til det første led i kæden. 49 vandrer hele vejen ned i kæden til det sidste led, hvor 49 så bliver stoppet da 7 går op 49.

```
$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4]
[async-threads:10] [hipe] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
1> c(codelisting1).
{ok,codelisting1}
2> Pid = spawn(fun() -> codelisting1:divProc(2) end).
2 is a prime
<0.41.0>
3> Pid ! {3}.
3 is a prime
{3}
4> Pid ! {4}.
{4}
5> Pid ! {5}.
5 is a prime
{5}
6> Pid ! {6}.
{6}
7> Pid ! {7}.
7 is a prime
{7}
8> Pid ! {8}.
{8}
9> Pid ! {49}.
{49}
10>
```

Bruge Erlang til at skrive en server

Erlangs styrke ligger i at kunne skrive services. Her kommer et eksempel på at lave en server i Erlang som viser at Erlang også kan bruges til at lave TCP-services.

Her bliver der lavet en klassisk echo server, der blot sender det som kommer ind tilbage igen.

Lad os se på koden:

```
-module(simple_echo).
-export([listen/1]).

listen(Port) ->
    {ok, ListenSocket} = gen_tcp:listen(Port, [binary, {packet, 0}, {active, false},
{reuseaddr, true}]),
    accept(ListenSocket).

accept(ListenSocket) ->
    {ok, Socket} = gen_tcp:accept(ListenSocket),
    spawn(fun() -> loop(Socket) end),
    accept(ListenSocket).

loop(Socket) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Data} ->
            io:format("Got ~p~n",[Data]),
            gen_tcp:send(Socket, Data),
            loop(Socket);
        {error, _} ->
            ok
    end.
```

Her bliver der lavet de indledende øvelser med modul og export.

Her efter kommer der tre funktioner. Listen sætter server socketen op til at modtage forbindelser. Der bliver sat op på de port man gerne vil have, samt med forskellige tcp options. Dernæst kaldes accept funktionen med den server socket som er blevet lavet i første funktion.

I accept funktionen ventes der på en forbindelse, når forbindelsen kommer, så bliver der lavet en ny process med den modtagende socket. Så kaldes accept funktionen igen, så den er klar til at håndtere endnu en forbindelse, mens den netop indkommende socket bliver behandlet.

Den sidste funktion er her hvor alt forretningslogikken ligger. I logikken venter vi på hvad vi nu end motager på socket'en. Det er en tuple der bliver modtaget, hvis det gik ok, starter den med ok, ellers starter den med error. Med ok, kommer der data ind i Data variablen, data skrives ud på skærmen og returneres til brugeren. Hvis der sker en fejl, så bliver det den anden del af case statement ramt, som blot returnere ok, så serveren ikke går ned, men processen bliver bare afsluttet.

Lad os se på brug:

```
1> c(simple_echo).
{ok,simple_echo}
2> simple_echo:listen(1337).
Got <<"test\r\n">>
Got <<"hest\r\n">>
```

I linje 1 bliver modulet indlæst og oversat. I linje 2 bliver serveren startet på port 1337. Dernæst kan det ses at der modtaget beskeden test og derefter hest.

Men lad os lige se den del som snakker med serveren.

```
$ telnet localhost 1337
Trying 127.0.0.1...
Connected to localhost.
```

```
Escape character is '^]'.
test
test
hest
hest
```

Her bliver der oprettet forbindelse til localhost på port 1337. Dernæst bliver der sendt test, serveren svarer tilbage med test. Så bliver der sendt hest, og der bliver svaret hest.

Fejlhåndtering og robuste Erlang programmer

Da Erlang var lavet til telefoni systemer, blev det bygget til at være yderst robust, så man kunne skrive programmer der havde svært ved at gå ned. Det er en meget ønsket feature i telefoni systemer.

Til at opnå denne robusthed kan processer blive linket sammen. Når den ene process fejler, så bliver der sendt et exit signal til den linkede process, og denne kan så vælge at håndtere det, så man kan få grebet fejlen, selvom det er seperate processer.

Lad os se på noget kode. Det første der skal laves er en exit handler, som fanger hvis der er en process der sender et exit signal.

```
-module(trap_exit).
-export([go/0]).

on_exit(Pid) ->
    spawn(fun() ->
            process_flag(trap_exit, true),
            link(Pid),
            receive
                {'EXIT', Pid, Why} ->
                    io:format("Process ~p died with error: ~p~n", [Pid, Why])
            end
        end).

the_failer() ->
    receive
        X -> list_to_atom(X)
    end.

go() ->
    FailPid = spawn(fun() -> the_failer() end),
    on_exit(FailPid),
    FailPid ! detteerentest.
```

Her bliver erklæret tre funktioner. on_exit er exit handleren som bliver brugt til at håndtere hvis en process fejler. Funktionen opretter en process som bliver linket sammen med den ønskede process. process_flag(trap_exit, true) fortæller erlang at dette er en system process. Derefter bruges link til at binde den oprettede process sammen med den ønskede, som kommer i form af et PID udefra. Til sidst lyttes der efter en tuple med EXIT, et pid og en grund til at processen stoppede. Den skriver dette ud på skærmen med io:format.

Den næste funktion er en funktioner som venter på at modtage et eller andet, og vil konvertere en liste til et atom. Den kommer til at fejle, men det vil vi se senere.

Den sidste funktion er den som også bliver exporteret, står for at binde det hele sammen. Først bliver der oprettet en process som vil fejle, derefter bliver der bundet en exit_handler på og til sidst bliver der sendt et atom til den funktion der vil fejle.

Lad os se på hvad der sker når det bliver brugt:

```
Eshell V7.3  (abort with ^G)
1> c(trap_exit).
{ok,trap_exit}
2> trap_exit:go().
Process <0.41.0> died with error: noproc
```

I linje et bliver modulet oversat og indlæst. I linje to bliver ***go*** funktionen kaldt. Det kan ses at den skriver den fejlmeddelse ud som man kan se i on_exit handleren.

Dette med at linke processer sammen kan bruges til at lave mere stabile systemer, som kan genoprette sig selv, hvis der sker nogle fejl.

Link og andet

- https://www.youtube.com/watch?v=uKfKtXYLG78 - En sjov video med Joe Amstrong der viser noget af det som Erlang kan i forbindelse med telefoni.
- http://learnyousomeerlang.com/ - Udemærket bog til at lære Erlang efter.
- https://pragprog.com/book/jaerlang2/programming-erlang - Den bog som er skrevet af Joe Amstrong og virkelig god til at komme rundt om emnerne i Erlang.

Kode-eksemplerne kan downloades fra dkuug.dk. ■

Kommando(linie)centralen

Nødvendigheden af kommandolinie-kendskab

Et moderne mobilsystem (Android, iOS, MS-Phone mv) har touch screen funktionalitet, som man hurtigt kan vænne sig til, - det er intuitivt.

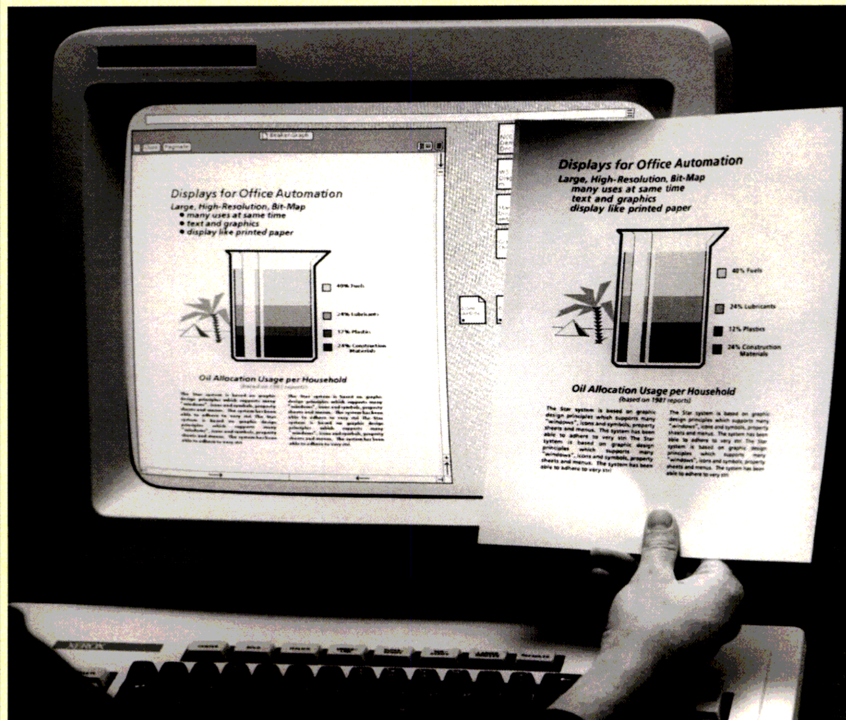
Det er blevet redaktionen bekendt, at nogle læsere bliver forskrækkede "kommando-vindue" -- man foretrækker et vindue med tick-bokse og "rullegardiner" - folde ud tabs og what not.

Vinduer plejer man at se, det er det normale. Men bag et ikon er der en kommando, navne, ord, bogstaver. Og kommandolinien giver større frihed i administration af servere, især, når der er mange af dem.

Mus, grafik og vinduer

Intuitive interfaces blev første gang forsøgt, da Rank Xerox erkendte at kopi-maskinens højdepunkt var nået. Papir med billede og tekst, som kunne kopieres, klippes og klistres med en kopier fra en kopimaskine havde gået sin sejrsgang gennem 1960'erne. En computer ville kunne gøre det samme - det vidste man allerede i 1970 (og før), men man havde ikke isenkrammet.

Rank Xerox' lavede i 70'erne en maskine, der lignede en PC'er, men som havde et grafisk interface og en pointer device - en mus. Prisen var titusinder af dollar (halv mio i nutidskroner).



Men Rank Xerox' maskiner kunne jo også meget mere end en PC. Steve Jobs fik lov at se produktet og kunne se ideen - og brugte den.

GUI forholder sig til kommandolinie som mobil forholder sig til en server.

Som det kan ses af billedet var ideen at computeren var en skrivebords-anordning. Det var overhovedet ikke på tale at bruge computeren på Internettet (som ikke var offentligt tilgængeligt i 1980) eller som server i et lokalt netværk.

Konceptet var at en computer var et tegne- og skrivebord, det var et værktøj til at skrive og tegne, en grafisk enhed: ***Skriv - tegn - print.***

De første skridt i retning af netværk var at bruge telefonlinier til at sende digitaliserede billeder - fax - men det var indlysende at det ville være en fordel med lokalt netværk til i det mindste backup og fælles arkiver. Dermed kom system-administration ind i alle virksomheder, fra såvel 1-mands som multinationale.

Software leverandører som Apple og Microsoft erkendte hurtigt at Novell Netware kunne udkonkurreres med mere primitivt netværk, blot det var nemt at opsætte og administrere. Det skulle kunne gøres af almindelige brugere.

Man begyndte at tale om "power-users" i stedet for system-administratorer.

Gør som systemet vil eller få systemet til at gøre, hvad du vil.

Grafisk administration udmærker sig ved tydeligt at vise, hvad man bør gøre. System-designerne kan få brugerne til at holde sig indenfor almindeligvis fornuftige valg.

Eftersom (næsten) alle systemer har fejl, er det vigtigt at vise den anbefalede måde - best practice - i de vinduer og prompts, som guider brugeren gennem opsætning og/eller fejlfinding. Det grafiske user-interfaces (GUI) kan klare det meste, men fastholder brugeren i rollen som den, der vælger imellem forskellige muligheder i stedet for at skabe nye muligheder.

Det er selvfølgelig også fornuftigt, og giver fungerende systemer, hvis man holder sig til en bestemt leverandør.

Det kan imidlertid også give anledning til at en hel verden har samme fejl. Der er brug for kyndige system-planlæggere.

De skal kunne se om bag systemets funktionsmåder.

Ofte er de kyndigste administratorer også programmører, system-programmører kalder man det somme tider. (Måske skulle vi have en artikel om titler!)

Server-administration

For server-administration er sagen mere kompleks: Der kommer 3.parts produkter, som man gerne skulle kunne bruge sammen med operativsystemet, og derfor må man selv bygge administrative rutiner.

Servere i store miljøer (Amazon, Google, NASA, Yahoo, Facebook) er baseret på Unix (BSD/Linux) og har behov for tusindvis af servere, som skal køre konstant.

Derfor er der behov for systemer til administration af konfiguration.

Det vil være spild at belaste servere i en 24/7 service med grafiske systemer, så man *"nøjes med kommandolinien"*.

Det er nu ikke så ringe endda, for den moderne kommandolinie-fortolker kan hjælpe med både at gætte hvad man mener og med at finde filer, programmer, options, dokumentation osv.

Kommandolinie-fortolkeren på Linux/Unix kaldes en shell, den er skallen omkring computerens power.

Det har vi forsøgt at vise lidt af i de foregående numre af DNyt, og i de følgende numre vil vi tage handsken op og give et grundkursus i hvordan og hvorfor indenfor serveradministration.

■

Internet *of* Things

SuperUsers nye IoT-kurser:

IoT Overblik:
På vores 1-dags kursus får du det samlede overblik over anvendelser, platforme og teknologier indenfor IoT.

IoT Programmering:
Kom på et af vores 3-dages programmerings-kurser og lav din egen IoT enhed. Her bliver du både udstyret med viden og hardware til selv at kunne lave IoT-enheder.

IoT Sikkerhed:
Når alle ting pludselig kan tilgås fra internettet, er der også nogle sikkerhedsmæssige udfordringer. På vores 2-dages kurser lærer du at udforme sikre IoT-løsninger.

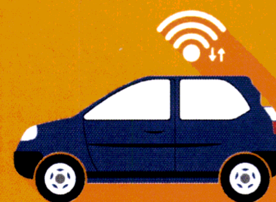
Windows 10 IoT

Linux IoT

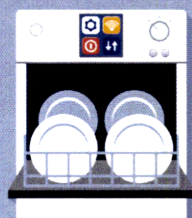
Microcontroller IoT

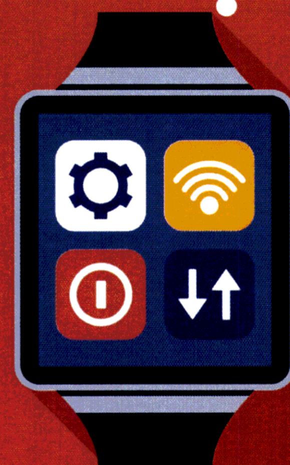
SuperUsers nye IoT-kurser

Se mere på: www.superusers.dk/iot

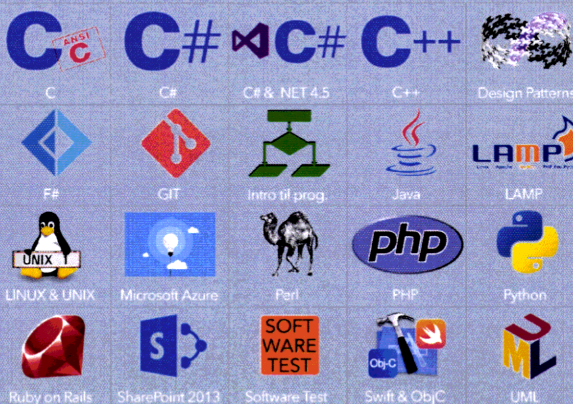


IoT - Internet Of Things - er den nye bevægelse, hvor alle tænkelige enheder bliver koblet på internettet. IoT kan være alt fra industrielle sensorer/transmitters til enheder i det private hjem, som med forbindelse til internettet får nye anvendelsesmuligheder. IoT er i ekstrem vækst. I dag er der 7 mia. enheder knyttet til internettet!





Som Danmarks største IT-kursushus har SuperUsers også mange andre programmeringskurser



SUPERUSERS

Danmarks største IT-kursushus | Afdelinger i Aarhus og Hillerød