Technical Report DIKU-TR-96/38
ISSN 0107-8283
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 KBH Ø
DENMARK

December 1996

Java og WWW Services

Morten Frank & Max Melchior

Synopsis

Java udvider radikalt World Wide Webs muligheder. Med udgangspunkt i den eksisterende arkitektur af WWW tilbyder Java transparent distribution af programmer til et stort heterogent distribueret system (Internettet), og et sikkert homogent miljø for disse programmer at afvikles i. Java understøtter distribuerede brugerorienterede applikationer ved at implementere tråde og synkronisering, understøtte kommunikation indenfor flere modeller og tilbyde mulighed for konstruktion af ensartede brugergrænseflader. Java kan endvidere udvides til at opfylde krav om nye kommunikationsmodeller og øget sikkerhed.

Forord

Denne rapport er resultatet af et projekt under kurset Distribuerede Operativsystemer og blev afleveret til bedømmelse maj 1996. I denne udgave er enkelte fejl og uklarheder blevet rettet, men rapporten foreligger ellers uforandret.

Emnet for rapporten er i øjeblikket genstand for intensiv interesse, hvilket har resulteret i et stort antal udgivelser — med denne rapport er tendensen yderligere forstærket. Heldigvis foregår der også en intensiv udvikling på området, hvilket på kort tid vil gøre mange af disse publikationer overflødige. Vi har allerede selv måtte konstatere, at flere af oplysningerne i denne rapport ikke længere er aktuelle. Dette gælder f.eks. oplysninger vedrørende konkrete produkter og flere af de foreslåede standarder. Her håber vi, at denne rapport vil inspirere læseren til at opsøge relevante kilder og yderligere fordybe sig i dette spændende område.

Vores arbejde er foregået med udgangspunkt i emner behandlet under kurset Distribuerede Operativsystemer. Målet har været en kortfattet beskrivelse af Java og WWW i relation til disse emner, og ikke en generel introduktion. For læseren med en bredere orientering vil visse dele af teksten derfor virke noget indforstået, og vi henviser i disse tilfælde til litteraturlisten eller til den føromtalte strøm af nye udgivelser for en yderligere orientering.

Indholdsfortegnelse

Indledning 1

1 World Wide Web – Arkitektur og Services 3

1.1 Indtroduktion til WWW . . . 3

1.1.1 Universal Resource Identifiers . . . 5

1.1.2 Hypertext Transfer Protocol . . . 6

1.1.3 Hypertext Markup Language . . . 6

1.2 WWW-baserede services . . . 7

1.2.1 W3 som simpelt publiktionsmedie . . . 8

1.2.2 W3 som publikationsmedie med typede dokumenter . . . 8

1.2.3 W3 som brugergrænseflade til resourcer . . . 9

1.2.4 W3 som brugergrænseflade til arbitrære applikationer . . . 10

1.2.5 Diskussion . . . 11

2 Java som klientsprog 14

2.1 Kort om Java . . . 14

2.2 Distribution af Java programmer . . . 15

2.2.1 Fordele . . . 16

2.2.2 Ulemper . . . 16

2.3 Arkitekturneutralitet . . . 17

2.3.1 Java Virtual Machine . . . 17

2.3.2 Betydning for distribution . . . 17

2.3.3 Implementationer af Java VM . . . 17

2.4 Sikkerhed . . . 18

2.4.1 Java er fortolket . . . 18

2.4.2 Sproglige faciliteter . . . 18

2.4.3 Loadning af klasser . . . 19

2.4.4 Beskyttelse af lokale resourcer . . . 19

2.4.5 Sikkerhed i runtime-systemet . . . 20

2.4.6 Implementation af sikkerheds-klasser . . . 20

2.5 Andre faciliteter i sproget . . . 20

2.5.1 Objekt orienterede konstruktioner . . . 21

2.5.2 Dynamisk loadning . . . 21

2.5.3 Tråde . . . 21

2.5.4 Synkronisering . . . 22
2.5.5 Kommunikation fra applets . . . 23

3 Eksempel **24**
3.1 Serveren . . . 24
3.1.1 Protokol . . . 25
3.1.2 Implementation . . . 25
3.2 Klienten . . . 26
3.2.1 Brugergrænsefladen . . . 26
3.2.2 Beregninger . . . 26
3.2.3 Kommunikation . . . 27
3.3 Klientens opbygning . . . 27
3.3.1 ValutaTable . . . 27
3.3.2 ValutaProvider . . . 28
3.3.3 ValutaServerProxy . . . 28
3.3.4 ValutaConverter . . . 29
3.3.5 ValutaApp . . . 29

4 Evaluering **30**
4.1 Kommunikation i Java . . . 30
4.1.1 Generelt om implementationer af protokoller . . . 31
4.1.2 Kommunikation via sockets . . . 31
4.1.3 Kommunikation via URL . . . 31
4.1.4 Remote Procedure Calls . . . 32
4.1.5 Remote Objects . . . 32
4.1.6 Objekt Mobilitet . . . 33
4.1.7 Agent-baseret kommunikation . . . 33
4.1.8 Konklusioner og perspektiver . . . 34
4.2 Sikkerhedsaspekter . . . 35
4.2.1 Klientsikkerhed . . . 36
4.2.2 Serversikkerhed . . . 37
4.2.3 Klient og serversikkerhed . . . 38
4.2.4 Konklusioner . . . 39
4.3 Arkitekturneutralitet og generelle bemærkninger . . . 40

5 Konklusion **42**

Litteraturliste **44**

A Kildeteksten til ValutaApp.java **47**

B Kildeteksten til server.c **48**

Indledning

I denne rapport vil vi undersøge generelle Client/Server applikationer baseret på World Wide Web, herunder specielt brugen af Java.

Applikationer til World Wide Web (W3) er vidt forskellige, fra helt simpel publikation af informationer til avancerede elektroniske værktøjer. Det, der samlet kendetegner disse services er, at de alle tilgåes på en simpel måde via et HTTP klientprogram – en Web Browser. I den simple ende af dette servicespektrum findes simpel HTML baseret publikation af tekst: Regelsamlinger, tabeller, artikler og meget andet. Gradvist er kompleksiteten af disse applikationer blevet forøget, f.eks. er andre typer af dokumenter blevet integreret i World Wide Web konceptet, det er blevet muligt at vise dokumenter med indlejrede billeder, det er blevet muligt at tilgå databaser og slutteligt er det blevet muligt direkte at overføre eksekverbare programmer til klienten.

Denne udvikling af Web baserede services er emnet for det første kapitel, hvor vi først beskriver de centrale ideer bag World Wide Web og derefter følger udviklingen frem imod *transportable front-ends* – platformuafhængige applikationer. En af de muligheder man har for at udvikle denne type applikationer er Java systemet, som er emnet for resten af rapporten.

I det andet kapitel vil vi beskrive Java sproget og Java omgivelserne. Vi vil specielt koncentrere os om de centrale komponenter i dette system – f.eks. hvorledes arkitekturneutraliteten implementeres og hvilke sikkerhedsmæssige overvejelser der er gjort.

For at illustrere Java systemets anvendelighed som udviklingsplatform for Web baserede applikationer har vi implementeret et simpelt eksempel, som vi beskriver i kapitel tre.

I kapitel fire evaluerer vi hele systemet – mere specifikt vil vi diskutere kommunikation og sikkerhed, to vigtige emner for distribuerede applikationer. Her vil vi beskrive hvilke muligheder der pt. er i Java systemet og hvilke muligheder man med fordel kunne tilføje for at gøre systemet endnu mere anvendeligt for applikationsprogrammøren. Til sidst i denne rapport vil vi prøve at konkludere ud fra de erfaringer vi har gjort os.

Emnet berører mange forskellige dicipliner, og vi har i vores gennemgang prøvet både at beskrive Java systemet i en rimelig dybde (kapitel 2,3 og 4) samt at placere hele konceptet i en bredere sammenhæng (kapitel 1). Grundet dette er vores litteraturliste

blevet temmelig lang, idet der var meget der skulle undersøges i den samlede vurdering af systemet. Der er blevet brugt enkelte generelle baggrundsbøger og artikler, men langt hovedparten af materialet er af mere specifik art. Det meste af materialet har været så nyt, at det endnu kun er publiceret i elektronisk form. I tråd med emnet for rapporten, har vi angivet URL adressen for de fleste af referencerne i litteraturlisten.

Kapitel 1

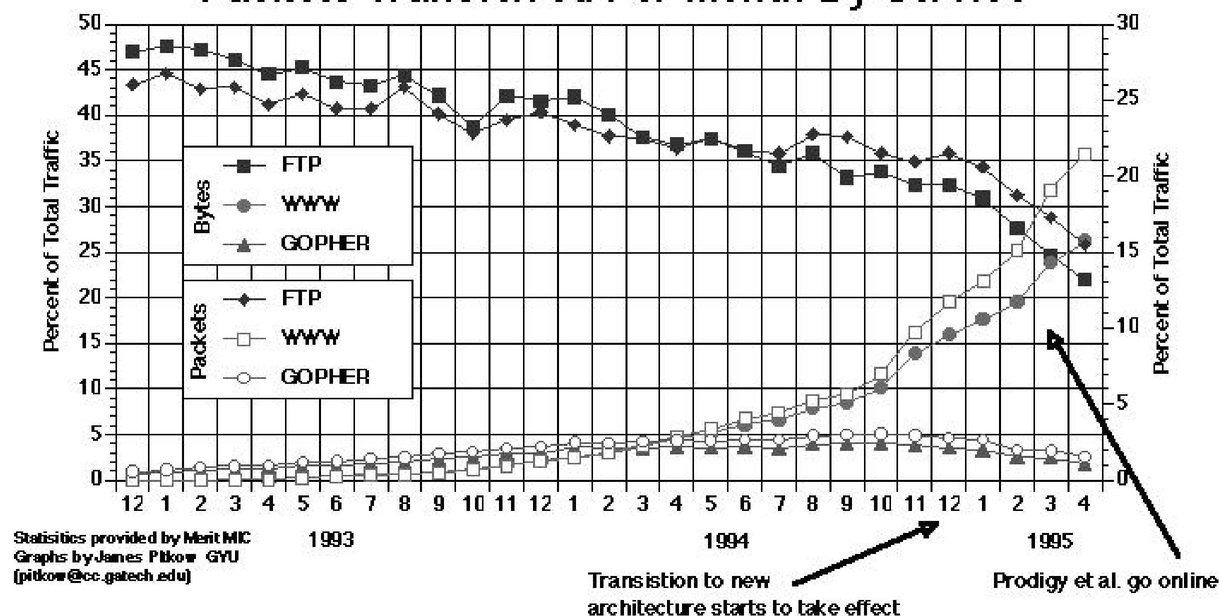
World Wide Web – Arkitektur og Services

Brugen af WWW (W3) er eksploderet på meget kort tid [7]– se figur 1.1, der viser forholdet mellem trafikken fordelt på netværksservice for et amerikansk *back-bone net*, som vi antager er repræsentativt. Dette kan kun skyldes, at der er nogle ideer, muligheder og kvaliteter i dette system, som har vist sig at være meget anvendelige og har tiltalt en stor gruppe af brugere. For slutbrugeren har en af de afgørende punkter givetvis været enkeltheden i "klik dig frem" konceptet kombineret med den flotte opsætning af tekst og billeder. For udbyderne af information har et vigtigt punkt i valget af dette medie nok været en kombination af en stor (mulig) målgruppe samt den ensartede måde, hvorpå information kan udbydes på. Brugen af det gængse formateringssprog – HTML – har muliggjort, at ikke-eksperter har kunnet udbyde information med en opsætning, struktur og layout der ser meget professionelt ud.

Men anvendelsesmulighederne for W3 rækker langt ud over den nuværende anvendelse. Et af de steder, hvor der først nu er ved at ske noget afgørende er på Client/Server området, hvor udbydere af services med W3 har fået en distributionsmekanisme af hidtil ukendt potentiale.

1.1 Indtroduktion til WWW

Det, der i dag kendes som World Wide Web (W3) blev udviklet med intentionen om at være en samling af menneskelig viden, som ville tillade geografisk adskilte parter at dele deres ideer og samarbejde om et fælles projekt. Udviklingen startede i CERN, det europæiske partikelfysik laboratorium i Geneve. Fysikere og ingeniører i CERN samarbejder med mange andre institutter og enheder, og har derfor brug for en enkel måde at udveksle information på, herunder naturligvis også på elektronisk form. Ideen om et Web



Figur 1.1: WWW statistik [7]

udviklede sig hurtigt her, efter positive erfaringer med et lille hjemmebrygget hypertekst system, brugt til at holde styr på personlige informationer i et distribueret projekt [4]. De centrale ideer i W3 er:

- Distribution af viden
- Samarbejde mellem geografisk adskilte parter
- God skalering
- Ingen centrale komponenter
- Fejltollerant

Af ovennævnte punkter er skaleringen meget central. At systemet skalerer godt betyder f.eks., at hvis systemet bruges uafhængigt til to seperate projekter og der så senere knyttes kontakter på tværs af disse, så skal integrationen kunne foretages nemt og uden centrale ændringer. Det er netop denne evne til at skalere, der har gjort, at W3 har kunnet udbrede sig så hurtigt, som det har gjort. W3 er kommet til at stå for en række punkter, der bør adskilles. Disse indkluderer:

- Ideen om en grænseløs verden af informationer, hvor hvert element har en “adresse” med hvilken man kan finde det

- Et adresseringssystem (URI) [3]
- En netværksprotokol (HTTP) [6]
- Et formateringssprog (HTML) [5]

1.1.1 Universal Resource Identifiers

Via W3 har man mulighed for at tilgå en lang række af resourcer – objekter. Tilgangen til disse objekter kan ske vha. eksisterende protokoller (FTP, gopher mm) eller protokoller udviklet i forbindelse med W3 (HTTP). W3 er desuden designet til at kunne inkorporere endnu ikke udviklede protokoller. Identifikationen af og tilgangen til et specifikt objekt er under nogle protokoller indkodet som en adressestreng, andre protokoller benytter en form for navngivning af objekter. Hver protokol definerer således sit eget adresserum. For at kunne tale om disse generelle (netværks)objekter bruger man begrebet *den universelle mængde af objekter* samt *den universelle mængde af navne og adresser på objekter*.

En *Universal Resource Identifier* (URI) [3] er et medlem af mængden af navne og adresser på objekter – til hvert tilgængeligt objekt kan der tilknyttes en URI. I URI syntaksen indgår angivelsen af en *scheme* (metode/netværksprotokol), der angiver adresserummet samt definerer fortolkningen af resten af URI-strengen. Der eksisterer URI metoder for FTP, NNTP, Gopher, WAIS, HTTP mm, og det er simpelt at tilføje nye protokoller. Dette adresserings/navngivnings system tillader altså, at (netværks)objekter med vidt forskellige karakteristika bliver tilgået på en ensartet måde.

En *Uniform Resource Locator* (URL) er en form for URI, hvor der i adressen (identifikationen af objektet) er angivet en rute til objektet. Dette system gør det mere simpelt at lave rutningsalgoritmer på basis af URL'en, men referencen er i sagens natur ikke lokalitetstransparent. URL'er er nødvendige for at kunne tilpasse de fleste eksisterende protokoller til URI syntaksen.

En *Uniform Resource Name* (URN) er en URI, hvor blot navnet på objektet er angivet. Ideen er, at mere persistente objekter kan refereres via navn og ikke lokalitet, og dermed kan et givent objekt replikeres transparent. Dette system er endnu ikke udbygget i særlig grad, da det kræver brugen af resolutions algoritmer (f.eks. i form af specialiserede nameservers). Både URL'er og URN'er delmængder af URI.

Brugen af URI's er central for W3 arkitekturen. Dette muliggør en simpel adressering af et objekt ethvert sted på Internettet, hvilket er essentielt for at opnå en god skalering samt for at informationsrummet er uafhængigt af aktuel netværks og server teknologi. [4]

1.1.2 Hypertext Transfer Protocol

Hypertext Transfer protokollen (HTTP) [6] er en applikations-orienteret protokol, som er bygger ovenpå en TCP forbindelse. Protokollen er en *stateless request-recieve* protokol, idet TCP forbindelsen kun bliver opretholdt til én forespørgsel-svar. Dette valg af en *stateless* protokol skyldes anvendelsen, hvor brugen af hyperlinks forårsager spring mellem mange forskellige og uafhængige (netværks)objekter – spring i kontekst.

En HTTP forspørgsel fra en klient starter med en *operation code* – den metode, der skal anvendes på den resource som den efterfølgende *request*-URI udpeger. Metoderne i brug er bl.a. GET og POST. GET metoden benyttes når der skal hentes information. Operationen er idempotent, dvs. at den ikke ændrer tilstanden af det refererede objekt. (Dog kan den forårsage statestikopsamling, f.eks. i forbindelse med betaling for services). POST metoden anvendes når data skal sendes fra en klient til en netværksresource – f.eks. en opdatering til en database eller data til behandling i et applikationsprogram.

I hovedet af HTTP svaret angives information om typen af det data der sendes (meta-information). Denne angivelse er i overenstemmelse med MIME [2] – Multipurpose Internet Mail Extension – som er et standard format for udvekslingen af elektronisk post indholdende afsnit af forskellig type (PostScript, audio/video clip mm). Protokollen kan derfor anvendes ved overførsel af enhver type data. Det er dermed også muligt selv at definere nye formater og anvende disse uafhængigt af en "central registrering". Ved forespørgslen kan der foregå en "forhandling", hvor klienten sender en vægtet liste med de formater der kan accepteres, og serveren svarer med data i et af de formater den kan producere, dette muliggør simpel integrering af ikke udbredte formater. Den version af HTTP der er i brug på nuværende tidspunkt er version 1.0, der har afløst version 0.9. Version 0.9 tillod kun en simpel GET metode med tilhørende svar, version 1.0 har flere metoder tilknyttet.

1.1.3 Hypertext Markup Language

På trods af mulighederne for at "forhandle" formaterne ved brug af HTTP, er der i W3 brug for et fælles basissprog for udveksling af hypertekst dokumenter. Dette sprog er HyperText Markup Language (HTML) [5]. Sproget er et formateringssprog, der er designet simplet, således at både slut brugeren samt applikationsprogrammer uden større problemer kan producere dokumenter af denne type. HTML er et generelt hypertekst sprog, og kan anvendes alle steder, hvor det ønskes at definere dokumenthierakier/strukturer. Sproget indeholder konstruktioner til overskrifter, lister, tabeller, inklusion af grafik mm. – alle konstruktioner der er meget anvendelige under opbygningen af dokumenter beregnet til præsentation af information. HTML sproget er kommet i flere versioner, hvor hver ny version har fået tilføjet flere konstruktioner. Noget af det seneste er tilføjelsen af *forms*, hvor en klient har mulighed for at indtaste data inde i et dokument – disse kan så eventuelt sendes tilbage til en HTTP server via en POST metode. HTML sproget er udviklet med basis i *Standardized Generalized Markup Language* (SGML) [35].

1.2 WWW-baserede services

Mange virksomheder, institutioner samt personer ønsker at dele og udbyde services af enhver art. Eksempler på disse er:

- Publikation af data:
  - Uformelt udbud af data
  - Kommercielt udbud af data
  - Samarbejde indenfor en gruppe vha. distribueret data
- Udbud af elektroniske værktøjer (også kommercielt):
  - Beregningsopgaver
  - Database systemer
  - Simulering
  - Spil

Mange af disse services henvender sig til almindelige brugere, det vil sige ikke computereksperter men dog kvalificerede brugere – brugeren ved noget om det data/værktøj der skal benyttes, men vil ikke belæmres med tekniske detaljer. Et generelt problem for udbyder og bruger er distributionen af data og hjælpe-applikationer. Det normale er pt., at klienten selv – manuelt - anskaffer sig det ønskede, hvilket giver andledning til et generelt distributionsproblem mht. opdatering af nye data eller applikationer. Hvis disse services kan udbydes via W3 på en måde, så klienten altid har adgang til den nyeste version, er man fri for dette distrubutionsproblem. Yderligere opnår man ved dette, at alle disse services kan tilgåes på en ensartet og letforståelig måde. Enhver med en UNIX arbejdsstation, Macintosh eller PC, en netforbindelse og passende HTTP klient programmer – f.eks. Netscape – kan dermed på en enkel måde udnytte disse services. Man bør dog overveje alternativerne, idet der er adskillige problemer ved at bruge Internettet som distributionsmedie:

- Sikkerhed – data kan principielt læses under transport
- Pålidelighed – netværksforbindelser kan være afbrudt
- Båndbredde – det tager tid at sende store datamængder
- Tilgængelighed – ikke alle har adgang til Internettet

Af alternativer kan nævnes distribution via CD-ROM, disketter eller magnetbånd. Hvis det gælder distribution af store mængder af statisk data (f.eks. lovsamlinger eller materialeegenskabs tabeller) er CD-ROM et fornuftigt alternativ. En CD-ROM kan indeholde store mængder af data, er billig at producere og sikrer slutbrugeren stabilitet i modsætning til et netværk. Hvis en klient skal sende *store* datamængder til en server (f.eks. en CAD tegning til en produktionsvirksomhed), kan et magnetbånd også være et godt alternativ. Hvis de indvolverede parter endvidere ønsker en høj grad af sikkerhed (sikring mod f.eks. industrispionage), kan magnetbånd/disketter være et godt alternativ. Problemerne vedr. sikkerhed er dog ved at blive løst i almindelighed også på Internettet via kryptering og digitale fingeraftryk.

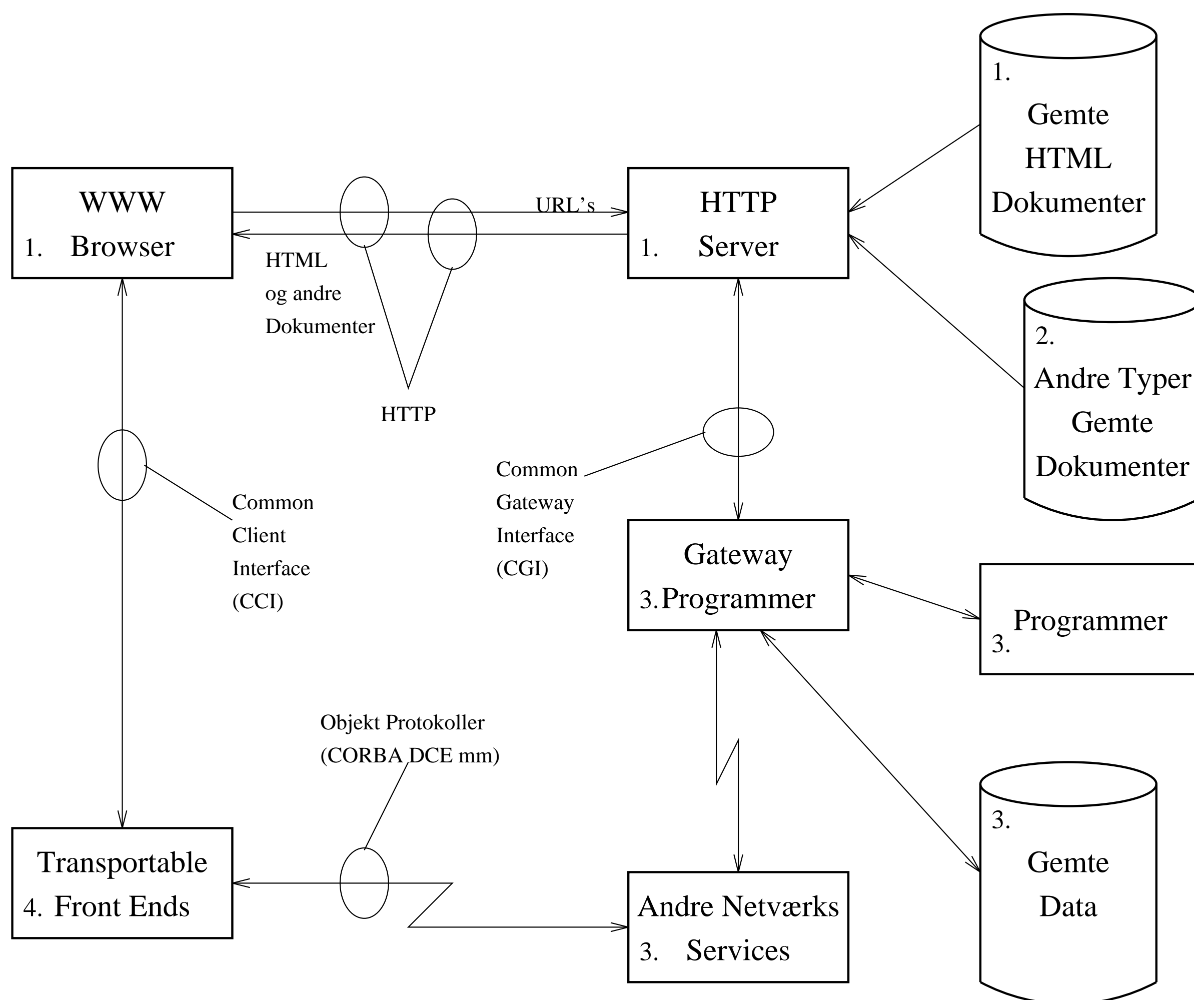
Med udgangspunkt i [27] vil vi i det følgende beskrive fire adskildte trin i W3's udvikling – dog falder punkt 2 og 3 rent tidsligt sammen, men bør holdes adskilt. Client/Server arkitekturen er et gennemgående træk i W3 – en klient sender via HTTP forespørgsler afsted til en eller flere servere. De services der bliver udbudt kan funktionelt inddeles i klasser, som nøje hænger sammen med de fire trin beskrevet nedenfor. Vi vil for hvert af de nedenstående punkter give eksempler på C/S applikationer indenfor denne klasse – se også figur 1.2.

1.2.1 W3 som simpelt publiktionsmedie

Den simpleste og mest nærliggende anvendelse for W3 er som et medium til at publicere HTML baserede dokumenter på. Disse dokumenter kan være af enhver art – forskningsartikler, manualer, regelsæt, underholdning og meget mere. På nuværende tidspunkt findes der allerede tusinde og atter tusinde af sådanne dokumenter [27]. Brugen af HTML sikrer gode muligheder for at strukturere sine dokumenter passende: Intern dokumentstruktur, dokumenthierarkier samt referencer (hyperlinks). Klassen af udbudte services kunne kaldes for *publikation af statiske HTML sider.*

1.2.2 W3 som publikationsmedie med typede dokumenter

Der findes andre formater for arbitrære typer af dokumenter. Dette kunne være PostScript, DVI samt mange former for binære filer, f.eks. CAD tegninger, audio og video klip. Da der allerede findes mange dokumenter af andre typer end HTML er det ønskeligt, at man også kan distribuere disse via W3. Senere versioner af HTTP (1.0) muliggør dette. Som udgangspunkt kan enhver type dokument hentes over netværket og gemmes lokalt. Ved brug af MIME kan serveren via HTTP protokollen tilføje information om dokumenters type, og klientens omgivelser kan sættes op, således at hentningen af en bestemt type dokument starter en passende hjælpe-applikation. F.eks. kan et PostScript dokument hentet via W3 automatisk starte ghostview. Klassen af udbudte services kunne kaldes for *publikation af statiske typede dokumenter.*



1. WWW som simpelt publikationsmedie
2. WWW som publikationsmedie med typede dokumenter
3. WWW som brugergrænseflade til resourcer
4. WWW som brugergrænseflade til abitrære applikationer

Figur 1.2: Udviklingsmodel for WWW services [27]

1.2.3 W3 som brugergrænseflade til resourcer

Der findes andre resourcer end dokumenter, f.eks. databaser. De ovenstående tekniker tillader ikke en aktiv klient, der sender forespørgsler til andet end dokumenter. Da der allerede findes et væld af resourcer der bygger på dynamiske klienter, der sender arbitrære forespørgsler, ville det være ønskeligt, at man kan tilgå disse resourcer via W3. De seneste versioner af HTTP og HTML tillader *forms*. Derved menes, at man inde i et dokument kan angive felter mm., som klienten kan manipulere (udfylde med data). Når klienten på denne måde har bygget en forespørgsel op, kan dette sendes tilbage til en

server. Her modtager HTTP serveren så forespørgslen og overfører denne til et *CGI* program – CGI står for *Common Gateway Interface*, og er en standard for kommunikationen mellem en HTTP server og et for denne lokalt program. HTTP serveren kommunikerer data fra forspørgslen via nogle variabler i CGI programmets omgivelser. Dette program håndterer så forespørgslen på passende vis, eventuel med tilgang til lokale resourcer. Et eksempel kunne være en forespørgsel til en database, hvor CGI programmet oversætter forespørgslen til en passende SQL forespørgsel, foretager opslaget i databasen, omformer svaret til en HTML side og giver denne tilbage til HTTP serveren. Klassen af udbudte services kunne kaldes for *form-baserede grænseflader til resourcer*. Vi vil ikke beskrive CGI programmeringen nærmere, for mere information se [17]

1.2.4 W3 som brugergrænseflade til arbitrære applikationer

Brugen af de ovennævnte *forms* har sine begrænsninger. Hvis en udbyder af en service gerne vil tilbyde klienterne en speciel brugergrænseflade, så kan dette kun lade sig gøre, hvis klienten har en af serveren udbudt applikation. *Forms* modellen kræver også, at alle forespørgsler sendes via HTTP samt at langt det meste af beregningsarbejdet kun kan foretages på serversiden. I forlængelse af *forms* muligheden falder det naturligt at bruge en *transportable front-end* – en applikation (dvs. et eksekverbart program), der kan indkapsles i en HTML side, sendes med i et HTTP svar, og som umiddelbart kan udføres af klienten. For at dette er praktisk muligt skal en række forudsætninger være opfyldt:

- Applikationen skal være arkitekturuafhængig (i stil med HTML)
- Klienten skal være sikker på, at applikationen kan udføres uden at gøre skade på det lokale system

Et gennemgående træk ved brugen af HTML, HTTP og URL'er er arkitekturneutraliteten. Uanset brugerens personlige platform, så sikrer definitionen af disse standarder, at referencen til et objekt (URL'en), tilgangen (HTTP forespørgslen) samt svaret (HTML dokumentet) er uniform. Dette krav må nødvendigvis også gælde det format, som en sådan applikation måtte have, og udelukker dermed brugen af arkitekturspecifik maskinkode. En anden mulighed er at benytte et velkendt programmeringssprog, f.eks. C++, og så oversætte dette lokalt hos klienten. Dette ville kræve, at kildeteksten var skrevet på en 100% portabel måde, så den uden problemer kunne oversættes lokalt og bringes til at køre automatisk – ikke nogen enkel opgave. Desuden ville det tvinge klienten til at investere i en brugbar oversætter.

Ser man lidt på måden HTML dokumenter bliver behandlet på – ved lokal fortolkning – så er det naturlige skridt, at *definere en standard for udførbar kode der skal fortolkes lokalt*. Med hensyn til sikkerhedsaspekterne er dette også en klar fordel, idet man under fortolkningen har langt bedre mulighed for at kontrollere sikkerhedsaspekterne end ved

eksekvering af maskinkode. Java sproget og især *Java Virtual Machine* [13] [20] [23] er et forsøg på at opstille en sådan standard. Andre forsøg er *Safe-Tcl* [12] og *TeleScript* [19].

Der findes et par andre teknikker, der delvist falder ind under dette punkt:

- *NCSA Mosaic Common Client Interface* [34]: En definition af en protokol til kommunikation mellem browseren Mosaic og eksterne applikationer. Der er ikke kommet nyt materiale vedr. CCI siden februar 1995, og der sker nok ikke mere på dette felt.
- *Netscape Navigator Plug-ins* [16]: Applikationer, der kan installeres på brugerens maskine (af brugeren), og som udvider Netscapes funktionalitet – f.eks. findes der plugins til at vise videoclip inde i browseren. Et meget omtalt eksempel på dette er multimedie plug-in systemet Shockwave [31] som tillader egentlige WWW-baserede multimedie præsentationer (baseret på Director formatet, et udbredt multimedie-udviklings værktøj). Plugins er selvstændige programmer, der kan kommunikere med Netscape browseren via et sæt metodekald i denne. Disse applikationer kan derfor også kommunikere med (net)services og bruge browserens vindue som brugergrænseflade. Dette må betegnes som lidt af en hybrid teknik: Den kan essentielt det samme som f.eks. Java miljøet, men applikationerne skal installeres af brugeren selv – og er dermed underlagt de omtalte distributions og sikkerheds problemer. Pt. findes sytemet kun til PC og Mac.

I resten af denne rapport vil vi beskæftige os med Java sproget og Java programmeringsomgivelserne set i relation til den forudgående diskussion.

1.2.5 Diskussion

Der er mange fordele ved at klienten kan udføre kode udleveret af serveren. For at illustrere dette vil vi her sammenligne et interface til en database, skrevet vha. CGI programmering og skrevet vha. en Java applet:

Forms/CGI

- Der skal skrives en HTML side indeholdende en grænseflade til databasen
- Der skal skrives et CGI program, som:
  - Varetager fortolkningen af klientens forespørgsel
  - Oversætter denne til SQL
  - Foretager opslaget i databasen
  - Opbygger en HTML side indeholdende resultatet af opslaget

  - Sende denne til HTTP serveren

Dette giver en stor belastning af de lokale resourcer på serversiden. Ydermere, så vil en ny forespørgsel skulle igennem hele systemet igen - også hvis det er den samme forspørgsel fra den samme klient, eller hvis den samme klient laver en forespørgsel der i princippet kunne håndteres lokalt, dvs. med de data klienten allerede er i besiddelse af. Problemet er, at HTTP serveren er *state-less* og dermed ikke gemmer information om tidligere forspørgsler. Dette problem findes der dog nogle metoder til delvist at omgå. HTML siden returneret som svar kan indeholde en *form* med skjulte inputfelter, hvor der kan gemmes information til brug i næste forspørgsel. Dermed kan man med den *state-less* HTTP protokol opnå en form for *state-full* protokol. Dette må dog betegnes som en noget uelegant løsning, der ikke ændrer ved de grundlæggende problemer bag denne teknik.

Java applet

- Der skal skrives en HTML side der indkapsler en Java applet
- Der skal skrives en Java applet, hvor kravene kunne være:
  - Appletten varetager præsentationen af data
  - Der fortages kun forespørgsler til serveren hvis det ønskede data ikke findes lokalt
  - Klientens forespørgsler kan kontrolleres for fejl før opslaget
  - Klientens forespørgsler oversættes til (optimeret) SQL på klientsiden, og opslaget foretages direkte

Denne model vil kunne reducere både serverbelastningen samt netværkstrafikken. Et problem er dog sikkerheden, både på serversiden og på klientsiden – mere herom senere.

Mere generelt, så er fordelene ved at klienten kan udføre kode udleveret af serveren, at man får adskilt *Repræsentationen*, *Præsentationen*, *Manipulationen* og *Kommunikation* af data:

- Data kan repræsenteres i en *mindste kanoniske form*, hvilket reducerer netværkstrafikken.
- Præsentationen af data varetages af klienten, hvilket reducerer beregningsarbejdet på serversiden
- Ved manipulation af data er der to muligheder:
  - En genberegning på basis af lokalt data kan foretages hos klienten

  - Kun forspørgslen efter *nye* data indvolverer serveren

  Dette reducerer både netværkstrafikken og serverbelastningen

- Kommunikationen kan separeres fra HTTP protokollen og dermed gøres mere effektiv i forhold til den ønskede applikation

Alt i alt kan dette give mere effektive applikationer, idet man ved god programmering kan opnå, at både netværkstrafikken samt beregningsarbejdet på serversiden minimeres. Samtidig er det også lettere at skrive applikationer, idet en adskilelse af de fire punkter falder i naturlige moduler for applikationsprogrammøren:

- Der skal designes en grænseflade til klienten – præsentationen. Dette varetages af et kodemodul hos klienten og kan ændres uafhængigt af serverens grænseflade
- Der skal skrives et beregningsmodul, der varetager klientens manipulation af data. Dette varetages af et kodemodul hos klienten, det er dette modul der afgør om en given manipulation kan varetages lokalt eller ej.
- Repræsentationen af data kan fastlægges udenom HTTP protokollen og dermed optimeres i forhold til applikationen
- Protokollen for kommunikationen af data mellem klienten og serveren kan vælges/designes så den passer til serverens grænseflade. Her findes der flere oplagte kandidater:
  - FTP
  - HTTP
  - RMI – *Remote Method Invokation*
  - User Agents
  - CORBA interface
  - ...

  Selve kommunikationen kan indkapsles i et Proxy-objekt, dvs. et objekt der både fungerer som klient og som server:

  - Klienten ser objektet som server, og kommunikationen forgår ved metodekald
  - Serveren ser objektet som en klient der sender forspørgsler. Det er denne kommunikation der skal følge en af de ovennævnte protokoller.

  Dette punkt vil blive diskuteret yderlige i et senere afsnit.

Kapitel 2

Java som klientsprog

I forrige kapitel så vi udviklingen mod *transportable frontends*, klientprogrammer, der implementerer en brugergrænseflade til services og funktioner tilbudt over W3. Java og *Java-enabled* browsere er et bud på et generelt system til at implementere disse.

Der er forskellige aspekter, som gør Java egnet til at implementere klient-applikationer til distribuerede systemer, specielt når disse baseres på Internettet og navnlig W3. Det drejer sig om de ydre mekanismer til distribution af programmerne, faciliteter ved de omgivelser Java-programmer udføres i, samt egenskaber ved selve sproget Java.

2.1 Kort om Java

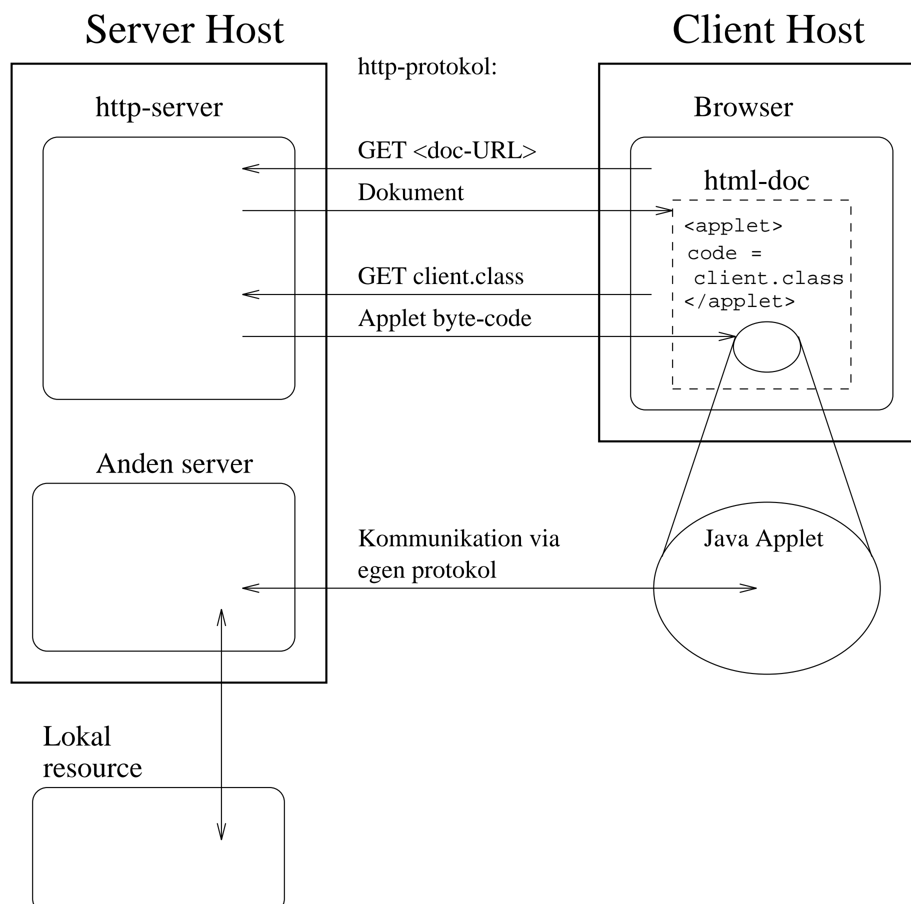
Java, kort for *Java programming language environment*, startede som en del af et projekt hos Sun Microsystems for at udvikle programmer til indlejrede systemer og enheder forbundne vha. netværk. Målet var at udvikle et lille, pålideligt, portabelt, distribueret, *real-time* system. Dertil er kommet krav om sikkerhed og dynamisk integration af nye programdele. Java består dels af programmeringssprog, oversætter og køretidsomgivelser (fortolker og standardbiblioteker). Se [20] for en generel introduktion.

Java sproget var oprindelig baseret på C++, men der er fjernet en del konstruktioner, dels for at gøre det enklere at programmere, men også af sikkerhedsmæssige årsager. I afsnit 2.4.2 og afsnit 2.5 gennemgår vi de vigtigeste ændringer.

Java er designet så det er let at distribuere og udføre programmer, såkaldte *applets*, via W3. Man kan dog også oversætte og udføre Java-programmer som *stand-alone* applikationer, og de vil her køre som almindelige programmer, med de sædvanlige rettigheder og egenskaber, og kunne derfor erstattes af C eller C++ programmer. Der gælder lidt forskellige regler for de to typer programmer, og vi vil i det følgende kun beskæftige os med distribuerede programmer.

2.2 Distribution af Java programmer

Kliententprogrammer baseret på Java forventes at køre som *applets*: Små programmer som indlejres i HTML-sider. Her forklares mekanismerne bag distributionen af selve programmet.



Figur 2.1: Distribution i en Java client/server-model

Som beskrevet tidligere er udgangspunktet i W3, at der findes en HTTP-server på servermaskinen og et browserprogram på klientens maskine. Browseren sender en HTTP-forespørgsel efter et dokument på en bestemt adresse (URL) til HTTP-serveren, der sender dokumentet tilbage. Under fortolkning, dvs. præsentation, af dokumentet finder browseren en reference (URL) til et applikationsprogram, og den beder HTTP-serveren om dette. Programmet sendes til browseren, og bringes til at køre (se figur 2.1).

2.2.1 Fordele

Java-applets kan opfylde præcist den samme funktionalitet som traditionelle C/S-applikationer hvad angår kommunikation med serveren (se afsnit 2.5.5). På figur 2.1 ses hvordan klient-applikationen kommunikerer direkte med en server via en egen protokol, som kan være bygget oven på sockets. Af sikkerhedsmæssige årsager er der derimod begrænsninger på en applets tilgang til lokale resourcer.

Distributionssystemet ovenfor tilbyder yderligere alle W3s metoder til at *finde* de relevante applikationer (hypertekst henvisninger, søgesystemer, etc.), samt en transparent mekanisme til at hente selve programmet til klient-maskinen.

2.2.2 Ulemper

Grundlæggende problemer

Det rejser en række praktiske og sikkerhedsmæssige problemer at hente et program ind på klient-maskinen fra en ukendt kilde og køre det. Disse problemer er løst først og fremmest gennem brug af et portabelt og maskinuafhængigt format for den eksekverbare kode, samt gennem en række sikkerhedsmekanismer i sprog og Web Browser, der skal beskytte klienten mod ondsindet/fejlbehæftet kode. Disse to aspekter bliver gennemgået i hvert sit afsnit.

Effektiv distribution af programmer

Hvis programmerne er store, vil den nævnte metode til distribution og opgradering af programmer give meget kommunikation, og dermed meget lange opstartstider, hver gang en klient ønsker at benytte serveren. Løsninger kan bestå af følgende:

- Programmerne caches lokalt, og en transparent versionskontrol sørger for at hente en ny version, når en sådan findes hos serveren. HTTP tilbyder faciliteter til dette.
- I forbindelse med ovenstående, kan man sørge for at kun de relevante programdele (objekt-klasser) opdateres (f.eks. protokol-håndteringen). Dette er muligt da al loadning og binding af programdele sker dynamisk og ved navnereferencer (se afsnit 2.5.2).
- Man kan evt. implementere sin egen opdateringsprotokol, så serveren eksplicit kan fortælle klientprogrammet, at det ikke er den nyeste version, hvorefter klientprogrammet sørger for at opdatere sig selv.

2.3 Arkitekturneutralitet

2.3.1 Java Virtual Machine

Java-programmer oversættes til et arkitekturneutralt maskininstruktionssæt kaldet *byte-codes*, som kan fortolkes på Java Virtual Machine [23]. Java VM implementerer en fuldstændig standardiseret maskine, både hvad angår instruktioner og repræsentation af simple data-typer, men ikke nødvendigvis for objekter. Ud over Java VM består køretidssystemet af et standard-bibliotek, som blandt andet implementerer maskinuafhængig i/o, herunder et vinduessystem til grafiske brugergrænseflader.

2.3.2 Betydning for distribution

Man er dermed sikret at et oversat Java-program vil opføre sig på præcis samme måde på alle maskiner, uafhængigt af arkitektur og operativsystem. Dette betyder først og fremmest at klientprogrammer kan udveksles frit. Korrekte Java programmer vil køre korrekt på alle maskiner, og der vil kun være brug for én eksekverbar version af hvert program. Man kan sikre sig at en klient altid har den nyeste version af et program, f.eks. hvis programmet altid hentes og startes fra en Web-side, og man er derfor udenom de sædvanlige problemer med programversioner, distribution og opgradering. Klienten vil slippe for manuel installation, etc., af de nye versioner. Serveren behøver ikke længere at understøtte flere versioner af et klientprogram indtil alle klienter har opgraderet, men sørger blot for at klienten får det sidste nye program samtidigt med at klienten ønsker at anvende serveren.

2.3.3 Implementationer af Java VM

Javas succes afhænger dels af om JavaVM accepteres som standard og dermed udbredes, dels af om arkitekturneutralitet faktisk kan opfyldes.

Suns egen fortolker, der tilbydes sammen med en generel udviklingspakke, findes i øjeblikket til Sun Solaris og Win95. OSF står for en portning til HP-UX 10 og ATT SysV.4 til Pentium. IBM har planer om at porte Java VM til alle deres platforme (se [30]). Kildeteksten og specifikationer til Java VM er frigivet, så alle kan porte til deres eget system. Netscape inkluderer en implementation af Java VM i de nye versioner af deres browser (2.0 og frem), som findes til alle større platforme (Mac, PC, de fleste UNIX-varianter).

I kapitel 4.3 kigger vi nøjere på om de aktuelle implementationer af Java VM faktisk er arkitekturneutrale.

2.4 Sikkerhed

Her følger en kort gennemgang af sikkerhedsaspekter ved udførsel af Java applets. Gennemgangen er taget fra [1]. De sikkerheds-faciliteter som Java understøtter drejer sig i vid udstrækning om beskyttelse af klienten mod ondsindet kode. Det klienten hovedsagligt ønsker at undgå er misbrug af lokale resourcer, f.eks. at applets ødelægger vigtige filer, overbelaster systemet, eller benytter en brugers rettigheder til at sende information ud af et system som ellers ikke var offentlig tilgængelige. Problemet kompliceres ved at programmer ofte skal have adgang til en vis mængde lokale resourcer (fx. læsning og skrivning af filer) for at være nyttige. Desuden er der et overordnet krav om effektiv programudførsel, som forhindrer meget omstændige sikkerhedsforanstaltninger.

2.4.1 Java er fortolket

Java programmer udføres ved fortolkning på en virtuel maskine, som logisk er adskilt fra den normale eksekverbar kode. Dette muliggør en høj grad af kontrol med Java-programmernes køretidsomgivelser, og mulighed for at holde øje med udførslen. Ren maskinkode kunne give en mere effektiv programudførsel, men vanskeliggør også muligheden for beskyttelse mod virus, eller for at programmer løber løbsk og overtager hele maskinen. Dette skal undgås, da klientprogrammer må forventes at køre navnlig på små personlige maskiner, fx. PC, Mac, eller lignende, som ikke implementerer nogen særlig beskyttelse af de vigtige resourcer, som det f.eks. sker i UNIX.

2.4.2 Sproglige faciliteter

Selve Java-sproget tilbyder følgende egenskaber for at understøtte sikkerhed. Mange af disse egenskaber er opnået ved at fjerne alle de konstruktion fra C++ som tillader at man 'roder rundt i lageret'.

- Ingen pointere — ingen explicitte referencer til lager
- Beskyttelse af private data og metoder
- Implicit lager-styring (garbage collection)
- Stærk type-kontrol, ingen implicitte type-konverteringer

Disse faciliteter understøttes i første række af oversætteren og senere af køretidssystemet.

2.4.3 Loadning af klasser

Siden man ikke kan være sikker på at koden er produceret af en korrekt oversætter udsættes bytecoden for yderligere kontrol når den hentes af browseren (se [37]).

Klasser fra andre maskiner hentes som en streng af bytes og konverteres til et passende internt format. Derefter bliver koden udsat for en række check.

- Bytecode verifier:

  Ved denne kontrol kan man før kørslen udelukke over- og under-løb på stakken, og man er sikker på at koden kalder andre objekter på en lovlig måde:

  - Der foregår ingen ulovlige typekonvertering mellem interne typer i JavaVM såsom heltal og objektreferencer, eller mellem referencer til forskellige klasser (f.eks. ved *castning* af pointere).
  - Begrænsninger ved tilgang til metoder/variable bliver overholdt.
  - Forbehold for nedarvning overholdes.
  - Kald foregår med lovlige argumentertyper.

- Referencer og navnekonflikter i den hentede kode:

  Alle referencer i en klasse-fil sker ved navn, hvilket kan give problemer, når man bringer klasser fra flere kilder sammen. Problemet bliver løst ved at hver kilde får sit eget adresserum. Som eksempel får klasser hentet lokalt et eget adresserum, og når de finder en reference til en anden klasse leder de først i det lokale adresserum. Hermed sikres, at klasser udefra ikke forveksles med lokale klasser, så applikationer lokkes til at bruge klasser udefra. Når der er fundet en gyldig klasse til et navn, kan navnereferencen erstattes med en mere effektiv form for reference.

Sikkerhedsmekanismen implementeres af ClassLoader-klassen. Efter denne kontrol kan koden udføres med minimal kontrol.

2.4.4 Beskyttelse af lokale resourcer

Systemresourcer bliver tilgået gennem standard bibliotekerne, som er en del af omgivelserne ved udførsel af Javaprogrammer. Det er derfor op til disse biblioteker at undersøge rettigheder, osv. De kan benytte systemets aktuelle SecurityManager-objekt til dette. SecurityManager-objektet har en række metoder til at undersøge og definere retten til at læse og skrive filer, eller udføre en række andre systemkald.

2.4.5 Sikkerhed i runtime-systemet

Under udførslen undersøges det, at alle operationer på objekter kun manipulerer de givne objekter. For eksempel undersøges det, at referencer i arrays kun foregår med indeks indenfor array'ets grænser. Desuden kan alle type-konverteringer kontrolleres dynamisk, da alle objekter indeholder eksplicit type-angivelse.

2.4.6 Implementation af sikkerheds-klasser

Standard bibliotekerne findes lokalt på klient-maskinen, og bruges uden sikkerhedskontrol. De bliver normalt leveret med browseren/Java VM. Som sagt er det op til standard bibliotekerne at implementere korrekt tilgang til resourcer, og dette ansvar er dermed lagt over på den aktuelle udbyder af Java VM.

Nescape har bla. implementeret sin egen ClassLoader og SecurityManager. Applikationer under Netscape må i øjeblikket kun (se [33])

- udføre beregninger, og andre operationer i hukommelsen.
- kontrollere skærmen indenfor et defineret område på klientens browser
- åbne og bruge vinduer, der automatisk mærkes som upålidelige
- kommunikere via sockets til serveren hvorfra applikationen blev hentet. Dette gælder blandt andet ved loadning af klasser.

Det er meningen at de to klasser ClassLoader og SecurityManager kan (gen)implementeres af systemadministratorer, der ønsker specielle adgangsforhold. Se [22] for yderligere beskrivelse af disse og andre klasser.

2.5 Andre faciliteter i sproget

Et af aspekterne ved Java/W3-systemet er at applikationer ikke nødvendigvis skal skrives i Java. Der findes allerede ADA-til-JavaVM-oversættere, der producerer bytecode, som overholder sikkerhedsreglerne. Det er derfor mere relevant at kigge på de generelle egenskaber ved Java-sproget, som er nødvendige for at skrive W3-baserede klientapplikationer, end at studere sprogtekniske detaljer. Vi kigger nærmere på følgende:

- Objekt orientede konstruktioner
- Dynamisk loadning og løsning af referencer
- Konstruktioner til kontrol af parallelle tråde indenfor en applikation
- Et net-bibliotek til fleksibel understøttelse af kommunikation fra Java applikationer.
- Et standardiseret bibliotek til programmering af systemuafhængige brugergrænseflader

2.5.1 Objekt orienterede konstruktioner

De generelle egenskaber og fordele ved objekt orienterede sprog er gennemgået af andre. Vigtigt for os er, at OO tillader en meget klar opdeling af et programs funktionalitet i afskærmede programdele.

I Java er det muligt at definere abstrakte klasser gennem *interfaces*, som er en måde at specificere grænsefladen til en række konkrete objekt-implementationer, der tilbyder de samme faciliteter. Denne facilitet anser vi for central i et distribueret miljø, hvor der vil være mange udbydere af services gennem konkrete klasser. En gruppe udbydere kan enes om et standardiseret interface, men tilbyde forskellige implementationer. Se desuden afsnit 4.1.5 for diskussion af forbindelsen til den gældende standard for definition af distribuerede objekter, CORBA IDL (Interface Definition Language).

2.5.2 Dynamisk loadning

De enkelte programdele (objekt-klasser) kan hentes hver for sig, og dette kan også ske under selve programudførslen. Referencer bindes først til konkrete objekter under selve loadningen, så man er ikke afhængig af en egentlig linkning af de forskellige programdele. Dette betyder både, at man kan opdatere de enkelte programdele hver for sig, men også at de forskellige dele kan komme fra forskellige kilder.

2.5.3 Tråde

Tråde kan oprettes i Java programmer på en meget enkel måde. Der er indbygget metoder til administration og prioritering af tråde, samt kommunikation og synkronisering mellem tråde.

Når koden i et objekt skal udføres i en separat tråd skal objektet enten nedarve fra Thread-klassen eller implementere Runnable-"interfacet". Begge disse indeholder en run() metode, som man overskriver. Koden der skal udføres i sin egen tråd skal ligge i denne run() metode.

```
class MinSlagsTråd extends Thread {
  public void run() {
    // Gør hvad der nu skal gøres ...
  }
```

For at sætte tråden igang konstrueres en ny tråd af MinSlagsTråd-klassen, og `start()` kaldes.

```
Thread th = new MinSlagsTråd();
th.start();
```

Nu bliver `MinSlagsTråd.run()` kaldt i sin egen tråd `th`, som vil udføres parallelt med den aktuelle.

Java-tråde implementeres i det underliggende operativsystem. Dette betyder, at skedulering af tråde er afhængig af operativsystemet. Man kan således være ude for at applikationer skal håndtere at køre både på time-sliced og ikke-time-sliced (fx. Win95) systemer. Dette brud på arkitektur-neutraliteten diskuteres i afsnit 4.3.

2.5.4 Synkronisering

Til hvert objekt i Java, hører der en monitor. Metoder i objektet kan erklæres som metoder i monitoren (dvs. der sikres udelelig adgang for kalderen af objektet) ved at erklæres som `synchronized`. Til hver monitor er der knyttet en (unavngiven) condition-variabel. Man kan vente på at en betingelse opfyldes ved at kalde `wait()`. Når en bestemt betingelse er blevet opfyldt, signaleres dette ved at kalde `notifyAll()`, der vækker alle ventende tråde. Herefter er det op til den enkelte tråd at undersøge om betingelsen er opfyldt:

```
while (! condition )
  wait();
```

Dette er en simpel, men ineffektiv og usædvanlig måde at opnå synkronisering på. Ved at kræve at betingelser hele tiden undersøges, kan det dog være at en del programmeringsfejl kan undgås, specielt hos uerfarne programmører. Hvis man ønsker en mere effektiv signalering, kan man benytte `notify()` metoden, der nøjes med at starte én ventende tråd, men hvis man ikke er 100% sikker på hvad den pågældende tråd ventede på, må dette alligevel undersøges af tråden selv, og man må evt. kalde `notify()` igen.

Det vil ikke være noget stort problem at implementere sin egen mere effektive condition-klasse, der opfylder den almindelige standard for condition-variable i monitorer. Dermed ville man med fordel kunne bruge mange løsninger fra litteraturen.

2.5.5 Kommunikation fra applets

En Java applikation kan have behov for at kommunikere med en server, med andre klienter, eller med det lokale system.

Kommunikationen mellem klient og services

Standard-biblioteket understøtter kommunikation vha. stream og datagram-sockets, samt ved forbindelser til URL-adresser. Begge typer sockets svarer til deres UNIX-modpart, men brugen er blevet en del forenklet, bla. tillades kun internet-adressering. Til både stream-sockets og URL-forbindelser er der tilknyttet stream-objekter, både til læsning og skrivning. Mest spændende er måske at der findes stream-typer, hvor man både kan læse og skrive de simple typer som heltal, etc. Selve kodningen af disse typer til et netværksformat, og konverteringen til den aktuelle arkitektur foregår helt transparent. Java definerer dog ikke en netværkskodning for objekt-instanser. Dog findes der specialiserede metoder, der kan læse objekter af ganske bestemte typer fra en URL. For eksempel kan en Applet indlæse et billede. I afsnit 4.1 kommer vi ind på flere metoder hvorved client/server kommunikationen kan foregå, og hvordan denne kommunikation kan indkapsles i objekter.

Kommunikation mellem klienter

Denne form for kommunikation er kontroversiel inden for en client/server model, men man kan dog have gode grunde (bla. hensyn til effektivitet) til at kommunikere direkte mellem klienter. Der er dog grundlæggende problemer: En applet vil normalt kun leve så længe som den bliver vist på en side, og man kan derfor ikke være sikker på hvornår en klient i form af en applet vil være tilgængelig. Derudover er der problemet med at finde og håndtere referencer til andre applets.

En applet har mulighed for at få en liste af de applets som findes på samme side, som den selv. Herefter kan den kommunikere direkte til disse applets ved metode-kald. Al kommunikation og referencer mellem andre applets må gå igennem en eller anden form for ekstern server.

Kommunikation med det underliggende system

Der er mulighed for at kalde lokale (*native*) programmer skrevet i f.eks. c fra Java. Dette er dog af gode grunde ikke tilladt i applets, der hverken kender eller bør have tilgang det underliggende system. Programmering og linkning mellem de lokale programmer og Java programmet er ikke trivielt.

Kapitel 3

Eksempel

Vi viser nu et eksempel på hvordan Java kan bruges til at implementere en grænseflade til en eksisterende service, og dermed gøre denne service tilgængelig over World Wide Web. Vi vil fokusere på mulighederne for at:

- definere grafiske grænseflader
- benytte egne kommunikationsprotokoller
- distribuere beregninger og derved minimere kommunikation

Vi har implementeret en klient-applikation – ValutaApp – som kommunikerer med en server, der tilbyder kurser for en række valutaer. Klienten opdaterer dynamisk sine data, og foretager lokale beregninger på foranledning af brugeren. Klient-applikationen hentes til en browser på klientmaskinen som vist i afsnit 2.2. Applikationen er indlejret i et HTML-dokument ved brug af <applet> konstruktionen (defineret i HTML version 3.2).

Vores applikation er kun et principielt eksempel. Der foregår så lidt beregning at det næsten kan være ligegyldigt hvor det foregår. Derimod vil netværksforsinkelsen ved forbindelse til en server være mærkbar. Under opbygningen af applikationen har vi formuleret og anvendt nogle generelle principper for konstruktion af applikationer af denne type, som med fordel vil kunne anvendes under udviklingen af realistiske applikationer.

3.1 Serveren

Vi forestiller os en eksisterende server, der tilbyder up-to-date kurser for en række valutaer. Serveren er gjort så simpel som mulig, bla. ved ikke at tilbyde specielle beregninger, osv. I stedet tilbydes der én standardservice: Hver gang man forbinder sig til serveren,

får man en liste af valutanavne og deres nyeste kurser, samt hvilken valuta kurserne er angivet i (kaldet *base valutaen*). Det er op til mere krævende klienter at implementere en øget funktionalitet selv. Dette stiller øgede krav til klienten, som ikke kunne opfyldes med en traditionel passiv browser. Ved brug af en aktiv klient sparer man serveren for beregningsarbejde, hvilket gør den mere effektiv.

Serveren befinder sig på en bestemt maskine og port. For at kunne bruge Netscape som platform for klient-applikationen, må vi føje de gældende sikkerhedsregler og lægge serveren på samme maskine som den HTTP-server HTML-dokukumentet blev hentet fra.

3.1.1 Protokol

Protokollen for kommunikation med serveren er som følger:

- Forespørgsel:

  Man foretager en forespørgsel om de nyeste kurser ved at koble sig til den port, som serveren sidder på. Dette sker konkret ved et connect kald til en socket på denne adresse.

- Svar:

  Serveren sender følgende data tilbage:

  - Hoved: Antal valutaer i beskeden angivet som en 4 byte integer (Big endian). Base valutaen angivet i en c-streng på 4 bytes, (sidste byte er altid '\0'). Der benyttes ASCII tegnsæt.
  - Krop: Et antal poster som angivet i hovedet, hver på 16 bytes. Hver post indeholder en Valuta angivet som en c-streng på 8 bytes (se ovenfor), og en Kurs angivet i en IEEE 64-bit double (Big endian)

3.1.2 Implementation

Vores server er skrevet i c, og er baseret på et program udleveret under kurset. Et praktisk problem stoppede os fra at implementere serveren i Java: Den eneste implementation af JavaVM, der pt. findes til DIKUs platforme er i Netscape Navigator2.x, og Netscapes version af sikkerhedsystemerne tillader ikke implementation af en serverfunktionalitet (se afsnit 2.4.6).

Serveren tilbyder kun kursen på nogle få valutaer, og kurserne ændres ikke.

3.2 Klienten

Vi ønsker følgende funktionalitet hos klienten: En kursliste med periodisk opdatering, frit valg af base-valuta samt forklaring af kurs-værdierne ved valg af en valuta. Som man ser kræver dette mere end den service, serveren kan tilbyde. Periodisk opdatering kræver at klienten kommunikerer med serveren og beder om nye data med jævne mellemrum. Valg af base-valuta kræver både en interaktiv brugergrænseflade og en mulighed for at beregne nye kurser på baggrund af de tilbudte. For at tilbyde den ønskede forklaring på en valgt valutas kurs, kræves også en interaktiv og opdaterbar grænseflade.

Ideelt set skal koblingen mellem interaktion og opdatering i grænsefladen være tæt, eller med andre ord: Hvis vi trykker på en knap gider vi ikke vente hundrede år på at der sker noget. Dette betyder at grænsefladeopdatering bør være så uafhængig af netværkskommunikation som mulig. Dette opnås i vores klient ved at alle beregninger, der er nødvendige for en bruger-initieret opdatering af grænsefladen, foretages lokalt. Vi sørger for at kommunikationen med serveren sker asynkront (i sin egen tråd) og uden indflydelsede fra brugeren.

Vi ser nu på de enkelte dele af klient-applikationen.

3.2.1 Brugergrænsefladen

Java standard biblioteket `java.awt` stiller en række værktøjer til rådighed ved konstruktion af brugergrænseflader. Blandt disse er komponenter som tekstfelter, lister, menuer, etc., men også systemer til at sammensætte og administrere disse komponenter. De fleste af disse komponenter tillader ændring af indehold under udførsel, samt registrering af interaktion – f.eks. valg af et menupunkt.

Der er to muligheder for interaktion med brugergrænsefladen (*begivenheder*): Valg af ny basevaluta fra en liste af muligheder, og valg af en valuta hvis kurs skal forklares. Dertil tilføjer vi en tredje begivenhed, der skal reageres på: Den periodiske opdatering af valutakurser. Behandlingen af begivenheder samles således et centralt sted i koden, hvor man kan afgøre hvordan der skal reageres. Vi sørger for at opdatering af valutakurser ikke blokerer for behandling af de andre begivenheder.

Vores færdige brugergrænseflade ses i figur 3.2, side 29. Brugergrænsefladen implementeres i klassen ValutaApp, se afsnit 3.3.5.

3.2.2 Beregninger

Vi modtager som sagt en række valutaer og kurser fra serveren, samt navnet på basevalutaen, den valuta kurserne er angivet i. Hvis man kender kursen på en ny basevaluta i den

nuværende basevaluta, er det enkelt at udregne nye kurser for alle valutaer. Vi beregner nye kurser hver gang der vælges en ny basevaluta, samt når der kommer nye data fra serveren. Når der vælges en ny basevaluta skal de nye kurser beregnes på baggrund af originale data fra serveren for at undgå unøjagtigheder ved gentagne konverteringer. Selve beregningen varetages af klassen ValutaConverter, se afsnit 3.3.4.

3.2.3 Kommunikation

Vi ønsker at indkapsle kommunikation til serveren og dermed skjule protokol-detaljer, dels for at forenkle resten af programmet, dels for at gøre det uafhængigt af en aktuel protokol. Vi definerer et objekt, der tilbyder serverens funktionalitet, men som gemmer kommunikationen med selve serveren. Et sådant objekt kaldes en *proxy*, som kommer af proximity (nærhed): Man har en server(repræsentant) liggende lokalt.

Selve kommunikationen foretages altså af dette objekt, der dermed fungerer som den egentlige klient (i serverens øjne). I biblioteket `java.net` defineres en socket-klasse, der bruges til at åbne den konkrete forbindelse. Til en socket er der tilknyttet en læse og en skrive-stream. I `java.io` defineres streams, der tillader læsning og skrivning af de grundlæggende typer, heriblandt arrays of bytes, integers og doubles.

C-Strengene læses som et array af bytes, det afsluttende '\0' klippes af, og det hele konverteres til Java Strings. Både repræsentationen af heltal og doubles svarer til den repræsentation som Java forventer, så de kan læses direkte. Kommunikationen varetages af ValutaServerProxy-klassen, se afsnit 3.3.3.

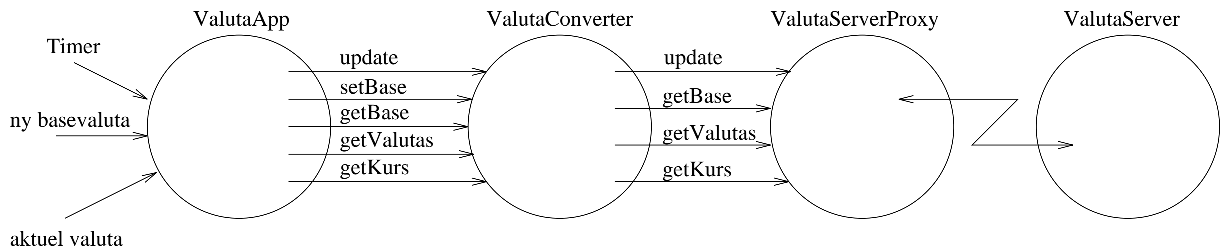
3.3 Klientens opbygning

Klient-applikationen består som sagt af en brugergrænseflade, beregning og kommunikation. Disse er implementeret i hhv. ValutaApp, ValutaConverter samt ValutaServerProxy. Disse objekter er vist i figur 3.1.

Nedenfor følger en forklaring af disse og andre vigtige objekt-klasser:

3.3.1 ValutaTable

Denne datastruktur anvendes til at opbevare en række konkrete valuta/kurs par, samt hvilken valuta kurserne er angivet i. Vi har behov for at gøre denne klasse *thread-safe*. Ved kun at anvende indbyggede klasser, der altid tilgås atomisk, sikrer vi at de enkelte variable i tabellen tilgås udeleligt. Brugere, der opdaterer tabellen, må selv sørge for at base kurs og de enkelte valuta kurser er indbyrdes konsistente.



Figur 3.1: ValutaApplet arkitektur

3.3.2 ValutaProvider

Dette interface definerer grænsefladen for objekter, der tilbyder kursinformation. En `ValutaProvider` tilbyder aktuelle kurser for en række valutaer angivet i en bestemt grundvaluta. Det er op til den enkelte implementation af `ValutaProvider` hvor den får sine kurser fra. Der er defineret følgende metoder:

```
void update();
String getBase();
Enumeration getValutas();
Double getKurs(String Valuta);
```

`update`	opdaterer aktuelle valutaer og kursangivelser. Kurser kan godt opdateres løbende og uafhængigt af hinanden.
`getBase`	returnerer basevalutaen.
`getValutas`	giver en liste af de valutaer, der findes kurser for, Listen returneres som en `Enumeration`. Denne klassen har følgende metoder: `boolean hasMoreElements()` og `Object getNextElement()`, som kan bruges til at gennemgå valutaerne en efter en.
`getKurs`	angiver den aktuelle kurs for en valuta.

3.3.3 ValutaServerProxy

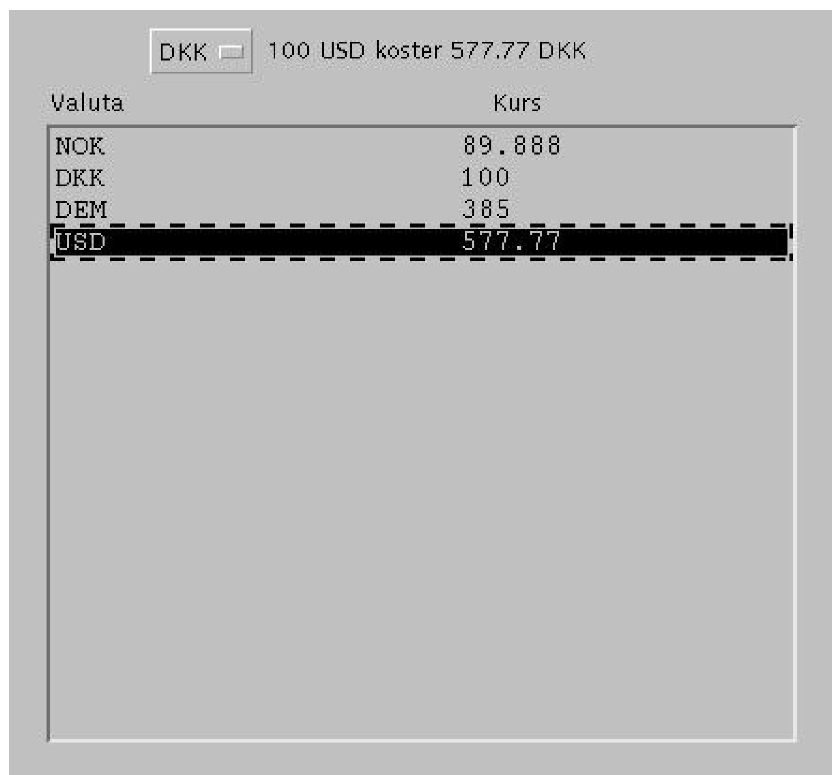
Denne klasse implementerer en `ValutaProvider`. Den sørger for forbindelse til og opdatering fra en server, der tilbyder kursservice. Kursinformation opbevares i en `ValutaTable`. Kommunikation foregår vha. sockets til en angiven server (host, port). Protokollen er beskrevet ovenfor i afsnit 3.1.1. Eftersom en opdatering kan tage lang tid, sørger vi for at kald til `update` ikke blokerer for de andre metoder.

3.3.4 ValutaConverter

Denne implementation af `ValutaProvider` tilføjer muligheden for selv at vælge basevalutaen ved kald af `void setBase(String Valuta)`. Klassen implementerer denne service ovenpå en anden `ValutaProvider`, som sørger for aktuel kursdata. `ValutaConverter` opdaterer sine data i den private metode `calculate()`, og denne er derfor erklæret `synchronized`. Ved korrekt brug af objektet, må kun en tråd opdatere basevalutaen og hente kursværdier, ellers er man ikke sikret konsistens mellem kurser og basevalutaen.

3.3.5 ValutaApp

Dette er selve brugergrænsefladen. Kursdata fås fra en `ValutaConverter`. Disse data opdateres med passende mellemrum. Basevalutaen vises i en `Choice`, en grænsefladekomponent, der tillader valg af ny basevaluta på en enkel måde. Der vises en liste af valutaer og deres kurser i den aktuelle basevaluta. Ved at klikke på en af disse valutaer får man derudover en mere udførlig beskrivelse af hvad 100 enheder af den pågældende valuta koster i basevalutaen.



Figur 3.2: Klientens brugergrænseflade

Kapitel 4

Evaluering

Efter at have beskrevet baggrunden for Java, detaljer ved sproget samt vist et eksempel på en applikation skrevet i Java, vil vi nu foretage en generel evaluering af Java, herunder om Java lever op til sine ambitioner.

Java tilbyder et system til udvikling af distribuerede applikationer via W3. Her vil vi diskutere to centrale aspekter af distribuerede systemer, nemlig kommunikation og sikkerhed, og beskrive hvilke muligheder Java har i disse henseende i forhold til eksisterende systemer.

En af Javas største fordele er muligheden for transparent distributionen af applikationsprogrammer. Denne transparens baseres til dels på arkitekturneutralitet, og vi diskuterer derfor i et afsnit om den nuværende implementation rent faktisk overholder kravet om arkitekturneutralitet.

4.1 Kommunikation i Java

Der findes flere modeller for client/server kommunikation, hvoraf mange er overlappende og bygger ovenpå hinanden. Forskellige applikationer vil have forskellige behov for kommunikation, og for at kunne tilfredstille disse behov må Java kunne dække et bredt spektrum. Grundlæggende kan man skelne mellem synkron og asynkron kommunikation.

I et sprog som Java, hvor der er mulighed for parallel udførsel af tråde, er skellet mellem synkron og asynkron kommunikation dog ikke så skarp. Asynkron kommunikation kan implementeres via synkrone kommunikationskald, hvis blot disse udføres i en separat tråd, der ikke stopper udførslen af hovedprogrammet.

For hver model overvejes mulighederne i Java/W3. De nyeste af de modeller vi beskriver er stadig under udvikling, og vi kan derfor hellere ikke forvente nogen færdige løsninger indenfor Java.

4.1.1 Generelt om implementationer af protokoller

Vi vil i de følgende afsnit beskrive hvordan forskellige former for kommunikation kan implementeres i Java-objekter. Hvis kommunikationen kan indkapsles helt i disse objekter, opnår man at applikationen slipper for at kende til den aktuelle protokol, men blot skal sørge for at overhold grænsefladen til objektet. En af fordelene ved denne konstruktion er, at protokollen kan ændres uden at applikationen skal skrives om.

En server kan således tilbyde et passende objekt, der varetager kommunikationen for klienten. Hvis serveren selv står for klient-applet'en, kan kommunikationsdelen sendes med appletten. Ellers må klient applikationen sørge for at hente kommunikationsdelen hos den server man ønsker at benytte. Bemærk: Når vi i dette afsnit har talt om at tilbyde objekter mener vi selvfølgelig objekt-klasser, der kan distribueres som `.class` filer.

4.1.2 Kommunikation via sockets

Sockets er en fundamental konstruktion til kommunikation mellem især UNIX maskiner, og de findes både til synkron kommunikation som stream-sockets, der bygger på en TCP-forbindelse, eller til asynkrone datagrammer ovenpå UDP. Ved at kommunikere gennem sockets tillader man brug af 'vilkårlige' protokoller, og man vil derfor kunne forbinde sig til eksisterende serverprogrammer (*legacy servers*).

Java understøtter begge former for sockets (se afsnit 2.5.5), men i modsætning til UNIX tillades kun adresseringen af en specifik servers socket ved host-navn og port-nummer (internet adressering). Det er ikke muligt at adressere mere specifikke objekter. Dette skal ske i den protokol som kommunikationen foregår i.

Man kan håndkode socket-kald direkte, men hvis man ønsker en pænere (objektorienteret) grænseflade til kommunikationen, kan disse kald indkapsles 'for hånden' i passende objekter. Et eksempel er ValutaServerProxy-objektet i vores applikation (afsnit 3.3.3).

4.1.3 Kommunikation via URL

En specifik protokol til kommunikation via sockets er HTTP. HTTP tilbyder at sende beskeder via forespørgsler og derefter vil man modtage svar. Kommunikation i W3 foregår normalt ved at sende HTTP-beskeder til URL-adresserede *aktive* objekter (se afsnit 1.1.2 for nærmere beskrivelse af protokollen). De refererede objekter skal kunne forstå og reagere på beskederne, og de vil sædvanligvis bestå af CGI-programmer (se afsnit 1.2.3).

URL'er kan understøtte vilkårlige protokoller (schemes), og enhver server burde således kunne tilgås ved en URL, og serverens egen protokol. De protokoller der bruges som URL schemes skal definere sit eget adresserum, og bla. derfor skal protokollerne registreres

og understøttes før de kan anvendes generelt (se evt. afsnit 1.1.1). Det er derfor ikke ønskeligt at prøve at benytte sin egen hjemmelavede protokol, og man må bruge en af de registrerede. Ingen af de andre protokoller som for tiden er registrerede som URI-schemes, tillader samme fleksibilitet som HTTP, og det er derfor den mest oplagte kandidat.

Java tilbyder en URL-klasse der kan håndtere de almindelige schemes. Data kan sendes og modtages fra et URL-adresseret objekt ved at læse og skrive stream objekter. Hvis man kender formatet kan vilkårlig data sendes via HTTP. URL tilbyder derfor ved brug af HTTP grundlæggende samme funktionalitet som stream-sockets pakket ind i et adresseringsystem. URL'er er en del af et fremtidsorienteret adresseringssystem, som kan være at foretrække frem for 'ren' internetadressering. Man vil dog være hæmmet af HTTP-protokollens funktionalitet: Det er ikke muligt at undgå den grundlæggende *stateless send-receive* mekanisme, eller skræddersy protokollen til mere effektiv kommunikation.

4.1.4 Remote Procedure Calls

I imperative sprog er Remote Procedure Calls (RPC) en naturlig måde at kommunikere med en server på. Målet med RPC er bla. at gemme alle kommunikationsmæssige detaljer, og f.eks. tilbyde lokalitetstransparent tilgang til en service, og konvertering af kald parametre og returværdier mellem forskellige arkitekturer. Der findes en række eksisterende RPC-systemer, bla. er RPC grundlaget for al kommunikation i DCE. Vi diskuterer flere aspekter af DCE i afsnit 4.2.2.

RPC systemer tilbyder at indkapsle kommunikation med en service som kald til f.eks. c-funktioner. Dette kræver at Java skal kalde disse som *native* procedurer (se afsnit 2.5.5), hvilket er fuldt ud muligt, men hvilket besværliggør udviklingen af Javaprogrammet. Desuden skal disse procedurer distribueres og intergreres i klientens system på anden måde end selve klientapplikationen. For at lette programmeringen af Java-applikationen kan man dog sørge for at kald til disse native procedurer indkapsles i passende objekter, så man opnår den illusion at kalde metoder på et objekt hos serveren.

4.1.5 Remote Objects

Både i afsnit 4.1.2 og 4.1.4 så vi ønsket om at betragte og kommunikere med ikke-lokale resourcer som objekter. Dette kræver både mekanismer til at lokalisere ikke-lokale objekter og til rent faktisk at kalde deres metoder. CORBA (*Common Object Request Broker Architecture*) [10] fra OMG, tilbyder begge mekanismer, samt sproget IDL (*Interface Definition Language*) til at definere navne og metoder på objekter. CORBA søger at gøre objekter sprog og arkitektur uafhængige, blandt andet tilbyder IDL en række standardiserede datatyper. På klientsiden bruges CORBA ved at oversætte en IDL specifikation af et serverobjekt til det sprog man ønsker at kalde objektet i. Herved genereres et passende *client-stub* objekt, som klientprogrammet kan kalde. Denne stub anvender en

ORB-*runtime*, der står for selve kaldet af metoder på det ønskede objekt. Både stub og runtime skal findes hos klienten. Hvis de implementeres som Java-klasser, kan de enkelt distribueres med den applet som skal bruge dem.

Der findes IDL-til-Java oversættere, som tillader Javaprogrammer at kommunikere med CORBA objekter defineret i en IDL-specifikation. Da serverobjekter som tilbydes via en CORBA/IDL er sproguafhængige, kan Java programmer på denne måde kommunikerer med ikke-Java objekter.

Java Remote Method Invocation (RMI) [25] er et forsøg på at integrere kald til ikke-lokale objekter direkte i sproget. RMI tilbydes af Sun som produktet Joe i forbindelse med NEO, som er Suns implementation af CORBA. Joe og NEO understøtter derfor OMGs IDL. Dette betyder også, at Java-objekter kan tilgås af applikationer skrevet i andre sprog. Joe beskrives i [11], RMI i [25], og forholdet mellem IDL og Java i [24]. RMI begrebet omhandler ikke flytning af objekter (*object migration*), men kun referencer til objekter på en anden maskine.

4.1.6 Objekt Mobilitet

For at flytte objekter (som det f.eks. er muligt i Emerald) skal der defineres et format til at sende objekter mellem maskiner. Java tillader allerede at man sender simple data-typer, og udførbar kode i form af Java objekt-klasser i `.class` filer. Det vil være fuldt ud muligt at definere objekter, der kan pakke sig selv ned og sende sig selv via en socket eller ved en anden metode. Klasse-definitioner kan i Java håndteres som objekter på lige linje med andre typer. Man kan evt. tænke sig et standardiseret *on-the-wire* format for objekter, og en udvidelse til de eksisterende oversættere, der automatisk definerer metoder til op og nedpakning til dette format, men endnu er ingen sådan standard annonceret. Dog arbejder OSF på sagen.

Selv hvis det er muligt at flytte objekterne, opstår der dog problemer vedrørende referencer og navnekonflikter. Disse referenceproblemer skal løses begge veje mellem et objekt og dets nye omgivelser. Dette er dog et generelt problem for alle former for objekt mobilitet.

4.1.7 Agent-baseret kommunikation

En anden foreslået mulighed for at varetage kommunikationen mellem klient og server er at benytte mobile agenter [14]. Hvis agenter implementeres som objekter, fx. nedarvet fra en abstrakt Agent klasse, vil denne løsning være en form for objekt mobilitet.

Den forventede fordel ved kommunikation via mobile agenter er mindre kommunikation og hurtigere databehandling, idet agenten kan træffe beslutningen for klienten lokalt på serveren, uden at delresultater og nye forespørgsler skal kommunikeres frem og tilbage.

Java opfylder næsten alle de krav Chess, et. al [15] stiller til et programmeringssprog for mobile agenter: Fortolket, objekt orienteret, muligheder for at dele objekter og fortsætte programudførslen i begge objekter, mulighed for tilgang til *fjerne* objekter for eksempel via proxy-objekter og mulighed for synkronisering mellem tråde. Derimod opfyldes disse krav ikke: Mulighed for at flytte objekter på en enkel måde, mulighed for at pakke objekter op og ned (*flatten and unflatten objects*), og administration af delte objekter.

Som Java/W3 omgivelserne ser ud i øjeblikket vil det ikke være ligetil at implementere mobile agenter i Java. For det første understøtter Java ikke direkte, at man sender konkrete objekter frem og tilbage (og navnlig ikke hvis disse er under udførsel: Der kendes ingen standard repræsentation af en aktiv tråds tilstand). Det eneste man kan gøre i det nuværende system, er at sende klasse-definitioner frem og tilbage. Vi har ingen mulighed for dynamisk at opbygge specifikke agent-klasser, med mindre vi da indkludere en oversætter i vores program. Derfor må vi nok nøjes med at bruge agent-klasser, der er defineret og oversat før kørslen. En statisk agent-klasse vil dog sandsynligvis være for generel til at løse et specifikt problem; vi mangler information om den aktuelle probleminstans.

Derudover skal man løse det grundlæggende problem med at opbygge passende omgivelser for agenter på de maskiner, hvor de skal være aktive. Hvis man får standardiseret agenter og agent-omgivelserne vil dette dog give en mulighed for at løse referenceproblemet på en ensartet måde: Både agenter og agent-omgivelserne ved hvad de kan forvente af hinanden.

4.1.8 Konklusioner og perspektiver

Som det ses er der flere forskellige måder at kommunikere på. De mest etablerede er via sockets og URL'er. Her tilbyder Java fuld funktionalitet. Java er svagere indenfor RPC, indtil der kommer en Java implementation af f.eks. DCE. Udviklingen går mod objektorientede systemer, og her er CORBA ved at blive en etableret standard. Der findes allerede flere implementationer af CORBA systemer til Java. Som en sidebemærkning kan nævnes, at Sun er ved at udvikle et distribueret objektorienteret operativsystem kaldet Spring, som vil benytte NEO, Suns implementation af CORBA. Dette system vil tillade brug af både C++ og Java (gennem Joe). Der arbejdes også for at opnå en generel model for distribuerede services i en ny generation af W3, som forventes bruge CORBAs objektmodel. Et eksempel vil være at erstatte HTTP med CORBA objekter med samme funktionalitet.

Selv med RMI tilbyder Java ingen indbygget mulighed for ægte objekt-mobilitet, hvilket vil være nødvendigt for at implementere mobile agenter. Både objekt-mobilitet og mobile agenter må dog siges at være på et meget eksperimentelt stadie endnu, men det er ret sandsynligt at nye systemer vil blive udviklet i Java. Der kan nævnes et konkret system fra Stanford baseret på Java [18].

4.2 Sikkerhedsaspekter

I dette afsnit vil vi diskutere sikkerhedsaspekter vedr. distribuerede applikationer i relation til Java og i lyset af den forudgående diskussion af kommunikation. Når applikationer udbydes i et offentligt tilgængeligt distribueret miljø på stor skala (Internettet), er et af hovedaspekterne de sikkerhedsmæssige problemer. Kompleksiteten af disse problemer er væsentligt større, end f.eks. de sikkerhedsmæssige overvejelser ved traditionelle *timesharing* systemer som UNIX. De netværksforbindelser man arbejder med er på alle måder usikre/ustabile mht. sikkerhed og fortrolighed. I et UNIX miljø anses det lokale netværk (LAN) mellem de enkelte maskiner at være sikker, dermed kan disse forbindelser uden større problemer benyttes til f.eks. RPC kald samt transport af data og eksekverbar kode. Når en bruger starter en *session*, skal han/hun først legitimere sig. Dette sker én gang, herefter "ved" det underliggende sytem hvem brugeren er, samt hvilke rettigheder vedkommende har til de lokale resourcer. I store distribuerede systemer findes der ikke ét underliggende system, der kan varetage dette. Alle sikkerheds og kontrol systemer må nødvendigvis selv være distribuerede, og hovedreglen er, at man principielt ikke kan stole på andet end sig selv:

- Knuder i netværksforbindelsen kan opsnappe data der passerer
- Knuder i netværksforbindelsen kan ændre data der passerer
- Kommunikationen kan i sig selv være "fjendtlig", dvs. at der udefra bliver gjort forsøg på at tilgå resourcer på en ikke tiltænkt måde

En af de grundlæggende ideer i Java systemet er, at applikationer frit kan udveksles, uden at brugeren behøver at tænke i sikkerhedsmæssige baner. Dette er dog kun ét af de punkter, som er interessante i forbindelse med mere generelle distribuerede applikationer.

Der er tre hovedklasser af sikkerhedsaspekter:

- Klientsikkerhed – sikkerhedsaspekter der berører en klient som benytter en applikation, f.eks. beskyttelse af lokale resourcer
- Serversikkerhed – sikkerhedsaspekter der berører en server som stiller services til rådighed, f.eks. uautoriseret tilgang til servicen
- Fælles sikkerhed – sikkerhedsaspekter der berører både klienten og serveren, her tænkes specielt på, at data i transit ikke opsnappes og/eller ændres

I det følgende diskuteres disse tre hovedaspekter.

4.2.1 Klientsikkerhed

For klienten drejer sikkerhedsaspekterne sig først og fremmest om beskyttelse af lokale resourcer. Hvis en ikke-lokal applikation hentes og bringes til at køre har klienten principielt ingen sikkerhed for at applikationen ikke forvolder skade på lokale resourcer som f.eks. filsystemet. Der er flere klasser af applikationer der skal tages hensyn til:

1. Applikationer der er skrevet i "ond tro" – dvs., at klienten prøver at benytte en applikation der er skrevet med den intention at gøre skade.
2. Applikationer der er skrevet i "god tro":
   (a) Applikationer der er fejlbehæftede, og som kan forette lokal skade uden at det var hensigten fra udbyderens side:
      i. Programmeringsfejl
      ii. Dokumentationsfejl
   (b) Applikationer der ved utilsigtet brug gør lokal skade - f.eks. grundet et dårligt interface

Der er to metoder der kan og bør kombineres, for at minimere de ovenstående sikkerhedsrisici:

1. Koden for applikationen skal ved hentning kontrolleres, og der skal udføres køretidscheck. En diskussion af denne teknik findes i afsnit 2.4, og er et af de centrale punkter i hele Java konceptet. Dette sikrer i høj grad, at "onde" eller fejlbehæftede applikationer ikke kan gøre skade.
2. Brugeren skal kunne kontrollere graden af rettigheder en given applikation skal have, idet det ofte er ønskeligt, at applikationer tildeles lokale rettigheder, f.eks. til læsning og skrivning af filer. Dette tænkes kontrolleret ved at man i browseren kan sætte en række privilegier, som en given applikation skal tildeles. Derved kan alle privilegier f.eks. frakobles første gang en applikation køres. Hvis det så viser sig, at man har "tiltro" til koden, så kan man give applikationen særlige privilegier. Denne teknik skal integreres med kodekontrol, således at brugeren kan iagtage hvilke resourcer applikationen forsøger at tilgå, og derefter sætttte applikationens rettigheder i forhold til den ønskede funktionalitet. Det næste skridt er så at oprette referencer over betroede servere. Applikationer fra disse servere kan så hos klienten automatisk tildeles udvidede rettigheder. Teknikken har i princippet ikke noget med Java systemet at gøre, men vedrører udviklingen af browsere. Selve kontrollen af tilgangen til de lokale resourcer styres af Security Manager Objektet, og det er så den aktuelle implementering af dette objekt, der afgør applikationers rettigheder. Den her diskuterede teknik for udvidelse af rettighederne for en given applikation er planlagt for næste version af Netscape.

4.2.2 Serversikkerhed

For serverens side er sikkerhedsaspekterne:

1. Hvordan beskyttes mod “ondsindede” angreb
2. Hvis det er en “offentlig” server, hvordan beskyttes der mod utilsigtet brug
3. Hvis det er en “lukket” server:
    - Hvordan legitimerer en lovlig klient sig
    - Hvordan sikres de forskellige adgangsrettighederne for de legale klienter

Man kan dele klienterne af disse servere op i to grupper: Legale og illegale. Hvis serveren er “offentlig”, findes der kun legale (og anonyme) klienter. Her er de generelle sikkerhedsforanstaltninger:

- En skarp grænseflade til serveren – hermed menes der, at der før implementationen af de forskellige services er blevet udarbejdet en specifikation af:
    1. Hvilke services der tilbydes
    2. Hvordan disse invokeres af klienterne
    3. Hvilke garantier serveren tilbyder – ved at være meget klar på dette punkt kan man tildels undgå et klassisk sikkerhedsproblem: At serveren arbejder med et lavere sikkerhedsniveau end en af dens klienter. En klient har f.eks. noget data, der lokalt er “klassificeret” på et eller andet niveau. Dette data sendes nu af klienten til en server i forbindelse med f.eks. et RPC kald. Data bliver nu gemt af serveren, som ikke har samme grad af sikkerhedsbestemmelser. Nu kan man havne i den situation, at dette data kan tilgåes af andre, der fra klientens synspunkt ikke burde have adgang til at se disse informationer.

    Det er så meget vigtigt, at denne specifikation overholdes under implementeringen.
- Kontrol af klienternes forspørgsler mht. tilgang af for serveren lokale resourcer. Som eksempel på dette, kunne man forstille sig et distribueret filsystem, hvor kun en del af filsystemet er offentlig. Serveren skal så for hver forspørgsel kontrollere om denne refererer til den offentlige del af filsystemet.
- Ingen sikkerhedshuller. Dette punkt har som bekendt vist sig at være meget svært i praksis at opnå.

Hvis det er en "lukket" server, kommer der udover det overstående yderligere to punkter til:

- En lovlig klient skal legitimere sig
- Rettighederne for de enkelte lovlige klienter skal bestemmes og kontrolleres

Disse to punkter er (blandt meget andet) blevet implementeret i *DCE - Distributed Computing Environment* [9], [32] kapitel 10 samt [36]. I DCE systemet er legaliseringen og tildelingen af adgangsrettigheder (i DCE terminologi: *Authentication* og *Authorization*) adskilt. Hele modellen bygger på såkaldte *tickets*, som er krypterede datastrukturer, og benyttes af klienterne til at få adgang til div. resourcer. *Tickets* findes i flere varianter, der hver især benyttes som "adgangsbillet" overfor en bestemt type server. Fælles for dem er, at de på en sikker (krypteret) måde indkoder information såsom brugerens identitet, rettigheder, udløbstiden for den pågældende *ticket* mm. For at en klient kan få adgang til ikke-lokale resourcer skal der kontaktes flere *security servers*, der hver især udsteder div. *tickets*. Klienten ender op med at have en *ticket*, der kan benyttes ved brug af en ikke-lokal resource. For hver resource varetages der en liste over forskellige klienters adgangsrettigheder. På basis af klientens *ticket* og listen af adgangsrettigheder for en given resource kan en klients forespørgsel så enten godkendes eller afvises.

Hele systemet er temmelig kompliceret, og for en mere detaljeret gennemgang henvises til [32], kapitel 10 – på dette sted er det nok at vide, at systemet kan modstå en lang række af mulige angreb, og at det er muligt at få en sikker forbindelse over et usikkert ("fjendtligt") netværk. I relation til Java applikationer ville dette system være en mulighed. Metoderne kan i princippet benyttes uden at sproget skulle ændres, men ville kræve meget implementationsarbejde. Det ville også være muligt at skrive Java applikationer, der anvender eksisterende DCE resourcer. Dette kunne gøres ved at implementere en række Java klasser, der skulle håndtere kommunikationen med DCE serverne. Hvis de føromtalte DCE RPC kald er blevet implementeret kan disse benyttes, idet sikkerhedssytemet i DCE bygger på RPC. Se også [36] og [8] for en nærmere diskussion.

4.2.3 Klient og serversikkerhed

1. Hvordan sikres evt. nødvendigheden af *atomic transactions*
2. Hvordan sikres evt. nødvendigheden af *secret transactions*

Disse to sikkerhedsaspekter vedrører både klienten og serveren. De er relevante i forbindelse med applikationer som f.eks. databaseopdateringer og *home-banking*.

Begrebet *atomic transaction* er velkendt indenfor distribuerede systemer, og der findes mulige implementeringer – se f.eks. [32] kapitel 3.4. I forbindelse med Client/Server

applikationer baseret på Java er begrebet nyttigt, når en klient skal tilgå resourcer på en "enten alt eller intet" måde. Her tænkes f.eks. på *home-banking*: En bank tilbyder sine kunder at kunne foretage transaktioner ved deres PC. Dette sker via en Java *front-end* til bankens kontosystem. Da banken af tillidsgrunde gerne skulle sikre, at der ikke sker fejl under en transaktion, skal transaktioner være atomiske. Selve implementationen (kan) fortages via en protokol mellem klienten og serveren. På (Java) klientsiden kræver dette en ny klasse, som implementerer denne protokol. På serversiden kræves der noget implementation, både til at varetage kommunikationen med klienten og til at varetage selve funktionaliteten af den atomiske transaktion.

Hvis data skal sendes fortroligt, så skal det på en eller anden måde krypteres. Der findes adskillige algoritmiske løsninger på dette, både udelukkende med hemmelige nøgler og med en kombination af hemmelige og offentlige nøgler. Hvis en klient og en server vil sende data fortroligt med hinanden, skal de derfor på en eller anden måde være enige om krypteringsalgoritme samt kende enten en fælles hemmelig nøgle eller kende hinandens offentlige nøgler. Det sidste er det mest praktiske og fleksible, idet det samtidig muliggør digitale fingeraftryk – hermed menes, at det ikke kun er data der er krypteret, men den ene part kan være sikker på, at data er afsendt af den anden part. Se f.eks. [21] side 318-328 for en introduktion til den sidst nævnte teknik – *Public-Key Cryptography*. Udvekslingen af (offentlige) nøgler udgør endnu et sikkerhedsproblem, idet disse af gode grunde ikke kan distribueres via et "usikkert" netværk – på en knude i netværket kan transporten af en offentlig nøgle opsnappes. I stedet for at vidersende nøglen erstattes denne nu af en ny offentlig nøgle, hvor både den hemmelige og den offentlige nøgle kendes. Hvis modtageren af denne nye offentlige nøgle benytter denne til at sende fortrolig data til den oprindelige server, kan man nu på den "onde" knude dekryptere meddelelsen (og dermed læse data), kryptere den med den oprindelige offentlige nøgle og vidersende det hele til serveren. I DCE sker kommunikation mellem en klient og en server som før omtalt via RPC. Klienten har en lang række af muligheder for at manipulere med sikkerhedsniveauet i disse RPC kald, alt efter den ønskede grad af fortrolighed. Det er f.eks. muligt at få de enkelte netværkspakker krypterede – transparent for brugeren. Hvis de føromtalte DCE RPC kald er implementeret som Java klasser, kan disse bruges til at sende fortroligt data med.

4.2.4 Konklusioner

For at summere op på sikkerhedsaspekterne: Der er gennemgået og diskuteret hovedaspekterne af de sikkerhedsproblemer der er ved distribuerede applikationer i relation til Java. Det nuværende sikkerhedssystem i Java fokuserer primært på knude-niveauet, dvs. klientsikkerheden. Java systemet indeholder på nuværende tidspunkt ingen generelle metoder (klasser) til at sikre netværkssikkerhed: Legalisering af klienter, integritet og hemmeligholdelse af data i transit mm. Disse sikkerhedsaspekter er i høj grad løst, både teoretisk og praktisk – sidstnævnte i f.eks. DCE systemet, og ville kunne implementeres i en applikation via klasser og dedikerede sikkerhedsservere. Der er dog et sidste niveau, der endnu ikke er blevet omtalt – det ville være ønskeligt, også at have sikkerhed på

kode niveau. Denne teknik kan bruges til at sikre, at en given applikation kommer fra en anerkendt server – dvs. en server man har tillid til. Dette tænkes implementeret ved at kode i transit er krypteret. For tiden arbejdes der med udarbejdelsen af S-HTTP (*Secure HTTP*), der bla. imødekommer dette behov [29].

4.3 Arkitekturneutralitet og generelle bemærkninger

I dette afsnit vil vi diskutere aspekter vedr. arkitekturneutraliteten i den nuværende specifikation af Java sproget og Java VM, samt de aktuelle implementationer af dette system. I det følgende bruges begrebet *platform* om kombinationen af en maskine og et operativsystem.

En af de centrale ideer i Java er som nævnt mange gange arkitekturneutraliteten, dvs. at korrekte Java programmer (bytecode) eksekveres korrekt uanset brugerens platform. At et givent program kører korrekt i termer af platformsuafhængighed vil ifølge specifikationen sige, at:

- Funktionaliteten af et program er ens på forskellige platforme
- Den grafiske grænseflade til brugeren *ligner* grænsefladen, som applikationer skrevet til netop denne platform har. Den samme Java applikation ser altså ikke ens ud på forskellige platforme.
- Interaktionen med det underliggende operativsystem er ens for alle Java programmer, uanset platformen

Spørgsmålet er nu, om denne platformsuafhængighed i praksis er opfyldt - både mht. den fremsatte standard og mht. de aktuelle implementationer.

Det er de enkelte biblioteksrutiner i Java klassebiblioteket, som understøtter arkitekturneutraliteten, specielt `java.awt` (*Abstract Window Tool*) og `java.io` (i/o systemkald). Disse biblioteker er naturligvis arkitekturspecifikke, idet de bygger på systemkald, men giver udadtil et ensartet billede – kaldet for API (*Application Programmers Interface*). For at dette system virker korrekt, er det en nødvendighed, at de forskellige operativsystemer funktionelt tilbyder de samme systemkald. Det er netop på dette punkt, at Java specifikationen har nogle svagheder (hvis man vælger at anse dette som svagheder), idet der findes Java bibliotekskald som kræver funktionalitet af det underliggende operativsystem, som ikke alle systemer tilbyder. Der er minimum to problemer:

1. Tråde: Hvis det underliggende operativsystem tilbyder *multithreading*, så benytter Java Trådbiblioteket dette. Ellers implementeres der *user threads*. Dette kan give problemer på platforme, hvor hverken *multithreading* eller *time sliceing* er muligt,

f.eks. på en del PC platforme. En flertrådet applikation, der kører fint på en UNIX platform, kan være umulig på en sådan maskine. Her er det op til applikationsprogrammøren at sikre, at bestemte tråde er højt prioriterede, samt at de med jævne mellemrum frivilligt laver et kontekstskift via et *yield* kald.

2. Den grafiske grænseflade: Personer der udvikler applikationer og som arbejder ved en farveskærm overvejer normalt ikke muligheden af, at applikationen skal køres på en ikke-farveskærm. Brugen af f.eks. farvede bogstaver ovenpå en farvet baggrund kan betyde, at applikationen er ubrugelig på ikke-farveskærme. Igen er det her op til programmøren reelt at gøre applikationen platformsuafhængig. Ud over dette kan det også nævnes, at forskellige systemer opfattelse af komponenters egenskaber ikke harmonerer – som eksempel på dette vil en knap hvis farve ikke specificeres få samme farve som dens baggrund under Motif, og under Windows vil den få en standard farve.

Grunden til, at de ovennævnte punkter ikke nødvendigvis må betegnes som svagheder er, at man har valgt ikke at designe efter "laveste fællesnævner". Java er designet, så det opfylder mange af de krav, man pt. stiller til brugbare udviklingssystemer – herunder brugen af tråde. Og i langt de fleste tilfælde kan applikationer bruges på alle platforme.

Der er på nuværende tidspunkt flere implementationer af Java VM. I hver af disse er både `ClassLoader`'en samt `SecurityManager`'en blevet implementeret. Implementationen af disse er endvidere vidt forskellige, f.eks. er der meget stramme regler for, hvad en applet der kører under Netscape må, hvorimod der er vide regler for en applets opførsel hvis den køres under Suns's Appletviewer. Dette har indflydelse på de enkelte applikationer, hvor programmøren nøje må overveje om den ønskede funktionalitet kan opnåes under afvikling på den Java VM, som målgruppen må formodes at have. Kendes dette ikke, så er en løsning naturligvis at kode efter laveste fællesnævner.

Det er netop på dette område, hvor vi under udviklingen af vores eksempel (se kapitel 3) stødte ind i nogle problemer. Oprindeligt ønskede vi at skrive både klientapplikationen og serverapplikationen i Java. Da der på DIKU pt. kun findes en brugbar version af Java VM – Netscape – var ideen, at også serveren skulle fortolkes via Netscape. Dette var dog et brud på en af Netscapes sikkerhedsforanstaltninger, idet man ikke kan oprette en `ServerSocket` fra en Server (den maskine hvorfra klientens applet stammer) til den pågældende klient. Efter noget prøven frem og tilbage endte vi op med at implementere vores server i C. Det næste problem var, at Netscape kun tillader en applet at åbne socket-forbindelser tilbage til den maskine, hvorfra bytekoden stammer. Da det på DIKU ikke er tilladt at køre programmer (vores server skrevet i C) på den maskine, hvorpå DIKU's officielle HTTP-server befinder sig (www.diku.dk), blev vi nødt til at køre vores egen HTTP-server på en af de maskiner, hvortil vi har adgang. Denne restriktion i hvad en applet må, har betydning for hvilke applikationer man mere generelt kan skrive. Hvis man f.eks. ønsker en gruppe af kommunikerende applets uden central kontrol, så er man alligevel nødt til at have en central HTTP-server, hvorfra bytekoden skal hentes, samt en anden central server (på samme maskine), som skal varetage kommunikationen mellem de enkelte applets.

Kapitel 5

Konklusion

I dette projekt har vi undersøgt Java systemet som udviklingsplatform. Her vil vi kort sammenfatte vores resultater med henvisninger til relevante afsnit.

Java systemet passer naturligt ind i W3's udvikling – systemet opfylder et konkret behov: Muligheden for at skrive platformsuafhængige og let distribuerbare applikationer, der kan tilgås via W3 (afsnit 1.2.4).

Java giver nogle åbenlyse fordele, både for de enkelte klienter og for udbydere af applikationer:

- Transparent distribution af applikationer til klienterne (afsnit 2.2)
- Stor potentiel målgruppe (afsnit 2.3.3)
- Sikkerhedssystem til beskyttelse af den enkelte klient (afsnit 2.4)

Der er centrale faciliteter i selve Java sproget, der muliggør og letter udviklingen af distribuerede applikationer (afsnit 2.5):

- Objektorienterede faciliteter
- Mulighed for flere parallelle tråde
- Et omfattende klassebibliotek
- Fleksible kommunikationsmuligheder

Java og dets omgivelser er endnu under udvikling, og der er minimum tre punkter, hvor systemet bør udvikles yderligere:

- Kommunikationen bør standardiseres på passende vis, for at kunne opbygge et mere stabilt miljø for distribuerede applikationer. CORBA standarden er her en mulig og fornuftig løsning (afsnit 4.1.8).
- Java fokuserer pt. på klientsikkerheden, men adskillige andre sikkerhedsaspekter skal løses, for at kunne udvikle applikationer der indebærer sikre og bindende transaktioner, specielt af økonomisk art. Disse aspekter er løst i DCE, der kunne bruges som udgangspunkt/model (afsnit 4.2.4).
- For at sikre slutbrugerne applikationer, der er mere funktionelle og fleksible, bør de aktuelle Browseres sikkerhedspolitik kunne nuanceres. Dette for at udvalgte applikationer fra udvalgte servere kan tildeles lokale rettigheder. En transparent tildeling af rettigheder indebærer brugen af autentificerede servere og sikker overførsel af kode fra disse (afsnit 4.3).

Disse tre punkter er blandt de centrale arbejdsområder i førsøget på at integrere W3 og Java, og derved skabe en homogen platform for udviklingen af generelle distribuerede elektroniske services.

Litteraturliste

[1] Joseph A. Bank. Java security. Technical report, MIT, 1995.
http://www-swiss.ai.mit.edu/ jbank/javapaper.ps.

[2] N.Borenstein Bellcore and N. Freed. *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies.* Network Working Group, September 1993.
http://ds.internic.net/rfc/rfc1521.txt.

[3] Tim Berners-Lee. *Universal Resource Identifiers in WWW.* Network Working Group, June 1994.
http://ds.internic.net/rfc/rfc1630.txt.

[4] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Communications of the acm*, 37(8):76–82, August 1994.

[5] Tim Berners-Lee and D.Connolly. *Hypertext Markup Language - 2.0.* Network Working group, November 1995.
http://ds.internic.net/rfc/rfc1866.txt.

[6] Tim Berners-Lee, R.Fielding, and Henrik Frystyk Nielsen. *Hypertext Transfer Protocol - HTTP/1.0.* HTTP Working Group, Februar 1996.
http://www.w3.org/pub/WWW/Protocols/HTTP/1.0/spec.html.

[7] Gvu's nsfnet backbone statistics page.
http://www.cc.gatech.edu/gvu/stats/.

[8] The java program page.
http://www.gr.osf.org/java/.

[9] Osf distributed computing environment.
http://www.osf.org/dce/.

[10] The common object request broker: Architecture and specification. OMG Document Number 93.12.43, 1993.
ftp://ftp.omg.org/pub/docs/93-12-43.ps.

[11] Joe: Developing client/server applications for the web. SunSoft White Paper, 1996. http://www.sun.com/sunsoft/neo/external/whitepapers/Joe-wp-new.html.

[12] Nathaniel S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail. In *Proceedings of the 1994 ULPAA Conference*, 1994. http://minsky.med.virginia.edu/sdm7g/Projects/Python/safe-tcl/ulpaa-94.txt.

[13] Mary Campioni and Kathy Walrath. *The Java Tutorial.* Sun Microsystems Inc. http://java.sun.com/tutorial/.

[14] D. M. Chess, C. G. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? IBM Research Report, RC 19887, 1994. http://www.research.ibm.com/massdist/mobag.ps.

[15] David Chess, Benjamin Grososf, Colin Harrison, David Levine, and Colin Parris. Ininerant agents for mobile computing. IBM Research Report RC 20010, 1995. http://www.research.ibm.com/massdist/rc20010.ps.

[16] NETSCAPE COMMUNICATIONS CORPORATION. Inline plug-ins. http://home.netscape.com/comprod/products/navigator/version_2.0/plugins/index.html.

[17] The National Center for Supercomputing Applications. *The Common Gateway Interface.* The University of Illinois at Urbana - Champaign , IL, USA. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.

[18] H. Robert Frost. Java agent template. Eksperimentelt system. http://cdr.stanford.edu/ABE/JavaAgent.html.

[19] General Magic, Inc., 420 North Mary Avenue, Sunnyvale, CA 94086. *The Telescript Language Reference Version 1.0*, October 1995. http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html.

[20] James Gosling and Henry McGilton. The java(tm) language environment: A white paper. Technical report, Sun Microsystems Inc., 1995. http://java.sun.com/whitePaper/java-whitepaper-1.html.

[21] David Harel. *Algorithmics - The Spirit of Computing.* Addison-Wesley Publishing Company, 1987.

[22] Java api documentation. Technical report, Sun Microsystems, 1996. http://www.javasoft.com/JDK-1.0/api/packages.html.

[23] The java virtual machine specification. Technical report, Sun Microsystems, 1995. ftp://ftp.javasoft.com/docs/whitepaper.A4.tar.Z.

[24] Remote objects for java. Technical report, JavaSoft Inc., 1996. http://splash.javasoft.com/pages/intro.html.

[25] Java remote method invocation. Technical report, JavaSoft Inc., 1996. http://splash.javasoft.com/pages/rmi.html.

[26] Bill Clarke Jim Flynn. How java makes network-centric computing real. *Datamation*, pages 42–43, March 1996.

[27] J.W.Erkes, K.B.Kenny, J.W.Lewis, B.D.Sarachab, M.W.Sobolewski, and R.N.Sum Jr. Implementing shared manufacturing services on the world- wide web. *Commun. ACM*, 39(2):34–45, Februar 1996.

[28] Vance McCarthy. Gosling on java. *Datamation*, pages 30–38, March 1996.

[29] E. Rescorla and A. Schiffman. *The Secure Hypertext Transfer Protocol*, Februar 1996. http://ds.internic.net/internet-drafts/draft-ietf-wts-shttp-01.txt.

[30] J. William Semich. Why java? *Datamation*, page 5, March 1996.

[31] Shockwave software reviewer's guide, April 1996. http://www.macromedia.com/Tools/Shockwave/Info/index.html.

[32] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall International Editions, 1995.

[33] Steve Thomas. The navigator java environment: Current security issues. Technical report, Netscape Communications Corporation, 1996. http://developer.netscape.com/standards/java-security2.html.

[34] Dave Thompson. Common client interface protocol specification. Technical report, National Center for Supercomputing Applications at the University of Illinois in Urbana-Champaign., 1995. http://yahoo.ncsa.uiuc.edu/mosaic/cci.spec.html.

[35] Standard generalized markup language. http://www.sgmlopen.org/sgml/docs/sgmldesc.htm.

[36] M. Weiss, A. Johnson, and J. Kiniry. Distributed computing: Java, corba, dce. Technical report, Open Software Foundation Research Institute, 1996. http://www.osf.org/mall/web/SW-java/corba.htm.

[37] Frank Yellin. Low level security in java. Technical report, Sun Microsystems, 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.

Bilag A

Kildeteksten til ValutaApp.java

```
/*
   ValutaApp
   Implements Client side of Valuta Applikation as a Java Applet
   Morten Frank & Max Melchior
*/

import java.util.*;
import java.applet.Applet;
import java.awt.*;
import java.lang.*;
import java.net.*;
import java.io.*;


public class ValutaApp extends Applet implements Runnable {
  // Two Objects for calculations and communications
  private ValutaConverter VC;
  private ValutaProvider VS;

  // Objects for the User Interface
  private Label InfoLabel, HeaderLabel;
  private List KursInfo;
  private Choice ValutaChoice;

  // Integer to hold the userselected valuta
  private int SelectedIndex;

  // Timer Object, used to get periodical updates from server
  private Thread Timer;


  public void init() {
    // Initialize Data-entities
    URL db = this.getDocumentBase();
    String Host = db.getHost();

    VS = new ValutaServerProxy( Host, 7992 );
    VC = new ValutaConverter( VS );

    // Create User-interface
    //   Use two panels, one for the choice and infolabel
    //   one for the kurslist with a headerlabel
    Panel ptop, plist;
```

```
  // Build first panel
  //   Initilize the infolabel
  InfoLabel = new Label("INFOLABEL - vælg valuta og se kurs");

  //   Initilize the choice
  ValutaChoice = new Choice();
  for (Enumeration Valutas = VC.getValutas();
       Valutas.hasMoreElements(); ) {
    ValutaChoice.addItem( (String) Valutas.nextElement() );
  }
  ValutaChoice.select( VC.getBase() );

  //   Set layout and add the choice and inforlabel to the top panel
  ptop = new Panel();
  ptop.setLayout(new FlowLayout());
  ptop.add(ValutaChoice);
  ptop.add(InfoLabel);

  // Build second panel
  //   Initilize the headerlabel
  HeaderLabel = new Label(
    "Valuta                                   Kurs");

  //   Initilize kurslist
  KursInfo = new List(10 , false);
  KursInfo.setFont( new Font( "Courier", Font.PLAIN, 14 ));
  SelectedIndex = -1;

  //   Set layout and add the headerlabel and kurslist to panel plist
  plist = new Panel();
  plist.setLayout(new BorderLayout());
  plist.add("North", HeaderLabel);
  plist.add("Center", KursInfo);

  // Add the two panels to the Applet.
  setLayout(new BorderLayout());
  add("North", ptop);
  add("Center", plist);

  // Startup
  updateKursInfo();
  update();
}


public void updateKursInfo() {
  // Update kursinfo - on new select in choice and on fresh data
  if ( SelectedIndex >= 0 )
    KursInfo.deselect( SelectedIndex );
  KursInfo.clear();
  for (Enumeration Valutas = VC.getValutas();
       Valutas.hasMoreElements(); ) {
    String v = (String) Valutas.nextElement();
    Double k = VC.getKurs( v );
    KursInfo.addItem(
      v + "                             " + k.toString() );
  }
  if ( SelectedIndex >= 0 )
    KursInfo.select( SelectedIndex );
}


public void updateInfoLabel() {
  // Update Infolabel - on new select in kurslist
  if ( SelectedIndex >= 0 ) {
    String Selected = KursInfo.getSelectedItem();
    String v = Selected.substring(0,3);
```

```
    Double k = VC.getKurs(v);
    InfoLabel.setText("100 " + v + " koster " +
                       k.toString() + " " + VC.getBase() );
  }
}


public void update() {
  // The update method, makes changes visible
  validate();
  repaint();
}


public boolean handleEvent(Event event) {
  // The eventhandler - reacts to events by calling appropriate methods

  // Find the target object
  Object target = event.target;

  if (target == ValutaChoice) {
    if (event.id == Event.ACTION_EVENT) {
      // The user has maked a new choice of base valuta
      String newBase = ValutaChoice.getSelectedItem();
      VC.setBase( newBase );
      updateKursInfo();
      updateInfoLabel();
      update();
    }
  }
  if (target == KursInfo) {
    if (event.id == Event.LIST_SELECT) {
      // The user has selected an item in the kurslist for futher info
      SelectedIndex = KursInfo.getSelectedIndex();
      updateInfoLabel();
      update();
    }
  }

  if (target == this) {
    if (event.arg == "Timer") {
      // The timer has caused an interrupt
      VC.update();
      updateKursInfo();
      updateInfoLabel();
      update();
    }
  }
  return super.handleEvent(event);
}


/***************************
 Timerthread and control:
***************************/


public void run() {
// The timer sleeps for a while, then causes a "Timer" event
// and goes to sleep again.....
  while(Timer != null) {
    try {
      Thread.currentThread().sleep(10000);
    } catch (InterruptedException ie) {}
    handleEvent( new Event( this, 0, "Timer"));
  }
}
```

```
  public void start() {
    if (Timer == null) {
      Timer = new Thread(this);
      Timer.start();
    }
  }

  public void stop() {
    if (Timer != null) {
      Timer.stop();
      Timer = null;
      System.out.println("Timer dead!");
    }
  }
}



/****************************************
  ValutaProvider
    standard interface for
    providers of valutadata
****************************************/

interface ValutaProvider {
  public Enumeration getValutas();
  public Double getKurs( String Valuta );
  public String getBase();
  public void update();
}



/***************************************
  ValutaServerProxy
    object interface for communicating
    with the valutaserver via sockets
***************************************/

class ValutaServerProxy implements ValutaProvider {
  int Portno;
  String Host;

  private ValutaTable ServerTable;
  // This is used for dummy ValutaServer

  public ValutaServerProxy( String host, int portno) {
    Portno = portno;
    Host = host;
    // The next two lines can be used for info while debugging
    //  System.out.println(" HOST " + Host +
    //        " port:" + Integer.toString(Portno));
    ServerTable = new ValutaTable();
    update();
  }

  public Enumeration getValutas() {
    return ServerTable.getValutas();
  }

  public Double getKurs( String Valuta ) {
    return (Double) ServerTable.getKurs( Valuta );
  }

  public String getBase() {
    return ServerTable.getBase();
```

```
  }

  public void update() {
    Socket sessionsocket;
    DataInputStream dis;

    try {

      // Send request

      sessionsocket = new Socket( Host, Portno );
      System.out.println(" Connected ");

      dis = new DataInputStream(sessionsocket.getInputStream());

      // Receive reply-header

      int valutacount = dis.readInt();

      byte BV[] = new byte[4];
      int bsc = dis.read( BV, 0, 4 );
      String BaseValuta = new String( BV, 0, 0, 3 );

      ServerTable.setBase(BaseValuta);
      ServerTable.setKurs(BaseValuta, new Double(100.00) );

      // Receive reply-body

      for (int i=0; i<valutacount; i++) {
        byte V[] = new byte[8];
        int cs = dis.read( V, 0, 8 );
        String Valuta = new String( V, 0, 0, 3 );
        double Kurs = dis.readDouble();

        ServerTable.setKurs( Valuta, new Double(Kurs) );
      }

      // Cleanup

      dis.close();
      sessionsocket.close();

    } catch (Exception e) {
      System.err.println( this.toString() +  ": Failed update");
    }

  }
}



/***************************************
  ValutaConverter
    A flexible ValutaProvider.
    Lets user specify the basevaluta
***************************************/

class ValutaConverter implements ValutaProvider {
  private ValutaTable UserTable;
  private ValutaProvider Server;

  public ValutaConverter( ValutaProvider S ) {
    Server = S;
    UserTable = new ValutaTable();
    UserTable.setBase( Server.getBase() );
    update();
  }
```

```
  public Enumeration getValutas() {
    return UserTable.getValutas();
  }

  public Double getKurs( String Valuta ) {
    return (Double) UserTable.getKurs( Valuta );
  }

  public void setBase( String Valuta ) {
    UserTable.setBase( Valuta );
    calculate();
  }

  public String getBase() {
    return UserTable.getBase();
  }

  public void update() {
    Server.update();
    calculate();
  }

  private synchronized void calculate() {
    // This private method recalculates the table
    // Must be synchronized to avoid concurrent calls
    Double nybasekurs = Server.getKurs( UserTable.getBase() );
    for (Enumeration Valutas = Server.getValutas();
         Valutas.hasMoreElements(); ) {
      String v = (String) Valutas.nextElement();
      Double k = new Double(100.00 *
                            Server.getKurs(v).doubleValue() /
                            nybasekurs.doubleValue());
      UserTable.setKurs( v, k );
    }
  }
}



/****************************************
  ValutaTable
    A table for (Valuta, Kurs) pairs
****************************************/

class ValutaTable {
  private Hashtable Table;
  private String BaseValuta;

  public ValutaTable() {
    Table = new Hashtable();
    BaseValuta = "***";
  }

  public void setBase( String Valuta ) {
    BaseValuta = Valuta;
  }

  public String getBase() {
    return BaseValuta;
  }

  public Double getKurs( String Valuta ) {
    return (Double) Table.get(Valuta);
  }

  public void setKurs( String Valuta, Double Kurs ) {
```

```
    Table.put( Valuta, Kurs );
  }

  public void delete( String Valuta ) {
    Table.remove( Valuta );
  }

  public Enumeration getValutas() {
    return Table.keys();
  }
}
```

Bilag B

Kildeteksten til server.c

```
/*
-----------------------
server.c
-----------------------
*/
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <string.h>
#include <errno.h>

/* Antagelser:
       maskinen har big-endian arkietektur
       double er 8 byte IEEE
       char er en byte
*/

typedef struct vk {
  char Valuta[8]; /* Tre tegn og '\0' */
  double Kurs;
} ValutaKurs;

int sessionsocket;
int listensocket;

struct hostent *hp;
int serverport = 7992;

int child;

long timevar;
char *ctime();

static int count = 0;

struct linger linger;

struct sockaddr_in myaddr_in;
```

```
struct sockaddr_in peeraddr_in;

char *localargv[2];

void childcatcher()
{
  pid_t child;
  child = waitpid( -1, NULL, WNOHANG );
  signal( SIGCHLD, childcatcher );
}

void intcatcher()
{
  printf("Closing server\n");
  close(listensocket);
  exit(0);
}

main(argc, argv)
int argc;
char *argv[];
{
  int addrlen;

  signal( SIGINT, intcatcher );
  signal( SIGCHLD, childcatcher );

  printf("ValutaServer started\n");

  localargv[1] = NULL;

  memset((char *) &myaddr_in, 0, sizeof(struct sockaddr_in));
  memset((char *) &peeraddr_in, 0, sizeof(struct sockaddr_in));

  myaddr_in.sin_family = AF_INET;
  myaddr_in.sin_addr.s_addr = INADDR_ANY;
  myaddr_in.sin_port = serverport;

  listensocket =  socket(AF_INET, SOCK_STREAM, 0);
     if (listensocket == -1) {
       perror(argv[0]);
       fprintf(stderr, "%s: unable to create socket\n", argv[0]);
       exit(1);
     }
  if (bind(listensocket, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
       perror(argv[0]);
       fprintf(stderr, "%s: unable to bind address\n", argv[0]);
       exit(2);
       }
  if (listen(listensocket,5) == -1) {
       perror(argv[0]);
       fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
       exit(3);
       }
  setpgrp();
  addrlen = sizeof(struct sockaddr_in);


  /* Main server loop:
       Wait for connection,
       Create server-process
  */
  for (;;) {
    while ( (sessionsocket =
     accept(listensocket, &peeraddr_in, &addrlen)) < 0) {
      if (errno != EINTR) {
        fprintf(stderr,"Something bad happened while accepting\n");
```

```
        exit(5);
      }
    }
    count++;
    switch (child = fork()) {
    case -1:
      fprintf(stderr,"Could not make serverthread\n");
      close(listensocket);
      close(sessionsocket);
      exit(6);
    case 0:
      /* Child */
      close(listensocket);
      server();
      exit(7);
    default:
      /* Parent */
      close(sessionsocket);
    }
  }
} /* end main() */


int server()
{
  char BaseVal[4];
  ValutaKurs transmitbuf[40];

  char *inet_ntoa();
  char *hostname;
  int len, len1;

  int bufcount = 0;
  int bufsize;

  int i;

  double Kurs;

  strcpy ( BaseVal, "DKK" );

  strcpy ( transmitbuf[bufcount].Valuta, "USD" );
  transmitbuf[bufcount].Kurs = 577.77;
  bufcount++;

  strcpy ( transmitbuf[bufcount].Valuta, "NOK" );
  transmitbuf[bufcount].Kurs = 89.888;
  bufcount++;

  strcpy ( transmitbuf[bufcount].Valuta, "DEM" );
  transmitbuf[bufcount].Kurs = 385.00;
  bufcount++;

  // Add more here.....

  bufsize = sizeof( ValutaKurs ) * bufcount;

  hp = gethostbyaddr((char *) &peeraddr_in.sin_addr,
                     sizeof (struct in_addr),
                     peeraddr_in.sin_family);
  if (hp == NULL) {
    hostname = inet_ntoa(peeraddr_in.sin_addr);
  }
  else {
    hostname = hp->h_name;
  }
```

```
  time(&timevar);

  printf("Startup from %s port %u at %s\n", hostname,
         ntohs(peeraddr_in.sin_port), ctime(&timevar));
  linger.l_onoff = 1;
  linger.l_linger = 1;
  if (setsockopt(sessionsocket, SOL_SOCKET, SO_LINGER,
      &linger, sizeof(linger)) == -1)
    {
       fprintf(stderr, "Server: Connection with %s aborted on error\n");
       exit(9);
    }

  if ( send(sessionsocket,(char *) &bufcount, sizeof(int) ,0)
       != sizeof(int) )
    printf(stderr,"Server: could not send count\n");

  if (send(sessionsocket, BaseVal, 4, 0) != 4 )
    printf(stderr,"Server: could not send baseval\n");

  if (send(sessionsocket,(char *) transmitbuf, bufsize,0) != bufsize)
    printf(stderr,"Server: could not send all data\n");

  close(sessionsocket);
  time(&timevar);
  printf("Completed %s port %u, %d requests, at %s\n", hostname,
         ntohs(peeraddr_in.sin_port), count, ctime(&timevar));

}
```